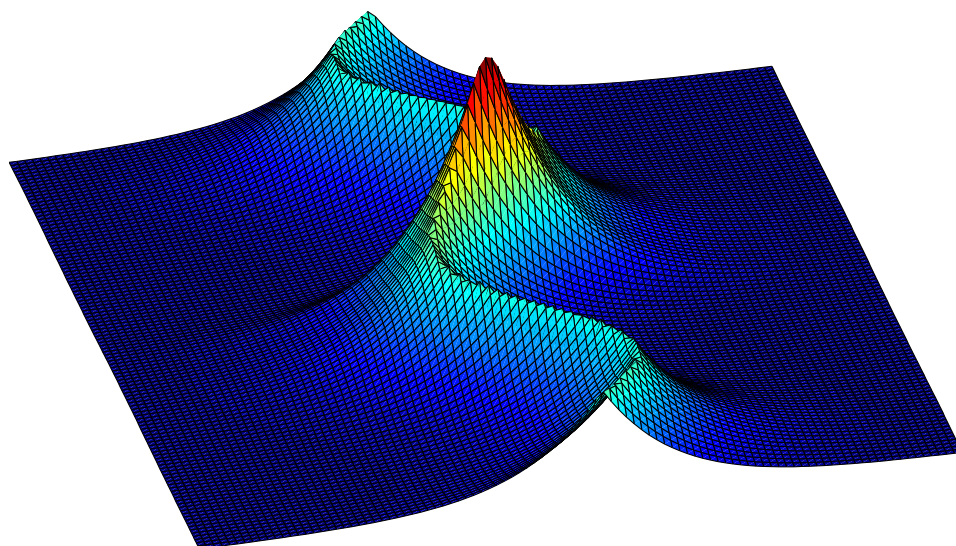# INTRODUCTION TO
SCIENTIFIC COMPUTING
IN PYTHON

Lance J. Nelson and Matthew R. Zachreson

Department of Physics

INTRODUCTION TO SCIENTIFIC COMPUTING IN PYTHON

Lance J. Nelson and Matthew R. Zachreson

Department of Physics

Brigham Young University–Idaho

Last Revised: January 11, 2019

Acknowledgements

Preface

This is a tutorial to help you get started in Python. This is a book for beginners. It assumes that you have no programming experience at all.

Examples of Python code in this book are in font `like this`. As you read through the text, type and execute in Python all of the examples. Longer sections of code are set off and named.

There are three major parts to this book: *Preliminaries*, *The Basics*, and *Advanced Topics*.

Preliminaries walks you through how to get Python installed and running, and then gives tips on what to do when your programs don't work.

We designed *The Basics* section to teach you Python, piece by piece. Each chapter builds on the last, and teaches you something new. You should work through each chapter, one by one, in order.

Advanced Topics covers things that are very useful in Python, but it serves more as a reference guide. Go through the chapters there either when you need to use what is in them, or if you just want to learn something new. Each chapter works as a stand alone tutorial, assuming that you've already covered *The Basics*.

In addition to teaching you Python, 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.

Please tell us about mistakes and make suggestions to improve the text (nelsonla@byui.edu).

Contents

Part I

Preliminaries

Chapter 1

Getting Started

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.

There are two ways to run Python. One is at the command line[1] 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. This text is written for Python 3, so choose the Python 3.x download that matches your operating system (Windows, Mac, or Linux).

1.1 Running Your First Program

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. (You may have to choose "Create a new file".) The Canopy Editor has three main windows: The file browser, the editor itself, and the Python console (Labeled "Python"). The file browser shows your local file directory. It is nice to have when you write larger Python programs. To keep big programs easy to read, we often break them into several files. The editor itself will be where we write our Python programs. We'll address the Python Console in a moment. Right now, type

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

into the editor. Save your program as `myFirstProgram.py`, then click the green arrow above the editor. If you look down at the console it should say:

```python
%run "where you saved the file/myFirstProgram.py"
Hello World
```

(If you don't see the words "Hello World", double check that what you've written in the editor and matches the above code exactly.) Congratulations, you just ran your first Python program. You told the computer to write "Hello World" on the console. Now try changing this line:

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

to this:

```python
print('I can do Python!')
```

and run your program again.[2]
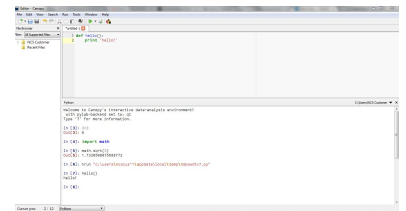
Figure 1.1 Canopy's environment for running Python code.

[1] If you are already comfortable with command line, then you probably don't need any help getting started. Feel free to use whatever editor you'd like. You'll need to make sure that you have Python 3 installed along with the following packages: numpy, scipy, matplotlib, iPython, sympy, and nose. These packages come pre-bundled with the Canopy distribution of Python, so if you installed Canopy, you already have them.

[2] You only need to do a "save as" on new files. Once you've given it a filename, Canopy will save and run your file every time you click the green arrow button.

1.2 The Python Console

The Python console serves two purposes: First, it shows our program's outputs. (The "Hello World" from earlier.) Second, it allows us to do interactive Python. Try typing

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

directly into the console, then press enter. You should see it print `Hello World`, just like it did for your program. The console will run any Python command that you enter into it. It can be very useful for short, quick calculations, or for looking at data when you are trying to figure out what your program is doing. However, it is impossible to save the commands that you enter into the console, so you should do most of your programming in the editor.

1.3 It's a Calculator

The very easiest, yet meaningful thing you can do with Python is to perform simple math. Simple math can be performed pretty much just as you would expect. Here are a few examples[3]

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

[3] The `#` marks something called a comment. Python knows to ignore anything on a line that comes after a `#`. They exist so that you can leave notes to anyone reading the program, without affecting what the program does.

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.

1.4 Getting the Most Out of This Book

In this book, all Python code will be offset like this:

```python
# A simple Print Statement
print('I Can do Python')
```

In order to make the textbook easier to read and understand, it usually omits print statements and direct instructions to type the examples into a Python script and run it. For example, here's a selection from one of the early chapters:

> You can define multiple variables by putting the assignments on seperate lines:
>
> ```python
> a = 2
> b = 4
> c = a * b
> ```
>
> Here we have stored the value 2 in `a`, the value 4 in `b` and then used those assignments to calculate a new value: 2×4 and assign that value to the variable `c`.

As the reader, you should type *all examples* into a Python script (program) and run it. As a general rule, you should also `print`[4] all of the variables, or at the very least the ones that highlight the new ideas that the example demonstrates. Remember: if you don't tell Python to print something, it won't.

So, as an example, when you reach the selection above while reading you should type the following (or something similar) into a Python script and run it:

```
a = 2
b = 4
c = a * b
print(a)
print(b)
print(c)
#Or print multiple variables at once
print(a,b,c)
```

Typing every example by hand (don't copy and paste) will help you better process what each program does. It also forces you to read it like Python will: one line at a time, top to bottom.

[4] As long as they aren't buried inside a function, you can also see what is stored in each variable by typing the variable name into the console and pressing enter.

Chapter 2

Debugging - Fixing Broken Code

This chapter is a little different from the others. The rest of this book is designed for you to read through and write Python as you go. This chapter gives you some tips and tricks on finding mistakes in your code, but it isn't very useful until you have some. We put this chapter at the beginning of the book so that you, the reader, would know it is here, and know to come back to it when you get stuck.

Skip this chapter. Go on to the next one. Come back when one of your programs starts spitting out error messages that you don't understand, or when it isn't doing what you think it should. That's when this chapter will be helpful.

2.1   What is a Bug?

In programming, we refer to any mistakes in your code as a "bug". The term "bug" also refers to anytime your code does something that you don't expect it to. As a programmer, you will be spending a good amount of your time trying to find and fix these bugs. We call that "debugging".

All programmers make mistakes, even the experts. Making bugs is such a common occurrence that it's often said, "the only bug free code is no code at all."

Bugs generally come in two types: ones that break your code and ones that don't do what you think.

2.2   Bugs That Break Your Code - Error Messages

As Python reads through your program, if it reaches a line of code that it can't read, or doesn't know what to do with, it will stop and print out an error message. The message that it prints out is intended to help you, so you should learn to read it.

Here's an example of an error message:

```
Traceback (most recent call last):
  File "errorExample.py", line 136, in <module>
    calculated_var=my_func(x,y)
  File "errorExample.py", line 63, in my_func
    divisor=int(xx/y)
NameError: name 'xx' is not defined
```

Let's break it down, piece by piece, starting with:

```
Traceback (most recent call last):
```

This lines tells you that Python is going show you the steps that led up to the error, or trace back to it. The next several lines show you those steps. Here's the first place Python noticed the problem:

```
File "errorExample.py", line 136, in <module>
  calculated_var=my_func(x,y)
```

This tells me that in my file `errorExample.py`, something went wrong when Python tried to execute the command `calculated_var=my_func(x,y)`. The next set of lines tells me that the problem is in the function `my_func`:

```
File "errorExample.py", line 63, in my_func
  divisor=int(xx/y)
```

Specifically, something went wrong on line 63, in the file `errorExample.py`. Line 63 contains the code `divisor=int(xx/y)`.

The last line of the error statement tells me what I did wrong:

```
NameError: name 'xx' is not defined
```

I used a variable `xx` without defining it first. The `NameError:` at the beginning tells me what kind of mistake I made, then the `name 'xx' is not defined` gives more details about the problem. Here are the different types of errors that you can get, along with ideas on things you should try to fix them.

AttributeError

Attribute errors occur when you try to use a method (one of the `.` functions) that doesn't exist. For example, if you set `x=5`, then `x.append(7)` will create an attribute error, since you can't `append` to an integer.

You will also get this error if you misspell the method. Even if `y` is a list, `y.appned(5)` will create an error because `appned` isn't a method. `y.append(5)` will work just fine.

SyntaxError

Syntax errors occur when you've made a mistake formatting your code. Check these sorts of things:

- You've forgotten a `:` at the end of a `def`/`ifforwhile` line.

- You've forgotten to put quotes around a string, or you've mismatched the type of quotes.

- You have mismatched brackets or parentheses.

The parenthesis can be a little harder to track down, because Python will tell you the line number where it noticed there was a problem with parentheses. Since Python also uses parentheses to break one long line of code into multiple lines, the mismatch might be earlier in your program.

NameError

Name errors occur whenever Python doesn't recognize the name of a variable or a function. Here are some common reasons:

- You misspelled the variable/function/method namem

- You use a function from a package that you forgot to import

- You forgot to define a variable

- You use a function before you define it

- You forgot to put quotes around a string so Python is trying to read it as a variable.

TypeError

Type errors occur whenever you try to do an operation on the wrong data type. You can't do float operations on a list, for example. Nor can you divide by a string.

Here's an example of the most common type error I see from beginning programmers:

```
v=[5.0, 6.0, 7.5]
t=5.0
x=v*t
```

Python does not know how to multiply a list (`v`) by a float (`t`), so (`v*t`) gives a type error.

However, this code will run fine:

```
v=[5.0, 6.0, 7.5]
t=5.0
x=v[1]*t
```

since `v[1]` is `6.0` (a float), Python has no problem multiplying `5.0*6.0`.

IndentationError

Python uses indentation to figure out when functions and loops begin and end. If you have too many spaces, (or too few) you will get an indentation error.

You also aren't allowed to mix tabs and spaces. Even if two rows look like they are lined up, if one was indented with a tab and the other with a few spaces, you will get an indentation error. If you are using Canopy, you probably won't run into this problem. Every time you use tab to indent, Canopy automatically replaces tab with four spaces.

There are a few other types of errors that you may run into, but these are the most common. If you are ever completely stumped by an error message, you can usually find answers by just copying the error type and details (the stuff after the `:`) and searching for it on Google.

Errors When Using Imported Packages

If you've imported a package like Numpy, and give the wrong thing to one of the Numpy functions, you will get a very long error message that lists every function that the Numpy function you used depends on. Here's an example:

```
Traceback (most recent call last):
  File "errorExample.py", line 4, in <module>
    plt.plot(data,y)
  File "/Users/test_usr/canopy/lib/Python3.4/site-packages/matplotlib/...
        pyplot.py",  line 3099, in plot
    ret = ax.plot(*args, **kwargs)
  File "/Users/test_usr/canopy/lib/Python3.4/site-packages/matplotlib/...
        axes/_axes.py",  line 1373, in plot
    for line in self._get_lines(*args, **kwargs):
  File "/Users/test_usr/canopy/lib/Python3.4/site-packages/matplotlib/...
        axes/_base.py",  line 304, in _grab_next_args
    for seg in self._plot_args(remaining, kwargs):
  File "/Users/test_usr/canopy/lib/Python3.4/site-packages/matplotlib/...
        axes/_base.py",  line 282, in _plot_args
    x, y = self._xy_from_xy(x, y)
  File "/Users/test_usr/canopy/lib/Python3.4/site-packages/matplotlib/...
        axes/_base.py",  line 223, in _xy_from_xy
    raise ValueError("x and y must have same first dimension")
ValueError: x and y must have same first dimension
```

When that happens, the most useful thing to do is start at the top of the traceback and find the last time that it says the error is in your program. Then, read the error at the bottom. Here are the relevant bits of that long error message:

```
Traceback (most recent call last):
  File "errorExample.py", line 4, in <module>
    plt.plot(data,y)
ValueError: x and y must have same first dimension
```

This lets me know that there is something wrong with my plot command on line 4. The `must have the same first dimension` part of the error lets me know that there is something wrong with the sizes of `data` and `y`.

2.3   Bugs that don't make errors

As you practice programming in Python, you will start to make fewer and fewer mistakes that Python will catch. Your program will run without any errors, but it doesn't do what you expect it to.

When that happens, it tells you that you've written your Python perfectly, but you've told your computer to do the wrong thing. For example, a plot might not have the right axes or you might calculate a distance that is three times further than what you expected.

To catch these sorts of bugs, hunt for them with a bottom-up approach[5]. Start at the place furthest down in your code where you know something has gone

[5] If you have a very large program with lots of pieces, sometimes it can be faster to do a top-down approach to find out which piece is broken, then do a bottom-up approach on just that piece.

wrong. If you are having a problem with a plot, first check the plot commands themselves to see if everything is in order.

If everything looks ok in your plot command, check the variables that you are plotting. If you have a problem with the command

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

Tell your program to print x and y just before you try to plot them:

```
print(x)
print(y)
plt.plot(x,y)
```

then run your program again. When they print[6], check and see if x and y have the sort of information in them that you expect. If they don't, look back at the part of the program where you made x and y, then continue looking backwards until you find your mistake.

[6] You can also see what is stored in the variable x by typing x into the console and hitting enter.

2.4   Toy Problems

Often times when we write programs that do calculations for us, we test them out with toy problems. For example, if you write a program that fits a line to some data, you can test it out by giving it data where you already know the slope and intercept. If you set

```
x=[0,1,2,3,4,5]
y=[0,1,2,3,4,5]
```

Your line fitting function should return a slope of 1.0 and an intercept of 0.0. If it doesn't, you definitely know that you've done something wrong.

As another example, if you are writing a program that will calculate the path a projectile will take when fired through the air while including air resistance, first leave out air resistance and see if your program will match what you expect from kinematics.

Toy problems won't help you catch all of your bugs, but they are a quick and easy way to catch many of them.

Part II

The Basics

Chapter 3

Variables and Data Types

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 its use and limitations. This chapter will focus on assigning, manipulating, and using different variable types.

Python has many types of variables, but you will mostly use the following types: integers, floats, lists, and arrays. What follows is a brief description of each variable type (we'll save our discussion of arrays until later) with some examples to illustrate.

3.1   Integer variables

Some of you may be used to programming in C++[7], 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:

```
a = 20
```

This statement creates the variable a and assigns[8] 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. [9] Note that *variable names in Python are case sensitive,* so watch your capitalization.

You can define multiple variables by putting the assignements on seperate lines:

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

Here we have stored the value 2 in a, the value 4 in b and then used those assignments to calculate a new value: 2×4 and assign that value to the variable c. Other common mathematical operations can be performed on integer variables. For example:

```
a = 20
b = 15
c = a + b  # add two numbers
```

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

[8] 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 varible. This can be done like this:
```
a = input("What is your age?")
```
When you run your program, you will be prompted to enter your age. When you do, that number will be saved to the variable a.

[9] 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.

```
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. This can pose a serious problem for division if you think about it for a second. For example, what would be the value of variable d above if the result has to be an integer? Beginning in version 3 of Python, this problem was resolved so that dividing two integers yields a float.

3.2   Float variables

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

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)
```

You may wonder what would happen when a float and an integer are used in a calculation, 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. Floats can be entered using scientific notation like this

```
1.23e15
```

3.3   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.

3.4   String Variables

String variables contain a sequence of characters, and can be created and assigned using quotes, like this[10]

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

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 functions that can be used with strings. For example, strings can be concatenated (joined) together using the + operator:

```
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 by using `[]` and placing the number of the character you want to access inside of the brackets,like this:

```
a = 'Hello, my name is B. Nelson'
a[0]
a[18]
```

Notice that strings are zero-indexed: the first character is the zeroth element, the second is the 1st, etc. If you want to extract a substring, you can do it by placing multiple numbers in the brackets, seperated by a`:`, like this:

```
a = 'Hello, my name is B. Nelson'
a[2:10]
a[2:10:3]
```

The first statement here will extract the substring starting at the third element (element 2 is the third character) and going up to the tenth element[11]. You can add a stepsize, like we've done in the second statement. In this case, the three means that it will extract every third character starting at the third and ending at the tenth. Some of the more commonly used functions for strings are shown in Table 3.2

3.5   Lists

It is very important that you fully grasp the concept of a list variable. They are heavily-used 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. Let's see how we might create a list.

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

`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 3.2 A sampling of "house-keeping" functions for strings.

[11] Python does not include the last element. `a[1:4]` includes `a[1]`, `a[2]`, and `a[3]` but not `a[4]`.

Creating Lists

The easiest way to create a list is by putting the list elements inside of square brackets, like this:

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

Note that square brackets (`[]`) must be used when creating the list. If you accidentally use parenthesis (`()`)[12] or curly brackets (`{}`)[13] you'll end up creating something other than a list. If you want a list of integers evenly spaced out over a given range, Python's `range` function can help:

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

This will construct a list of <u>integers</u> starting at 10, ending at 51, 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 8 stepping
                 # in increments of 1 (default value)
b = range(10)    # Creates a list starting at 0(default), ending at 9
                 # 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[14]:

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

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:

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

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

[14] You should be careful about associating such a variable with a mathematical matrix. It's just a data container!

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

At times you may want to extract more than just a single element, but not the entire list. You can do this with the : operator, like this

```
aList = [4,5,10,1560,23,19]
#Try printing each of these to see what is in each slice
aList[1:]
aList[1:3]
aList[:3]
aList[1:4:2]
aList[5:2:-1]
```

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 going to the 4th list and accessing the 2nd element. (remember: Python lists are zero-indexed)

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 3.3 lists some of the more common built-in functions/operations for lists. Please note that this is not a comprehensive list.

3.6   Naming Variables

So far in this book, we've named most of our variables using the first few letters in the alphabet. (a,b,c,...) In Python, you can assign a variable any name you'd like, as long as you follow these two rules:

1. Variables must start with a letter or an underscore (_).

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 3.3 A sampling of "housekeeping" functions for lists.

2. The rest of your variable name can only consist of letters, numbers, and underscores

Here are a few examples of allowed names:

```
susan = 72
susan2 = 21
This_is_allowed_but_you_would_never_want_a_name_this_long = 'Hello'
thisStyleIsCalledCamelCase = susan
```

Here are a few names that aren't allowed:

```
2susan = 56
no spaces allowed = 'But you can put spaces in strings'
```

`{}`	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

Table 3.4 Formatting strings available when printing.

3.7 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* with the print statement:

```
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:

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

#This style also works
joe_string="Hi, I am Joe. My GPA is {:5.2f} 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 odd characters inside of the curly braces are a format code: they indicate how you would like the variable formatted when it is printed. The `:d` indicates an integer variable and `:f` indicates a float. 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 selection of what available format statements is given in table 3.4

Chapter 4

Functions and Libraries

One of the most fundamental constructs for any programming language is the function. A function is nothing more than a set of instructions packaged up and given a name. The function can be as long and complex or short and succinct as you wish.

The main purpose of a function is to take in information, do some calculations with or otherwise manipulate that information, then produce a result. Here's a function you should already be comfortable with:

```python
print('Print is a function.')
```

The `print` function takes a string, and tells the computer to write that string to the console.

You do not need to know all of the details of how a function works in order to use it. You just need to know what to give it, and what the result is.

All Python functions have the same general format: first you type the name of the function (`print`), then you put round parentheses `()` around whatever information[15] you are giving to the function. If a function needs more than one piece of information, you separate them with a comma:

```python
x=[1,2,3]
y=[4,5,6]
zipped=zip(x,y) #Join two lists into one, item by item
print(zipped)
```

[15] Not all Python functions require inputs. You use those by typing the name of the function followed by empty parentheses `()`.

You can think of a function as a black box. Someone created the contents of the box, specified what information needs to enter the box for it to be able to accomplish its task, and what information will exit the box. Why a black box? Well, one benefit of functions is that the user doesn't need to know what's inside. They only need to know what information the box needs and what information the box will give back to them.

Python functions generally fall into three groups: functions that come standard with Python (called native functions), functions that you can import into Python, and functions that you write yourself.

4.1 Native functions

There are a few functions that are always ready to go whenever you run Python. They are included with the programming language. We call these functions native functions. You have already been using some of them, like these

```python
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 useful built-in functions can be found in margin tables 3.1, 3.2, 3.3, and 3.4.

4.2 Imported Functions and Libraries

[16] For example, Python doesn't include `sin()` and `cos()` as Native functions.

Many times, you will need to go beyond what Python can do by itself[16]. However, that doesn't mean you have to create everything you need to do from scratch. Most likely, the function that you need has already been coded. Somebody else created the function and made it available to anyone who wants it. Groups of functions that perform similar tasks are typically bundled together into libraries ready to be imported so that the functions that they contain can be used.

It is critical that you know what information(variables) the function expects you to give it and what useful information the function will give back to you. This information can be found in the library's documentation. Most libraries have great documentation with lists of the included functions, what the functions do, what the functions expect, and examples on how to use the most common ones. You can usually find the library documentation by searching the internet for the library's name, plus "Python documentation".

Providing a complete list of all available libraries and function is well beyond the scope of this book. Instead, we'll illustrate how to import functions and use them. As you use Python more and more you should get in the habit of searching out the appropriate library to accomplish the task at hand. When faced with a task to accomplish, your first thought should be, " I'll bet somebody has already done that. I'm going to try to find that library."

[17] 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.

Let's see how to import libraries and use their functions[17]. You've already seen how to perform very simple mathematical calculations. $(5/6, 8^4$, etc..) For more complex mathematical calculations, like $\sin(\frac{\pi}{2})$ or $e^{2.5}$, you'll need to import these functions from a library.

```
import math
```

This imports a library called `math`. This command is like telling Python to go get the `math` book off of the shelf. 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
```

The `math.` before each function is equivalent to telling Python "Use the `sqrt()` function that you find in the `math` book I told you to grab." If you just type

```
sqrt(5.2)
```

Python won't know where to find the `sqrt` function and will give you an error.

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

```python
import math as mt
```

Here, the short name `mt` was chosen for this library. This tells Python "I'm going to call the `math` book `mt`." The desired functions can then be called like this

```python
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

```python
from math import sqrt,sin,pi
```

This code tells Python "Go grab `sqrt`, `sin`, and `pi` from the `math` book. Then, you can use the `sqrt` and `sin` functions without the library name before it, like this

```python
sqrt(5.5)
sin(pi)
```

but, you will only have access to the functions you imported, not all of the functions in the `math` library.

If you want to import every function inside of a library and not have to use a prefix, do this[18]

```python
from math import *
```

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

[18] For larger libraries, importing all of the functions this way can take a long time. A better choice is to import only the functions that you need. You will also get some unexpected results if you use this method to import two different libraries that have functions with the same name.

4.3   User-defined functions

Sometimes, you will need to do something over and over again that you can't find in a library. You (the programmer) will need to write your own function. You do it 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 (the thing that exits the black box). Every user-defined function must begin with the keyword `def` followed by the function name (you can choose it). 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

```
def myFunction(a,b):
    #Everything that is part of the function
    #needs to be indented.
    c = a + b
    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
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 (`f`) is passed out of the function and stored in the variable `result` for later use.

A word on local vs. 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 by the function when it is called and then immediately forgotten. To see what I mean try the following and observe the results

```
result = myFunction(r,t)
print(c)
```

Notice the 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
result = 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 is usually considered to be bad form and confusing.** If you are going to use global variables there better be a very good reason. For example, assigning physical constant, like k_B, G, or ϵ_0, to be global variables is one example of proper use because their values never change and may be used repeatedly in multiple functions. Generally speaking however, every variable that is used in a function ought to be either i) passed in, or ii) defined inside of the function.

Let's look at one more example. Here's (roughly) what Python does every time you use `math.sin(x)`:

```python
def sin(x):
    from math import factorial
    result=0
    for k in range(20): #This Starts a "for loop"
        sign=(-1)**k
        denom=factorial(2*k+1)
        result+=sign*x**(2*k+1)/denom
    return result
```

This function takes in a number (`x`) and returns the sine of `x`. When you use `sin()`, the rest of your program has no idea that the variables `result`, `k`, `sign`, and `denom` were assigned along the way. The main program only knows that `sin()` took `x` and returned a number that has the value of $\sin(x)$.

Importing User Defined Functions

If you ever write a function 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 another file and import them just like a Python library.

As an example, assume that you've written your own `sin`, `cos`, and `sum` functions and saved them in a file called `my_funcs.py`. As long as the program that will use these functions is saved in the same folder[19], you can import and use them like any other library.

[19] You can specify a path to a different folder during your import, or make your functions available to any program using Python on your computer. However, the steps to do so are beyond the scope of this book.

```python
#Method #1
import my_funcs
my_funcs.sin(15)

#Method #2
from my_funcs import sin
sin(15)

#Method #3
import my_funcs as mf
mf.cos(50)

#Method #4
from my_funcs import *
cos(50)
```

Chapter 5

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.

The calculations that you will perform can be roughly categorized into two types: (i) simple calculations involving a couple of numbers (e.g. $5.5\sin(6.2 \times \frac{\pi}{2})$ or $5.5 J_0(3.5 \times \frac{\pi}{4})$) and (ii) more complicated calculations involving many numbers (e.g. $5.5 \sum_i (x_i - D)^2$ where x is a list of 200,000 numbers)

5.1 Simple Calculations

Any mathematical function that you could possibly need to perform a simple calculation can be found in the library `numpy` or `scipy`. Here is an example:

```python
from numpy import sin, pi
from scipy.special import j0 # Bessel function of the 1st kind of
                             # order 0

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

Most likely, `Numpy` has the mathematical function that you need and if it doesn't, then `scipy.special` probably has it. Tables 5.2 and 5.3 provide a small sampling of the functions available. See online help for a more extensive listing.

5.2 More Complex Calculations: Don't Use Lists

What about more complicated calculations involving large data sets? It may *seem* natural to use a list to perform this type of calculation. However, lists were not designed to do math. For example, suppose you defined the following list:

```python
a = [5.1 , 3.2 , 6.8]
```

conceptualizing it as a mathematical vector, and then tried to calculate:

```python
b = 3 * a
```

expecting `b` to become `[15.3,9.6,20.4]`. Try it. Can you explain the result? Or maybe you define two lists and try to add them up, like this:

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

Can you explain the results? So lesson # 1 is: Lists are not mathematical vectors or matrices and doing vector or matrix math cannot be done with a list. If you were allowed to do that, what should Python do when your lists were filled with non-numerical data, like this:

```
a = ['Physics', 'is']
b = ['the', 'coolest', 'topic']
c = a + b
```

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 cannot be done using the list data type.

Second, it may be tempting to associate a 2-D list with a matrix and try to use the : to slice out a submatrix. For example, what if you defined the following 2D list

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

and interpreted it as this matrix:

$$
\begin{pmatrix}
1 & 2 & 3 \\
4 & 5 & 6 \\
7 & 8 & 9
\end{pmatrix}
\tag{5.1}
$$

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

$$
\begin{pmatrix}
5 & 6 \\
8 & 9
\end{pmatrix}
\tag{5.2}
$$

and you tried to do it like this:

```
a[1:3][1:3]
```

you would be disappointed to find out that it did not work that way. You should take a few minutes to understand what it did do. In the end you should always remember to not treat 2D lists as matrices.

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 using the Numpy library. (pronounced num-pie, short for numerical Python)

5.3 Numpy Arrays

If lists are not designed for mathematical calculations, what should be used to do this sort of thing? The answer is a Numpy array. Let's explore this a little more carefully using a specific example that is designed to help you see what we mean. Let's say you have a large 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 want to calculate the summation

$$\sum_{i=1}^{N} (x_i - D)^3 \tag{5.3}$$

where x_i are the data given above and $D = 5$. You could calculate, one-by-one, each contribution in the sum and then add them up. However, there is an easier way involving a library called `numpy`. The main object used in this library is called an `array`, which is very similar to a list except that an array is intended for mathematical use. If `x` were a Numpy array, the code to evaluate the sum in equation 6.2 would be as simple as:

```
D=5 #define D so the next line looks exactly like the equation
my_sum=sum((x-D)**3)
```

Let's explore arrays a little more.

Array Creation

There are several ways to create an array. If you already have a list of numbers and you just want to convert it to an array, you can do it with `numpy`'s `array` function:

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

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 5.1 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.

If you are looking for a function to create an array from scratch, there are plenty of options. The function `arange` is very similar to the native `range` function that you have already seen. The difference is that `arange` creates an array object instead of a list object and `arange` allows the stepsize to be less than one. Here is an example:

```
from numpy import arange
myArray = arange(0,10,.1)
```

This will create an array that looks like this:

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

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:

```
from numpy import linspace
myArray = linspace(0,10,10)
```

This will create an array that looks like this:

```
[0,1.11111,2.222222,3.333333,4.4444444,5.55555555,6.666666,
            7.7777777,8.8888888,10]
```

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

```
from numpy import linspace
myArray,mydx = linspace(0,10,10,retstep = True)
```

Notice that since `linspace` is now returning two things(the array and the step size), we must have two variables to assign them to on the left hand side of the equals sign.

Many other useful function for creating arrays are available. Online documentation is freely available. Table 5.1 gives some of the more heavily-used ones:

Simple 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'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 by
                     # 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.

Mathematical functions with Arrays

Mathematical functions (like sin(x), sinh(x)...etc.) can be evaluated using arrays as their arguments and the result is an array of function values. (Just imagine wanting to calculate the sin of 1,000,000 numbers!!) However, you must use functions from either the `numpy` or `scipy` library, and not from the `math` library. The functions from `numpy` and `scipy` library are 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.

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

5.4   Accessing and Slicing Arrays

Accessing and slicing arrays can be done in exactly the same way as is done with lists. However, there is some additional functionality for accessing and slicing arrays that do not apply to lists.

Accessing Multiple Elements

Let's consider the following 1-d array:

```
from numpy import array
a = array([1,2,3,4,5,6,7,8,9,10])
```

Let's say you wanted to extract elements 2,3,5,and 9. You can extract all of these elements in one shot by using a list of these numbers for the index, like this:

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

You can't do that with lists.

When using two-dimensional arrays, like this one:

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

single elements of this array can easily be extracted, just as with lists:

```
from numpy import array
a = array([[1,2,3],[4,5,6],[7,8,9]])
b = a[0][2]
```

This will extract the number 3, which is the third element in the first list. In other words, it's element 0,2 in array `a`. What if you wanted to extract multiple elements in one shot. Could you do that? In fact you can and here is how:

```
from numpy import array
a = array([[1,2,3],[4,5,6],[7,8,9]])
b = a[[0,2,1],[1,1,2]]
```

Pay special attention to the last line. It will extract the numbers 2,8, and 6. The first list (`[0,2,1]`) indicate the rows (first dimension of the array) where the element is to be extracted and the second list (`[1,1,2]`) indicate the columns that correspond to the rows provided. So elements `[0,1]`, `[2,1]`, and `[1,2]` from list `a` will be extracted. The lists of indices can be as long as you want them to be and all of the corresponding elements will be extracted.

Boolean Slicing

Imagine wanting to extract the elements of an array that meet some provided criteria. An array can be sliced according to the provided criteria very easily by replacing the index with the selection criteria as a boolean statement.

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

Here, only those elements that are greater than 2 will be extracted from the array. It will work on multi-dimensional arrays as well

Multi-dimensional Slicing

We've already shown you how to slice a list using the : operator. The same can be done with arrays. However, for 2D (and higher) arrays the slicing is more powerful (intuitive). It can be helpful to visualize an array as a matrix, even if it is not being treated that way Mathematically. For example, let's say that you define the following array:

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

which could be interpreted as this matrix:

$$\left(\begin{array}{ccc} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{array}\right) \tag{5.4}$$

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

$$\left(\begin{array}{cc} 5 & 6 \\ 8 & 8 \end{array}\right) \tag{5.5}$$

you could do it like this:

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

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:

$$\left(\begin{array}{cc} 2 & 3 \\ 5 & 6 \\ 8 & 8 \end{array}\right) \tag{5.6}$$

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

5.5   Miscellaneous Mathematical Functions

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
```

`airy(z)`	Airy function
`jv(z)`	Bessel function of 1st kind
`yv(z)`	Bessel function of 2nd kind
`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 5.3 A very small sampling of functions belonging to the `scipy.special` library.

If you are summing up the elements of a two-dimensional list, the built-in version of `sum` 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)
```

[20] By default, `axis=0`.   Therefore `sum(a)` and `sum(a,axis=0)` perform the same operation.

Notice the extra argument to the `sum` function in the last line. The `axis=1`[20] 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.

sum	sum up the elements of an array.
prod	find the product of the elements of an array.
cumsum	cumulative sum up the elements of an array.
cumprod	cumulative product up the elements of an array.
diff	discrete difference between all adjacent elements.
cross	cross product of two vectors.
inner	dot(inner) product of two vectors.

Table 5.4 A sampling of miscellaneous functions in numpy. See online documentation for further details.

5.6 Complex Arithmetic

Python can work with complex numbers just as easily as real numbers. The variable j is the imaginary number $\sqrt{-1}$ and a complex number can be specified by placing the j after the imaginary number:

```
a = 1 + 2j
b = 2 + 3j
print(a+b)
```

You may also create a complex number using the `complex` function:

```
a = complex(1,2j)
b = complex(2,3j)
print(a+b)
```

There are several native functions that work on complex numbers:

```
a = 1 + 2j
b = 2 + 3j
c = a.real
d = a.imag
e = a.conjugate()
```

More functions are available in the `cmath` package.

Chapter 6

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.

6.1 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. Just as with functions, Python uses indentation to indicate where the loop ends. [21] The list that you iterate over can contain anything. For instance, here is a loop over a list of strings:

```
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[22]:

$$\Sigma_{n=1}^{1000} \frac{1}{n^2} \tag{6.1}$$

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

[21] Technically, you can iterate on any object that has a member function called `next`, but unless you understand classes, this is probably confusing to you.

[22] `theSum += 1/n**2` is equivalent to `theSum = theSum + 1/n**2`, but runs a little faster and is a little bit easier to read, once you get used to it. See Table 6.1 for more shorthand notations.

Operation	Shorthand Equivalent
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 6.1 A list of shorthand variable reassignment notation.

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 *= n #Multiply theProduct by 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)
```

6.2 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\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
```

==	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.

Study the examples above until it makes sense. Any logical statement can be constructed out of the elements in table 6.2

A word of caution about comparing Python floats is in order here. 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. [23] Try the following to get a feel for this

Table 6.2 Python's logic elements.

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

```
a = 0.1
b = 3 * a
c = 0.3
print(b==c)  # Are they the same number?  You would think they would
             # be right?
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
```

The first two print statements display the value of `b` to one decimal place. The second two print statements force Python to display the value of `b` and `c` out to 45 decimal places. Notice that the true value of `b` is not <u>exactly</u> equal to 0.3. This is why the statement `print(b==c)` returns `False`. The take home message here is that comparing two floats to see if they are equal <u>is always a bad idea.</u> 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)
```

6.3   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 continue to execute until some condition is met. As an example, let's compute the sum

$$\sum \frac{1}{n^2} \tag{6.2}$$

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 += 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<1e-10`. Note that unlike the `for` loop, here you have to do your own counting. 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 wasn't the loop's first logical test would fail and the loop wouldn't execute at all.

Sometimes `while` loops are awkward to use because you can get stuck in an infinite loop[24] if your check condition is never false. The `break` command is

[24] Usually, you can force your program to stop by pressing ctrl+c or cmd+c

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 += 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 useful statement that is used with loops is the `continue` statement. When `continue` is used, the remainder of the loop is skipped and the next iteration of the loop begins. So for example, if you wanted to do the sum from equation 6.2 but only include those terms for which n is a multiple of 3, it could be done like 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
    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 sum will not be updated and the associated terms are effectively skipped.

Chapter 7

Basic 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.

7.1 Linear Plots

Making a Grid

Simple plots of x vs. y are done using a library called `matplotlib`. In order to build make a plot, `matplotlib` needs arrays of x and y points that are going to be plotted. It cannot generate these points on its own. To build an array `x` of x-values starting at $x = 0$, ending at $x = 10$, and having a step size of .01, you'll need numpy's `arange` function:

```
from numpy import arange
x=arange(0,10,0.01)
```

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

```
from numpy import sin
y=sin(5*x)
```

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

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

☞ Remember that you must use <u>numpy's</u> `sin` function if you want to be able to pass an array of values to it. Using math's version will produce an error.

Plotting the Function

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

```
from matplotlib import pyplot
pyplot.figure()
pyplot.plot(x,y)
pyplot.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[25]

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

[25] You determine the filetype by the file extension. `'filename.png'` makes a png image, while `'filename.pdf'` makes a pdf.

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:

```
from matplotlib import pyplot
pyplot.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:

```
from matplotlib import pyplot
nhalf = len(x)/2
pyplot.plot(x[0:nhalf],y[0:nhalf])
pyplot.plot(x[nhalf:],y[nhalf:])
pyplot.show()
```

Controlling the Axes

Python chooses the axes to fit the functions that you are plotting. You can override this choice by specifying your own axes, like this:

```
pyplot.axis([0, 10, -1.3, 1.3])
```

Or, if you want to specify just the x-range or the y-range, you can use `xlim`:

```
pyplot.plot(x,y)
pyplot.xlim([0, 10])
```

or `ylim`:

```
pyplot.plot(x,y)
pyplot.ylim([-1.3, 1.3])
```

[26] Without equal axes, plots of circles are elliptical instead of perfectly round.

And if you want equally scaled axes, so that the x and y axis are the same size[26], use

```
pyplot.axis('equal')
```

Logarithmic Plots

To make log and semi-log plots use the matplotlib's commands `semilogx`, `semilogy`, and `loglog`. They work like this:

```
from matplotlib import pyplot
from numpy import arange,exp
pyplot.clf()  # Clear the figure
x=arange(0,8,0.1)
y=exp(x)

pyplot.semilogx(x,y)  # Replace with semilogy or loglog
pyplot.title('semilogx')

pyplot.show()
```

Plots with Error Bars

To make plots with errorbars, use matplotlib's command `errorbar`. It works like this:

```python
from matplotlib import pyplot
from numpy import arange

x=arange(0,8,0.1)
y=x**2

#Set error in x and y
#Notice that x and y are are different sizes
x_error = 0.05 #Constant error in x
y_error = y/10 #Each error bar will be 10% of x

#Make an errorbar plot without an x error
pyplot.errorbar(x,y,yerr=y_error)
pyplot.show()

#Make an errorbar plot with x error
pyplot.errorbar(x,y,xerr=x_error,yerr=y_error)
pyplot.show()
```

7.2   Plot Appearance

Line Color and Style

To specify the color and line style of your plot, use the following syntax

```python
pyplot.plot(x,y,'r-')
```

The `'r-'` option string tells the plot command to plot the curve in red connecting the dots with a continuous line. Many other colors and line styles are possible, and instead of connecting the dots you can plot symbols at the points with various line styles between the points. For instance, if you type

```python
pyplot.plot(x,y,'g.')
```

you get green dots at the data points with no connecting line. Table 7.1 has a list of the possiblilities

Labeling your plots

To label the x and y axes, do this after the `plot` command:

```python
pyplot.xlabel('Distance (m)')
pyplot.ylabel('Amplitude (mm)')
```

And to put a title on you can do this:

```python
pyplot.title('Oscillations on a String')
```

`'-'`	solid line style
`'--'`	dashed line style
`'-.'`	dash-dot line style
`':'`	dotted line style
`'.'`	point marker
`','`	pixel marker
`'o'`	circle marker
`'v'`	triangle down marker
`'^'`	triangle up marker
`'<'`	triangle left marker
`'>'`	triangle right marker
`'1'`	tri down marker
`'2'`	tri up marker
`'3'`	tri left marker
`'4'`	tri right marker
`'s'`	square marker
`'p'`	pentagon marker
`'*'`	star marker
`'h'`	hexagon1 marker
`'H'`	hexagon2 marker
`'+'`	plus marker
`'x'`	x marker
`'D'`	diamond marker
`'d'`	thin diamond marker
`'|'`	vline marker
`'_'`	hline marker

Table 7.1 Line styles and plot markers.

To write on your plot, you can use Python's `text` command in the format:

```
pyplot.text(10,.5,'Hi');
```

which will place the text "Hi" at position $x = 10$ and $y = 0.5$ on your plot.

You can also build labels and titles that contain numbers you have generated; just construct a formatted string and then use this string variable as the argument of the commands `xlabel`, `ylabel`, and `title`, like this:

```
s = 'Oscillations with k={:.d}'.format(5)
pyplot.title(s)
```

In this example we hard-coded the number 5, but you can do the same thing with variables.

Greek Letters, Subscripts, and Superscripts

α	\alpha
β	\beta
γ	\gamma
δ	\delta
ϵ	\epsilon
ϕ	\phi
θ	\theta
κ	\kappa
λ	\lambda
μ	\mu
ν	\nu
π	\pi
ρ	\rho
σ	\sigma
τ	\tau
ξ	\xi
ζ	\zeta

Table 7.2 The lowercase Greek letters in LaTeX

When you put labels and titles on your plots you can print Greek letters, subscripts, and superscripts by using the LaTeX syntax. To print Greek letters the string should be preceded with an 'r' and the math should be enclosed in `$`, like this:

```
pyplot.xlabel(r'$\theta$')
pyplot.ylabel(r'F($\theta)$')
```

And to put a title on you can do this:

```
pyplot.title(r'F($\theta$)=$\sin$(5 $\theta$)')
```

You can use formatted strings to insert variables into your titles

```
s=r'F($\theta$)=$\sin$({:d} $\theta$)'.format(5)
pyplot.title(s)
```

You can also print capital Greek letters, like this \Gamma, i.e., you just capitalize the first letter.

To put a subscript on a character use the underscore character on the keyboard: θ_1 is coded by typing `\theta_1`. And if the subscript is more than one character long do this: `\theta_{12}` (makes θ_{12}). Superscripts work the same way only using the `^` character: use `\theta^{10}` to print θ^{10}.

7.3   Multiple Plots

You may want to put one graph in figure window 1, a second plot in figure window 2, etc. To do so, put the Python command `pyplot.figure()` before each plot command, like this

```
from numpy import arange, sin, cos
from matplotlib import pyplot

x=arange(0,20,0.01)
f1=sin(x)
```

```
f2=cos(x)/(1+x**2)
pyplot.figure()
pyplot.plot(x,f1)
pyplot.figure()
pyplot.plot(x,f2)
pyplot.show()
```

Overlaying Plots

Often you will want to overlay two plots on the same set of axes. Here's the first way—just ask for multiple plots on the same plot line

```
pyplot.plot(x,f1,'r-',x,f2,'b-')
pyplot.title('First way')
```

Here's the second way: use two separate plot commands before calling `pyplot.show()` or `pyplot.savefig()`.

```
pyplot.figure()
pyplot.plot(x,f1,'r-')
pyplot.plot(x,f2,'b-')
pyplot.title('Second way')
pyplot.show()
```

The second way is convenient if you have lots of plots to put on the same figure. For example, you could create each of the plots in a for loop, then show the plot after the end of the loop.

Subplots

It is often helpful to put multiple plots in the same figure, but on separate axes. The command to produce plots like this is `subplot`, and the syntax is `subplot(rows,columns,plot number)` This command splits a single figure window into an array of subwindows, the array having `rows` rows and `columns` columns. The last argument tells Python which one of the windows you are drawing in, numbered from plot number 1 in the upper left corner to plot number `rows*columns` in the lower right corner, just as printed English is read. See online help for more information on subplots. For instance, to make two-row figure, do this

```
pyplot.subplot(2,1,1)
pyplot.plot(x,f1)
pyplot.subplot(2,1,2)
pyplot.plot(x,f2)
```

Chapter 8

File Input and Output

8.1 Reading Files

Often you will find yourself wanting to read the contents of a file into Python so that you can perform a calculation or make a plot. There are two possible types of files you may want to read: (i) highly-structured files with only numerical data and ii) randomly-formatted files with non-numerical data mixed in with numerical data. By highly structured I mean files where i) every row of your file has the same number of entries, ii) all of your entries are seperated by the same character (maybe just a space but it could be something else too), and iii) no non-numerical entries. The following data file is highly structured:

```
1.2 4 5 2
3.1 2.9 4.9 1.9
4.2 1 9 2
3.1 2.9 4.9 1.9
1.2 4 5 2
3.1 2.9 4.9 1.9
```

It contains only numbers, each row has the same number of entries and the numbers are all seperated by white space. In contrast, here is an example of a data file that is not highly structured:

```
x y z t r(optional)
1.2 4 5 2 6
3.1 2.9&& 4.9 1.9
4.2 ,1 9 2
3.1 2.9, 4.9, 1.9& 5
1.2 4 5 2
3.1 2.9 4.9 1.9
```

Notice that: i) there are non-numerical entries, ii) rows have a different number of entries, iii) entries are seperated by different delimiters[27] (sometimes &&, sometimes a comma is used, and sometimes white space is used).

[27] A **delimiter** is a character that separates data points.

If your file is the first type(highly structured, numerical data only), the Numpy library has functions that will make your life very simple. If your file falls into the latter category, you will have to use native Python functions to read and parse the file contents manually based on the formatting of the file.

Structured data files with numerical data only

Let's start with highly-structured data files with only numerical entries. In this case, Numpy's `loadtxt` is your friend. It will load the file and save the numerical data as an array. Here is an example:

```
from numpy import loadtxt
data = loadtxt('dataFile.txt')
```

If your entries were seperated by something other than white space (commas for example), you could add the optional argument `delimiter =','` to `loadtxt` to specify the seperator. It's worth mentioning that `loadtxt` can handle the presence of non-numerical entries if you need it to. You'll want to use the optional argument `dtypes` to help you specify what type of data `loadtxt` should expect to find. Thank you Numpy!

Randomly-formatted files with non-numerical data

If your data file contains characters and does not have a well-defined structure, you'll have to read the file using Python's native functions and then parse the contents manually. To read in a file you first need to open the file using the `open` command. Next, you can read the contents using one of the following: `read`, `readlines`, or `readline`. Finally, you should always close the file using the `close` command. Let's say you created a file called `dataFile.txt` that had the following contents:

```
x y z t r(optional)
1.2 4 5 2 6
3.1 2.9&& 4.9 1.9
4.2 ,1 9 2
3.1 2.9, 4.9, 1.9& 5
1.2 4 5 2
3.1 2.9 4.9 1.9
```

You could read this file like this

```
f = open('dataFile.txt','r')  #Open the file
data=f.readlines()   # Read the entire file and save it as a list of strings
f.close()
```

The `'r'` in the `open` command indicates that you are opening the file for the purpose of reading it. Other options are `'w'` for writing to files (more to come later on that), `'a'` for appending, and `'r+'` for reading **and** writing. The `readlines` command will read the entire contents of the file (it's your problem if the size of the file exceeds your RAM storage capability) and saves it as a **list of strings**: one string for each line in your file. If you were to use `read` instead of `readlines`, the entire file would be read and saved to a **single string**. The `readline` command will read a single line in the file. An alternate way to read data that automatically ensures that the file closes after the read completes is done like this:

```
with open('dataFile.txt','r') as file:
    data = file.readlines()
```

This will do the exact same thing that the above code does. Python will automatically close the file once the program is no longer indented.

Notice that the data inside of a file is saved as a string (or list of strings). That's most likely not the most convenient form if you are dealing with numerical data. For the above example, using `readlines` to read the example file from above would save the following list in Python:[28]

```
[' x y z t r(optional)\n','1.2 4 5 2 6\n', ' 3.1 2.9&& 4.9 1.9\n',
 ' 4.2 ,1 9 2\n',' 3.1 2.9, 4.9, 1.9& 5\n','1.2 4 5 2\n','3.1 2.9 4.9 1.9\n']
```

How do you get at the data that you need? This is where your ability to work with and modify lists and strings will come in handy. The exact details of what you should do will depend on your specific data file. For this particular file, the commands `del`, `split`, and `float` would probably come in handy.

8.2   Writing to Files

Writing Python data to file is as simple as is reading a file. Just like when you are reading a file, determining which function to use will be determined by the type of data that you are writing. If your data is strictly numerical information stored in an array, Numpy has the function you want to use. On the other hand, if your data is rife with inconsistencies, non-numerical data, etc, you'll have to use something else.

Writing well-structured, numerical data

If you have an **array** of data and you want to save that data to a file, you can do it with `Numpy`'s `savetxt` command, like this:

```
from numpy import savetxt
savetxt('filename.txt',myArray)
```

Maybe you have multiple one-dimensional arrays and you want them saved to a file as columns. You can do that like this:

```
savetxt('filename.txt',(x,y,z))
```

where `x`,`y`, and `z` are the one-dimensional arrays and all have the same length. You can specify a delimiter (seperator) to include in between data entries using the `delimiter=' '` option and putting the delimiter in quotes. See online documentation for other optional arguments.

Non-numerical and randomly-structured file writes

If you need to write a non-data file, you can write it line by using the `write` command. Just like the `readlines()` command, first you need to open the file. Then, use the `write`[29] command to write in your open file:[30]

```
with open('myFile.txt','w') as f:
    f.write('The first line of the file\n')
```

---

[28] The string, `'n'` is called either "new line" or "line break" and is the equivalent of hitting enter on a keyboard in a text editor.

[29] `write` follows the same formatting rules as `print`

[30] Note the use of `'n'` to tell Python to start a new line in the file.

```python
for i in range(2,10):
    f.write('Line Number {}\n'.format(i))
f.write('The last line of the file')
```

Python closes the file once the program is no longer indented.

Part III

Advanced Topics

Chapter 9

Advanced Plotting

Python will display functions of the type $F(x, y)$, either by making a contour plot (like a topographic map) or by displaying the function as height above the xy plane like a perspective drawing. You can also display functions like $\mathbf{F}(x, y)$, where $\mathbf{F}$ is a vector-valued function, using vector field plots.

9.1 Making 2-D Grids

Before we can display 2-dimensional data, we need to define arrays X and Y that span the region that you want to plot, then create the function $F(x, y)$ over the plane. First, you need to understand how Python goes from one-dimensional arrays x and y to two-dimensional matrices X and Y using the commands `meshgrid` and `mgrid`. Begin by executing the following example.

Listing 9.1 (ch9ex1.py)

```python
from numpy import arange,meshgrid,cos,exp,pi,mgrid
from matplotlib import pyplot
from mpl_toolkits.mplot3d import Axes3D   #<- You need this to make 3D plots

# Define the arrays x and y
# Don't make the step size too small or you will kill the
# system (you have a large, but finite amount of memory)
x=arange(-1,1,0.1)
y=arange(0,1.5,0.1)

# Use meshgrid to convert these 1-d arrays into 2-d matrices
# of x and y values over the plane
X,Y=meshgrid(x,y)
#X,Y=mgrid[-1:1:0.1,0:1.5:0.1]   # You could also do this in place of meshgrid
# Get F(x,y) by using F(X,Y). Note the use of .* with X and Y
# rather than with x and y
F=(2-cos(pi*X))*exp(Y)

# Plot the function
fig = pyplot.figure()
ax=fig.gca(projection='3d')
ax.plot_surface(X,Y,F)
pyplot.xlabel('x')
pyplot.ylabel('y')
pyplot.show()
```

The picture should convince you that Python did indeed make things two-dimensional, and that this way of plotting could be very useful. But exactly how Python did it is tricky, so pay close attention. First, notice that anytime you want a 3 dimensional plot, you will need the following lines:

```
from mpl_toolkits.mplot3d import Axes3D  #<- You need this to make 3D plots
fig = pyplot.figure()   # Create the figure object
ax=fig.gca(projection='3d')  # Set the axes to be 3 dimensional
```

Next, we need to understand how `meshgrid` turns one-dimensional arrays `x` and `y` into two-dimensional arrays `X` and `Y`, consider the following simple example. Suppose that the arrays `x` and `y` are given by

```
x=[1,2,3]
y=[4,5]
```

The command `X,Y=meshgrid(x,y)` produces the following results:

Figure 9.1

$$X = \begin{bmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \end{bmatrix} \qquad Y = \begin{bmatrix} 4 & 4 & 4 \\ 5 & 5 & 5 \end{bmatrix}. \tag{9.1}$$

Compare these matrices to Fig. 9.1 to see how they correspond to the small area of x-y space that we are representing. Note that the y dimension entries are flipped from what you might expect because of the difference in conventions for writing matrices (left-to-right and top-to-bottom) versus plotting functions (left-to-right and bottom-to-top).

With `meshgrid`, the first index of both `X` and `Y` is a y-index, since matrix elements are indexed using the `X(row,column)` convention. For example, the elements `X(1,1)` and `X(2,1)` both have value 1 because as the y-index changes, x stays the same. Similarly, `Y(1,1)=4` and `Y(2,1)=5` because as the y-index changes, y does change. This means that if we think of x having an array index i and y having index j, then the two-dimensional matrices have indices

$$X(j,i) \quad Y(j,i). \tag{9.2}$$

But in physics we often think of two-dimensional functions of x and y in the form $F(x,y)$, i.e., the first position is for x and the second one is for y. Because we think this way it would be nice if the matrix indices worked this way too, meaning that if $x = x(i)$ and $y = y(j)$, then the two-dimensional matrices would be indexed as

$$X(i,j) \quad Y(i,j) \tag{9.3}$$

instead of in the backwards order made by `meshgrid`.

Numpy has a command called `mgrid` which is similar to `meshgrid` but does the conversion to two dimensions the other way round. For instance, with the example arrays for x and y used above `X,Y=mgrid[1:4,4:6]` would produce the results

$$X = \begin{bmatrix} 1 & 1 \\ 2 & 2 \\ 3 & 3 \end{bmatrix} \qquad Y = \begin{bmatrix} 4 & 5 \\ 4 & 5 \\ 4 & 5 \end{bmatrix} \tag{9.4}$$

These matrices have the indexes in the $X(i,j)$, $Y(i,j)$ order, but lose the spatial correlation that `meshgrid` gives between Eq. (9.1) and Fig. 9.1.

A third argument can be added to the `mgrid` command to indicate the stepsize. For example, try the following:
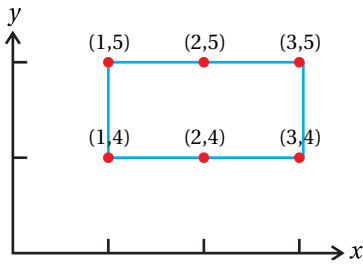
```
from numpy import mgrid
X,Y=mgrid[1:4:0.5,4:6:0.5]
```

The 0.5 that was added indicates that I want a mesh from one to four in increments of 0.5. If the third argument is imaginary, like this:

```
from numpy import mgrid
X,Y=mgrid[1:4:10j,4:6:10j]
```

then the meaning changes to indicating the number of elements you'd like in the range. This is very similar to the difference between `arange` and `linspace`.

Plots made either with `plot_surface(X,Y,F)` or `contour(X,Y,F)` (discussed below) will look the same with either grid. However, `streamline` plots require your data to be in the format provided by `meshgrid`. You will need to be familiar with both methods of creating a grid. You will have need to do it both ways, depending on the circumstance.

9.2 Surface Plots

Now that we understand how to make two-dimensional grids, let's make some plots. Surface plots can be made with either the `pyplot.plot_surface` command or the `pyplot.plot_wireframe` command. Run the following example to see what a wireframe plot looks like. Make sure you read through the comments and watch what it does.

Listing 9.2 (ch9ex2.py)

```python
from numpy import arange,meshgrid,cos,exp,pi,mgrid
from matplotlib import pyplot
from mpl_toolkits.mplot3d import Axes3D  #<- You need this to make 3D plots

# Define the arrays x and y
# Don't make the step size too small or you will kill the
# system (you have a large, but finite amount of memory)
x=arange(-1,1,0.1)
y=arange(0,1.5,0.1)

# Use meshgrid to convert these 1-d arrays into 2-d matrices
# of x and y values over the plane
#X,Y=meshgrid(x,y)
X,Y=mgrid[-1:1:0.1,0:1.5:0.1]  # You could also do this in place of meshgrid
# Get F(x,y) by using F(X,Y). Note the use of .* with X and Y
# rather than with x and y
F=(2-cos(pi*X))*exp(Y)

# Plot the function
fig = pyplot.figure()
ax=fig.gca(projection='3d')
ax.plot_wireframe(X,Y,F)
pyplot.xlabel('x')
pyplot.ylabel('y')
pyplot.show()
```

9.3   Contour Plots

Python generates contour plots very similarly to surface plots. Instead of us-
ing `plot_surface` or `plot_wireframe`, the command `contour` is used. Look at the
example below.

Listing 9.3 (ch9ex3.py)

```python
from numpy import arange,meshgrid,cos,exp,pi,mgrid
from matplotlib import pyplot

# Define the arrays x and y
# Don't make the step size too small or you will kill the
# system (you have a large, but finite amount of memory)
x=arange(-1,1,0.05)
y=arange(0,1.5,0.05)

# Use meshgrid to convert these 1-d arrays into 2-d matrices
# of x and y values over the plane
X,Y=meshgrid(x,y)
#X,Y=mgrid[-1:1:0.1,0:1.5:0.1]  # You could also do this in place of meshgrid
# Get F(x,y) by using F(X,Y). Note the use of .* with X and Y
# rather than with x and y
F=(2-cos(pi*X))*exp(Y)

# Plot the function
pyplot.contour(X,Y,F)  #Also try pyplot.contourf(X,Y,F)
pyplot.xlabel('x')
pyplot.ylabel('y')
pyplot.show()
```

9.4   Vector Field Plots

Python will plot vector fields for you with arrows. This is a good way to visualize
flows, electric fields, magnetic fields, etc. The command that makes these plots
is `quiver` and the code below illustrates its use in displaying the electric field
of a line charge and the magnetic field of a long wire. Note that the vector field
components must be computed in Cartesian geometry.

Listing 9.4 (ch9ex4.py)

```python
from numpy import arange, meshgrid,sqrt,log
from matplotlib import pyplot
x = arange(-5.25,5.25,0.5) # define the x and y grids (avoid (0,0))
y = arange(-5.25,5.25,0.5) # define the x and y grids (avoid (0,0))

X,Y=meshgrid(x,y)

# Electric field of a long charged wire
Ex=X/(X**2+Y**2)
Ey=Y/(X**2+Y**2)
```

```
pyplot.quiver(X,Y,Ex,Ey) # make the field arrow plot
pyplot.title('E of a long charged wire')
pyplot.axis('equal')  # make the x and y axes be equally scaled

# Magnetic field of a long current-carrying wire
Bx=-Y/(X**2+Y**2)
By=X/(X**2+Y**2)

# make the field arrow plot
pyplot.figure()
pyplot.quiver(X,Y,Bx,By)
pyplot.axis('equal')  # make the x and y axes be equally scaled
pyplot.title('B of a long current-carrying wire')

# The big magnitude difference across the region makes most arrows too small
# to see.  This can be fixed by plotting unit vectors instead
# (losing all magnitude information, but keeping direction)
B=sqrt(Bx**2+By**2)
Ux=Bx/B
Uy=By/B

pyplot.figure()
pyplot.quiver(X,Y,Ux,Uy);
pyplot.axis('equal')  # make the x and y axes be equally scaled
pyplot.title('B(wire): unit vectors')

# Or, you can still see qualitative size information without such a big
# variation in arrow size by having the arrow length be logarithmic.
Bmin=B.min()
Bmax=B.max()
# s is the desired ratio between the longest arrow and the shortest one
s=2  # choose an arrow length ratio
k=(Bmax/Bmin)**(1/(s-1))
logsize=log(k*B/Bmin)
Lx=Ux*logsize
Ly=Uy*logsize

pyplot.figure()
pyplot.quiver(X,Y,Lx,Ly)
pyplot.axis('equal')  # make the x and y axes be equally scaled
pyplot.title('B(wire): logarithmic arrows')
pyplot.show()
```

There may be too much detail to really see what's going on in some field plots. You can work around this problem by clicking on the zoom icon on the tool bar and then using the mouse to define the region you want to look at. Clicking on the zoom-out icon, then clicking on the figure will take you back where you came from. Or double-click on the figure will also take you back.

9.5   Streamlines

In addition to plotting little arrows at each point in the vector field, you can plot "streamlines." In fluid dynamics, streamlines show the path that a small particle would follow if it was entrained in the fluid. In electricity and magnetism, streamlines for the electric and magnetic fields are the field lines that you learned about in your introductory electricity and magnetism course.

As mentioned above, the streamline plotting routine assumes that the grid data was created in the `meshgrid` format rather than the `mgrid` format.

Listing 9.5 (ch9ex5.py)

```python
from numpy import arange, mgrid,ones,array
from matplotlib import pyplot
x = arange(0,1,0.1)
y = arange(0,1,0.1)

#Define the position arrays x and y
X,Y = meshgrid(x,y)
#X,Y = mgrid[0:1:0.1,0:1:0.1]  #This won't work for streamplots.!!!

#Define the flow velocity arrays u and v
u = X
v = -Y

#Create a quiver plot of the flow velocity
pyplot.figure()
#pyplot.quiver(X,Y,u,v)

#Plot streamlines that start at different points along the line y=1.
startx = arange(0.05,0.95,0.05)
starty = ones(len(startx)) -0.2
sp = array([startx,starty])
pyplot.streamplot(x,y,u,v,start_points=sp.T,color='crimson',linewidth = 2)
pyplot.show()
```

3-D Line Plots

Python will draw three-dimensional curves in space with the `plot` command. However, there are a few additional lines of code that must be added to make the plot 3 dimensional. Here is how you would do a spiral on the surface of a sphere using spherical coordinates. I've hightlighted the additions needed for the 3D plot

Listing 9.6 (ch9ex6.py)

```python
from matplotlib import pyplot
from mpl_toolkits.mplot3d import Axes3D  # <- Important
from numpy import pi, arange, sin, cos

dphi=pi/100 # set the spacing in azimuthal angle
```

```
N=30 # set the number of azimuthal trips
phi=arange(0,N * 2 * pi, dphi)

theta=phi/N/2 # go from north to south once

r=1  # sphere of radius 1

# convert spherical to Cartesian
x=r*sin(theta)*cos(phi)
y=r*sin(theta)*sin(phi)
z=r*cos(theta)

# plot the spiral
fig = pyplot.figure()
fig.gca(projection='3d')  # Create 3D axis
pyplot.plot(x,y,z)
pyplot.axis('equal')
pyplot.show()
```

9.6 Animations

Often you will want to see how a plot evolves over time. Maybe you are plotting the waveform on a string and you want to see how it changes in time. Animations can be built by repeatedly constructing a new plot and then waiting before you clear the canvas and plot again. Look at the example below to see how to do this:

Listing 9.7 (ch9ex7.py)

```
from numpy import arange,sin,cos
from matplotlib import pyplot

t = 0
x = arange(0,5,0.01)    #Domain over which I want to plot the function.
 while t < 10:
    y = sin(5 * x - 3 * t) * cos(2 * x)  #Construct array of function
                                   # values at current time.
    pyplot.clf()  # Clear the canvas, otherwise all plots end up on
                  # top of each other
    pyplot.plot(x,y,'r-',linewidth=3)
    pyplot.xlabel('x')
    pyplot.ylabel('f(x)')
    pyplot.title('t = {:.4f}'.format(t))
    pyplot.draw()            # draw the plot, but don't wait for
                             # someone to close the window.
    pyplot.pause(0.01)       # Wait before plotting the next one.
    t = t + .1               # Advance time.
```

Here I am plotting snapshots of the function $\sin(5x - 3t)\cos(2x)$ from $x = 0$ to $x = 5$. The animation runs from $t = 0$ to $t = 10$ s. You'll want to make sure that you use `pyplot.draw()` instead of `pyplot.show()` like you've done before. If you use

`pyplot.show()` the animation will stop at each snapshot and you'll have to close the figure window to allow the animation to proceed. Also notice the use of the `pyplot.clf()` command which clears out any previous plots. If you forget to do this, all of the animation's plots will get put on top of each other. And finally, the `pyplot.pause(0.01)` command does exactly what it says: waits before proceeding to the next plot. Changing the pause time will affect the speed of the animation.

Chapter 10

Statistics and Random Number Generation

10.1 Simple Statistical Functions

Doing science often involves collecting large amounts of data and then analyzing the data to extract conclusions. You may find yourself wanting to calculate the mean, standard deviation, and/or other statistical functions. Numpy can do that for you (here I've assumed that `x` contains the data set):

```
from numpy import mean, std

avg = mean(x)
standardDev = std(x)
```

Other useful statistical functions can be found in table 10.1

Let me explain the function `nanmean` briefly. NaN is shorthand for "not a number" and this "value" appears when an illegal math operation is attempted. The "value" `inf` may also appears when mathematical operations are attempted that lead to a very large result. For example, try the following:

```
from numpy import array,sqrt
a = array([[-2.,0.,1.,2.,3.,4.]])
print(sqrt(a))
print(a/0)
```

and look at the results. You should see arrays with non-numerical entries (`nan` or `inf`). If, in the process of post-processing your data, you end up with a `nan` or `inf` buried deep in your data set, the function `mean` will not work correctly. It would be annoying if you had to sift through your data manually and delete all of the `nan`s before you could calculate the mean. Numpy thought of this. The function `nanmean` does the same thing that `mean` does except that it ignores any entries that are nans. Many of the other standard functions have "nan" counterparts too.

10.2 Random Number Generation

Often while programming, you'll need to generate a random number over some interval. What you actually mean when you ask for a random number is to sample from a statistical distribution. For example, the Gaussian (Normal) distribution is peaked in the center and drops off to zero on both sides of the peak. If I were to sample from a Gaussian distribution, numbers close to the peak are more likely to be chosen over numbers that are close to the tails of the distribution. There are many statistical distributions, each with their place in science, and you should always think about which distribution is appropriate for your problem.

median	Median of data
average	Compute weighted average
mean	Compute the arithmetic mean
nanmean	Compute the arithmetic mean, ommiting nan (not a number) entries
percentile	Compute a given percentile of the data
std	Standard deviation
histogram	Compute histogram of data.

Table 10.1 A very small sampling of functions belonging to the `numpy` library.

`rvs(size=100)`	Sample from the distribution. size defaults to 1 if omitted.
`pdf(x)`	Probability density function.
`cdf(x)`	Cumulative density function.
`logpdf(x)`	Log of the probability density function.
`logcdf(x)`	Log of the cumulative density function.
`stats()`	Returns the mean,variance,skew, and/or kurtosis of the distribution.

Table 10.2 A sampling of functions available to any statistical distribution inside of `scipy.stats`.

The simplest distribution is the uniform (or flat) distribution and numpy has a sub-libary that can sample from it (and many others)

```
from numpy.random import uniform

a = uniform(2,5,1000)
```

This will generate an array of 1000 random samples from the uniform distribution in the range (2,5)

Samples from a Gaussian distribution can be generated too

```
from numpy.random import normal

a = normal(5,0.5,1000)
```

This will generate 1000 random samples from a Gaussian distribution centered at 5(location of peak) with a variance of 0.5 (width of peak) A small sampling of the statistical distributions available via numpy are given in table 10.3. For all of the distributions, the optional argument `size=` can be added to specify the number of samples you'd like.

Acceptance-Rejection Method

If you need to generate a random distribution that isn't uniform nor Gaussian, you can use the acceptance-rejection method. The method involves three steps:

1. Use a uniform random number generator to choose a value

2. Find the value of the random number plugged into your desired distribution function.

3. Choose a second random number. If it is smaller than the value found in the previous step, keep the random number from step one. Otherwise, reject the number from step one and repeat.

In order for this method to work, you need to normalize your distribution function so that its maximum value is 1.

Here's an example function that creates a list of random numbers with a Maxwell-Boltzmann distribution[31] given a thermal velocity and the desired number of random numbers.

[31] The Maxwell-Boltzmann Distribution is

$$f(v) = \sqrt{\frac{2}{\pi}} \frac{v^2}{v_{th}^2} \exp\left(-\frac{v^2}{2v_{th}^2}\right)$$

where v is the speed of a particle in a gas, and $v_t h$ is the thermal velocity of the gas. When the peak value is normalized to one, the distribution becomes

$$f_{norm}(v) = \frac{v^2}{2v_{th}^2} \exp\left(-\frac{v^2}{2v_{th}^2}\right)$$

Listing 10.1 (ch10ex1.py)

```
def rand_max_boltz(vth, N):
    #Import the necessary functions from numpy
    from numpy import sqrt, exp
    from numpy.random import rand

    #Load the Maxwell-Boltzmann distribution into a function
    max_boltz = lambda v: v**2/2/vth**2*exp(1-v**2/2/vth**2)
```

```python
    #initialize an empty list to fill with random numbers
    rand_list = []

    #Now repeat acceptance-rejection until the program has reached
    #the desired number of random numbers
    while len(rand_list)<N:
        #Generate the first random speed between 0 and 4*vth
        rand_speed=rand()*4*vth #Technically, the speed should be anywhere from
                                #zero to infinity, but only about 0.1% of particles
                                #Have a velocity greater than 4*vth


        #Find the function value of the random speed
        func_val = max_boltz(rand_speed)

        #Now check func_val against a second random number
        if rand()<func_val:
            rand_list.append(rand_speed)
    return rand_list

#Now use the function

my_dist = rand_max_boltz(100,10000)
```

10.3   Statistical Plotting

Statistical plotting tools range from plotting a continuous distribution to making a histogram of some collected data. Plotting a continuous distribution is nothing more than plotting the function corresponding to that distribution. However, instead of having to look up that function, `scipy.stats` makes your life a little easier by making the functions available to you.

Continuous Distributions

While `numpy` has functions for performing simple statistical operations and sampling a distribution, `scipy.stats` has some additional functionaliy. One extra feature contained in this library is the ability to plot the distribution. Here is an example:

```python
from scipy.stats import norm #import the distribution you want.
from numpy import linspace
from matplotlib import pyplot

x = linspace(-4,4,100)
dist = norm.pdf(x)
pyplot.plot(x,dist,lw=5,alpha=0.6)
pyplot.show()
```

Notice the use of the `pdf` (short for probability density function) function that is associated with the distribution `norm`. To fully understand what is happen-

ing here requires that you understand classes. When you imported `norm` from `scipy.stats`, you actually didn't import a function like you thought. Instead, you imported an "object", and associated with that object are a long list of variables and functions. One of those functions is `pdf`. Other functions that are available for distributions from `scipy.stats` are givein in table **??**.

Histograms

A histogram is generated by binning your domain and then counting up the number of data points in each bin. A histogram can be generated as follows

```python
from numpy.random import normal
from matplotlib import pyplot

a = normal(3,0.5,1000)
pyplot.hist(a)
pyplot.show()
```

Here we have generated some data from a Gaussian distribution and then plotted a histogram of the samples.

Chapter 11

Linear Algebra

Linear algebra is the study of sets of linear equations and their solutions. Matrices and column vectors are common mathematical objects used in this field of Mathematics. In this chapter we will illustrate how to use Python (numpy actually) to solve common problems encountered in the field of linear algebra.

11.1 Matrices

We have already seen the many mathematical advantages associated with using arrays. Numpy also has a matrix object that was specially designed to perform operations that are typical to matrices. If you want to do matrix math or linear algebra, `numpy`'s `matrix` object is a good choice.

Creating Matrices

You can create a `matrix` object in many different 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 an array into a matrix. Note that if you print a matrix object to the 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.

Math with Matrices

Once the matrix is defined, you can use it for lots of cool and useful math. 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.
```

Many other useful functions are available to matrix objects. Online documentation is very helpful. A small sampling of what is available is found in Table 11.1

11.2 Solving a Set of Linear Equations

A common problem encountered in Linear Algebra is that of solving a set of linear equations. Here is an example of a set of two linear equations, with two unknown variables:

$$3x + y = 9 \qquad x + 2y = 8 \tag{11.1}$$

This problem can be represented in matrix form like this:

$$\mathbf{Ax} = \mathbf{b} \tag{11.2}$$

where

$$\mathbf{A} = \begin{pmatrix} 3 & 1 \\ 1 & 2 \end{pmatrix} \tag{11.3}$$

,

$$\mathbf{b} = \begin{pmatrix} 9 \\ 8 \end{pmatrix} \tag{11.4}$$

, and

$$\mathbf{x} = \begin{pmatrix} x \\ y \end{pmatrix} \tag{11.5}$$

,

Numpy has a function called `solve` that will solve this problem. This is how it is used:

```
from numpy.linalg import solve
from numpy import matrix

a = matrix('3,1;1,2')
b = matrix('9;8')
x = solve(a,b)
```

Please note that *a* and *b* need not be matrix objects for the `solve` function to work correctly. Array objects will work just fine. However, if you do use matrix objects, you could also solve the problem like this:

```
x = a.I * b # Multiply the inverse of A by b.
```

`a.trace()`	Calculate trace of matrix a.
`a.shape`	Returns the shape of matrix a.
`a.H`	Returns the conjugate transpose of matrix a.
`a.diagonal()`	Return diagonal elements of matrix a.
`(a==b).all()`	Check if all elements of a matrix satisfy a condition.
`(a==b).any()`	Check if any elements of a matrix satisfy a condition.

Table 11.1 A sampling of functions that work on matrix objects in numpy. See online documentation for further details.

11.3   Eigenvalue Problems

Eigenvalue problems show up in many branches of science including quantum mechanics, signal processing, and geology. The eigenvalue problem is similar to the linear system of equations problem except that the r.h.s is not known. Here is the mathematical statement for an eigenvalue problem.

$$\mathbf{A}\mathbf{x} = \lambda \mathbf{x} \tag{11.6}$$

The solution to this problem yields a value for λ and a corresponding vector, $\mathbf{x}$. Here is how the problem would be solved in Python:

```python
from numpy.linalg import eig
from numpy import matrix

a = matrix('1,-1;1,1')
vals,vecs = eig(a)
```

Notice that the function `eig` returns two things: i) the eigenvalues (λ) and ii) the eigenvectors. Since it returns two things, I need two variables on the l.h.s to save them to. The first thing, `vals`, is the eigenvalues and the second thing, `vecs` is the corresponding eigenvectors.

Chapter 12

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 mimics that data that you've gathered. In this chapter you'll see how to do that.

12.1   Fitting to Linear Functions

The most common way of fitting to linear data is to use a linear least squares fit. Linear least squares is often used because it is easy (once you get used to it), and it has a known solution. (It's a plug and chug formula) Most other types of fits have a computer try several different values[32] and see which one fits the best. Sometimes the computer will miss the actual best fit for something that is just better than the choices nearby, and even if the computer does find the best fit, it will take much longer to get there.

You can use linear least squares whenever you predict that your data should match

$$y = mx + b$$

. However, there is no single line that will go through every single data point. There's always a little bit of error, often written as χ (represented by the little green lines in Fig. 12.1). Therefore, the equation that actually matches the data is

$$y_i = mx_i + b + \chi_i$$

where y_i is the y value corresponding to each individual x value, x_i. χ_i gives how far away each data point is from the line. We can solve for how far away from the line each data point is:

$$\chi_i = y_i - b - mx_i$$

and come up with a function that gives us a total error:

$$E_{tot} = \sum_i^N \chi_i^2 = \sum_i^N (y_i - b - mx_i)^2$$

which is just the sum of how far off every single data point is from the line. We use χ_i^2 as an easy way[33] to get the absolute value of each error. We really only care about how far each data point is from the line, not whether it is above or below the line.

Using calculus, you can find the slope and intercept of the line that will minimize the total error. To do so, take the derivative of the error function with respect to the slope and intercept independently, and set them equal to zero:

$$\frac{\partial E_{tot}}{\partial m} = 0; \ \frac{\partial E_{tot}}{\partial b} = 0$$

[32] The computer chooses which values to try algorithmically. One of the more commonly used methods is the Gauss-Newton Method.
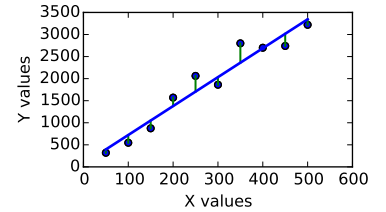

Figure 12.1 A plot of data fitted with a least squares line. The green marks show χ_i, or how far each data point is from the line.

[33] For those with more of a statistics background, using χ_i^2 relates to the standard deviation of the data. Just replace the mean with the value predicted by the equation.

67

If you do those derivatives, and use the two equations to solve for m and b, you get this:

$$m = \frac{\langle xy \rangle - \langle x \rangle \langle y \rangle}{\langle x^2 \rangle - \langle x \rangle^2}$$

$$b = \langle y \rangle - m \langle x \rangle$$

where m and b are the slope and intercept of the line that gets closest to all of the data points and $\langle \rangle$ means the average of the thing inside, i.e. $\langle x \rangle$ is the average of all the xs, and to calculate $\langle xy \rangle$ you'd multiply each x value by its corresponding y value, then take the average. [34]

Here is an example function that takes in lists of x and y data, then returns the slope and intercept of a linear least squares fit line.

[34] Using error propagation, you can find the uncertainty of the fit. The error in the slope is:

$$\sigma_m = \frac{\sigma_y}{\sqrt{N\left(\langle x^2 \rangle - \langle x \rangle^2\right)}}$$

and the error in the intercept is:

$$\sigma_b = \sigma_y \sqrt{\frac{\langle x^2 \rangle}{N\left(\langle x^2 \rangle - \langle x \rangle^2\right)}}$$

σ_y represents the average error of each y value, and is calculated using:

$$\sigma_y = \sqrt{\frac{1}{N-2} \sum_i^N \left(y_i - b - mx_i\right)^2}$$

```python
def linearLeastSquares(x,y):
    #Import numpy
    import numpy as np
    #Get the number of data points
    N=len(x)

    #Make sure x and y are numpy arrays to make array math easy
    x=np.asarray(x)
    y=np.asarray(y)

    #Calculate the average values needed to find the slope and intercept
    xbar=np.mean(x) #Average Value of the x data
    ybar=np.mean(y) #Average value of the y data
    xbar2=np.mean(x**2) #Average value of the x data squared
    xybar=np.mean(x*y) #Average value of xdata*ydata


    #Use the linear least squares formula to calculate
    #the slope and the intercept of the best fit line
    slope=(xybar-xbar*ybar)/(xbar2-xbar**2)
    intercept=ybar-slope*xbar
    return slope, intercept
```

12.2   Fitting to an Arbitrary Function

While fitting to a line is quick and (relatively) easy, not all data will fit a line. To help, the `scipy.optimizes` library contains the `curve_fit` function (among many others). Sounds promising, right? Let's try it out. Assume you've collected the following data

```
x = [4.30949, 5.33127, 2.21479, 5.56794, 3.49002, 0.00272514, 1.3348, \
7.90191, 6.79002, 0.0857548]

y = [0.418161, -0.328667, -1.58646, -0.305927, 0.56044, 0.0272141, \
1.16629, -0.0092056, 0.142341, 0.817534]
```

and you decide it would be a prudent choice to attempt to fit this data to the function

$$y(x) = a\sin(bx)\exp(cx) \qquad (12.1)$$

Your task then is to find values for `a`, `b`, and `c` that make the function the best approximation to the data. To do that we must send our data into the function `curve_fit`. First, we need to import the library and define the function being fitted to:

```
import scipy.optimize as opt

def func(x,a,b,c)
    retun a * sin(b*x) * exp(c * x)

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

Notice that I have defined the function that I am trying to fit the data to and called it `func`. When defining this function, take note that the first argument must be the independent variable (x in this case) and after that you are free to put as many adjustable parameters as you need. In this case we had three (a,b, and c). The `curve_fit` function requires three arguments: The function that is being fitted, the independent variable values, and the dependent variables values (in that order). The `curve_fit` function returns a list of two things. The first thing is a list of the parameter values. The second thing is the variance (uncertainty) in the parameter values. You can access these just as you would do with any other list. 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 = [0.01,0.02,0.1,0.2,0.05,0.2,0.5,0.01,0.02,0.01,0.1,0.05]
fit = opt.curve_fit(func,x,y,sigma=uncertainty)
```

You can also find the uncertainty in the fit parameters:

```
#Change the fit line to:
fit,pcov = opt.curve_fit(func,x,y,sigma=uncertainty)
#pcov holds the estimated covariance of the different fit parameters.
#Convert the covariance into one standard deviation fit error estimates:
fit_err = np.sqrt(np.diag(pcov))  #requires that numpy is imported as np
```

12.3   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 7.
Here I'll show you again (recall that the variable `fit` contains the fit results)

```python
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

xVals = 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 = fitA * sin(fitB * xVals) * exp(fitC * xVals)
pyplot.scatter(x,y)  #Plot the data
pyplot.plot(xVals,yVals)  #Plot the fit function
pyplot.show()  #Show results.
```

Chapter 13

Interpolation and Extrapolation

In computational physics we usually represent functions as arrays of values at discrete points in time and space. But we often want to be able to find function values at points not in the arrays. Finding function values between data points in the array is called *interpolation*; finding function values beyond the endpoints of the array is called *extrapolation*.

You can do both either using sophisticated functions built into the Scipy library or using quick, manually coded methods. This chapter discusses both approaches.

13.1   Manual Interpolation and Extrapolation

If you are going to Interpolate or Extrapolate large datasets, you are usually better off using the Scipy library. However, you will often have a need to quickly calculate a boundary condition, or grab a single point between two others. In these situations, knowing how to manually interpolate and extrapolate will significantly improve your program's performance. This section discusses linear and quadratic interpolation and extrapolation.

Linear approximation

Two points define a line. The equation of the line through those two points (say (x_1, y_1) and (x_2, y_2)) is

$$y = y_1 + \frac{(y_2 - y_1)}{(x_2 - x_1)}(x - x_1) \tag{13.1}$$

where x is where you want to interpolate, and y is the value given by the interpolation.[35]

When using linear interpolation, you must be careful that your points are close enough together that a line gives a good approximation to your data. In Fig. 13.1, for instance, the linear approximation to the curved function represented by the dashed line "a" is pretty poor because the points $x = 0$ and $x = 1$ on which this line is based are just too far apart. Adding a point in between at $x = 0.5$ gets us the two-segment approximation "c" which is quite a bit better. Notice also that line "b" is a pretty good approximation because the function doesn't curve much.

You can also use eq. 13.1 to extrapolate. Often, you will need to find just one more point at the end of your data set. As long as your data is evenly spaced in x, eq. 13.1 simplifies to

$$y_{N+1} = 2y_N - y_{N-1} \tag{13.2}$$

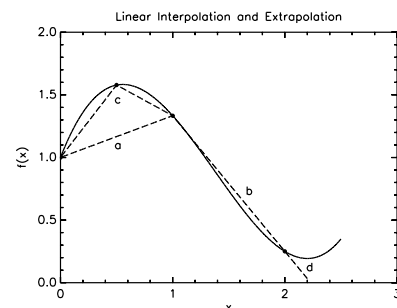where y_N is the last data point, and y_{N-1} is the second to last data point.

Figure 13.1 Linear interpolation only works well over intervals where the function is straight, or when the interval is small enough that the function is approximately straight.

[35] If x in eq. 13.1 is half-way between x_1 and x_2 (at $x = (x_1 + x_2)/2$), then the linear interpolation formula simplifies to $y = (y_1 + y_2)/2$.

Just like with interpolation, you must be sure that the data point you are extrapolating to is roughly linear with the two points you are using to extrapolate: segment "d" in Fig. 13.1 is the linear extrapolation of segment "b", but because the function starts to curve "d" is a lousy approximation once $x > 2$.

Quadratic approximation

Quadratic interpolation and extrapolation is more accurate than linear with curved functions because the quadratic polynomial, $ax^2 + bx + c$, fits curve segments better than the linear polynomial, $ax + b$. Though once again, you must be sure that the data you are fitting to is roughly parabolic. As an example, look at Fig. 13.2. It shows two quadratic fits to the curved function. The one marked "a" just uses the points $x = 0, 1, 2$ and is not very accurate because these points are too far apart. But the approximation using $x = 0, 0.5, 1$, marked "b", is really quite good, and is much better than a two-segment linear fit[36] to the same three points.

[36] See the fit labeled "c" in Fig. 13.1.
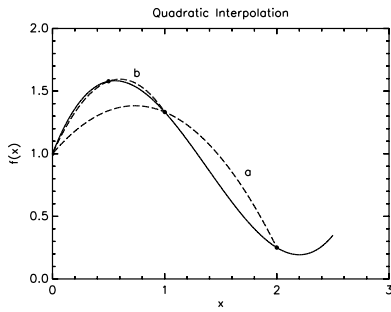
Figure 13.2 Quadratic interpolation follows the curves better if the curvature doesn't change sign.

To derive the quadratic interpolation and extrapolation function, we need three known points, (x_1, y_1), (x_2, y_2), and (x_3, y_3). If our parabola $y = ax^2 + bx + c$ passes through all three points, then these equations will be true:

$$
\begin{aligned}
y_1 &= ax_1^2 + bx_1 + c \\
y_2 &= ax_2^2 + bx_2 + c \\
y_3 &= ax_3^2 + bx_3 + c
\end{aligned}
\tag{13.3}
$$

Unfortunately, when you solve this set of equations for a, b, and c, the formulas are ugly and hard to work with. Unless your data is evenly spaced, you are better off using a more robust fitter from a library. However, if the data is evenly spaced, the result is much simpler. If we assume that x_1, x_2, and x_3 are separated by a distance h (so that $x_1 = x_2 - h$ and $x_3 = x_2 + h$) the solutions are[1]

$$
\begin{aligned}
a &= \frac{y_1 - 2y_2 + y_3}{2h^2} \\
b &= \frac{y_3 - y_1}{2h} - 2x_2 a \\
c &= y_2 + x_2 \frac{y_1 - y_3}{2h} + x_2^2 a
\end{aligned}
\tag{13.4}
$$

---

[1] It is common in numerical analysis to derive this result using Taylor's theorem, which approximates the function $y(x)$ near the point $x = a$ as

$$ y(x) \approx y(a) + y'(a)(x - a) + \frac{1}{2} y''(a)(x - a)^2 + \cdots $$

If we ignore all terms beyond the quadratic term in $(x - a)$) near a point (x_n, y_n), use an array of equally spaced x values, and employ numerical derivatives as discussed in Chapter 14, the Taylor's series becomes

$$ y(x) \approx y_n + \frac{y_{n+1} - y_{n-1}}{2h}(x - x_n) + \frac{y_{n-1} - 2y_n + y_{n+1}}{2h^2}(x - x_n)^2 . $$

which is a quadratic function of $(x - x_n)$, and you can just read off the coefficients a, b, and c.

With these coefficients, we can quickly find approximate y values near our three points using $y = ax^2 + bx + c$. This formula is very useful for getting function values that aren't in the array. For instance, we can use this formula to obtain the interpolation approximation for a point half way between two known points, i.e. $y_{n+1/2} \equiv y(x_n + h/2)$

$$y_{n+1/2} = -\frac{1}{8}y_{n-1} + \frac{3}{4}y_n + \frac{3}{8}y_{n+1} \tag{13.5}$$

and also to find the quadratic extrapolation rule for a data point one grid spacing beyond the last point, i.e. $y_{N+1} \equiv y(x_N + h)$

$$y_{N+1} = 3y_N - 3y_{N-1} + y_{N-2} \,. \tag{13.6}$$

13.2 Python interpolaters

Interp1

The Scipy library has an interpolation routine called `interp1d` which does the things discussed in the previous two sections automatically. Suppose you have a set of data points $\{x, y\}$ and you have a different set of x-values $\{x_i\}$ for which you want to find the corresponding $\{y_i\}$ values by interpolating in the $\{x, y\}$ data set. You simply use any one of these three forms of the `interp1` command (assuming x and y are already loaded with the data points):[37]

```
from scipy.interpolate import interp1d
yi=interp1d(x,y)  # Linear interpolation
yi=interp1d(x,y,kind = 'slinear')  # Spline interpolation to 2nd order
yi=interp1d(x,y,kind = 'quadratic')
```

Here is an example of how each of these three types of interpolation works on a crude data set representing the sine function.

Listing 13.1 (ch13ex1.py)

```
from scipy.interpolate import interp1d
from numpy import pi,linspace, sin,random
from matplotlib import pyplot

# make the crude data set with dx too big for
# good accuracy
dx=pi/5
x=linspace(0,2 * pi,10)
y=sin(x)

# make a fine x-grid
xi=linspace(0,2 * pi,200)
# interpolate on the coarse grid to
# obtain the fine yi values

# linear interpolation
yi=interp1d(x,y,kind = 'linear')
```

[37] Spline interpolation breaks the data into chunks, then fits a polynomial to each chunk. We refer to fitting to different chunks of data as a piece-wise fit. Splines tend to do an excellent job fitting to smooth functions.

```python
# plot the data and the interpolation
pyplot.plot(x,y,'b*',xi,yi(xi),'r-')
pyplot.title('Linear Interpolation')

# quadratic spline interpolation
yi=interp1d(x,y,kind='quadratic');

# plot the data and the interpolation
pyplot.figure()
pyplot.plot(x,y,'b*',xi,yi(xi),'r-')
pyplot.title('Quadratic Spline Interpolation')

# cubic spline interpolation
yi=interp1d(x,y,kind = 'cubic');

# plot the data and the interpolation
pyplot.figure()
pyplot.plot(x,y,'b*',xi,yi(xi),'r-')
pyplot.title('Cubic Spline Interpolation')
pyplot.show()
```

The data used to interpolate need not be spaced out regularly over the domain of interest, but the quality of the interpolation will not be good when there are big gaps between data points. Note also that `interp1d` is an *interpolator* not an *extrapolator*. Hence if you try to "interpolate" at a location that is beyond the domain of the sampled values (that's called extrapolation), you will get an error.

13.3   Two-dimensional interpolation

The Scipy library also has function that will allow you to do 2-dimensional interpolation on a data set of $\{x, y, z\}$ to find approximate values of $z(x, y)$ at points $\{x_i, y_i\}$ which don't lie on the data points $\{x, y\}$. In the completely general situation where your data points $\{x, y, z\}$ don't fall on a regular grid, you can use the command `griddata` to interpolate your function onto an arbitrary new set of points $\{x_i, y_i\}$, such as an evenly spaced 2-dimensional grid for plotting. Examine the code below to see how `griddata` works, and play with the value of `N` and see how the interpolation quality depends on the number of points.

Listing 13.2 (ch13ex2.py)

```python
from scipy.interpolate import griddata
from numpy import mgrid,cos,sin,arange,random,pi
from matplotlib import pyplot
from mpl_toolkits.mplot3d import Axes3D


# Make some "data" at random points x,y points
N=200
points = (random.rand(N,2) - 0.5) * 6
#z = points[:,0]*(1-points[:,0])*cos(4*pi*points[:,0]) *
sin(4*pi*points[:,1]**2)**2
```

```
z=cos((points[:,0]**2+points[:,1]**2)/2)

# Define grid of points to interpolate at
grid_x, grid_y = mgrid[-3:3:100j, -3:3:200j]

# Interpolate to points located at grid_x,grid_y and save to F
F = griddata(points,z,(grid_x,grid_y),method='cubic')

#Plot a contour map
pyplot.imshow(F.T, extent=(0,1,0,1), origin='lower')

#Plot a surface plot
fig = pyplot.figure()
ax = fig.gca(projection='3d')
ax.plot_surface(grid_x,grid_y,F)

# overlay the "data" as dots
pyplot.plot(points[:,0],points[:,1],z,'r.')
pyplot.axis('equal')
pyplot.show()
```

The `griddata` command is very powerful in the sense that you can ask it to estimate $z(x, y)$ for arbitrary[38] x and y (within your data range). However, `griddata` is slow, especially for large data sets. In the case that your data set is already on a regular grid, it's much faster to use the `interpn`[39] command, like this:

<div align="center">

Listing 13.3 (ch13ex3.py)

</div>

```
from scipy.interpolate import interpn
from numpy import mgrid,cos,sin,arange,random,pi, vstack
from matplotlib import pyplot
from mpl_toolkits.mplot3d import Axes3D

# Generate the rough x and y grids
x_grid_rough, y_grid_rough = mgrid[-3:3.1:0.4,-3:3.1:0.4]
# calculate a function usings the rough x and y, then build a surface plot
Z=cos((x_grid_rough**2+y_grid_rough**2)/2)
fig = pyplot.figure()
ax = fig.gca(projection='3d')
ax.plot_surface(x_grid_rough,y_grid_rough,Z)
pyplot.title('Original')
pyplot.show()

#Build the finer grids to interpolate onto
#Note: these grids do not have to have uniform spacing, but they do need
#to be meshable
x_grid_fine, y_grid_fine = mgrid[-3:3.1:0.1,-3:3.1:0.1]

#interpn fits to tuples of arrays, so here we set up those tuples

#These are the original data points.  x_grid_rough[:,0] is a 1-d array
#that contains all of the x positions, y_grid_rough[0,:] holds the y positions
rough_points = (x_grid_rough[:,0],y_grid_rough[0,:])
#Make a tuple out of the grids that will be interpolated onto.
```

[38] `griddata` will work with data of any shape and interpolate it onto a grid, which can be very useful, but is a very computationally expensive process.

[39] `interpn` is built to interpolate in multiple dimensions. If your data is strictly two-dimensional, consider looking into one of these other interpolating functions in the `scipy` library: `interp2d` or `RectBivariateSpline`

```python
fine_grid= (x_grid_fine,y_grid_fine)

#interpn uses three different methods: nearest,linear, and splinef2d.
#Try each of them


#nearest - sets the value of the interploated point equal to the nearest data point
Z_fine = interpn(rough_points,Z,fine_grid,method='nearest')
fig = pyplot.figure()
ax = fig.gca(projection='3d')
ax.plot_surface(x_grid_fine,y_grid_fine,Z_fine)
pyplot.title('Nearest')
pyplot.show()

'''
linear interpolation
Notice that when you plot it, the mesh is finer, and it is much
smoother than nearest.
However, since the interpolation is linear, the sharp corners of the
rough grid are still there.
'''

Z_fine = interpn(rough_points,Z,fine_grid,method='linear')
fig = pyplot.figure()
ax = fig.gca(projection='3d')
ax.plot_surface(x_grid_fine,y_grid_fine,Z_fine)
pyplot.title('Linear')
pyplot.show()

'''
splinef2d
splinef2d does a two dimensional piecewise polynomial fit,
which can fit curves. Notice the increased smoothness
'''

Z_fine = interpn((x_grid_rough[:,0],y_grid_rough[0,:]),Z,(x_grid_fine,y_grid_fine),method='s
fig = pyplot.figure()
ax = fig.gca(projection='3d')
ax.plot_surface(x_grid_fine,y_grid_fine,Z_fine)
pyplot.title('2d Spline')
pyplot.show()
```

Chapter 14

Derivatives and Integrals

In numerical physics we represent functions like $f(x)$ as discrete points on a grid. If you are careful, you can use these discrete values to quickly calculate numerical approximations to the derivative $f'(x)$ and the integral $\int_a^b f(x)\,dx$.

14.1 Derivatives

The instructor of your first calculus class probably introduced the derivative with a formula like this:

$$\frac{df}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}. \tag{14.1}$$

To do a derivative numerically on a grid, we use the following slightly different, but numerically more accurate, form:

$$\frac{df}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x-h)}{2h}. \tag{14.2}$$

It's more accurate because it's centered about the value of x where we want the derivative to be evaluated.

To see the importance of centering, consider Fig. 14.1. In this figure we are trying to find the slope of the tangent line at $x = 0.4$. The usual calculus-book formula uses the data points at $x = 0.4$ and $x = 0.5$, giving tangent line a. It should be obvious that using the "centered" pair of points $x = 0.3$ and $x = 0.5$ to obtain tangent line b is a much better approximation.

As an example of what a good job centering does, try differentiating $\sin x$ at $x = 1$ this way:

```
from numpy import sin
dfdx=(sin(1+1e-5)-sin(1-1e-5))/2e-5
```

Now take the ratio between the numerical derivative and the exact answer $\cos(1)$ to see how well this does

```
from numpy import cos
print(dfdx/cos(1))
```

You can also take the second derivative numerically using the formula

$$\frac{d^2 f}{dx^2} = \lim_{h \to 0} \frac{f(x+h) - 2f(x) + f(x-h)}{h^2}. \tag{14.3}$$

For example,

```
from numpy import sin
d2fdx2=(sin(1+1e-4)-2*sin(1)+sin(1-1e-4))/1e-8
```
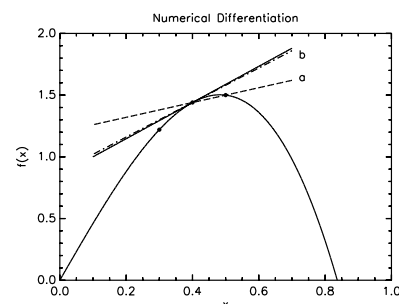

Figure 14.1 The centered derivative approximation works best.

Again, we take the ratio between the numerical derivative and the exact answer $-\sin(1)$ to see how well this does

```
print( d2fdx2/(-sin(1)) )
```

You may be wondering how to choose the step size h. This is a little complicated; take a course on numerical analysis and you can see how it's done. But until you do, here's a rough rule of thumb. If $f(x)$ changes significantly over an interval in x of about L, approximate the first derivative of $f(x)$ using $h = 10^{-5}L$; to approximate the second derivative use $h = 10^{-4}L$.

If you want to differentiate a function defined by arrays x and f, then the step size is already determined; you just have to live with the accuracy obtained by using $h = \Delta x$, where Δx is the spacing between points in the x array. *Notice that the data must be evenly spaced for the example we are going to give you to work.*

The idea is to approximate the derivative at $x = x_j$ in the array by using the function values f_{j+1} and f_{j-1} like this

$$f'(x_j) \approx \frac{f_{j+1} - f_{j-1}}{2h}. \qquad (14.4)$$

This works fine for an N element array at all points from x_2 to x_{N-1}, but it doesn't work at the endpoints because you can't reach beyond the ends of the array to find the needed values of f. So we use this formula for x_2 through x_{N-1}, then use linear extrapolation to find the derivatives at the endpoints, like this

Listing 14.1 (ch14ex1.py)

```python
from numpy import arange, sin, cos, zeros_like
from matplotlib import pyplot

dx=1./1000
x=arange(0,4,dx)
N=len(x)
f=sin(x)

# Do the derivative at the interior points all at once using
# the colon command

dfdx = zeros_like(f)  # Create array of zeros.
dfdx[1:N-1]=(f[2:N]-f[0:N-2])/(2*dx)  # Populate array with derivative values

# linearly extrapolate to the end points
dfdx[0]=2*dfdx[1]-dfdx[2]
dfdx[N-1]=2*dfdx[N-2]-dfdx[N-3]

# now plot both the approximate derivative and the exact
# derivative cos(x) to see how well we did
pyplot.plot(x,dfdx,'r-',x,cos(x),'b-')

# also plot the difference between the approximate and exact
pyplot.figure()
pyplot.plot(x,dfdx-cos(x),'b-')
pyplot.title('Difference between approximate and exact derivatives')
```

```
pyplot.show()
```

Here is an example of a function that takes as inputs an array y representing the function $y(x)$, and dx the x-spacing between function points in the array. It returns yp and ypp, numerical approximations to the first and second derivatives.

Listing 14.2 (derivs.py)

```python
# This function numerically differentiates the array y which
# represents the function y(x) for x-data points equally spaced
# dx apart.  The first and second derivatives are returned as
# the arrays yp and ypp which are the same length as the input
# array y.  Either linear or quadratic extrapolation is used
# to load the derivatives at the endpoints.  The user decides
# which to use by commenting out the undesired formula below.

from numpy import arange,cos,zeros_like
from matplotlib import pyplot

def derivs(y,dx):

    # load the first and second derivative arrays
    # at the interior points

    N=len(y)
    yp = zeros_like(y)  # Initialize array of zeros for first derivative
    ypp = zeros_like(y)  # Initialize array of zeros for second derivative

    yp[1:N-1]=(y[2:N]-y[0:N-2])/(2*dx)
    ypp[1:N-1]=(y[2:N]-2*y[1:N-1]+y[0:N-2])/(dx**2)

    # now use either linear or quadratic extrapolation to load the
    # derivatives at the endpoints


    # linear
    #yp[0]=2*yp[1]-yp[2]
    #yp[N-1]=2*yp[N-2]-yp[N-3]
    #ypp[0]=2*ypp[1]-ypp[2]
    #ypp[N-1]=2*ypp[N-2]-ypp[N-3]

    # quadratic
    yp[0]=3*yp[1]-3*yp[2]+yp[3]
    yp[N-1]=3*yp[N-2]-3*yp[N-3]+yp[N-4]
    ypp[0]=3*ypp[1]-3*ypp[2]+ypp[3]
    ypp[N-1]=3*ypp[N-2]-3*ypp[N-3]+ypp[N-4]

    return yp, ypp


# Build an array of function values
x=arange(0,10,.01)
```

```
y=cos(x)

# Then, since the function returns two arrays in the form
# yp,ypp, you would use it this way:
fp,fpp=derivs(y,.01)

# plot the approximate derivatives
pyplot.plot(x,fp,'r-',x,fpp,'b-')
pyplot.title('Approximate first and second derivatives')
pyplot.show()
```

14.2   Integrals

Definite Integrals

There are many ways to do definite integrals numerically, and the more accurate these methods are the more complicated they become. But for everyday use the midpoint method usually works just fine, and it's very easy to code. The idea of the midpoint method is to approximate the integral $\int_a^b f(x)\,dx$ by subdividing the interval $[a,b]$ into N subintervals of width $h=(b-a)/N$ and then evaluating $f(x)$ at the center of each subinterval. We replace $f(x)\,dx$ by $f(x_j)h$ and sum over all the subintervals to obtain an approximate integral. This method is shown in Fig. 14.2. Notice that this method should work pretty well over subintervals like $[1.0,1.5]$ where $f(x)$ is nearly straight, but probably is lousy over subintervals like $[0.5,1.0]$ where the function curves.
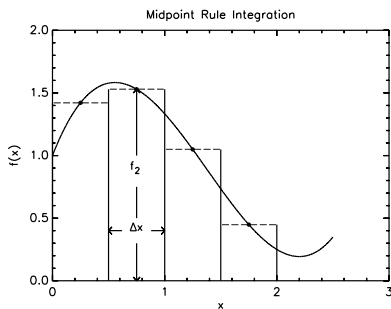
Figure 14.2 The midpoint rule works OK if the function is nearly straight across each interval.

Listing 14.3 (ch14ex3.py)

```
from numpy import arange, cos,sin

N=1000
a=0
b=5.
dx=(b-a)/N

x=arange(.5*dx,b - 0.5 * dx, dx)   # build an x array of centered values
f=cos(x)   # load the function

# do the approximate integral
s=sum(f)*dx

# compare with the exact answer, which is sin(5)
err=s-sin(5)
print(err)
```

If you need to do a definite integral in Python, this is an easy way to do it. And to see how accurate your answer is, do it with 1000 points, then 2000, then 4000, etc., and watch which decimal points are changing as you go to higher accuracy.

Indefinite integrals

And what if you need to find the indefinite integral? For example, maybe you need to calculate the integral of

$$f(x) = e^{-x^2} \tag{14.5}$$

which has no analytical solutions. You can integrate this function numerically in exactly the same way that you did the definite integral. The only difference is that instead of summing up all of the contributions into one number, you keep a cumulative sum. Here is one way that you could do it with Python:

```python
from numpy import arange,exp
from matplotlib import pyplot
N=1000
a=0
b=5.
dx=(b-a)/N

x=arange(.5*dx,b - 0.5 * dx, dx)  # build an x array of centered values
f=exp(-x**2)  # load the function

# do the approximate integral
s=[sum(f[:n]) * dx for n in range(len(f))]  #<- pay attention to this line.
print(s)
pyplot.plot(x,s)
pyplot.show()
```

You'll want to study the line starting with s= until it makes sense. If this line doesn't make sense to you, we can make it more explicit (but less elegant programatically) by replacing the integration line with:

```python
s = []
thesum = 0
for i in f:
    thesum += i * dx
    s.append(thesum)
```

Here we have implemented the rectangle rule for integrating the function. More complicated integration methods (trapezoidal, Simpson's rule, etc.) can be implemented by changing the thing that is being added in the loop.

14.3   Python Integrators

The `numpy` and `scipy` libraries have several functions designed to perform numerical integration. When integrating, there are two types of situations that are most likely: i) You know the function that you want to integrate or ii) You don't know the function you are integrating but you have samples (data points) from the it. If you know what function you want to integrate, `quad`[40], `romberg`, and `quadrature` are good choices. Here is an example of how to use `romberg`:

[40] `dblquad` and `tplquad` are also available for double and triple integrations respectively. Look for online help for more details.

```python
from numpy import sin, exp
from scipy.integrate import romberg

# First define the function you are wanting to integrate
def func(x):
    return sin(2 * x**2) * exp(-x)

integral = romberg(func,0,5)
print integral
```

The functions `quad` and `quadruature` can be used in the same way.

For example, `numpy` has a function called `cumsum` (sounds helpful here right?) that integrates using the rectangle rule, just as we did above. Here is how it works:

```python
from numpy import arange,exp,cumsum
from matplotlib import pyplot
N=1000
a=0
b=5.
dx=(b-a)/N

x=arange(.5*dx,b - 0.5 * dx, dx)   # build an x array of centered values
f=exp(-x**2)   # load the function

# do the approximate integral
s=cumsum(f*dx)
pyplot.plot(x,s)
pyplot.show()
```

The Scipy libary has a sublibrary called `integrate` which contains a host of useful functions for integrating. function will do a similar thing except that it will use. This function takes an array of function values in y and an x-spacing `dx` and returns an approximate indefinite integral function $g(x) = \int_a^x y(x')dx'$ using the trapezoid rule. This rule says to use as the height of the rectangle on the interval of width h the average of the function values on the edges of the interval:

$$\int_x^{x+h} y(x')dx' \approx \frac{y(x) + y(x+h)}{2}h \tag{14.6}$$

Because `cumtrapz` uses the trapezoid rule instead of the midpoint rule, the array of function values must start at $x = a$ and be defined at the edges of the subintervals of size h rather than at the centers. Once you have your function values stored in an array, say f, you calculate the indefinite integral like this:

```python
from scipy.integrate import cumtrapz
from numpy import arange,abs,cos,pi
from matplotlib import pyplot
dx = 0.01
x = arange(0,2 * pi,dx)
f = cos(x)
g=cumtrapz(f,x,initial = 0)
pyplot.plot(x,g,'ro',x,f,'b-')
pyplot.show()
```

where dx is the point spacing.

Here is a function that does essentially the same calculation as cumtrapz, but is coded to be readable. This function takes dx as an argument and multiplies it at each step of the calculation, whereas cumtrapz assumes a step size of one. This code is more intuitive, but less efficient. Study this code to see how it works, but use cumtrapz and multiply by dx as above when you actually do calculations.

<div align="center">

Listing 14.4 (indefint.py)

</div>

```
function g=indefint(y,dx)

% returns the indefinite integral of the function
% represented by the array y.  y(1) is assumed to
% be y(a), the function value at the lower limit of the
% integration.  The function values are assumed to be
% values at the edges of the subintervals rather than
% the midpoint values.  Hence, we have to use the
% trapezoid rule instead of the midpoint rule:
%
% integral(y(x)) from x to x+dx is (y(x)+y(x+dx))/2*dx

% The answer is returned as an array of values defined
% at the same points as y

% the first value of g(x) is zero because at this first value
% x is at the lower limit so the integral is zero

g(1)=0;

N=length(y);

% step across each subinterval and use the trapezoid area
% rule to find each successive addition to the indefinite
% integral

for n=2:N

   % Trapezoid rule
   g(n)=g(n-1)+(y(n-1)+y(n))*.5*dx;

end
```

For simple integrals of functions sampled at discrete points (e.g. measured data) it is usually fine to use trapz to calculate a definite integral, or cumtrapz to calculate an indefinite integral. Both functions treat the function as little line segments connecting the data points in your array. Performing integrals on raw measured data is usually a bad idea unless your data is very clean. If your data is noisy, you'll probably want to consider smoothing it or fitting it to a curve before trying to take an integral.

If you want to calculate the integral of a function rather than data stored in an array, you have more options because you can calculate function values at arbitrary points rather than having to interpolate between array elements. For

this type of problem, use the command `integral` (or `integral2`, or `integrate3` for multiple dimension integrals). These functions take references to a Python function in their arguments (like `fzero`, `fsolve`, and `fminsearch` did) and adjust the grid step size to optimize accuracy.

The Python routine `integral` adaptively approximates the function with parabolas (as shown in Fig. 14.3) or other functions instead of the rectangles of the midpoint method. The parabolic method is called Simpson's Rule. As you can see from Fig. 14.3, parabolas do a much better job, making `quad` a standard Python choice for integration. To use the `integral` command, you need to define your function using an anonymous function. For example, to define an anonymous function to represent the function $f(x) = \cos(x)e^{-x}$, you use this syntax:

```
f=@(x) exp(-x).*cos(x)|
```

After you have defined `f` this way, you can use it just like built-in Python functions like `sin` and `cos`. For instance, try these commands after defining `f` as above:

```
f(1)
f(1:10)
```

Using the anonymous function syntax, we can use `integral` to integrate $\cos(x)e^{-x}$ from 0 to 2 like this:

Listing 14.5 (ch14ex5.py)

```
clear; close all;

f=@(x) exp(-x).*cos(x)

integral(f,0,2,'AbsTol',1e-8)
```

Python also has a command `integrate2` that does double integrals. Here's how to use it to integrate the function $f(x, y) = \cos(xy)$.

Listing 14.6 (f2int.py)

```
clear; close all;

f = @(x,y) cos(x.*y);

integral2(f,0,2,0,2,'AbsTol',1e-10)
```
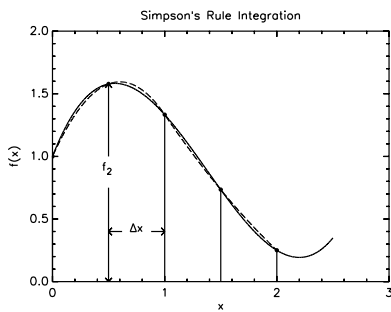

Figure 14.3 Fitting parabolas (Simpson's Rule) works better.

Chapter 15

Advanced Python Techniques

There are several very useful Python techniques where full walkthroughs and descriptions are beyond the scope of this book. However, many of them you may not come across through the course of a normal science/engineering education.

This chapter contains many of those, with brief descriptions and a few examples so that you as a reader can see first, that they exist; second, when they might be useful; and third, what they are called so you can further research them on your own.

15.1 Iterating with the enumerate function

You've seen how to iterate over a list, extracting each element, one-by-one. There are times when, in addition to extracting the elements of the thing you are iterating over, you'd like to also extract it's location in the list. You can do that with the `enumerate` function. Here's an example where every fourth element is added up:

```python
a = [4,5,2,8,99,65,78,32,39,67,92,120,567,23]
thesum = 0
for index,val in enumerate(a):
    if index % 4 == 0:
        thesum +=val
```

The function `enumerate` returns the list element **and** it's location in the list.

15.2 Lambda (unnamed) functions

Sometimes, a function that 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:

```python
f = lambda x: x**3   #Define the function
print f(5)    # Evaluate the function
g = lambda x,y: x + y # Function of two variables
print 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.[41] (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:

[41] Stop and process that for a minute.

```python
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 that 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`.

15.3   Inline If Statements

Many times, you will need to set up this sort of if statement:

```python
if x>3:
    y=10
else:
    y=-10
```

This line of code does exactly the same thing:

```python
y= 10 if x>3 else -10
```

This format is called a "ternary operator" or a "conditional expression". They can make your code a little easier to read (sometimes), but they are exceptionally helpful when you perform operations that will not let you write a multiline if/else statement, like when setting values in a list comprehension.

All conditional expressions follow the format: <value if true> if <conditional> else <value if false>. Here's another example:

```python
mood='Bad'
print('Life is {}.'.format('terrible' if mood=='Bad' else 'wonderful'))
```

15.4   List Comprehensions

A list comprehension is a more compact way to write a `for` loop. Also, because of how Python handles memory, a list comprehension will be much faster than a `for` loop. As an example, here is a `for` loop that takes a list of the numbers from 0 to 10 and creates a list of those numbers, squared:

```python
n_squared=[]
for n in range(10):
    n_squared.append(n**2)
```

This code will produce the exact same result, but faster:

```python
n_squared = [n**2 for n in range(10)].
```

This structure is referred to as a list comprehension. Here is the general format:[42]

```
[<item_in_new_list> for <item_in_old_list> in <old_list>]
```

Notice that you can only do list comprehensions on existing lists. They do not work well for things like Euler's method, where the next value in a list depends on a previous value.

Here are a few more examples of useful list comprehensions:

```
#Get only odd numbers from 0 to 20
a = [n for n in range(20) if n%2==1]

#Use a conditional expression to halve even numbers and square odds
b = [n/2 if n%2==0 else n**2 for n in range(10)]

#You don't just have to use numbers
c = ['Physics','is','the','best']
d = [word.upper() for word in c]
```

You can also put multiple `for` statements in one comprehension. Execute this block of code and try to wrap your head around what each comprehension does.

```
list_1 = [5,3,7,8]
list_2 = [2,7,10]

#Make a list of every possible pairing
combinations = [[x,y] for x in list_1 for y in list_2]
print(combinations)

#Remove any pairs where both numbers are the same
remove_doubles = [x for x in combinations if x[1]!=x[0]]
print(remove_doubles)

#Put all the numbers into one long list
long_list = [x for pair in remove_doubles for x in pair]
print(long_list)
```

Notice in the `long_list` example that `pair` must be set with the first `for` statement, then iterated over in the second. Python reads left to right. This format will create an error since Python runs into the variable `pair` before it knows what it means:

```
long_list = [ x for x in pair for pair in remove_doubles]
```

You can also use comprehensions with tuples and dictionaries[43]:

```
#Tuple comprehension  -> use ()
my_tuple = ((-1)**n/n for n in range(10))

#Dictionary comprehension
my_dict = {x:x**2 for x in range(30)}
```

[42] Putting <> around a phrase is a stand in for "put this thing here". You cannot copy this exact code and expect it to run in Python.

[43] For more information on tuples and dictionaries, see section 15.5.

15.5   Data Structures

The "Data Structure" of a variable tells you how information is stored in that variable. So far in this book, we've worked mostly with three different types: single variables, lists, and Numpy arrays. This section introduces three other very useful structures: the tuple, the dictionary, and the class[44].

[44] You may want also want to look into data trees, as well as common data storage formats like json and xml.

Tuples

This book has mentioned tuples a few different times. They are very similar to lists, but with two key differences:

[45] This property is referred to as being **immutable**

- Tuples cannot change size or shape[45]. So `append`, `insert`, `pop`, and other similar methods do not work on Tuples.

- Tuples are faster than lists, since they cannot change size or shape.

Creating a tuple is ver similar to creating a list. Just replace the `[]` with `()`.

```python
my_list = [5,'Hello',[72, 63], -1.431]
my_tuple = (5,'Hello',[72, 63], -1.431)

print(my_list[1])
print(my_tuple[1]) #Notice that tuples still use [] for indexing
```

Since tuples cannot be modified , they are great when you don't want to accidentally modify your data set.

Dictionaries

Imagine that your program needed the name, age, and birthday of several of your classmates. You could store that data as a list of lists, like this:

```python
classmates=[['Christine','Whitney',25, 'March 10'],
            ['Marvin','Flick', 21, 'April 17']]
```

Then you would have to remember that the first thing in the list is the first name, the second thing is the last name, then the age, then the birthday. You could get the last name of the second student with `classmates[1][1]`, and their birthday with `classmates[1][3]`.

Dictionaries make it much easier to store this sort of information. Rather than using indices like a list, dictionaries store information in what are called key value pairs. Here's our `classmates` list rewritten as a list of dictionaries:

```python
classmates=[{'First':'Christine','Last':'Whitney','Age':25,'Birthday':'March 10'},
            {'First':'Marvin','Last':'Flick','Age':21,'Birthday':'April 17'}]
```

In the first element of our list, the key `'First'` correspondes to the value `'Christine'`. This makes it much easier to keep strack of what kind of data is stored where. Now, to get the first student's last name, you can use `classmates[0]['Last']`.

Dictionaries do not preserve order. From Python's perspective, both of these dictionaries would be the same:

```
dict_1={'First':'Christine','Last':'Whitney','Age':25,'Birthday':'March 10'}
dict_2 = {'Age':25,'Last':'Whitney','First':'Christine','Birthday':'March 10'}
```

Both `Age'dict_1'` and `Age'dict_2'` would have the same value, 25.

Here's a few operations you can do with dictionaries:

```
student = {'First':'Marvin','Last':'Flick','Age':21,'Birthday':'April 17'}
student['Age']=22 #Change a value
student['Height']=1.77 #Add a key/value pair
student.pop('Last') #Remove a key/value pair

#Iterate over a dictionary's keys
for key in student:
    print(key)

#Iterate over key/value pairs
for key,value in student.items():
    print(key,value)
```

Classes

In Python, a class is a dictionary, with a few extra features. They are most commonly used when you have lots of different objects that will have the same data structure. With the name, age, birthday example, you will have several different students[46], each with a name, age, and birthday. By setting up a class, you can tell Python what to expect. Here's an example:

[46] The overall data structure is called a class, each individual student that you load with the class is referred to as an "object" or "instance".

```
class Student: #Declare the class name
    #This function runs whenever you make a new object with this class
    def __init__(self,First,Last,Age,Birthday):
        self.First=First  #self refers to the current object that you are making
        self.Last=Last
        self.Age=Age
        self.Birthday = Birthday

#Now Load the students:
firstStudent=Student('Christine','Whitney',25, 'March 10')
secondStudent = Student('Marvin','Flick', 21, 'April 17')

#Print the first student's last name
print(firstStudent.Last)
#You can also reference it like a dictionary
print(firstStudent['Last'])


#Make a list of students
student_list = [firstStudent, secondStudent]
```

In addition to saving information like a dictionary, you can also add functions to a class:

```python
class Student: #Declare the class name
    #This function runs whenever you make a new object with this class
    def __init__(self,First,Last,Age,Birthday):
        self.First=First  #self refers to the current object that you are making
        self.Last=Last
        self.Age=Age
        self.Birthday = Birthday

    def getOlder(self):  #By passing self to this function,
                         #The function has access to whatever is in the object
        self.Age+=1

#Now Load the students:
firstStudent=Student('Christine','Whitney',25, 'March 10')

#And use our function
firstStudent.getOlder()
print(firstStudent.Age)
```

A Physics Example

Try to read through this next example and see if you can figure out how it works. As a general outline, the program fills a two dimensional box with moving particles, lets the particles move around, and keeps the particles in the box by making them bounce off of the walls. The program defines two classes: a box and a particle.

       The particle class keeps track of where the particle is, and what its velocity is. The particle class will also assign a random position and velocity (within a given range) whenever it creates a new particle. Each particle has the ability to move itself, given an amount of time.

       The box class sets up the box. It keeps track of how big the box is, and what particles are in it. It can move all of the particles in the box. It will also check if any particles have left the box, and makes them bounce off the walls instead.

       Once the classes are built, this program uses those classes to build a box with 2 particles, then plot the position of the particles as they are moved 50 times. Here is the example code:

Listing 15.1 (ch15ex1.py)

```python
#Define a particle class
class Particle:
    #Set up the initialization function
    def __init__(self,x_min,x_max,y_min,y_max,v_max):
        #Import a random number generator
        from random import random


        #Set the initial x position by randomly
        #choosing a value between x_min and x_max
        self.x=(x_max-x_min)*random()+x_min

        #Repeat for y
        self.y=(y_max-y_min)*random()+y_min
```

```python
        #Choose a random speed between 0 and v_max
        v0=v_max*random()


        #This next part shows a fancy way to randomly generate an angle

        #First, choose a random sin(theta)
        sin_theta = random()

        #use sin^2+cos^2 = 1 to find cos(theta)
        cos_theta=(1-sin_theta**2)**0.5

        '''sin_theta and cos_theta are limited to positive values
        Randomly choose whether they should be positive or negative
        By multiplying by either 1 or negative 1.'''
        sin_theta*= 1.0 if random()>0.5 else -1.0
        cos_theta*= 1.0 if random()>0.5 else -1.0

        #Done generating the angle =================



        #Now give the particle its initial velocity
        self.vx=v0*cos_theta
        self.vy=v0*sin_theta


    #Set up a function that will move particles on a straight line path
    def move(self,dt): #dt is the time for which to move the particles
        self.x+=self.vx*dt
        self.y+=self.vy*dt



#Now build a class for the box
class Box:
    #Set up the initialization function
    '''
    The initialization function takes 4 inputs:
        x_size -> a list giving the size of the box in x -> [x_min, x_max]
        y_size -> a list giving the size of the box in y -> [y_min, y_max]
        num_parts -> the number of particles in the box
        v_max -> the maximum speed of a particle in the box '''

    def __init__(self,x_size,y_size,num_parts,v_max):
        #Load the box size
        self.x_size = x_size
        self.x_min = x_size[0]
        self.x_max = x_size[1]

        self.y_size = y_size
        self.y_min = y_size[0]
        self.y_max = y_size[1]
```

```python
            #Fill the box with a list of particles
            self.particles=[
                Particle(self.x_min,self.x_max, #Use the particle class
                         self.y_min,self.y_max, v_max)
                         for _ in range(num_parts) ] #repeat num_parts times

    #Set up a function that keeps particles in the box by
    #bouncing them off the walls of the box.
    def check_walls(self):
        #Check each of the particles
        for particle in self.particles:
            #If a particle is outside of the box in the x direction,
            #Reflect it off of the walls until it it no longer outside
            #the box
            while particle.x > self.x_max or particle.x < self.x_min:
                #Find out whether the particle is above x_max or below x_min
                x_ref = self.x_max if particle.x > self.x_max else self.x_min
                #This part brings the particle back into the box
                #If it is one meter above x_max, this sets the particle
                #One meter below x_max, and will do the same for x_min
                particle.x=2*x_ref-particle.x

                #Flip the direction of its x-velocity
                particle.vx*=-1

            #Repeat for the y direction
            while particle.y > self.y_max or particle.y < self.y_min:
                y_ref = self.y_max if particle.y > self.y_max else self.y_min
                particle.y=2*y_ref-particle.y
                particle.vy*=-1

    #Set up a function that moves the particles
    def move_parts(self,dt):
        #First, move all of the particles
        [particle.move(dt) for particle in self.particles]

        #Bounce the particles off of the walls
        self.check_walls()

#End of class definitions, the rest of this program uses them ===========================


#Box size
box_x = [-1,1]
box_y = [-2,2]

#build a box
my_box = Box(box_x,box_y,2,10)

import matplotlib.pyplot as plt

for i in range(50):
    #Plot the particles' positions
    plt.scatter(my_box.particles[0].x,my_box.particles[0].y,color='blue')
    plt.scatter(my_box.particles[1].x,my_box.particles[1].y,color='red')
```

```
    #Move the particles
    my_box.move_parts(.01)

plt.xlim(box_x)
plt.ylim(box_y)

plt.show()
```

15.6 Pandas

Pandas is a Python package built to work with spreadsheet-like data, and it is very good at its job.

Pandas stores data in something called a "dataframe". A dataframe is simply data stored in rows and columns. As an example, here is some sample data taken by an accelerometer sitting on an elevator floor:

```
time,gFx,gFy,gFz
0.007,-0.0056,-0.0046,1.012
0.008,0.0007,0.0024,1.0022
0.008,0,0.0059,1.0039
0.009,0.0054,-0.0022,1.0032
0.009,-0.0015,-0.0056,1.0042
0.009,0.0037,-0.002,0.9951
0.01,-0.002,-0.002,1.002
0.014,0.009,-0.0024,1.0159
0.015,0.0012,-0.0037,1.01
0.017,-0.0115,-0.002,1.0012
0.019,-0.0022,-0.0015,1.001
0.021,0.0024,-0.0022,1.0166
```

Here is that same data loaded into a dataframe:

```
     time     gFx     gFy     gFz
0   0.007  -0.0056  -0.0046  1.0120
1   0.008   0.0007   0.0024  1.0022
2   0.008   0.0000   0.0059  1.0039
3   0.009   0.0054  -0.0022  1.0032
4   0.009  -0.0015  -0.0056  1.0042
5   0.009   0.0037  -0.0020  0.9951
6   0.010  -0.0020  -0.0020  1.0020
7   0.014   0.0090  -0.0024  1.0159
8   0.015   0.0012  -0.0037  1.0100
9   0.017  -0.0115  -0.0020  1.0012
```

It has 10 rows (0-9) and 4 columns (time, gFx, gFy, gFz). Here's an example of the few things you can do with Pandas. This example assumes that the elevator data is saved in the same folder as the program and is titled `elevator.csv`[47].

Listing 15.2 (ch15ex2.py)

```python
import pandas as pd
```

[47] "csv" stands for comma separated values. It is a very common datafile type, where each row is another set of data, and each value is separated by a comma. Two other common data file types that are useful to look into are "JSON" and "xml".

```python
import numpy as np


#Load the datafile into a dataframe
elevator_data=pd.read_csv('elevator.csv')

#Print the first five lines of the dataframe
print(elevator_data.head(5))

#Load column names into a list

column_names = list(elevator_data.columns)

#Re-Zero the gFz column using the first five data points
elevator_data['gFz']-=elevator_data['gFz'][0:5].mean()

#Print the first 3 times
print(elevator_data['time'][0:3])

#convert accelerations from units of g to m/s^2
accel_columns=['gFx','gFy','gFz']
elevator_data[accel_columns]*=9.8

#Create a new column with the magnitude of the acceleration
elevator_data['a_mag']=np.sqrt((elevator_data[accel_columns]**2).sum(axis=1))

#Find the maximum accleration
max_accel = elevator_data['a_mag'].max()

#Zero out any accelerations that are less than 10% of the maximum.
zero_spots=elevator_data['a_mag']<0.1*max_accel
#loc is useful when you need to set values on only part of a dataframe
elevator_data.loc[zero_spots,accel_columns+['a_mag']]*=0
#Usually, you can do the same thing with
#elevator_data[zero_spots][accel_columns+['a_mag']]*=0
#But there are a few exceptions, so pandas will give a warning
#if you don't use .loc[]

#plot t vs a_mag
import matplotlib.pyplot as plt
plt.plot(elevator_data['time'],elevator_data['a_mag'])
plt.show()

#Save the dataframe to a file
elevator_data.to_csv('elevator_processed.csv')
```

15.7   Regular Expressions

Python by itself is good at finding regular strings inside of others. For example,

```python
my_string = "The quick brown fox"
print(my_string.find('fox'))
```

will print 16 since the word fox starts at `my_string[16]`. However, `find` will not help you get all of the dates out of this sort of data:

```
date, temperature
05/08/2016, 69
05/09/2016, 59
05/10/2016, 51
05/11/2016, 56
05/12/2016, 70
05/13/2016, 78
05/14/2016, 78
```

You could search individually for every single date string, but there is a better way.

Regular expressions, or regex for short, are a way to match patterns in strings. Here's an example that will match any date in the mm/dd/20yy format:

```
'[0-1][0-9]/[0-9]{2}/20[0-9]{2}'
```

Let's break down that example piece by piece. Square brackets `[]` will match whatever is listed inside the brackets. So, `[0-9]` will match any digit from zero to nine while `[0-1]` only matches zero or one. Therefore, you can use `[0-1][0-9]` to match 00, 01, 02, all the way up to 19. Putting curly braces `{}` around a number tells the computer how many times the thing before is repeated. Therefore, `21{5}` tells the computer to look for 211111, and `[0-9]{2}` matches any two digit number. There is a list of common regular expressions in Tables 15.1 and 15.2.

In Python, you can use regex with the `re` library. However, regex is widely used in many programming languages, not only in Python. Knowing regex is an incredibly useful skill for anyone who needs to analyze data with a computer.

Regex	Description	Example
\w	Any word character (letter, number, or subscript)	`an\w` matches `any` and `and`, but not `an`,
\W	Anything that isn't a word character	`friend\W` matches `friend.` and `friend`, but not `friends`
\s	Any white space (spaces, tabs, etc)	
\S	Anything but white space	
\n	Newline (line break)	
.	Any character (wildcard)	
\	Escape character. Let's you use special characters as the actual character.	. is a wildcard. \. is a period.
[ ]	Anything listed in the brackets.	`[ABC]` matches A, B, or C.
[^]	Anything not listed in the brackets	`[^0-9]` matches anything but the digits 0 through 9
^	beginning of a line	`^A` only matches A when it is the first character in a line of text.
$	end of a line	`A$` only matches A when it is the last character in a line of text.

Table 15.1 Common regular expressions that represent characters.

Regex	Description	Example
+	one or more occurrences	`AB+` matches `AB`, `ABB`, and `ABBBBBB`
*	zero or more occurrences	`AB*` matches `A`, `AB`, and `ABBBBBB`
?	zero or one occurrence	`Hell?o` matches `Hello` and `Helo`
{n}	*n* occurrences	`[0-9]{3}` matches any three digits
{n,}	*n* or more occurrences	`[0-9]{3,}` matches `121` and `17572`
{n,m}	*n* to *m* occurrences	`[0-9]{3,4}` matches `121` and `1984` but not `17572`
()	Marks a pattern group	`A(BC){3}` matches `ABCBCBC`
|	"or". Must be used inside ()	`A(BC|DE)` matches `ABC` and `ADE`

Table 15.2 Common regular expression quantifiers. These quantifiers apply to the proceeding character or group.

Index