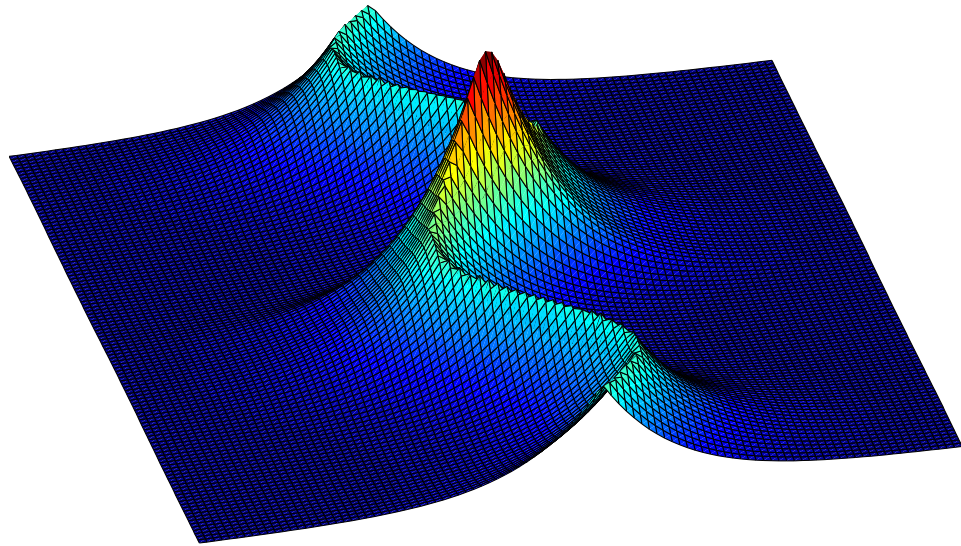


INTRODUCTION TO PYTHON



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

© 2019 Michael J. Ware, Brigham Young University

Last Revised: January 9, 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 acknowledgement 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.

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

Contents

Table of Contents	ix
1 The Basics, Numpy, Scipy, and Plotting	1
1.1 Get Anaconda Installed	1
1.2 Your First Program	1
1.3 The Python Console	2
1.4 Variables and Data Types	3
1.5 Integers	3
1.6 Float variables	4
1.7 Boolean variables	4
1.8 String Variables	5
1.9 Formatting Printed Values	5
1.10 Functions and Libraries (Numpy)	5
1.11 Numpy Arrays	7
1.12 Making x - y Plots	10
Index	11

Chapter 1

The Basics, 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. Anaconda is geared toward scientific computing, is available as a free download on all major platforms, and comes with a nice 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 you can issue Python commands directly to be evaluated and stored.

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
```

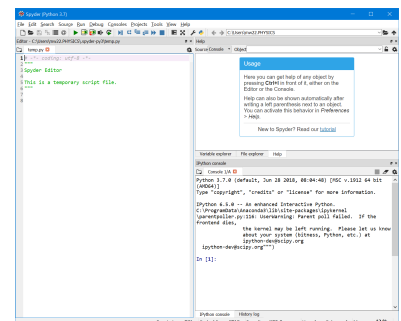


Figure 1.1 The Spyder IDE window.

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: anything 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
```

Table 1.1 shows some common housekeeping functions that you can use on float variables.

1.7 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 start using loops and logical statements.

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

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

1.8 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 a 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

1.9 Formatting Printed Values

The `print` command will take pretty much any variable dump it to screen as Python pleases. 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: {:.2f}".format(a, b))

#This style also works
joe_string="My GPA is {:.2f} and I am {:d} years old."
print(joe_string.format(b,a))
```

Notice the structure of this print statement: A string followed by the `.` operator and the `format()` function. The variables to be printed are provided as arguments to the `format` statement and are inserted into the string sequentially wherever curly braces (`{}`) are found. The characters inside of the curly braces are a format code 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 more examples.

1.10 Functions and Libraries (Numpy)

As you've probably noticed, 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)
```

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

Table 1.2 A sampling of “house-keeping” functions for strings.

<code>{}</code>	Use the default format for the data type
<code>{:4d}</code>	Display integer with 4 spaces
<code>{:.4f}</code>	Display float with 4 numbers after the decimal
<code>{:8.4f}</code>	Display float with at least 8 total spaces and 4 numbers after the decimal

Table 1.3 Formatting strings available when printing.

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 `c` and `d` 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 `sin()` and `cos()` 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"). 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 `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

```
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.

```
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
```

This syntax tells Python "I'm going to call the numpy library `np`." We recommend that you typically use this syntax always for this class unless there is some compelling reason to do otherwise. It is so common to do this, that in this manual we will omit writing the `import numpy as np` at the beginning of each example. **All of the example code through the remainder of this book will assume that you have the this line of code at the beginning of your program:**

```
import numpy as np
```


Virtually any mathematical function that you will need to perform a scientific calculation can be found in the library NumPy or another common library called SciPy (say “sigh pie”). Here is an example involving both libraries:

```
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)
```

Most likely, numpy has the mathematical function that you need and if it doesn't, then `scipy.special` probably has it. Tables 1.4 and 1.5 provide a small sampling of the functions available. See online help for a more extensive listing.

1.11 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

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

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. (1.1) 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:

<code>airy(z)</code>	Airy function
<code>jv(z)</code>	Bessel function of 1st kind
<code>yv(z)</code>	Bessel function of 2nd kind
<code>kv(n,z)</code>	Modified Bessel function of 2nd kind of integer order n
<code>iv(v,z)</code>	Modified Bessel function of 1st kind of real order v
<code>hankel1(v,z)</code>	Hankel function of 1st kind of real order v
<code>hankel2(v,z)</code>	Hankel function of 2nd kind of real order v

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

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

```
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 with Matlab's colon syntax.

Another very useful function for array-creation is `linspace`, which 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:

```
myArray, mydx = 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.

Many other useful function for creating arrays are available. Table 1.6 gives some of the more heavily-used options.

Math with Arrays

Once the array object is created, a whole host of mathematical operations become available. For example, you can square the array and Python knows that you want to square each element, or you can add two arrays together and Python knows that you want to add the individual elements of the arrays. You can add a constant value to every element of an array, or even multiply two arrays together and the elements of the first array are multiplied by the corresponding element in the second. Here are some examples:

```
xList = [2 , 3 , 5.2 , 2 , 6.7]
xArray = np.array(xList)      # Create first array
yArray = np.array([4,8,9.8,2.1,8.2,4.5]) # Create second array

c = xArray**2    # Square the elements of the first array
d = xArray + 3   # Add 3 to every element of the first array
e = xArray * 5   # Multiply every element of the first array by 5
f = xArray + yArray # Add the elements of array one to the elements of
                    # array two
g = xArray * yArray # Multiply the elements of array one by
                    # the elements of array two
```

In most respects, math with Numpy arrays behaves like the “dotted” operators with Matlab matrices, where the operations are performed on corresponding elements.

Functions of Arrays

Mathematical functions (like $\sin(x)$, $\sinh(x)$...etc.) 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 commonly available math library (which also provides a sine function), are not designed to work on arrays. Here is an example of this issue:

```
import numpy as np
import math

xList = [2 , 3 , 5.2 , 2 , 6.7]
xArray = array(xList)

c = np.sin(xArray)  # Works just fine, returning array of numbers.
d = math.sin(xArray) # Returns an error.
```

Accessing Elements of 1-D Numpy Arrays

You can access individual array elements 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, much the same way as you did in Matlab, this way:

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

But if you want to extract a set of non-contiguous elements, say 2, 3, 5, and 9, you need to use a list of indexes like this:

```
b = a[[2,3,5,9]]
```

Notice that you need nested square brackets in this syntax: the outer brackets indicate that you'll be accessing elements of `a`, and the inner brackets define the list of indexes that you want.

1.12 Making x - y Plots

Once you have arrays of values, you'll often want to plot them. Simple plots of x vs. y can be made using NumPy arrays with a library called `matplotlib`. This library creates plots using a syntax very similar to the plotting routines used in Matlab. To build make a plot, `matplotlib` needs arrays of x and y points that are going to be plotted. To build an array x of x -values starting at $x = 0$, ending at $x = 10$, and having a step size of `.01`, we use numpy's `arange` function:

```
import numpy as np
x=np.arange(0,10,0.01)
```

To make a corresponding array of y values according to the function $y(x) = \sin(5x)$ simply type this

```
y = np.sin(5*x)
```

Both of these arrays are the same length, as you can check with the `len` command

```
len(x)
len(y)
```

Once you have two arrays of the same size, you plot y vs. x like this

```
import matplotlib.pyplot as plt

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

This will create a plot where the points are connected. If you omit the `pyplot.show()` command, the plot will not appear on your screen. You can save the plot to a file by replacing the `pyplot.show()` command with

```
plt.savefig('filename.png')
```

If you want to see the actual data points being plotted, you can either add the string `'ro'` inside of the plot command (the `'r'` means make the data points red and the `'o'` means plot circle markers) or use the `scatter` function, like this:

```
plt.scatter(x,y)
```

And what if you want to plot part of the x and y arrays? The colon operator can help. Try the following code to plot the first and second half separately:

```
nhalf = len(x)/2
plt.plot(x[0:nhalf],y[0:nhalf])
plt.plot(x[nhalf:],y[nhalf:])
plt.show()
```

Remember that you must use Numpy's `sin` function if you want to be able to pass an array of values to it.

You specify the filetype of the output by the file extension. `'filename.png'` makes a png image, while `'filename.pdf'` makes a pdf.

Index

Array, first or second half, 10

Assigning values, 3

Case sensitive, 3

Data types, 3

Plotting, xy, 10

Strings, 5