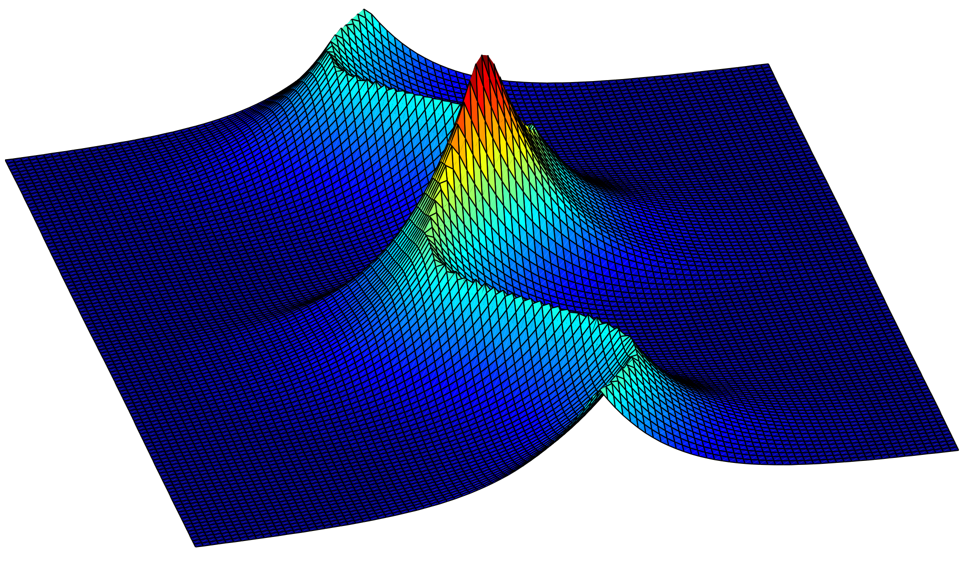
INTRODUCTION TO PYTHON



Lance J. Nelson

Department of Physics

INTRODUCTION TO PYTHON

Lance J. Nelson

Department of Physics

Brigham Young University–Idaho

Last Revised: June 5, 2017

Preface

This is a tutorial to help you get started in Matlab. Examples of Matlab code in this pamphlet are in typewriter font `like this`. As you read through the text, type and execute in Matlab all of the examples, either at the ≫ command line prompt or in a test program you make called `test.m`. Longer sections of code are set off and named. This code can be found as files on the Physics 330 web page at physics.byu.edu/Courses/Computational.

This booklet can also be used as a reference manual because it is short, it has lots of examples, and it has a table of contents and an index. It is almost true that the basics of Matlab are in chapters **??**-**??** while physics applications are in chapters **??**-**??**. Please tell us about mistakes and make suggestions to improve the text (michael_ware@byu.edu).

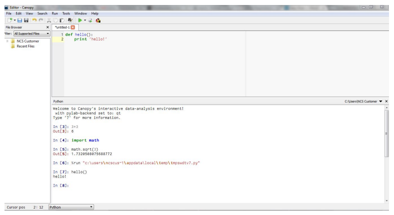
To find more details see the very helpful book *Mastering MATLAB 7* by Duane Hanselman and Bruce Littlefield.

Contents

Chapter 1

Running Python

Python is a computer programming language (don't freak out) with broad applicability in science and engineering. For those that are brand new to computer programming, Python is simply a way to communicate a set of instructions to your computer. You'll quickly learn that your computer is great at doing exactly what you tell it to do. If you find that your computer isn't doing what you think it should, it is not because the machine is malfunctioning, rather you just probably don't fully understand what you are telling it to do.



```
def hello():
    print 'hello!'
```

```
Welcome to Canopy's interactive data-analysis environment!
 with pylab-backend set to: qt
Type '?' for more information.

In [3]: 3+3
Out[3]: 6

In [4]: import math

In [5]: math.sqrt(3)
Out[5]: 1.7320508075688772

In [6]: %run "c:\users\ncscus~1\appdata\local\temp\tmpswdtv7.py"

In [7]: hello()
hello!

In [8]:
```

Figure 1.1 Canopy's environment for running Python code.

There are two ways to run python. One is at the command line and the other is through an IDE (integrated development environment). If you don't know what the command line is, I recommend you start out using an IDE. A good IDE for python is called Canopy, which can be downloaded here. If you know what the command line is then you probably don't need any help getting started. Feel free to use whatever editor you'd like. Once you have downloaded and installed Canopy, launch the editor. You should see a window that looks similar to the one displayed in the margin. As you read this book, type the commands that appear in `this font` in the editor window and run the code by pushing the green arrow.

When writing a computer program, there are several big ideas, or main pillars, that you should grasp to be successful. Two of those pillars are variables and functions. This chapter will focus on helping you get comfortable with these ideas. Later chapters will focus on other main ideas and some more specific tasks that can be accomplished using these foundational ideas.

1.1 It's a Calculator

Simple math can be performed pretty much just as you would expect. Here are a few examples

```
print(1+2)
print(5.0/6.0)
print(5**6)  # 5 raised to the power 6
print(678 * (3.5 + 2.8)**3.)
```

Note that typing the calculation alone, without the `print` statement, will not produce any output to the screen. Even though the calculation has been performed, you will not see the result unless you `print` it. Throughout this book, the `print` statement will usually be omitted in code examples to maintain brevity. You should always print your result so you can see what you have done.

1.2 Variables

The simplest, and most fundamental object when programming a computer is the variable, which is nothing more than a name which is given to a piece of information. It allows the information to be saved (stored) so that it can be recalled and used later. There are different types of variables and each has it's use and limitations. Python has many types of variables, but you will mostly use the following types: integer, float, and list. What follows is a brief description of each variable type with some examples to illustrate.

Integer variables

Some of you may be used to programming in C++[1], where the variables are declared before a value is assigned. In python, variables are created and assigned in one statement using the assignment operator (=). The simplest type of data is an integer, a decimal-less number. An integer variable is created and assigned a value like this:

[1] C++ is a compiled language, whereas Python is an interpreted language

```
a = 20
```

This statement creates the variable a and assigns it a value of 20 (an **integer**). The value of a can be modified in any way you choose. For instance, the statement

```
a = a + 1
```

adds one to whatever value was previously stored in a. [2] Note that *variable names in Python are case sensitive,* so watch your capitalization. If you want to see the value of a variable, you must use the print command, like this

[2] An assign statement in programming is not a mathematical equation. It makes perfect sense to write a=a+1 as assign statement: it tells the computer to get the value of a and add 1 to it and then store the evaluated quantity back in the variable a. However, the mathematical equation $a = a + 1$ is clearly false.

```
print a
```

[3] Other common mathematical operations can be performed on integer variables.[4] For example:

[3] If you are using version 3 or higher of python, printing a variable is done like this: print(a).

[4] For more complex mathematical calculation, wait until we discuss functions.

```
a = 20
b = 15
c = a + b  # add two numbers
d = a/b    # divide two numbers
r = a//b   # return only the quotient(an integer) of the 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 ^)
```

Notice that performing an operation on two integers usually yields another integer. The one exception is division if you are using Python version 2 or earlier. For example, what would be the value of d above if the result has to be an integer?

Float variables

Most meaningful calculations should be performed using floating point numbers, or floats in Python. Float variables are created and assigned in one of two ways. The first way is to simply include a decimal place 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)
```

P1.1 Check that the calculations that you performed above using integers yield different answers when floats are used instead

`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.

You may wonder what would happen when a float is multiplied by an integer, like this

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

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

```
1.23e15
```

Python floats are stored as a 53-digit, base-2 binary number(that's a mouthful). If you're interested in what that means, we can talk more. If you're not that interested, just know that when you define a float in python, the number that is stored in the computer is not **exactly** the number that you think it is. This can cause problems when you are comparing two numbers that you think should be equal but actually aren't equal in the computer. [5] Try the following to get a feel for this

```
a = 0.1
b = 3 * a
print(b)
print(" {:.45f} ".format(b))   #Formatted print: you haven't learned
                               #about this yet
```

The first print statement displays the value of `b` to one decimal place. The second print statement is called a formatted print and you'll learn about it a little later. In this case I am forcing Python to display the value of `b` out to 45 decimal places. Notice that the true value of `b` is not exactly equal to 0.3. You'll want to keep this fact in mind when you are comparing two numbers that you think are equal. When we study logical statements this should become more clear.

[5] There is a library called `Decimal` that will fix a lot of these problems.

Boolean variables

A boolean variable is one that stores 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

```
myVar = True
```

The purpose for using a boolean variable will become more clear as you study loops and logical statements, so stay tuned.

Lists

Lists are heavily-used data types in mathematical and scientific programming. You can think of a list as a container that holds multiple pieces of information. The list could hold integers, floats, strings, or really just about anything. A list is created and assigned in a manner similar to the way floats and integers are, like this:

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

Note the use of brackets ([]) when creating the list. If you accidentally use parenthesis (())[6] or curly brackets ({})[7] you'll end up creating something other than a list.
Python has a built-in function called `range` that can be used to construct a list of integers, like this

```
a = range(10,52,3)
```

This will construct a list of integers starting at 10, ending at 52, while stepping in increments of 3. This function can also be called with 1 or 2 arguments and default values will be assigned to the missing ones.

```
a = range(3,9)   # Creates a list starting at 3, ending a 9 stepping
                 # in increments of 1 (default value)
b = range(10)    # Creates a list starting at 0(default), ending at 10 stepping
                 # in increments of 1(default value)
```

You can make a list of anything. For example, here is a list of strings:

```
a = ['Physics' , 'is' , 'so' , 'great']
```

The individual elements of any list need not be the same type of data. For instance, the following list is perfectly valid

```
# Here is a list of strings and integers
a = ['Ben',90,'Chad',75,'Andrew',22]
```

You can even define a list of lists:

```
a = [[4,3,2],[1,2.5,90],[4.2,2.9,10.5],[239.4,1.4],[2.27,98,234,16.2]]
```

though you should not think of this as a matrix and try to perform matrix math on it. To do that sort of thing, we'll need to learn about arrays.

[6] Use parenthesis to create a tuple, which is just like a list but cannot be modified.

[7] Use curly brackets to create a dictionary, which is like a list but can be indexed on any data type, not just integers

Accessing and Slicing lists

Accessing an element of a list (this will be done frequently so pay attention!) can be accomplished using square brackets, like this

```
a[0]  # Access the 1st element of array a
a[4]  # Access the 5th element of array a
a[-1] # Access the last element of array a
a[-2]  # Access the second to last element of array a
```

Take special note to the last two statements where a negative index is used. Using a negative index means that you are counting from the back of the list forward. Please note that python lists are zero-indexed: the first element of any list is 0, the second element is 1, etc. Lists can be easily modified by specifying which element you want to change and what you want it changed to:

```
a = ['Physics' , 'is' , 'so' , 'great']
a[3] = 'tough'  # Change the 4th element of a to "tough"
```

At times you may want to access entire sections of list, though not the entire list. You can do this with the : operator, like this

```
aList = [4,5,10,1560,23,19]
aList[1:]
aList[1:3]
aList[:3]
aList[1:4:2]
```

There can be three numbers inside the brackets, each seperated by the : symbol, like this `[x:y:z]`. The section of the list that is extracted starts at element x, ends at element y (but does not include element y), while stepping in increments of z. Note that the last number is optional, and omitting it will result in a default value of 1 for the step size.

When working with a list of list (also called a 2-dimensional list) accessing an individual element requires two indices, like this

```
a = [[4,3,2],[1,2.5,90],[4.2,2.9,10.5],[239.4,1.4],[2.27,98,234,16.2]]
c = a[3][1]
```

Here `[3][1]` means we are accessing the 2nd element of the 4th list. (remember: Python lists are zero-indexed) You may also think of this as the 4th row and 2nd column of the matrix. Warning: Don't conceptualize 2-d lists as matrices and try to slice out a sub-matrix. For example, try this

```
a = [[4,3,2],[1,2.5,90],[4.2,2.9,10.5],[239.4,1.4],[2.27,98,234,16.2]]
c = a[0:3][0:2]
```

Slicing a 2-d list like this is rather redundant. First you slice out the first three elements of the main list, resulting in a list of 3 lists. The second slice is performed on the result of the first slice giving a list of 2 lists. The same result could have been extracted with the following slice:

`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
`range(x,y,z)`	Create a list of integers, starting at x, ending at y, and stepping in increments of z.
`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 1.2 A sampling of "housekeeping" functions for lists.

```
a = [[4,3,2],[1,2.5,90],[4.2,2.9,10.5],[239.4,1.4],[2.27,98,234,16.2]]
c = a[0:2]
```

The take home message here is to just not ever think of lists as mathematical matrices. Just think of them as data containers.

Built-in Functions for Lists

Python has many built-in functions that work on lists. You've already seem some of them. Here are a few examples

```
myList = range(5,25,2)  # Create list starting at 5, ending at 25,
                        # in increments of 2
len(myList)             # Find how many elements are in myList
myList.append(520)      # Add the number 520 to the end of myList
```

Here, `range` will create a list of integers starting at 5, ending at 25 in increments of 2. The `len` function will find the length of a list, and `append` will add an element to the end of a list. Table 1.2 lists some of the more common built-in functions/operations for lists. Please note that this is not a comprehensive list. A complete list of available function can be found in the appendix.

Vector and Matrix Math: A warning

You may feel tempted to associate a list with a mathematical vector and try to perform vector math on them. We've mentioned this warning before but just want to further emphasis what list cannot be used to do. For instance, it may seem natural to try

```
a = [5.1 , 3.2 , 6.8 , 9.2]
b = 5 * a
```

or

```
a = [5.1 , 3.2 , 6.8 , 9.2]
b = [2.7 , 1.9 , 3.2 , 9.9]
c = a + b
```

P1.2 Try the above operations and explain the results.

Doing vector or matrix math (any math that involves an operation on lists of numbers, not just single numbers) cannot be done with a list. If you were allowed to do that, what would happen when your lists were filled with non-numerical data. This doesn't mean that the numbers stored in lists can't be extracted and used in mathematical calculations, like this

```
a = [4.5,8,2.1,10.8,12]
c = a[0]**a[1]  # Take the first element of a and raise it to a power
                # equal to the second element of a
```

It just means that mathematical calculations involving entire arrays of numbers (like vector and matrix math) cannot be done using the list data type. However, don't think for one second that Python is unable to handle this kind of math. Just keep reading and you'll learn how this is done.

String Variables

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

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

[8] Some Python functions require options to be passed to them using strings. Make sure you enclose them in quotes, as shown above. There are many useful function that can be used with strings. For example, strings can be concatenated (joined) together using the + operator:

[8] You may also enclose the characters in double quotes.

```
a = 'Hello'
b = ', my name is B. Nelson'
c = a + b
```

The number of characters in a string can be calculated using the `len` function.

```
a = 'Hello'
b = ', my name is B. Nelson'
c = a + b
d = len(c)
```

Individual characters inside of a string may be accessed and sliced in the same way that elements of a list are accessed and sliced.

```
a = 'Hello, my name is B. Nelson'
a[18]
a[2:15]
```

However, you'll have little luck modifying a single character in a string unless you first convert the string to a list, like this

```
a = 'Hello, my name is B. Nelson'
b = list(a)  # convert string \texttt{a} to a list
b[18] = 'S'  # Modify the 18th element in the list.
c = "".join(b)  # Join all the elements back together into one string.
```

The `join` function used here was probably unfamiliar to you. It is a built-in function for use with strings. It joins together all of the elements in list `b` into one string, putting whatever is in `""` in between each element. There are many built-in functions that can be used with strings. Some of the more commonly used ones are shown in Table 1.3

`a.join(b)`	Join all elements in list `b` while placing string `a` between each pair of elements.
`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 1.3 A sampling of "housekeeping" functions for strings.

1.3 Displaying Results

It's great to calculate something useful, but not that helpful unless you can see it. Beginners often ask the question, "I calculted XX but Python didn't do anything." Actually, Python did exactly what you told it: It calculted XX and called it a day. If you want to see XX, you have to print it. The fastest way to see something, as you've already seen is using the print statement:

```python
a = 5.3
b = [5,3,2.2]
c = 'Physics is fun'
print(a)
print(b)
print(c)
```

As you can see, you can `print` anything and Python will dump it to screen as it pleases. There are times when you may want to be more careful about the formatting of your print statments. For example:

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

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 odd characters inside of the curly braces indicate how you would like the variable formatted when it is printed. The `:d` is used to indicate an integer variable and `:f` is used for floats. Further specifications regarding spacing can also be made. 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. A summary of what is available is given in table 1.4

`{:4d}`	Display integer with 4 spaces
`{:.4f}`	Display float with 4 numbers after the decimal
`{:8.4f}`	Display float with at leasat 8 total spaces and 4 numbers after the decimal

Table 1.4 Formatting strings available when printing.

1.4 Functions

A function is simply a set of instructions that will be performed when called upon. The function can be as long and complex or short and succinct as you wish. You can think of a function as a black box. You create the contents of the box, specify what information needs to enter the box for it to be able to accomplish it's task, and what information will exit the box. Why a black box? Well, one benefit of functions is that the user of it doesn't need to know what's inside. She only needs to know what information the box needs and what information the box will give back to her.

User-defined functions

At times it will be beneficial for you to write a function that you have designed. You (the programmer) can create your own function 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.(what exits the black box) Every user-defined function must begin with the keyword `def` followed by the function name (you can choose it). The variables `a` and `b` are the arguments to the function. (the things that must enter the black box) Python does not use an `end` statement or anything like it to signal the end of a function. Instead, it looks for indentation to determine where the function ends.

The function can be called like this

```
r = 10
t = 15
result = myFunction(r,t)
```

In this case, when the function is called, `a` gets assigned the value of 10 and `b` gets assigned the value of 15. The result of this calculation is stored in the variable `result`, which is outside the function.

Lambda (unnamed) functions

The user-defined function above had 4 lines in it. Other functions could be even longer. Sometimes, however, the function you need to define is quite simple (maybe even one line) and it'd be nice if you could define it in one line. Luckily, you can. It's called a lambda function and here is an example:

```
f = lambda x: x**3  #Define the function
print f(5)   # Evaluate the function
g = lambda x,y: x + y # Function of two variables
pritn g(5,6)
```

Here we defined the function $f(x) = x^3$, a function of one variable, and quickly evaluated it. Your lambda functions can have as many arguments as you'd like: just seperate the arguments with commas. At this point, you may be asking yourself why this is such a big deal: Why would you ever really need to do this. Well, there are some python functions that take functions as their arguments.[9] (as opposed to taking simple numbers or strings) An example of this is the `filter` function which serves to extract elements of a list according to some criteria. The exact details of the criteria are specified using a lambda function. Here's an example:

[9] Stop and process that for a minute.

```
a = [1,3,4,6,9,2,3,8]
b = list( filter( lambda x: (x % 2 == 0) , a ) )
print b
```

Notice how the `filter` function is used. The first argument is the criteria function: the function the dictates which of the list elements you want extracted. It's not an integer or a float or a sting. It's actually a function: something that takes an argument and returns a result. In this case, the filter function was being used to extract the elements of `a` that were multiples of 2 and the lambda function serves to check each number in the list and return either `True` or `False`.

Imported Functions and Libraries

Thus far you have performed very simple mathematical calculations. ($5/6, 8^4$, etc..) However, you were probably left wondering about more complex mathematical calculations, like $\sin(\frac{\pi}{2})$ or $e^{2.5}$.

Many times, the function that you need is available already. Somebody else has already created the function and made it available to anyone who wants it. Groups of functions that perform similar tasks are usually bundled together into libraries. These libaries can then be imported and the functions that they contain can be used. Just as with user-defined functions, it is critical that you know what information(variables) the function expects you to give it and what useful information the function will hand back to you. This information can be found in the library's documentation. Google will be your friend in this regard.

If you are using Enthought's Canopy package, all of the libraries that you need are already installed. If they aren't installed you'll have to download the libary and install it. Once the library is installed, it can be imported like this:

```
import math
```

This imports a library called `math`. Functions inside the math library can be used like this

```
math.sqrt(5.2)  # Take the square root of 5.2
math.pi         # Get the value of pi
math.sin(34)    # Find the sine of 34 radians
```

Using the functions inside a libary requires that you know what functions are available. This information is usually available in the library's documentation. Google will be a great resource here.

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

```
import math as mt
```

Here, the short name `mt` was chosen for this library. The desired functions can then be called like this

```
mt.sqrt(5.2)  # Take the square root of 5.2
mt.pi         # Get the value of pi
mt.sin(34)    # Find the sine of 34 radians
```

Sometimes you may not want to import the entire library, just a few functions. This can be done like this

```
from math import sqrt,sin
```

and the `sqrt` and `sin` functions can then be used without the library name before it, like this

```
sqrt(5.5)
```

If you want to import every function inside of a libary, do this

```
from math import *
```

Now, every function contained in `math` is available without needing the `math.` prefix in front of it. [10]

[10] For larger libraries, importing all of the functions can take a few seconds. A better choice is to import only the functions that you need

Native functions

Python has many native functions: functions that are included with the programming language. You don't have to import native functions because they came with the langauge. You have already been using some of them, like these

```
len(mylist)  # Returns the length of a list.
float(5)     # Converts an integer to a float.
str(67.3)    # Converts a float to a string.
```

The functions: `len`, `float`, and `str` are all built-in functions, and they each take a single argument. Other built-in function are found in the margin tables in the previous section.

Throughout this book, we will use a variety of libaries to accomplish important tasks. Instead of giving a complete description of each library used and every function that it contains, we will simply discuss the functions needed for each specific task. The interested reader is referred to the documentation of the various libaries for further details.

Chapter 2

Calculating

In a scientific setting, much of what you will ask Python to do will involve math. You've already seen how to do very simple math. Here we will give you all the tools you will need to do any mathematical calculation you could want.

2.1 Mathematical functions

Common (and not-so common) mathematical functions like `exp` and `sqrt` are available via the libraries `numpy`, `scipy`, and `math`. There are some good reasons to not use the `math` library, which we will discuss shortly. Some commonly-used mathemtical functions from these libraries can be found in the tables.

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

2.2 Numpy Arrays

Often you'll find that you want to perform math on an entire set of data. For example, let's say you had the following data set

```
x = [2.42762254  2.53691271  3.15932278  1.7128872   2.54105921  2.54094893
  2.55284336  2.36430906  2.37972415  2.70342833  2.2846214   2.37636944
  2.74236195  3.06429336  2.29889954  1.99944808  2.46066766  1.86346638
  2.69619554  1.81298331  2.96144256  3.020208    2.71914935  2.59783385
  2.41512769  2.84674515  2.92394769  3.15879826  2.25886137  3.04074924
  3.14635756  2.60488105  2.79643916  2.67695452  2.77874282  1.94903284
  2.60399377  1.88255081  2.38624122  3.43726289  2.46514806  2.74985076
  2.33684695  2.58710514  2.10996793  3.19191947  3.93418676  2.90987071
  2.52449511  1.71514896  2.42465365  2.24485334  2.88390193  2.97911184
  2.86770773  2.97543667  2.00454583  2.56522443  2.99691011  2.79259592
  2.01617544  1.66098216  2.59230004  2.31295971  3.49570792  2.37890997
  2.14965171  2.40578128  2.44831872  2.0519382   2.41011389  3.07252157
  2.50662296  2.49878442  1.97225157  2.00764702  2.67472532  3.02465629
  2.45257132  2.9325564   2.69301075  2.81356219  2.49886432  1.97998459
  2.86166356  3.24091275  2.83846089  2.58103089  2.23525104  2.85815534
  3.33391592  2.6850452   2.3267767   3.27800198  2.17433118  2.17612604
  2.80002452  2.48975877  3.01856681  2.34280246]
```

and you wanted to calculate

$$\sum_{i=1}^{N} (x_i - D)^3 \tag{2.1}$$

where x_i are the data and $D = 5$. You could write a loop and, one-by-one, calculate each contribution in the sum. But there has to be an easier way. The easier way involves a library called `numpy` (pronounced num-pie, short for numerical python). The main object used in this library is called an `array`. A Python list can be converted into a numpy array using `numpy`'s `array` function, like this:

```
from numpy import array
xList = [2 , 3 , 5.2 , 2 , 6.7]
xArray = array(xList)
```

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's a sampling of examples.

```
from numpy import array
xList = [2 , 3 , 5.2 , 2 , 6.7]
xArray = array(xList)    # Create first array
yArray = 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  # Multipy 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 to the elements of
                     # array two
```

In short, you can do all of the math that you were hoping you could do when you first learned about Python lists.

Evaluating Functions on Arrays

What if you wanted to evaluate the `sin` function over an entire data set. You surely don't want to loop over every single value in the data set and evaluate the sine function on each number. It turns out that the `math` library and the `numpy` library both contain a function called `sin`. The one from the `numpy` library is designed to work on arrays but the one from the `math` library is not. Here is an example

```
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.
```

Notice that `math`'s version of `sin` does not know what to do when you give it an array of numbers. It only works for single numbers. `Numpy`'s `sin` function, on the other hand, does know what to do with an array of numbers: it calculates the sin of all the numbers.

Summing the elements

Suppose you have a list(or array) of numbers and you'd like to add up all of the elements. Python has a built-in `sum` function and there is also a `sum` function inside of `numpy` that will do this. They both do the same thing for one-dimensional lists.

```
a = [1.5,2.2,9.8,4.6]
b = sum(a)   #Use built-in sum function
from numpy import sum
c = sum(a)   # Use numpy's sum function
```

If you are summing up the elements of a two-dimensional list, the built-in version of the function will not work and you will have to use `numpy`'s version.

```
a = [[1.5,2.2],[9.8,4.6]]  # Define a 2-d list, a list of lists
b = sum(a)   #Use built-in sum function, notice the error
from numpy import sum
c = sum(a)   # Use numpy's sum function, no error.
d = sum(a,axis = 1)
```

Notice the extra argument to the `sum` function in the last line. The `axis=1` indicates that you want to sum up the elements in each individual list and return a list of sums. If you had a higher-dimensional list, you could use `axis=2` or `axis = 3` as well.

Accessing and Slicing Arrays

Often times, though not always, an array is used to represent a matrix. In that case it may be necessary to extract a sub array. For example, let's say that you define the following array:

```
from numpy import array
a = [[1,2,3],[4,5,6],[7,8,9]]
b = array(a)
```

which could be interpreted as this matrix:

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

If you wanted to slice out the following 2 x 2 sub-matrix:

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

you could do it like this:

```
b[1:3,1:3]  # Slice out a sub-array (Exactly what you wanted!)
```

This can't be done with lists, but using an array it's very simple. Also note that you must use the `[x1:y1,x2:y2]` notation rather than the `[x1:y1][x2:y2]` notation. Use of the latter will not fail, but it will not produce the sub-matrix desired.

2.3 Matrices

Matrix math is different from normal math. That will become more clear after you take a linear algebra class. If you want to do matrix math or linear algebra, `numpy`'s `matrix` object is what you want to use. A `matrix` object can be created in a few ways. Here are a few examples

```
from numpy import matrix

a = matrix('1 2; 3 4')  # Create a 2 x 2 matrix from string
b = matrix([[1,2],[3,4]])  # Create 2 x 2 matrix from list
c = matrix('1;2;3;4')  #Create column vector: a 4 x 1 matrix
```

The first definition is a nice way to create a matrix from a string. The `;` indicates the end of the rows. You can also convert a list or array into a matrix. Note that if you print a matrix object to screen, it will probably look the same as an array or list (or similar). The differences are the things you can do with a matrix object as compared to an array object, or list object.

Once the matrix is defined, lots of cool and useful math becomes available to you. Here are a few examples:

```
from numpy import matrix

a = matrix('1 2; 3 4')  # Create 2 x 2 matrix from string
b = matrix('5 6; 8 9')  # Create 2 x 2 matrix from string
col = matrix('3;4')  # Create 2 x 1 column vector

c = a.T   # Transpose the matrix
d = a.I   # Find inverse of matrix
e = a.H   # Find conjugate transpose of matrix
f = a * b # Matrix multiplication
g = b * col # Multiply matrix b to column vector
h = a**2   # Square the matrix. Not the same as squaring an array.
```

2.4 Statistics

`Numpy` has a statistical package that includes many useful functions. Below is an example that calculates the mean and standard deviation of a data set

```
import numpy as np

data = [1.3,7.8,4.5,9.83,2.23,3.67]  # Define the data set
dataMean = np.mean(data)     # Calculate the arithmetic mean
dataSD = np.std(data)        # Calculate the standard deviation
```

Sampling from a Distribution

Chapter 3

Loops and Logic

Loops are an essential part of programming a computer. A loop is a set of instructions to be performed repeatedly until some criteria is met or until the data being looped over is exhausted.

3.1 For Loops

A `for` loop is a good choice when you know exactly what things you want to loop over before hand. Here is an example of a `for` loop used to add up the elements of a 4-element list:

```
thesum = 0
for i in [3,2,1,9.9]:
    thesum = thesum + i
```

Here you are *iterating* over the list `[3,2,1,9.9]`. This means that the loop variable (`i` in this case) gets assigned the values of the list elements, one by one, until it reaches the end of the list. You can iterate over lists, arrays, and matrices.[11] The list that you iterate over can contain anything.[12] For instance, here is a loop over a list of strings:

[11] Technically, an interable is an object that has a member function called `next`, but unless you understand classes, this is probably confusing to you.

[12] Once again, notice that Python uses indentation to determine what the contents of the loop are.

```
for i in ['Physics', 'is','so','fun']:
    print i
```

You can use a function to generate the list that you want to iterate over

```
for i in range(5,50,3):  #Range function will generate a list
    print i
```

The `range` function doesn't allow your step size to be smaller than 1, which means that you could never use it to generate a list like this `0.1..0.2..0.3..` etc. To do this you'll want to use the `arange` function inside of `numpy`, like this:

```
from numpy import arange
for i in arange(5,50,.1):
    print i
```

These examples are so simple that you might wonder when a loop might actually be useful to you. Let's see if we can build a loop to calculate the following sum:

$$\Sigma_{n=1}^{1000} \frac{1}{n^2} \tag{3.1}$$

```
theSum = 0
for n in range(1,1000):
    theSum = theSum + 1/n**2
print theSum
```

Here, n is being assigned the values 1,2,3,4....1000, one by one, until it gets all the way to 1000. Each time through the loop, n is different and the expression 1/n**2 evaluates to a new value. The variable theSum is updated each time through to be the running total of all calculations performed thus far. Here's another example of a loop used to calculate the value of 20!:

```
theProduct = 1
for n in range(1,21):
    theProduct = theProduct * n
print theSum
```

Remember that the range function creates a list starting at 1, going up to 21 but not including it. The math library has a function called factorial that does the same thing. Let's use it to check our answer:

```
from math import factorial
factorial(20)
```

3.2 Logical Statements

Often we only want to do something when some condition is satisfied. This requires the use of logic. The simplest logic is the if statement, which works like this:

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

if a >= 0 | 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
```

==	Equal
!=	Not equal
>=	Greater than or equal
>	Greater than
<=	Less than or equal
&&	and
|	or
!	not

Table 3.1 Python's logic elements.

Study the examples above until it makes sense. Any logical statement can be constructed out of the elements in table 3.1

3.3 While Loops

Logic can be combined with loops using something called a while loop. A while loop is a good choice when you don't know beforehand exactly how many iterations of the loop will be executed but rather want the loop to contiune to execute until some condition is met. As an example, let's compute the sum

$$\sum \frac{1}{n^2} \tag{3.2}$$

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 = 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 = s + term     # Add 1/n^2 to the running total
```

This loop will continue to execute until `term<1e-10`. Note that unlike the for loop, here you have to do your own counting, being careful about what value n starts at and when it is incremented (`n=n+1`). Also notice that `term` must be assigned prior to the start of the loop. If it wasn't the loop's first logical test would fail and the loop wouln't execute at all.

Sometimes `while` loops are awkward to use because you can get stuck in an infinite loop if your check condition is never false. 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

```
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 = 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 = s + term     # Add 1/n^2 to the running total
    if n > 1000:
        print 'This is taking too long. I''m outta here...'
        break
```

Chapter 4

Plotting

Creating plots is an important task in science and engineering. The old addage "A picture is worth a thousand words!" is wrong.... it's worth way more than that if you do it right.

4.1 Plotting Data

Python does not plot functions. This is true of any computer software used to make plots. Rather, the computer samples the function at a series of points in the domain and plots those points. In python, we can create a plot like this using a library called `matplotlib`. A simple scatter plot can be generated like this

```
from matplotlib import pyplot  #Import necessary functions

xData = [1.2,2.3,8.3,10.5]  # x-coordinates of the data points
xData = [2.2,8.5,1.2,3.5]   # y-coordinates of the data points
pyplot.scatter(xData,yData)  # make the plot
pyplot.show()                # Display the plot to your screen.
```

Here we have plotted some data and displayed them to the screen.

Customizing Scatter Plots

You can easily modify the style,color, and size of the plot markers. The change the style of the plot marker, add the optional `marker` argrument to the scatter function, like this

```
pyplot.scatter(xData,yData,marker='+') # Use plus sign as the marker
```

A list of plot marker styles is given in table 4.1

'+'	Plus
'.'	Point
','	Pixel
'v'	Triangle down
'8'	Octagon
's'	Square
'p'	Pentagon
'x'	X

Table 4.1 Sampling of available plot markers.

4.2 Plotting Functions

Sometimes you actually do want to plot a function and don't really want to see the data points individually. You can do this using the `plot` function (also inside of `matplotlib`).

```
from math import sin
from matplotlib import pyplot
xData = range(0,10):
yData = [sin(x) for x in xData]
pyplot.plot(xData,yData)
pyplot.show()
```

The plot function connects the data points being plotted. In this case, we only had 10 data points, so the plot looked pretty jagged. To get a better plot of the function, we need to generate more data points in our range (0,10). Unfortunately, Python's built-in range function can't do that for us. However, the numpy library has a function called arange that does just that.

```
from math import sin
from numpy import arange
from matplotlib import pyplot
xData = arange(0,10,.01):
yData = [sin(x) for x in xData]  #<-  This is cool, but probably new
                                 #    to you.
pyplot.plot(xData,yData)
pyplot.show()
```

Notice that arange takes three arguments: starting position, ending position, and step size. You should take a look at the variable xData to see what you just created. Also note the line that was used to generate the function values (yData). This is a compact way to combine a loop with a list and is worthwhile to learn. Notice that I am evaluating sin(x) at each point in my list called xData and saving each number in a list (hence the brackets enclosing the whole things). When I'm all done, I have a list of function values evaluated at the exact points in xData.[13]

[13] Think about this until it makes sense. It will save you time.

4.3 Histograms

Histograms are very useful plots when thinking about mean values and spread in data sets. As with the other plots, the library matplotlib has the needed function to create this plot. Below is an example

```
from matplotlib import pyplot
measurements = [3.46,2.48,1.98,3.42,3.56,3.87,3.9,4.2,2.9]
pyplot.hist(measurements)
pyplot.show()
```

4.4 Displaying multiple plots

Frequently you will want to display multiple plots on the same figure. To do this, simply execute multiple plot commands and wait until the end to use the show command, like this:

```
from matplotlib import pyplot
measurements = [3.46, 2.48, 1.98, 3.42, 3.56, 3.87, 3.9, 4.2, 2.9]
data =         [2.44, 1.95, 2.64, 2.84, 1.83, 2.79, 3.5, 3.8, 1.9]
x =            [1.24, 3.15, 4.62, 1.99, 2.01, 4.78, 2.1, 5.7, 3.2]

pyplot.scatter(x,data)
pyplot.scatter(x,measurements)
pyplot.show()
```

Notice that the `pyplot.show()` is only performed once after all of the plot objects have been created.

If you want to display multiple plots on the same canvas, you have to use the `subplots` command, like this:

```
f, (ax1, ax2) = pyplot.subplots( 2, sharex=True)
```

This command creates a figure with 2 plots on it: one on top of the other. If you wanted the plots to be in a grid pattern, you could replace the 2 in the subplots command with `(2,2)` (for example). This would create a 2 x 2 grid of plots, a total of 4 plots. Notice that the subplots command is returning 2 things: the figure, and the 2 plots. The figure is the canvas that the plots are being placed on. You can create the plot objects like this:

```
from scipy.stats import norm  # Import norm to create some data to plot
f, (ax1, ax2) = pyplot.subplots( 2, sharex=True)
data= norm.rvs(2.5,0.5,100)  # Create some data to plot
ax1.hist(data,bins = 100)  # Create the first plot
data= norm.rvs(2.5,0.5,1000)  # Create some more data to plot
ax2.hist(data,bins = 100)  # Create the second plot

pyplot.show()  #Show the whole thing.
```

Chapter 5

Fitting Functions to Data

As a scientist(or engineer), you will frequently gather data. Usually when you gather data, the goal is to use that data to uncover some relationship between the two variables. In other words, you'd like to find a function that best mimicks that data that you've gathered. In this chapter you'll see how to do that.

5.1 Fitting to Linear Functions

When you think of fitting a function to a data set you most likely think of least-squares fitting. Mathematically, this can be stated as minimzing the following quantity:

$$\sum_{i=1}^{N} (y_i - f(x))^2 \tag{5.1}$$

Here, y_i represents the data point collected at x_i and $f(x)$ represents the fitting function evaluated at x_i. By varying the parameters in the function until the above sum is minimized, you are finding the function that best mimicks the data.

5.2 Fitting to an Arbitrary Function

The `scipy.optimizes` library contains a wealth of useful function, one of which is called `curve_fit`. Sounds promising, right? Let's use this function to fit to some made-up data. Let's extract some data from the function

$$y(x) = a\sin(bx)\exp(cx) \tag{5.2}$$

where $a = 3$, $b = 2$ and $c = -1$. We'll sample this function at the following points

```
[0.2,0.7,1,1.5,1.8,2,4,5,6,7,10,15]
```

and also add some random noise. Here is the python code to do all of that:

```
from numpy import array,sin,exp  # Import needed math functions
from scipy.stats import norm     # Import normal distribution to add noise.

# Define the function
def func(x,a,b,c):
    value = a * sin(b * x) * exp(c * x)
    return value

#Define sample points
x = array([0.2,0.7,1,1.5,1.8,2,4,5,6,7,10,15])

a = 3
```

```
b = 2
c = -1
#         Sample the function...........and add some random noise.
y = array([func(n,a,b,c)  for n in x]) + norm.rvs(0,.0069,len(x))
```

All of these things should be familiar to you if you have read through previous chapters. Notice the use of `norm.rvs` to sample a normal distrubution.

Now we're ready to send our data into the function `curve_fit` perform the fitting. First, we need to import the library

```
import scipy.optimize as opt

fit = opt.curve_fit(func,x,y)
```

The `curve_fit` function requires three arguments: The function that is being fitted, the independent variable values, and the dependent variables values. There are several optional arguments that can be passed to this function. For example, you can specify a guess at the solution and that guess will serve as a starting point.

```
guess = [2.5,1.2,-3]  # Guess for a, b, and c.
fit = opt.curve_fit(func,x,y,p0=guess)
```

You can specify bounds on the search parameters:

```
# Set bounds on a to (0,4)
# Set bounds on b to (0,3)
 # Set bounds on c to (-4,2)
paramBounds = [[0,0,-4],[4,3,2]]
fit = opt.curve_fit(func,x,y,bounds=paramBounds)
```

and you can specify the uncertainty in the data points

```
#Specify uncertainty on each data point
uncertainty = [0.01,0.02,0.1,0.2,0.05,0.2,0.5,0.01,0.02,0.01,0.1]
fit = opt.curve_fit(func,x,y,sigma=uncertainty)
```

5.3 Plotting the Fit

The function `curve_fit` will return several things. The first thing is the values of the fit parameters, which is the thing you are most interested in. Once you have those you can proceed to plot the function just as we showed you in chapter 4. Here I'll show you again (recall that the variable `fit` contains the fit results)

```
fitA = fit[0][0]  # Pull out value of a
fitB = fit[0][1]  # Pull out value of b
fitC = fit[0][2]  # Pull out value of c

xRange = arange(0,15,0.1)  # Define a grid of points on my domain

#Evaluate my fit function over the entire domain
#using my newly-found fit parameters
yVals = [func(n,fitA,fitB,fitC) for n in xRange]
pyplot.scatter(x,y)  #Plot the data
pyplot.plot(xRange,yVals)  #Plot the fit function
pyplot.show()  #Show results.
```