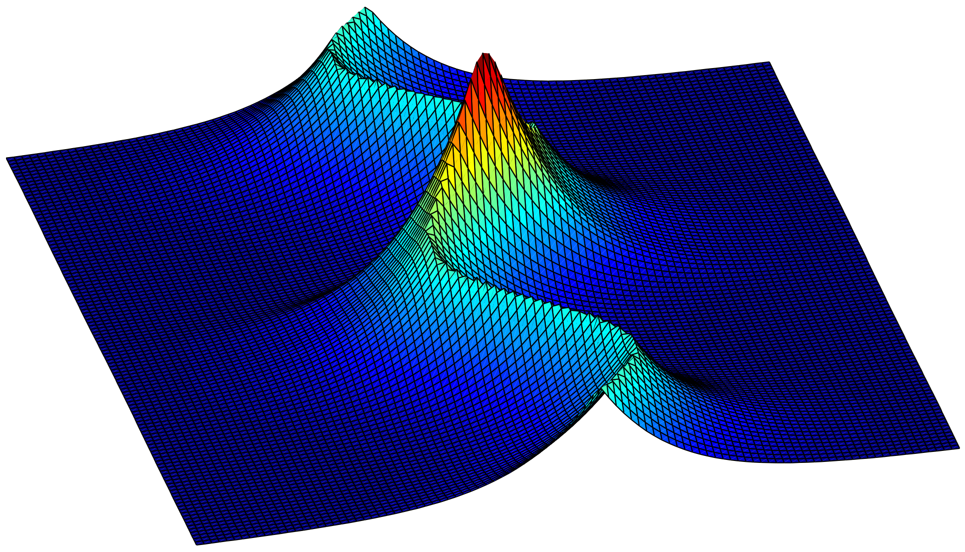
Introduction to Python



Michael J. Ware

Department of Physics and Astronomy
Brigham Young University

INTRODUCTION TO PYTHON

Michael J. Ware

Department of Physics and Astronomy
Brigham Young University

Last Revised: June 11, 2019

Acknowledgements

This book is based heavily on the book *Introduction to Scientific Computing in Python*, by Lance J. Nelson and Matthew R. Zachreson at BYU Idaho. With their permission, I've adapted their material to our use here at BYU Provo. In a sort of virtuous cycle, Ross Spencer and I had previously allowed Lance and Matthew to borrow material from our book *Introduction to Matlab* in the creation of their *Introduction to Scientific Computing in Python*. Thus, this work is a mix of content from the BYU Idaho Python book, our original Matlab book, and new and modified material specific to this edition. I express sincere thanks and acknowledgment to Ross, Lance, and Matthew for all of the work that has gone into development this material. I've put my name as author of this edition to reflect the fact that I am the one editing this particular offshoot of our combined work.

Preface

This book is a tutorial for physics students to get up to speed using Python for scientific computing as quickly as possible. It assumes that the reader is already familiar with the basics of scientific programming in another programming language, and does not spend time systematically going through the fundamentals of programming. This tutorial is designed to work hand-in-hand with the BYU Physics 430 lab manual. Each chapter in this tutorial is designed to give students enough understanding of the Python syntax and ecosystem to tackle a specific scientific computing lab, and in doing so will sample from a range of different topics.

Students in the class for which the book is designed have previously taken a three-credit introductory programming class in C++, a one-credit lab courses introducing them to Mathematica, and another one-credit lab course introducing them Matlab. In this book, we build on that foundation, without trying to re-teach (at least not too much) the material already covered in prior classes. We also present material in the order needed to complete the associated labs rather than as a systematic and complete treatment of each topic before moving to the next.

As you find mistakes or have suggestions, send them to me at ware@byu.edu.

Contents

Chapter 1

Numpy, Scipy, and Plotting

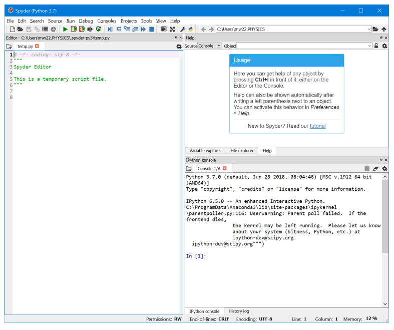
1.1 Get Anaconda Installed

Python is a popular general-purpose programming language that has become a standard language for many areas of scientific computing. It is open source, and there are many implementations of Python, many development environments for it, and multiple versions of the programming language itself. We will use the Anaconda distribution for Python version 3.7. The Anaconda distribution is geared toward scientific computing, is available as a free download on all major platforms, and comes with an integrated development environment (IDE) called Spyder. Anaconda Python is installed on the lab computers, and we recommend that you also install it on your personal computers so you can work at home (see anaconda.com).

1.2 Your First Program

Launch the Spyder IDE. You should see a window similar to the one in Fig. 1.1. The Spyder IDE interface is divided into three main windows:

- The code editor on the left is where you edit your Python code files (usually with a .py extension).
- The top right pane has three default tabs: a variable explorer where you can view current values stored in memory, a file explorer where you can browse your files, and a help tab where you can ask questions
- The lower right pane is a console window where your output will be displayed and you can issue Python commands directly to be evaluated.



```
# -*- coding: utf-8 -*-
"""
Spyder Editor

This is a temporary script file.
"""
```

Usage

Here you can get help of any object by pressing **Ctrl+I** in front of it, either on the Editor or the Console.

Help can also be shown automatically after writing a left parenthesis next to an object. You can activate this behavior in *Preferences > Help*.

New to Spyder? Read our tutorial

```
Python 3.7.0 (default, Jun 28 2018, 08:04:48) [MSC v.1912 64 bit
(AMD64)]
Type "copyright", "credits" or "license" for more information.

IPython 6.5.0 -- An enhanced Interactive Python.
C:\ProgramData\Anaconda3\lib\site-packages\ipykernel
\parentpoller.py:116: UserWarning: Parent poll failed.  If the
frontend dies,
                the kernel may be left running.  Please let us know
                about your system (bitness, Python, etc.) at
                ipython-dev@scipy.org
  ipython-dev@scipy.org""")

In [1]:
```

Figure 1.1 The Spyder IDE window.

To write your first program, press the "New File" button on the toolbar, erase any auto-generated text so you have a blank window, and then type

```
print('Hello World')
```

Save your program in a new directory (where you will save all your work for this class) with the name hello.py, then click the green arrow above the editor. Spyder may ask you in which console you'd like to execute the program. Just accept the default values, and then look down at the console window. There you will see some code that Spyder auto-generated to run your program, and under it should be the output from your program:

```
Hello World
```

Congratulations, you just executed your first Python program. Change the text string `'Hello World'` to something else, and then click the green arrow again. Notice that Spyder saves your code and executes the program again. If you get tired of clicking the green arrow, F5 is the keyboard shortcut to save and execute the code shown in the editor.

1.3 The Python Console

Spyder's console window is a powerful environment called Interactive Python (or IPython for short) where you can directly enter Python commands. Put your cursor in the console window at the `In [1]:` prompts, type

```
1+1
```

and press enter. You should see the answer, stored in a variable `Out[1]`. You can use this output in later calculations by typing the following at the `In[2]:` prompt:

```
Out[1]+2
```

Notice that your new result is stored in `Out[2]`. IPython behaves a lot like a Matlab's command window, with a series of inputs and variable values accumulating in the workspace. When you run your Python programs, your variables are placed in the console's workspace and can be accessed from the command line afterward. With your cursor in the command window, press the Up-Arrow key, and notice that you can access your command history with the up and down arrow keys.

The console can be useful for quick calculations or for looking at data when debugging, but you should do most of your programming in the editor. Add the following line to your hello.py program

```
1+1
```

and run it again. Note that the answer to `1+1` is *not* displayed in the console when you run the program. Now switch your program to read

```
print(1+1)
```

and run it again, and note that the answer displayed in all its glory. Python evaluates each line of code, but will not display the result of a calculation in the console unless you `print` it.

As we go through the remainder of the text, type all the indented example code into a *.py file in the editor by hand (don't copy and paste) and execute it using the green arrow (or the F5 keyboard shortcut). This method of interacting with the code will help you better process and understand what each command does. It forces you to read the code like Python will: one line at a time, top to bottom. Also, place each command on a separate line and don't indent any lines of code when you type them in. Python really cares about white space. We'll learn more about that in the next chapter.

1.4 Variables and Data Types

Variables in Python don't need to be declared before being used. You declare a variable and assign it a value in one statement using the assignment operator (=), like this

```
x = 20
```

(Did you type the line of code into your program and execute it? If not, do so now and get in that habit.) This statement creates the variable `x` and assigns it a value of 20. Python didn't print anything since we didn't include a `print` command. To convince yourself that the variable `x` was defined, click on the "Variable explorer" tab in the upper-right pane of Spyder to see that the variable exists, and has a value of 20. Also type `x` in the console window and hit enter, and note that Python displays its value. Now add the line

```
x = x + 1
```

to your program, execute it, and look at the new value of `x` to convince yourself that the program executed correctly. Multiple variables are defined by putting the assignments on separate lines, like this

```
a = 2
b = 4
c = a * b
```

Sometimes you may want your program to prompt the user to enter a value and then save the value that the user inputs to a variable. This can be done like this:

```
a = input('What is your age?')
```

When you run this line of code, you will be prompted to enter your age. When you do, the number you enter will be saved to the variable `a`.

Variable names must start with a letter or an underscore and consist of only letters, numbers, and underscores. Variable names are case sensitive, so watch your capitalization.

1.5 Integers

The simplest type of numerical data is an integer. Python implicitly declares variables as integers when you assign an integer values to the variable, as we did above. You can perform all the common mathematical operations on integer variables. For example:

```
a = 20
b = 15
c = a + b   # add two numbers
d = a/b     # floating point division
```

☞ Notice that we've introduced the Python commenting syntax here: any text following the symbol # on a line is ignored by the Python interpreter.

```
i = a//b   # integer division
r = a % b  # return only the remainder(an integer) of the division
e = a * b  # multiply two numbers together
f = c**4   # raise number to a power (use **, not ^)
```

☞ In Python 2.x the single slash between two integer variables performs integer division. If you are using this older version of Python, you should be sure to use floats when you want regular division.

Performing an operation on two integers *usually* yields another integer. This can pose an ambiguity for division where the result is rarely an exact integer. Using the regular division operator (as in the line defining `d` above) for two integers yields a float, whereas the double slash (used in the line defining `i`) performs integer division. Look at the variable values in the variable explorer and compare them to your code to convince yourself that you understand the distinction between these two types of division.

1.6 Float variables

Most calculations in physics should be performed using floating point numbers, called floats in Python. Float variables are created and assigned in one of two ways. The first way is to simply include a decimal point in the number, like this

```
a = 20.
```

You can also cast an integer variable to a float variable using the `float` command

```
a = 20
b = float(a)
```

When a float and an integer are used in the same calculation, like this

```
a = 0.1
b = 3 * a  #Integer multiplied by a float results in a float.
```

the result is always a float. Only when all of the numbers used in a calculation are integers will the result be an integer. Floats can be entered using scientific notation like this

```
y = 1.23e15
```

1.7 Functions and Help

`abs(x)`	Find the absolute value of x
`divmod(x,y)`	Returns the quotient and remainder when using long division.
`x % y`	Return the remainder of $\frac{x}{y}$
`float(x)`	convert `x` to a float.
`int(x)`	convert `x` to an integer.
`round(x)`	Round the number `x` using standard rounding rules

Table 1.1 A sampling of built-in functions commonly used with integers and floats.

Table 1.1 shows some common housekeeping functions that you can use with float variables. Practice using one of these functions in your code. Then place your cursor on the function name in your code and press Ctrl-I (think “I is for information) to display detailed information about this function in the help panel. You can use this method for getting details of calling conventions and function usage for all of the functions we use in this class.

1.8 Boolean variables

Boolean variables store one of two possible values: `True` or `False`. A boolean variable is created and assigned similar to the other variables you've studied so far

```
q = True
```

Boolean variables will be useful when we use loops and logical statements.

1.9 String Variables

String variables contain a sequence of characters, and can be created and assigned using quotes, like this

```
s='This is a string'
```

You may also enclose the characters in double quotes, which is mostly used when there is are single quote in your string:

```
t="Don't worry"
```

Some Python functions require options to be passed to them as strings. Make sure you enclose them in quotes, as shown above. Some commonly used functions for strings are shown in Table 1.2

len(c)	Get the number of characters in the string `c`
`a.count(b)`	Count the number of occurrences of string `b` in string `a`
`a.lower()`	Convert upper case letters to lower case
`a[x]`	Access element `x` in string `a`
`a[x:y:z]`	Slice a string, starting at element `x`, ending at element `y`, with a step size of `z`
`a + b`	Concatenate strings `a` and `b`.

Table 1.2 A sampling of "housekeeping" functions for strings.

1.10 Formatting Printed Values

The `print` command will take pretty much any variable dump it to screen in a format of Python's choosing. However, there are times when you'll want to be more careful about the formatting of your print statements. For example:

```
a = 22
b = 3.5
print("I am {:d} years old and my GPA is: {:5.2f}".format(a, b))

#This style also works
joe_string="My GPA is {:5.2f} and I am {:d} years old."
print(joe_string.format(b,a))
```

Notice the structure of this formatted print statement: A string followed by the `.` operator and the `format()` method. Those of you familiar with object-oriented coding will notice that this is an example of Python's fully object-oriented structure. The variables to be printed are provided as arguments to the `format` method and are inserted into the string sequentially where curly braces (`{}`) are found. The characters inside of the curly braces are format codes that indicates how you would like the variable formatted when it is printed. The `:d` indicates an integer variable and `:f` indicates a float. The `5.2` in the float formatting indicates that I'd like the number to be displayed with at least 5 digits and 2 numbers after the decimal. See Table 1.3 for format examples.

`{}`	Use the default format for the data type
`{:4d}`	Display integer with 4 spaces
`{:.4f}`	Display float with 4 numbers after the decimal
`{:8.4f}`	Display float with at least 8 total spaces and 4 numbers after the decimal
`{:1.2e}`	Scientific notation with 2 digits after the decimal

Table 1.3 Formatting strings available when printing.

1.11 Functions and Libraries (Numpy)

The syntax for calling Python functions is fairly standard: you type the name of the function and put parentheses `()` around the arguments, like this

```
x = 3.14
y = round(x)
```

If the function requires more than one argument or returns more than one value, you separate the arguments and outputs with commas, like this

```
x = 3.14
y = 2
b,c = divmod(x,y)
```

Pause a moment after executing this code to compare the values of `b` and `c` with the code above, and convince yourself that you understand how function calls work with multiple arguments and outputs in Python.

Python contains a set of standard functions, called native functions, that are always ready to go whenever you run Python. All the functions we've used so far fall into this category. While useful, these native functions are inadequate for scientific computing. However, user communities have created extensive collections of functions called *libraries* that you can import into Python to extend its native capabilities.

For example, Python doesn't natively include the sine and cosine functions. However, virtually any mathematical function that you will need to perform a scientific calculation can be found in a library called NumPy (say "num pie", not something that rhymes with grumpy). To use this library, we add an `import` statement to the beginning of our program, and then reference the functions in this library like this

```
import numpy

x = numpy.pi          # Get the value of pi
y = numpy.sin(x)      # Find the sine of pi radians
z = numpy.sqrt(x)     # Take the square root of pi
```

After importing the library, the code "`numpy.`" before a function tells Python to look in the NumPy library to find the function rather than in the native functions. If you just type

```
sqrt(x)
```

Python will look for a native function named `sqrt` and will give you an error.

A library can be imported and then referenced by a different name like this:

```
import numpy as np

x = np.sqrt(5.2)   # Take the square root of 5.2
y = np.pi          # Get the value of pi
z = np.sin(34)     # Find the sine of 34 radians
```

`sin(x)`
`cos(x)`
`tan(x)`
`arcsin(x)`
`arccos(x)`
`arctan(x)`
`sinh(x)`
`cosh(x)`
`tanh(x)`
`sign(x)`
`exp(x)`
`sqrt(x)`
`log(x)`
`log10(x)`
`log2(x)`

Table 1.4 A very small sampling of functions belonging to the `numpy` library.

This syntax tells Python "I'm going to call the `numpy` library `np`." We recommend that you use this syntax for the NumPy unless there is some compelling reason to do otherwise. It is so common to do this, that in this manual we will usually omit writing the code

```
import numpy as np
```

at the beginning of each example and just assume that you'll include it whenever you see functions with the "`np.`" prefix.

Virtually any mathematical function that you will need to perform a scientific calculation can be found in the library NumPy or in another common library called SciPy (say "sigh pie", not "skippy"). These two libraries are designed to work together. Here is an example showing how to use the J_0 Bessel function:

```
import numpy as np
import scipy.special as sps

a = 5.5
b = 6.2
c = np.pi/2
d = a * np.sin(b * c)
e = a * sps.j0(3.5 * np.pi/4)
```

If `numpy` doesn't have a mathematical function you need, then `scipy.special` probably has it. Table 1.5 provides a small sampling of its functions. See SciPy's online help for a more extensive listing.

`airy(z)`	Airy function
`j0(z)`	0th order Bessel function of 1st kind
`j1(z)`	1st order Bessel function of 1st kind
`jv(v,z)`	Bessel function of 1st kind, order v
`yv(v,z)`	Bessel function of 2nd kind, order v
`kv(n,z)`	Modified Bessel function of 2nd kind of integer order n
`iv(v,z)`	Modified Bessel function of 1st kind of real order v
`hankel1(v,z)`	Hankel function of 1st kind of real order v
`hankel2(v,z)`	Hankel function of 2nd kind of real order v

Table 1.5 A very small sampling of functions belonging to the `scipy.special` library.

1.12 Numpy Arrays

In scientific computing we often do calculations involving large data sets. Say you have a large set of numbers, x_i and you want to calculate the summation

$$\sum_{i=1}^{N}(x_i-5)^3. \tag{1.1}$$

You could use loops to calculate each term in this sum one-by-one, and then add them up to compute the final sum. However, the Numpy library provides a much slicker method for making these kinds of calculation using something called a NumPy array. For example, if `x` were a Numpy array containing all of the x_i values, the code to evaluate the sum in Eq. (2.1) is simply:

```
s=sum((x-5)**3)
```

This code subtracts 5 from each element in the array, cubes the result for each element, and then sums the resulting elements all without a loop in sight. Let's explore the details of how to do calculation using Numpy arrays.

Array Creation

Our first task is to create a NumPy array. Here you have several options. You can enter small arrays by providing a list of numbers to the NumPy `array` function, like this:

```
a = np.array([1,2,3,4,5,6,7,8,9,10])
```

Another common method for creating arrays is to use the function `arange` (think the two words "a range" not the one word "arrange" so you don't misspell this function). This function creates an array of evenly spaced values from a starting value, and ending value, and a step size, like this:

```
b = np.arange(0,10,.1)
```

This code creates the following array:

```
[0,.1,.2,.3,.4,.5,.6.... 9.5,9.6,9.7,9.8,9.9]
```

Notice that this array does not include the value at the end of the range (i.e. 10). Also, for those of you coming from a Matlab background, notice that the step size (0.1 in this case) comes third, not between the two limits as is done in Matlab.

The `linspace` function creates an array by specifying the starting value, ending value, and the number of elements that the array should contain. For example:

```
x = np.linspace(0,10,10)
```

This will create an array that looks like this:

```
[0,1.111,2.222,3.333,4.444,5.555,6.666,7.777,8.888,10]
```

When you use the `linspace` function, you aren't specifying a step size. You can ask the `linspace` command to tell you what step size results from the array that it created by adding `retstep=True` as an argument to the function, like this:

```
x,dx = np.linspace(0,10,10,retstep = True)
```

Note that since `linspace` is now returning two values (the array and the step size), we need two variables on the left hand side of the equals sign. Table 1.6 gives several other options for creating NumPy arrays.

`logspace`	Returns numbers evenly spaced on a log scale. Same arguments as `linspace`
`empty`	Returns an empty array with the specified shape
`zeros`	Returns an array of zeros with the specified shape
`ones`	Returns an array of ones with the specified shape.
`zeros_like`	Returns an array of zeros with the same shape as the provided array.
`fromfile`	Read in a file and create an array from the data.
`copy`	Make a copy of another array.

Table 1.6 A sampling of array-building functions in numpy. The arguments to the functions has been omitted to maintain brevity. See online documentation for further details.

Math with Arrays

Once a NumPy array object is created, a whole host of mathematical operations become available—you can square the array and Python knows that you want to square each element, you can add two arrays together and Python knows that you want to add the individual elements of the arrays, etc. Here are some examples:

```
x = np.array([2,3,5.2,2,6.7,3])      # Create the array x
y = np.array([4,8,9.8,2.1,8.2,4.5])  # Create the array y

c = x**2   # Square the elements of x
d = x + 3  # Add 3 to every element of x
e = x * 5  # Multipy every element of x by 5
f = x + y  # Add the elements of x to the elements y
g = x * y  # Multiply the elements of x by the elements of y
```

In most respects, math with Numpy arrays behaves like the "dotted" operators with Matlab matrices, where the operations are performed on corresponding elements. However, NumPy arrays are objects, so assigning one array to another, like this

```
x = np.array([2,3,5.2,2,6.7,3])
y = x
```

does *not* copy the values of `x` into a new variable `y`. Instead, it creates a new pointer to the existing object `x`, so that `x` and `y` now refer to the same object. Try changing a value of `y` and note that it also changes the value in `x`. To make a copy of an array, do this

```
x = np.array([2,3,5.2,2,6.7,3])
y = np.copy(x)
```

Switch the values in `x` and `y` and notice that they are now independent variables.

This array copy business is important and often misunderstand. This results in frustrated students when they try using the wrong syntax. You should probably go back and re-read the last paragraph and make sure you understand the distinctions to save yourself some frustration.

Functions of Arrays

Mathematical functions like $\sin(x)$ and $\sinh(x)$ from the NumPy and SciPy libraries can accept arrays as their arguments and the output will be an array of function values. However, functions from other libraries, such as the `math` library, are not designed to work on arrays. Here is an example of this issue:

```
import numpy as np
import math
x = np.array([2,3,5.2,2,6.7])
c = np.sin(x)   # Works just fine, returning array of numbers.
d = math.sin(x) # Returns an error.
```

The `math` library is much less capable than `numpy`, so we don't recommend using it for scientific coding. We just introduced it here to emphasize that NumPy arrays usually need to be used with NumPy functions.

Accessing Elements of Numpy Arrays

You can access individual array elements in a NumPy array using square brackets to index the elements like this

```
a = np.array([1,2,3,4,5,6,7,8,9,10])
x = a[1]
```

Look carefully at the value of `x` after running this code, and convince yourself that Python array indexes are zero-based. That is, the first element has index 0, the second element has index 1, and so forth.

You can extract a contiguous range of values using the colon command in square brackets like this

```
b = a[1:4]
c = a[0:6:2]
```

Note that there can be three numbers inside the brackets, each separated by the `:` symbol, like this `[x:y:z]`. The extracted elements start at element `x`, end just before element `y` (i.e. the element at `y` is not in the range), and steps in increments of `z`. Note that the last number is optional, and if omitting a default value of 1 is used for the step size. Python uses zero-based indexing, so `a[1:4]` starts at the second element, not the first.

You can also use negative indexes in the colon command to count from the end of the array toward the beginning:

```
b = a[-1]
c = a[1:-2]
```

The index `-1` refers to the last element, index `-2` refers to the second to last element, etc. You can also omit one of the endpoint arguments to go all the way to the beginning or end of the array, like this:

```
b = a[1:]
c = a[:3]
```

1.13 Making *x*-*y* Plots

Plots of *x* vs. *y* can be made with a library called `matplotlib`. This library creates plots using a syntax essentially the same as the plotting functions in Matlab using NumPy arrays as data inputs. Here is an example of the basic syntax:

☞ Spyder uses IPython for its console which automatically shows plots. For other consoles, you'll need to add the command `plt.show()` after making a plot to display it on the screen.

```
import matplotlib.pyplot as plt
import numpy as np
x = np.arange(0,10,0.01)
y = x**2

plt.figure(1)
plt.plot(x,y)
```

If you want to see the actual data points being plotted, you can add the string `'ro'` inside of the plot command

```
plt.plot(x,y,'ro')
```

The `'r'` means make the data points red and the `'o'` means plot circle markers. All of the usual plot commands from Matlab work about the way you'd expect. For instance, here is an example of overlaying two graphs

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

x = np.linspace(0,2.*np.pi,100)
y1 = np.sin(2*x)
y2 = np.cos(3*x)

plt.figure(1)
plt.plot(x,y1,'r.',x,y2,'b')
plt.legend(['Sine of x','Cosine of x'])
plt.xlabel('x')
plt.title('Two Trig Functions')

plt.figure(2)
plt.plot(y1,y2)
plt.title('Something Fancy')
```

1.14 Customizing the IDE

The Spyder development environment can be customized to suit your preferences. We recommend at least the following changes, and you can also look around at some of the other options while making these changes.

1. By default plots are displayed inline in the console. To have them pop out in their own window go to the Tools menu in Spyder and select Preferences → IPython Console → Graphics Tab and set the Graphics backend to "Automatic." The popup windows will allow you to zoom in on different parts of the graph. The separate window will also allow us to make simple animations by repeatedly redrawing plots in the same window. If you don't make this change, your animations will scroll by as separate plots in the console window rather than displaying on the same axis.

2. By default, all the variables that you define by direct interaction with the console and during the execution of your code remain in memory and available for subsequent use, both in the console and subsequent executions of programs. Sometimes this can be convenient, but it can also cause debugging problems. For example, if you define a variable using the console and then write a program that depends on that variable, then the mode of execution of your program depends on what you've manually entered

in the console. When learning to program, we recommend that you have Spyder clear all variables before executing a program. To do this go to the Tools menu in Spyder and select Preferences → Run and check the "Remove all variables before execution" option.

Chapter 2

Lists, Loops, Logic, and 2D Arrays

2.1 Lists

You'll encounter lists quite a bit in Python. In fact we already used some in the previous chapter without telling you. Lists are ordered collections of values. To create a list, put the elements inside of square brackets like this:

```
x = [5.6,2.1,3.4,2.9]
```

You can make a list of any variable type. Here is a list of strings:

```
a = ['electron','proton','neutron']
```

List elements don't need to be the same data type, so this list is perfectly valid

```
a = ['Ben',90,'Chad',75,'Andrew',22]
```

You can even define a list of lists:

```
a = [[4,3,2],[1,2.5,9],[4.2,2.9,10.5],[39.4,1.4],[2.7,98,42,16.2]]
```

When looking at the numerical lists above, you might be tempted to think of lists as mathematical matrices. Don't. Lists are not really designed for mathematics. To convince yourself of this, add the statement

```
a[1][2] = 'george'
```

after the list of lists above and then examine a. You can't calculate on elements that aren't numbers! As another example, execute this code

```
import numpy as np
x = np.array([1,2,3]) # x is a NumPy array
print(x+x)            # adding two NumPy arrays adds the values

y = [1,2,3]           # y is a Python list
print(y+y)            # adding lists concatenates the two lists
```

Study the output from this code and convince yourself that NumPy arrays are the right object for doing math, not lists. However, lists are otherwise quite useful in Python. Table 2.1 gives a list of common list functions. Note that lists are objects with methods (e.g. `a.sort()`), but can also be used as arguments to other functions (e.g. `len(a)`. Like many things in Python, the reasons for the different syntaxes are known only to those who developed Python. For example, why does `len(a)` work, but `a.len()` return an error? The organic nature of the development of Python syntax results in some inconsistencies in syntax that just need to be learned.

`a[x]`	Access element `x` in list `a`
`a[x:y:z]`	Extract a slice of list `a`
`a.append(x)`	Append `x` to list `a`
`a.pop()`	Remove the last element of list `a`.
`len(a)`	Find the number of elements in `a`
`a.insert(x,y)`	Insert `y` at location `x` in list `a`
`a.sort()`	Sort list `a` from least to greatest.
`filter(f,x)`	Filter list `x` using the criteria function `f` (see lambda functions).
`a.reverse()`	Reverse the order of list `a`.
`a.index(x)`	Find the index where element `x` resides.
`a + b`	Join list `a` to list `b` to form one list.
`max(a)`	Find the largest element of `a`
`min(a)`	Find the smallest element of `a`
`sum(a)`	Returns the sum of the elements of `a`

Table 2.1 A sampling of "housekeeping" functions for lists.

2.2 The `range` Function

Python's `range` function represents a sequence of integer values that behaves a lot like a list. The syntax of this function is like this

```
a = range(10,20,2)
```

This statement represents a sequence of integers that starts at 10, ends at 18, and steps in increments of 2. The `range` function does not include the item at the endpoint of the range. In Python 2.x, the `range` function returned an actual list object, but in Python 3.x (which we are using) it returns an object called an iterator. Rather than store a huge list of numbers, an iterator just stores the rule for generating the number at any index. So, if you print the range variable

```
print(a)
```

Python will just tell you that `a` represents a range, but if you execute this code

```
a[2]
```

you can get an actual value.

The `range` function can also be called with 1 or 2 arguments and default values will be assigned to the missing ones.

```
b = range(3,9)   # starts at 3, ends at 8, steps of 1 (default)
c = range(10)    # starts at 0 (default), ends at 9, steps of 1 (default)
```

2.3 For Loops and Shorthand Assignments

The primary use for the `range` function in writing `for` loops. A `for` loop iterates over each element in a list or range. Here is a `for` loop used to sum a list of values

☞ Usually you would just use a `sum` function, but we are learning about loops here.

```
s = 0
for x in [3,42,1,9.9]:
    s = s + x
print(s)
```

This code iterates over the list `[3,42,1,9.9]`, assigning the variable `x` each value in the list, one by one, until it reaches the end of the list. Notice the required colon at the end of the `for` statement that marks the beginning of the loop, and also that there is no code indicating the end of the `for` loop. Python uses indentation to denote the group of code that will be executed during each iteration. To see this, indent the `print(s)` statement to the same level as the statement above it and execute the code. Note that the `print` function now executes on each iteration of the loop instead of just displaying the final value of `s`.

You can also iterate over the values from the `range` function, like this

```
for y in range(5,50,3):
    print(y)
```

The `range` function only produces integers, so the most common use case is to use it with a `for` loop to index a separate array, like this

```
x = np.arange(5,10,.1)

for n in range(len(x)):
    print(x[n])
```

You can also iterate directly over the values of an array, like this:

```
x = np.arange(5,10,.1)

for y in x:
    print(y)
```

To practice `for` loops, let's build a loop to calculate the sum

$$\sum_{n=1}^{1000} \frac{1}{n^2}$$

```
s = 0
for n in range(1,1001):
    s += 1/n**2
print(s)
```

The code `s += 1/n**2` is equivalent to `s = s + 1/n**2`, but runs a little faster and is a little bit easier to read (once you get used to it). See Table 2.2 for more shorthand notations. Here's another example of a loop used to calculate the value of $p = 20!$ (20 factorial) using another shorthand notation

```
p = 1
for n in range(1,21):
    p *= n   # Multiply p by n
print(p)
```

Remember that the range function starts at the first argument (i.e. 1), and goes up to, but not including the second argument, so our product will only go up to $n = 20$.

Operation	Shorthand
`a = a + 1`	`a += 1`
`a = a - 2`	`a -= 2`
`a = 5*a`	`a *= 5`
`a = a/c`	`a /= c`
`a = a % 10`	`a %= 10`
`a = a**3`	`a **= 3`
`a = a // 12`	`a //= 12`

Table 2.2 A list of shorthand variable reassignment notation.

2.4 Logical Statements

Often we only want to run a section of code only when some condition is satisfied. This requires the use of logic. The simplest logic is the `if`\\`elif`\\`else` statement, which works like this:

```
a = 1
b = 3
if a > 0:
    c = 1
else:
    c = 0

if a >= 0 or b >= 0:   # If condition 1 met
    c = a + b
elif a > 0 and b >= 0: # If condition 2 met
    c = a - b
else:  # If neither condition is met.
    c = a * b
```

Note the locations of colons in these statements, and also note that indentation again defines the regions of code that execute as a group. Table 2.3 lists the standard elements for constructing logical statements.

`==`	Equal
`>=`	Greater than or equal
`>`	Greater than
`<`	Less than
`<=`	Less than or equal
`!=`	Not equal
`and`	True if both conditions joined by `and` are true
`or`	True if either of the conditions joined by `or` are true
`not`	True if the following condition is false.

Table 2.3 Python's logic elements.

A word of caution about comparing Python floats is in order here. Because of the way floats are represented in a computer, the number that is stored in a float is often not *exactly* the number that you think it is. For instance, execute the following, and study the output:

```
a = 0.1
b = 3 * a
c = 0.3
print(b==c)  # Are they the same number?
print(b)     # It sure looks like they are the same.
print(c)     # It sure looks like they are the same.
print(" {:.45f} ".format(b))  #b--- out to 45 decimal places
print(" {:.45f} ".format(c))  #c--- out to 45 decimal places
```

This can cause problems when you are comparing two numbers that you think should be equal but actually aren't equal in the computer. The take home here is that *comparing two floats to see if they are equal is a bad idea.* A better way to check to see if two floats are equal (or close enough that we can say they are equal) is to check if the absolute value of their difference is very small, like this:

```
a = 0.1
b = 3 * a
c = 0.3
print(abs(b - c) < 1e-10)
```

2.5 While Loops

A `while` loop iterates until a certain condition is met. This loop is a good choice when you don't know beforehand exactly how many iterations of the loop will be executed but rather what condition you want to be met before finishing. As an example, let's compute the sum

$$\sum \frac{1}{n^2} \tag{2.1}$$

by looping until the terms become smaller than 1×10^{-10}.

```
term = 1   # Load the first term in the sum
s = term   # Initialize the sum
```

```
n = 1      # Set a counter
while term > 1e-10:  # Loop while term is bigger than 1e-10
    n += 1           # Add 1 to n so that it will count: 2,3,4,5
    term = 1./n**2   # Calculate the next term to add
    s += term        # Add 1/n^2 to the running total
```

This loop will continue to execute until `term` is less than 10^{-10}. Note that indentation again defines the extent of the code that will execute each iteration. Unlike the `for` loop, you have to do your own counting in a `while` loop. Be careful about what value `n` starts at and when it is incremented (`n+=1`). Also notice that `term` must be assigned prior to the start of the loop. If it isn't the loop's first logical test will fail and the loop won't execute at all.

The `break` statement

Sometimes `while` loops are awkward to use because you can get stuck in an infinite loop if your check condition is never true. The `break` command is designed to help you here. When `break` is executed in a loop the script jumps to just after the end at the bottom of the loop. The `break` command also works with `for` loops. Here is our sum loop rewritten with break

☞ If you are stuck in a loop, you can force your program to stop by pressing ctrl+c or cmd+c

```
term = 1   # Load the first term in the sum
s = term   # Initialize the sum
n = 1      # Set a counter
while term > 1e-10: # Loop while term is bigger than 1e-10
    n += 1          # Add 1 to n so that it will count: 2,3,4,5
    term = 1./n**2  # Calculate the next term to add
    s += term       # Add 1/n^2 to the running total
    if n > 1000:
        print('This is taking too long. I''m outta here...')
        break
```

The `continue` statement

Another statement that is used with loops is the `continue` statement. When `continue` is used, the remainder of the code for the current iteration is skipped and the next iteration of the loop begins. If you wanted to do the sum in Eq. (2.1) but only include those terms for which n is a multiple of 3, you could do this

```
term = 1   # Load the first term in the sum
s = term   # Initialize the sum
n = 1      # Set a counter
while term > 1e-10: # Loop while term is bigger than 1e-10
    n +=  1         # Add 1 to n so that it will count: 2,3,4,5
    if n % 3 != 0:
        continue    # Skip the rest of the code, start the next iteration
    term = 1./n**2  # Calculate the next term to add
    s += term       # Add 1/n^2 to the running total
```

Now, when the value of `n` is not a multiple of 3, the last two lines of code in the loop will be skipped.

2.6 2D NumPy Arrays–Matrices

We introduced 1D NumPy arrays in the last chapter. The syntax for two-dimensional arrays (matrices) builds on this foundation. Small matrices can be built by giving the `array` function an argument that is a list of lists, like this

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

which could be interpreted as this matrix:

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \tag{2.2}$$

Once you have a 2D array, single elements can be extracted with this syntax:

```
b = a[0,2]
```

This code extracts the number 3, the element in the first row and third column in the array `a`. If you wanted to slice out the following 2×2 sub-matrix:

$$\begin{pmatrix} 5 & 6 \\ 8 & 9 \end{pmatrix} \tag{2.3}$$

you could do it like this:

```
b[1:3,1:3]  # Slice out a sub-array
```

If you want all of the elements in a given dimension, use the `:` alone with no numbers surrounding it. For example, the following:

```
b[:,1:3]
```

would extract all of the rows on columns 1 and 2:

$$\begin{pmatrix} 2 & 3 \\ 5 & 6 \\ 8 & 9 \end{pmatrix} \tag{2.4}$$

This kind of slicing can't be done with lists.

Creating Larger Matrices

You can create larger arrays using several functions from NumPy. To create an array full of zeros, you use this syntax:

```
A = np.zeros((100,200))
```

This creates an array with 100 rows and 200 columns, completely full of zeros. The argument that you provide to the `zeros` function, a set of numbers surrounded by round parentheses, is a data type called a *tuple*. Tuples behave a lot like lists, except they are immutable (i.e. after you create a tuple, you can't change the item values). We won't use tuples a whole lot in computational physics, but you should be aware of what they are since you periodically need to create them as arguments for functions. You can also ask a NumPy array what size it is using the array's `shape` property, like this

```
s = A.shape
```

Notice that the `shape` property returns a tuple.

To create an array full of ones, you use this syntax:

```
B = np.ones((100,200))
```

You can also create an identity matrix like this

```
I = np.identity(100)
```

Since identity matrices must be square, it only requires one number as an argument rather than a tuple.

Math with Matrices

Once a 2D array is defined, it behaves as a matrix and you can perform all the standard linear algebra functions with SciPy's linear algebra module. The matrix multiplication operator is the `@` symbol. Here are a few examples illustrating matrix operations:

```
import numpy as np
import scipy.linalg as la

a = np.array([[1,2],[3,4]])  # Create 2 x 2 matrix
b = np.array([[5,6],[8,9]])  # Create 2 x 2 matrix
col = np.array([[3],[4]])    # Create 2 x 1 column vector

c = a.T        # Transpose the matrix
e = a.conj().T # Find conjugate transpose of matrix
f = a @ b      # Matrix multiplication
g = b @ col    # Multiply matrix b to column vector
d = la.inv(a)  # Find inverse of matrix
```

☞ NumPy also has a linear algebra module. The SciPy version can do all that the NumPy version can, plus it has more advanced functions. In addition, the SciPy linear algebra package has some optimizations to make it run faster. We recommend you always use the SciPy linear algebra package.

The `linalg` name space within SciPy provides lots of matrix functions. Pretty much anything you learned about in your linear algebra class is available there.

2.7 Solving a Set of Linear Equations

To illustrate the use of matrix computations, let's solve a set of linear equations. Here is an example of a set of two linear equations, with two unknowns:

$$3x + y = 9 \qquad x + 2y = 8 \tag{2.5}$$

This problem can be represented in matrix form like this:

$$\mathbf{A}\mathbf{x} = \mathbf{b} \tag{2.6}$$

where

$$\mathbf{A} = \begin{pmatrix} 3 & 1 \\ 1 & 2 \end{pmatrix}, \tag{2.7}$$

$$\mathbf{b} = \begin{pmatrix} 9 \\ 8 \end{pmatrix}, \tag{2.8}$$

and

$$\mathbf{x} = \begin{pmatrix} x \\ y \end{pmatrix}. \tag{2.9}$$

SciPy has a function called `solve` that will solve this problem like this

```
import numpy as np
import scipy.linalg as la

a = np.array([[3,1],[1,2]])
b = np.array([[9],[8]])
x = la.solve(a,b)
```

2.8 NumPy Matrix

NumPy provides another object called a matrix that has been widely used for linear algebra to represent matrices. The matrix object is a subclass of the NumPy array and provides a little cleaner syntax for manipulating matrices than the array syntax. However, as of Python 3.5, the developers of NumPy have indicated that they plan to deprecate the usage of the matrix type, so we haven't taught it here. But you may still see it in existing code bases. It is essentially a wrapper around the NumPy array object to make some linear algebra syntax cleaner. All of the functionality of the matrix object is available with NumPy arrays with some small syntax changes.

Chapter 3

Functions

3.1 User-defined functions

Python's ecosystem is extremely rich, and you can usually find a pre-made library to do common manipulations. But you will also need to write your own functions. User-defined functions are created like this

```python
def myFunction(a,b):
    c = a + b
    d = 3.0 * c
    f = 5.0 * d**4
    return f
```

This function performs several simple calculations and then uses the `return` statement to pass the final result back out of the function. User-defined functions must begin with the keyword `def` followed by the function name (you can choose it). Python does not use an end statement to signal the end of a function. Rather, it looks for indentation to determine where the function ends, just like it did with loops and logic.

You can integrate this function into a larger program like this

```python
# The function code is not executed until the function is called below.
def myFunction(a,b):
    c = a + b
    d = 3.0 * c
    f = 5.0 * d**4
    return f

#The rest of this code is not part of the function.
r = 10
t = 15
x = myFunction(r,t)
```

In this case, when the function is called, the input `a` gets assigned the value of 10 and input `b` gets assigned the value of 15. The result of this calculation (`f`) is passed out of the function and stored in the variable `x`.

A word on local and global variables is in order here. In the example above, the variables `a`, `b`, `c`, `d`, and `f` are *local variables*. This means that these variables are used internally by the function when it is called and then immediately forgotten. To see what I mean, add the following `print` statement at then end and observe the results

```python
x = myFunction(r,t)
print(c)
```

The print statement just causes an error since Python does not remember that inside the function `c=a+b`.

In contrast, the variables `r`,`t`, and `result` are called *global variables*, which means that Python remembers these assignments from anywhere, including inside of functions. So, technically, you could do the following:

```
g = 9.8         # <--- g defined to be a global variable
def myFunction(a,b):
    c = a + g   # <--- Notice the reference to g here
    d = 3.0 * c
    f = 5.0 * d**4
    return f
#The rest of this code is not part of this function.
r = 10
t = 15
x = myFunction(r,t)
```

and there would be no error. Notice that `g` has been defined as a global variable, and the function `myFunction` knows it's value and can use it in a calculation. Using global variables inside functions is usually considered bad practice and can confuse those reading the code. In general, every variable used in a function ought to be either passed in or defined inside of the function.

3.2 Importing User Defined Functions

If you write some functions that you find yourself using over and over again, or if you've written so many functions for a program that it makes your program hard to read, you can save your functions in a separate file and import them just like a Python library.

As an example, create a blank python file and type a couple of functions to calculate some of the parameters of projectile motion that you learned about in Newtonian physics:

```
import numpy as np

def maxRange(v0,theta):
    R = v0**2 * np.sin(2*theta) / 9.8
    return R

def maxHeight(v0,theta):
    vy = v0*np.sin(theta)
    h = vy**2 / (2*9.8)
    return h
```

Then you save the code above in a file called `projectile.py`. Now create another file in that same directory, and practice importing your projectlie functions using the following four methods (just use one method at a time):

```
import numpy as np

#Method #1
import projectile
projectile.maxRange(10,np.pi/4)

#Method #2
from projectile import maxRange
maxRange(10,np.pi/4)

#Method #3
import projectile as pf
pf.maxRange(10,np.pi/4)

#Method #4
from projectile import *
maxRange(10,np.pi/4)
```

Using these methods, you can organize your functions into neatly divided units that you can reuse in multiple programs. By default Spyder doesn't give you great hints about user-defined functions. You can change this by going to Tools→ Preferences→Help, and clicking the boxes under "Automatic connections."

It is possible to save your user-created libraries in a different folder, and then map to them during your import, or even make your functions available to any program using Python on your computer. When you are ready to work on larger projects where these techniques are important, you can learn about modules and default search paths online. However, to keep things simple in this course, just keep your files together in the same folder.

3.3 Writing Readable Code

Comments

You should have all had several programming courses before this one, so I'll spare you the lecture about commenting your code. For your own sake, and for the sake of those who may someday need to read your code, always comment your code a little more than you think is necessary at the time you write it. Any text written after the # symbol is ignored by Python as a comment:

```
# A full line of commenting
E=m*c**2 # comments after a short line of code
```

Docstrings

Python also has an environment like a comment that is called a docstring, delimited by a triple quote (either single or double):

```
"""
This is a docstring, which can be automatically parsed to generate
```

```
documentation.  It is not the same as a comment.
"""
```

Docstrings are placed immediately after the beginning of a module or a function definition, and can be used to automatically generate documentation for your code. You'll see some people use them as a multi-line comment, but this is considered bad practice. If you just want to comment your code, use the # symbol in front of multiple lines. Docstrings are important when writing large code bases to keep things well documented, but we won't use them much in this course.

Line Continuation

To keep your lines of code from becoming too long, Python provides two ways of continuing a logical line of code on the next physical line. The preferred method is an implicit continuation which happens when you put a line break in your code before closing all parenthesis, like this:

```
x = (a + b
     + c)
```

Normally you wouldn't wrap lines of code this short, but this is just an example. You should indent the continued line of code so that it starts right after the unclosed opening parenthesis. You can almost always wrap lines of code this way simply by added parentheses. You can also use the backslash to continue a line of code, like this:

```
x = a + b \
     + c
```

Code cells

Spyder will allow you to split your code into individual cells that can be run separately, without executing the whole program like this

```
#%% This is the start of a cell
x=5
y=6
print(x*y)
#%% This is the start of another cell
a=3
b=4
print(a/b)
```

If you place your cursor in one of the cells and press Ctrl+Enter (or push the button to the right of the green arrow), Spyder will execute just the code in the current cell. We generally discourage writing code using cells. The current state of you program then depends on which cells you've previously executed which can be painful to debug. However, there are times that they can be useful.

Index