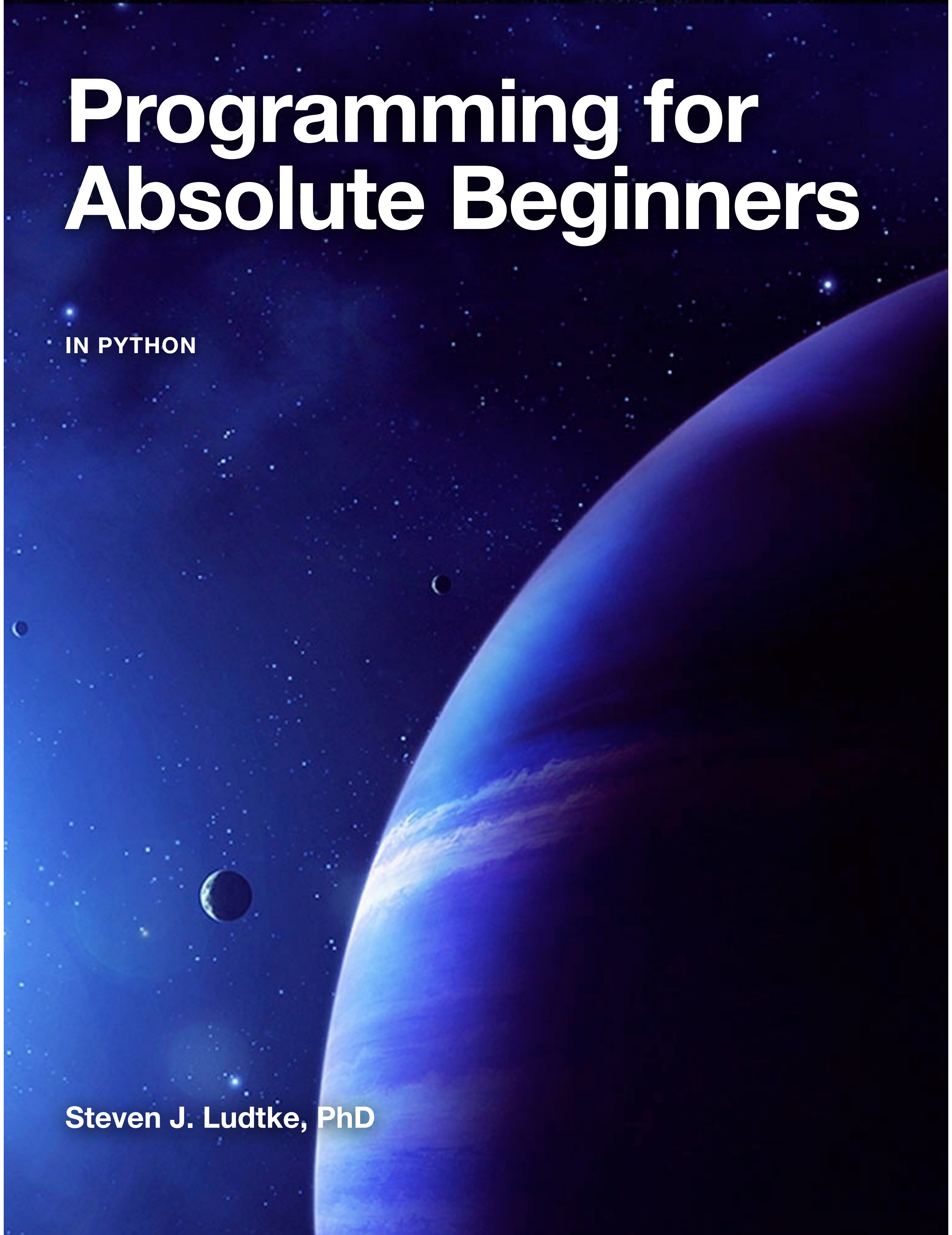
Programming for Absolute Beginners

IN PYTHON



Steven J. Ludtke, PhD

Preface

Preface

I first learned to program when I was about 10 years old, when my father purchased our first computer, a TRS-80 Model I. Most of you have probably never heard of this computer, as the year was 1978, and this was one of the first ‘personal computers’ to come to market. There were no home video games yet, and such devices were quite rare. In those days, there were no hard drives, and even floppy drives weren’t available yet. The sole mechanism for storing programs was on an audio cassette tape. The computer had the ‘BASIC’ programming language built in to it, but otherwise came with no software at all. If you wanted to play a game, you had to type in a program from a magazine first. Needless to say, such games were primitive even by the standards of what you might find on a cell-phone ten years ago. Nonetheless, it was cutting edge at the time.

My father made a deal with me. I could play games on the computer, but only if I went through the ‘learn to program’ book that came with the computer. This book was really quite amazing even by modern standards. It had little cartoons, and took you step by step through the art of programming in BASIC. I still remember bits of it, even 30 years later, and once I started learning how to get these fascinating machines to do something, I was hooked. After all, you really couldn’t do much interesting on the machines without knowing how to program.

Of course, The world has changed. Nowadays, you can buy or download a program for just about any task you can imagine. Some of these programs even include little programming languages of their own.

So, why learn to program at all ? Seriously, why ?

The same question could be asked of any hobby. For example, carpentry. You can go out and buy virtually any piece of furniture you could ever need, generally for less than it would cost to make yourself. If you learn how to do it yourself, you're doing it for fun, for the pleasure of creating something useful with your own hands. If you're good at it, you could turn it into a career, but fundamentally, you do it because you enjoy it.

So, before you start the journey to learn how to program, put yourself in the right mindset. Very likely you can find an existing program to do just about anything you need done. Sure, knowing how to program will give you a bit more flexibility in how you use programs, but for most people, the time you save will be less than the time you invest. You should learn how to program because you're honestly interested in how programming works. This should be a fun process, not a stressful one. While I'm no artist, and have no hope of producing something with quite the same spirit as the book I remember from my childhood, hopefully I've achieved something with a somewhat lighter spirit than your typical programming tome. The focus is on doing fun and interesting things rather than on learning the sort of formal programming you'd do as a professional. Certainly you can take what you learn in this book and turn it into something more formal later on if you find yourself inspired, but that's an afterthought, not a goal.

This book will introduce you to programming through a language called 'Python'. While languages such as C++ and Java may be more widespread among professional programers, Python is also quite widespread, and it is **far** easier to learn as a beginner. Knowing quite a few programming languages myself, I can also say without reservation that Python is the by far the most fun to program and work with.   So, relax and enjoy your journey !

Python ?

Why name a programming language after a big snake, you may ask ? The answer is, it isn't ! In fact, Python is named after another Python of British TV fame (try Wikipedia if you can't figure it out). So, when you run across odd references in the Python documentation, you'll know where they come from. This also makes it a doubly good language to use for this particular book.

NOBODY expects the Spanish Inquisition!

Chapter 1

Getting Ready to Program

Python is, itself, a program you run on your computer, which interprets the programs you enter. This chapter takes you through the process of setting your computer up to use Python, and some initial examples to show you what Python can do.

Section 1

How This Book Works

SUMMARY

1. ***`Text to type`***
2. *`Text you see on the screen`*
3. <key> to press
4. Example code at
5. Chapter ? contains a review of concepts

Conventions

I tried to write this book so it would be understandable to someone learning programing for the very first time. If you already know a little bit of programming from somewhere, you could also use this book to learn Python, though you may find some of it a little simplistic. To add a little spice, you will find occasional boxes that look like this: box, with notes for more advanced readers. If you're a beginner, you can ignore these.

There are a few basic conventions we use in this book that it's important to be familiar with, so you find the process fun, not frustrating. Most people will find these things pretty obvious, but let's just make sure everyone's on the same page. First, if you see ***`text that looks like this`***, you are supposed to (or can if you like) type exactly what you see into the computer. Generally you should press <enter> at the end of each line. When you see something between <> characters, this represents a key you are supposed to press, such as <p>, <space> or <enter>. Please note also that <enter> is the same as <return>. Different keyboards give it different names. Sequences like <ctrl-c> mean you should hold down the 'ctrl' key and press c. Text *`like this`* is text you should expect to see displayed on the screen.

The first chapter will explain how to install and run Python, and then give you a series of little examples just to demonstrate some of Python's capabilities, and to give you a flavor of what's to come. Later in the book, we will take apart some of these examples and see how they do what they do.

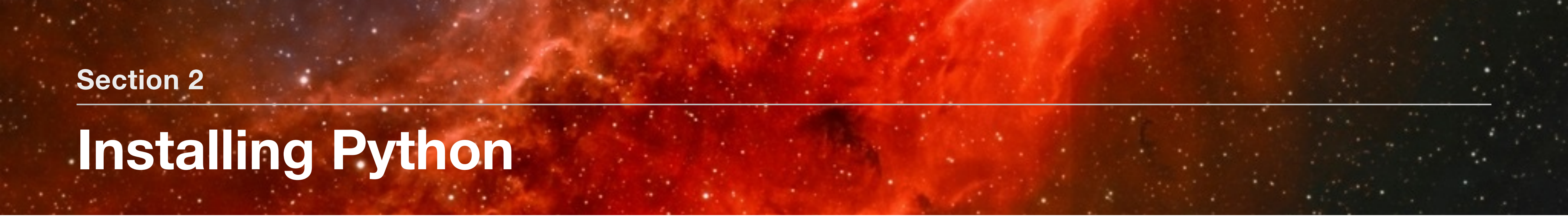
Too much Typing

This book has quite a lot of code (programs) that you need to type in. You will get the most out of the book if you actually go through the examples and type them in when you see them. In fact, don't be shy, go ahead and play around. Change the examples, and see what happens. It's pretty hard to do any serious harm with anything we're doing in this book. If we're playing with anything really risky, like deleting files from the hard drive, I'll warn you in advance. For the most part, however, if you find that you are playing around and things become completely messed up, you can just exit Python and start it up again from scratch.

If you're feeling lazy (or type really slowly), there is one alternative. Most of the significant chunks of code in this book are all assembled into one neat package at

Concepts

This book introduces new concepts gradually as we need them in each of the various fun little projects. The fun projects are presented in order such that we can introduce new concepts a steady, but hopefully not overwhelming, pace throughout the book. However, that means if you wanted to use this book as a reference, it could be challenging to remember exactly where a new idea was introduced. For this reason, we have Chapter ?, which contains a review of the concepts in the book, complete with additional examples. If you find a concept too difficult when it's first introduced, check this chapter for more examples that may help clear up your confusion. This chapter will also serve as a good reference after you finish the book. Also note that there are plenty of good resources on the web for learning about specific concepts if you still find something hard to grasp.



Section 2

Installing Python

SUMMARY

1. Python installation is platform-dependent (Linux, Mac, Windows).
2. This book will use Python 3.x.
3. We also install some useful Python libraries which aren't part of the standard distribution.
4. Wherever text appears in `this font`, this is an indication of something you are expected to type into your computer.

The Good

Python is, quite simply, a fun language to use. Whereas many programming languages force you to do any specific task one specific way, and make you carefully define every aspect of your program before you can actually do anything, Python is very relaxed, and free-form. For any given task, it is generally possible to come up with a half-dozen different ways to accomplish it in Python. While it permits you to be very rigid in your software design, it also gives you the freedom to simply play around (which is what we will spend most of this book doing).

Python is widespread enough that it is included as a standard part of most (but not all) modern operating systems. The specific version of Python you will have will vary with how old your OS is. When I started writing this book, Python 2.x was still dominant, but now we are beginning to see Python 3.x installed most (but not all) places. While the two versions are almost compatible, there are some important differences, so it's important to make sure you have Python 3.x installed.

While I may make some minor comments when there are differences between the two, we won't cover everything.

The Bad

Python is what is known as an interpreted programming language. When you write a program in a language like C++ or Fortran, your program is first passed through a **compiler**, to produce an **executable**. This executable runs directly on the **CPU** of the computer, and is very fast. In Python, your program runs immediately, without a compiler. However, since it hasn’t been compiled, it may run a LOT slower than the same program written in a language like C++. On the bright side, Many of the libraries of functions provided as a standard part of Python ARE compiled, and run at full speed. So, Python is often used as a **scripting language** to make other, faster programs and libraries do precisely what you want them to.

The Ugly

Many professional programmers, particularly those trained in formal C++ and Java programming, dislike Python’s free-form style. They claim it encourages bad programming habits. To some extent, they are correct. If you were writing the software system for a Bank, where everything had to work exactly according to specific rules, and 50 programmers all had to work together to produce one gigantic piece of code, you, too, might be fond of rigid rules and guidelines. In such situations, if one guy decides to do things ‘their own way’, the next thing you know, someone’s bank account has accumulated an extra $1m due to a programming 'error'. Python can, and has, been used for very large projects, but it really shines in situations where a lone programmer or a small group is trying to get something done quickly.

The Makers

Since 2011 or so you may have noticed the term "maker" occurring all over in popular media. Makers are people who like to build their own gadgets, often computerized. One of the most popular tools for doing this now is the Raspberry Pi, which is basically a fully functional Linux computer on a small ciruit-board which costs well under $50. These little computers are surprisingly powerful, and surprise, are often programmed in Python, which comes pre-installed with all of the Linux distributions on the Pi !

Installing Python

Mac Users

Luckily and unluckily for users of Macintosh computers, Python comes preinstalled with the operating system. Unfortunately, the Python on the Mac is still (2016) Python 2.x. If you open a **terminal window**, and type

```
python --version
```

you will see what you have. Alas this means you will need to install Python 3 from somewhere. On the bright side, Python2 and 3 are installed independently, and it is perfectly fine to have both installed on a single machine. Normally Python 3.x is executed with the command "python3". The simplest way to get a reliable version of Python3.x is from www.python.org. Simply click on the Download link and find the most recent Python3.x link. This will be downloaded as an installer you can just double-click on to set up.

If you think you will be doing a lot with Python and want to be able to easily set up hundreds of different Python libraries, you might want to try a "Superpackage", like Anaconda: https://www.continuum.io/downloads. If you do this, don't be confused by references to "Anaconda 4.x" or somesuch. The anaconda versions have nothing to do with Python versions. Indeed, you will find two downloads available for Anaconda x.x, one for Python2.x and one for Python3.x. Clearly you want to get the second one. This package is not much harder to install and gives some really impressive capabilities, even if you won't use most of them until you are well past the end of this book.

To make Anaconda actually work, you will need to make some changes to your environment so the system Python isn't still used. On my machines I do this like this:

```
unset PYTHONPATH
export PATH=/anaconda/bin:$PATH
```

Windows Users

Python will not be preinstalled on windows, however, a python installer for virtually any version of Windows is available from www.python.org. Simply download the appropriate (Python 3.x) installer and run it. On Windows, you can launch python in two different ways, which we'll talk about later.

You may also consider installing the more capable Anaconda "superpackage", which includes Python3 and many standard libraries, and makes hundreds of others easy to install:

https://www.continuum.io/downloads.

Linux Users

As of the time of writing (2017) Linux Python has gotten a bit confusing. It used to be that `python` ran Python2.x and `python3` ran Python 3.x, but a few distributions have changed things up and you have to type `python2` to get Python 2.x and `python` will get you Python 3.x. Sometimes only one of the two will be installed by default. Uggh! The solution? Type:

python --version

If that returns a 2.x number, or if it returns an error type:

python3 --version

if one of these has led you to an already installed Python3.x version, make a note of which command to use, and use it throughout the book. If not, you will need to use the package manager for your specific version of Linux and install Python3. I can't give you specific advice at this point, as there are many different flavors of Linux, and several different tools they use for managing installed software. Regardless, all are easy to use, and if you are using Linux in the first place, you can probably handle this.

Many Python libraries and tools are already easy to install on Linux, but like the other platforms, if you install the Anaconda "superpackage", you will gain easy access to hundreds of different Python libraries, which may be useful as you become more advanced:

https://www.continuum.io/downloads

IPython

In addition to the standard interactive mode that comes with Python, there is an open source project called IPython, which gives you an interactive mode with some additional capabilities. If you install Anaconda, you will already have this tool. If not, and you are feeling adventurous, you may consider downloading and installing IPython in addition to the normal python interpreter, and using ***`ipython`*** rather than ***`python`***, when prompted. Some of the capabilites of IPython will be discussed later in the book, but if you go this route, for the most part you'll need to read the manual to sort out how to use some of its advanced features. If you get frustrated easily, and are new to programming, you may want to hold off on trying IPython for the moment.

iPad Users

Originally Apple didn't permit programming languages on its iDevices, however sometime in 2011 they reversed this policy, and you can now purchase at least one version of Python for the iPad/iPhone. However, it may be challenging to continuously switch back and forth between this book and your iPad, and a few things we'll do will not be possible on the iPad due to its security restrictions, so my advice would be to read this book on your iPad and practice programming on the computer, but it's completely up to you. It is also worth noting that, at the time of this writing, turtle graphics is one of the thing the Python interpreter on the iPad doesn't support, and we use this for a number of the more entertaining early examples, so this may not be an optimal choice.

Section 3

Taking Python Out for a Spin

SUMMARY

1. **You can start Python by typing `python` (or `python3`) at the command-prompt.**
2. **Python can be used to do basic math like a calculator, for example `2*5+10`. If you need scientific functions, like sqrt() or cos(), first you have to type: `from math import *`**
3. **A string can be created by surrounding text with double quotes, such as : `"a test"`. You can also perform addition and multiplication with strings.**
4. **Python has built-in Turtle graphics, which can be used to do simple drawing operations. This emulates a real Turtle robot drawing with a pen.**

Starting Python

There are two fundamentally different ways you can use an **interpreted language** like Python. First, you can use a text editor to create a file containing your program, then you can run the program just like you do any other application on your computer. Alternatively, you can run Python in **interactive mode**, and just type commands into it one after another. It will immediately respond to each command. We will make use of both methods in this book. However, we will begin with the interactive mode, and use this for many of the simple exercises in the book.

On any of the three computer platforms we cover, Python can be run by opening a command prompt, and typing **`python`**. While you can start it using an Icon on most platforms as well, there are some reasons not to do it that way just yet.

So, go ahead and give it a try. Once you enter **`python`**, you should receive a prompt, looking something like:

```
odd% python
Python 3.5.1 |Anaconda 2.4.1 (x86_64)| (default, Dec  7 2015)
[GCC 4.2.1 (Apple Inc. build 5577)] on darwin
Type "help", "copyright", "credits" or "license" for more in-
formation.
>>>
```

>>> is the Python prompt. If you opted to use IPython instead of Python, you will see a prompt like *In [1]:* instead, but it has exactly the same meaning. Either way, this is the place where you type all of the nifty Python commands we'll be learning in the interactive exercises.

Your First Python Commands

Next chapter we will start learning Python properly, but let's get started with a few quick examples showing you some easy things you can do.

Python as a Calculator

This is where almost any introduction to Python starts, mainly because it's easy, and can be useful. At the prompt, type: ***1+1*** and press <enter>.

If everything is working as it should, you should see '2' followed by another prompt. Cool, huh ? Ok, ok, perhaps that was a little simple. How about something a little more complicated. Try this:

```
for i in range(10):
    print(i,i*i,i*i*i)
```

This one is a little trickier. Note that the second line is indented. This indentation is critical to python, as code that is indented the same amount will be executed together (we'll discuss this more in the next chapter). For now, just make sure you either use one or more spaces or a <tab> character to indent the second line.

You'll also note that after you enter the first line, the prompt will change from >>> to **...** . This means Python is waiting for you to complete a command you didn't finish on the first line. After you enter the command on the second line, you will see another **...** prompt. At this prompt you will need to press <enter> again on an empty line to let Python know you don't have any more commands to give.

Whew, quite a long explanation for 2 lines of code, huh ? Don't worry, things will get easier once you learn a few of these simple rules. If you typed everything correctly, you should have seen:

```
0 0 0
1 1 1
2 4 8
3 9 27
4 16 64
5 25 125
6 36 216
7 49 343
8 64 512
9 81 729
```

That is, the numbers from 0 - 9 and each number squared and cubed. Still not rocket science, but easier than doing the same thing with your pocket calculator.

Note that if you were using Python2.x, this program would be exactly the same, except the ( ) would be removed from the print statement. This is one of the more irritating changes between Python2 and Python3.

Just to get it out of the way, let's list the basic mathematical **operators** in python:

a+b	add	a/b	divide
a*b	multiply	a//b	integer divide
a-b	subtract	a%b	remainder
a**b	a to the b power	pow(a,b)	a to the b power

Those take care of all of the basic math you'll need to do. We'll get to more complicated math later on.

Floating Downstream

There are a few oddities when you first start programming, which may seem a bit odd. Enter the following: ***`5/2, 5//2, 5%2`***

You should see:

```
(2.5, 2, 1)
```

The "," operator combines several different math expressions into one, so you get 3 results on 1 line. There are 2 different kinds of division in Python, floating point division, '`/`' and integer division, '`//`'. When you do integer division, you also need the ability to find the remainder, hence '`%`'.

It is important to understand that Python, and computers in general, treat integers and decimal values in fundamentally different ways. We'll get into this more later in the book.

Fractions

Many people grow up hating fractions, and for many parents, their kids confronting them with homework involving fractions is their worst nightmare. Now, while this isn't true for your typical programmer, it's nice to know that Python has your back. Give this a try:

```
from fractions import Fraction
Fraction(3,45)
Fraction(2,9)*Fraction(3,4)+Fraction(1,2)
```

You'll note that it automatically simplifies the resulting fractions for you.

Mathematical Functions

Now lets try something a little more complex. Let's say we want the square root of a number. Try typing ***`sqrt(5.0)`***.

Aack! You probably got an error saying it doesn't know what sqrt means. What's up ? Did I misspell sqrt ? No, this is simply our first introduction to **modules** in Python. All of the math functions live in a module called 'math'. So, let's try one more time. Do this:

```
from math import *
sqrt(5.0)
cos(.25)
```

After typing that first line, you suddenly have access to mathematical functions of all sorts. What functions ? Quite a few to choose from. Try this:

```
import math
help(math)
```

This will give you help on the entire math module. Note that after you see the first page of math functions, you get a ***:*** prompt instead of the typical >>>. This prompt tells you that python has more than one page of stuff to show you. Press <space> to see the next page, and when you get tired, press <q>.

Strings

Strings ? Musical instruments ? Subatomic particles ? No, in programming, a **string** is a sequence of characters (letters, punctuation, numbers). While math is undeniably important, most of what we do with computers involves words too, so let's see a few simple things we can do with strings.

First, consider a simple string, letters between double quotes:

```
s="This is a string"
print(s)
```

If you enter this, you will create a variable, ***s***, and print it (the string) on the screen. Great ! We have "This is a string" in a variable. Other than print it out, what can we do with it ?

```
print(len(s))
```

Ah ha ! The length of the string ! Useful for some things, but not earth shattering.

```
print(s.count("i"))
```

We can find out how many "i"s there are in the string. Why ? Well, why not.

```
print("".join(sorted(s)))
```

That one is a just a bit less obvious. It sorts the letters in the string in alphabetical order. Probably not the most useful example, but it gives you a feel for some of the possibilities. Later, we'll have a whole chapter on programming for word games. How about math with strings ?

```
"2.0"+"3.0"
```

You should try this one yourself. If you haven't done much programming before, you might guess that this would produce *"5.0"*, when in reality, it produces *"2.03.0"*. So, strings can be added, but with strings this means they should be joined together (concatenated), not added mathematically. Strings can be multiplied as well:

```
"2"*4
```

This produces the same thing `"2"+"2"+"2"+"2"` would, that is `"2222"`.

Of course, there are many more interesting things we can do with strings, but that will have to wait until the next chapter. For now you just need to understand that they are fundamentally different than numbers, even if they seem to contain a number.

Turtles

Before the days of high resolution color monitors, there were simple black and white display, capable of showing only letters and numbers. No graphics at all.

Before this, were the TTYs: Imagine a typewriter connected to a computer. As you typed, each letter was printed on the paper, but was also sent to the computer. The computer's responses were then ghost-typed on the same paper. In these days, the idea of "graphics" on a computer was whatever you could do with letters and numbers on a printed piece of paper. Not only was there no color, but there weren't even lines, aside from the -, |, / and \ characters.

Ok, why am I boring you with history ? Back in the 50's, early robots known as 'turtles' were developed. In the late 70's/early 80's, these turtles were adapted to be educational tools as graphics devices for the computer. The turtle had a pen which could be moved up and down, and the robot could be given simple commands like move forward, turn 10 degrees left, etc. This could be used to draw pictures on paper. While the actual turtle robots aren't very common any more (though you can still get/make them), the concept is still alive and well. Most of the time the turtles are now little triangles that move around on your computer screen, trailing a line behind them if the "pen" is down.

Now that our history lesson is over, let's try it:

```
from turtle import *
s=Screen()
goto(0,0)
```

Look ! A window appeared, and when you typed goto(0,0), a little arrowhead (the turtle) appeared in the middle of the window. What fun ! What next ?

```
for i in range(36):
    forward(10)
    left(10)
```

Look, a circle ! (we have to start somewhere, don't we ?) How about something with a little more flair (Example 1.1):

```
reset()
goto(-125,-125)
clear()
for i in range(61):
    forward(250)
    left(118)
```

Ever play with a Spirograph before ? You can do interesting things with turtles without much effort. Note that the turtles are intentionally slowed down so they act more like a turtle robot might. Clearly the computer is capable of drawing a lot faster than this.

Example 1.1 What you should see

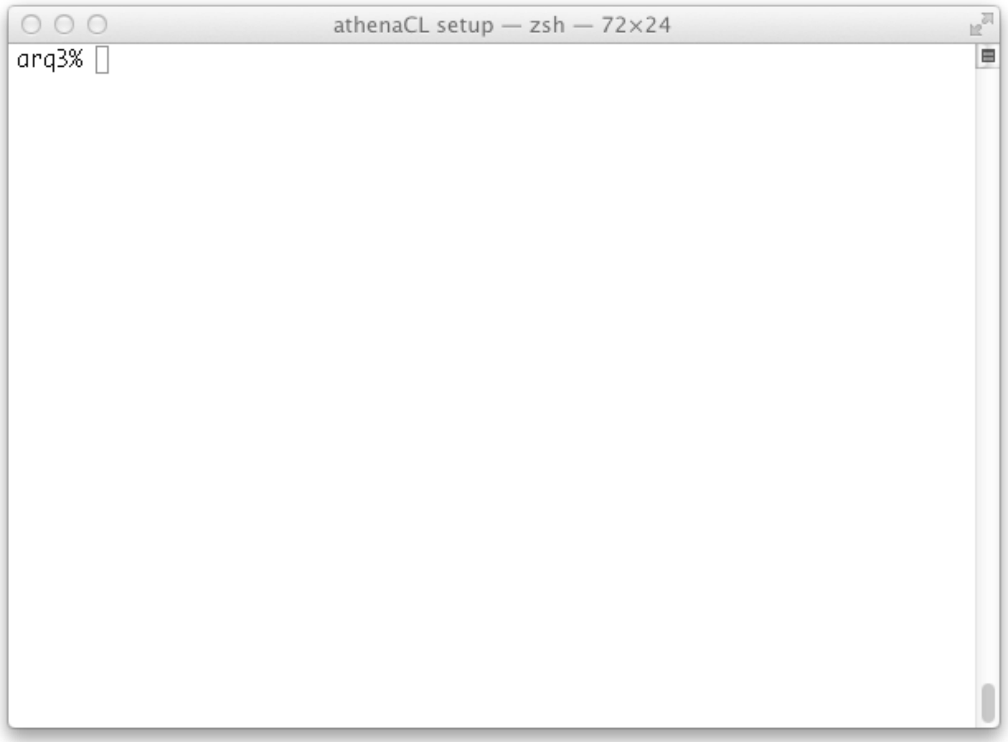
Section 4

Problems

While you haven't really been 'taught' anything yet, if you're clever, you may be able to figure out the examples you've seen enough to try your hand at a few simple problems. Of course, the answers are provided as well.

Problem 1 - Print the integers from 0 to 10 and the square root of each.

Solution 1.1



```
arq3% 
```

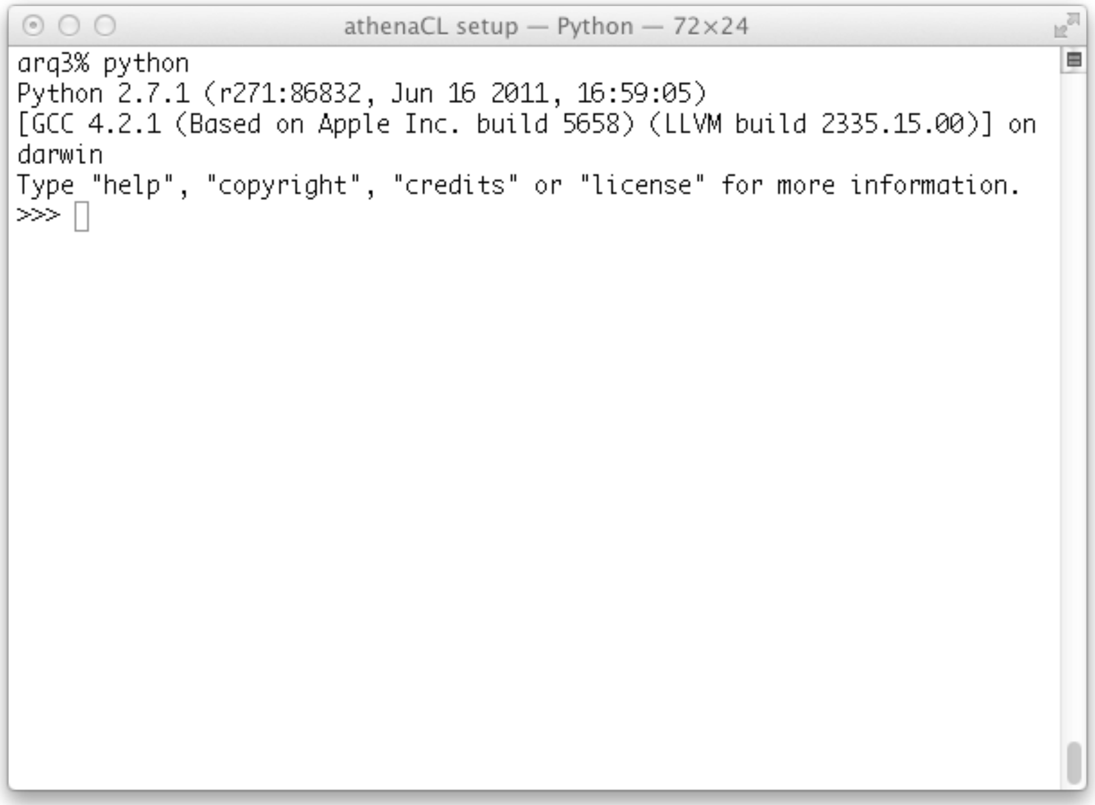
Scroll through the images to see the solution.

Problem 2 - Modify the turtle examples and see if you can draw:

a) A hexagon

b) The spirograph example is loosely based on triangles, modify it so its based on squares instead.

Solution 1.2



```
arq3% python
Python 2.7.1 (r271:86832, Jun 16 2011, 16:59:05)
[GCC 4.2.1 (Based on Apple Inc. build 5658) (LLVM build 2335.15.00)] on
darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> 
```

Again, start by running ***`python`***.

Chapter 2

Turtlerific

Turtles have the most potential for doing something interesting quickly, so we'll take a couple of turtle examples apart to see what we can learn from them.

Section 1

Spirograph Example

SUMMARY

1. **Python has many built in modules, including math and the turtle graphics module we have already used. To use a module you must either *`import module`* or *`from module import *`*.**
2. **Python has a built-in help function, which can be used to get documentation for modules or functions, such as *`help(math)`*.**
3. **There are over 20 different commands you can give the turtle. The most useful of these are summarized.**
4. **Lists can be created using square brackets and commas, such as: *`[1,2,3,4]`*.**
5. **The *`for`* loop allows us to repeat an operation for each element in a list.**

Let's start with our spirograph example from the last chapter:

```
from turtle import *
s=Screen()
reset()
goto(-125,-125)
clear()
for i in range(61):
    forward(250)
    left(118)
```

This example shouldn't be too difficult to figure out. Let's start with the turtle graphics functions: *Screen(), reset(), goto(), clear(), forward()* and *left()*. These functions wouldn't be available except for the first line:

```
from turtle import *
```

Import

Python comes with a wide range of standard libraries to do all sorts of useful and interesting things. We've seen two of these libraries in the examples in the first chapter: math and turtle. While these libraries are distributed with the Python language, to use them, you still have to let Python know that you want to use each one in any given session.

The import command is how this is done. There are three different ways this command can be used. It's good to understand all three, since you will run across all three at one point or another. The three methods are (don't actually type these):

```
import module
from module import *
from module import name1, name2
```

The first form makes **module** available, the second form makes all of the functions from **module** available without having to type `module` before them, and the third form makes only the specific functions **name1** and **name2** available.

Let's use the *math* module as an example. Say we need to use the *cos()* function. We could do this three different ways:

```
import math
math.cos(0.5)

from math import cos
cos(0.5)

from math import *
cos(0.5)
tan(0.5)
```

In the first example, we get access to the entire math module, but we have to put `math.` in front of any function we use. The second form gives us access to just the *cos* function, but we can omit the `math.` The third form gives us access to the entire *math* module, and we don't have to use `math.` for any of the functions.

At first glance it would seem like the 3rd form is always going to be the best choice. After all, it saves you from having to type an awful lot of `math.`'s everywhere, and it gives you access to all functions without having to name them all individually. Why then would we not want to do this ?

The problem arises when the same function exists in more than one module. *math* is a particularly good example for this, because there is another module called *cmath*, which has a virtually identical set of functions. Why would this be so ? The *cmath* module contains routines for **complex math**. The difference isn't terribly important here, and could be rather confusing for people who haven't taken a lot of advanced math. Suffice it to say that in the *math* module `sqrt(-1)` returns an error, and in the *cmath* module it doesn't.

So, if we do this:

```
from math import *
from cmath import *
print sqrt(-1.0)
```

What happens ? Do we get an error, or do we get an answer ? You can try it, but you'll find that indeed, no error is raised. When we import *cmath*, it overwrites all of the functions from *math*.

One solution to this problem is to use the other form of import:

```
import math
import cmath
print cmath.sqrt(-1)
print math.sqrt(-1)
```

You'll find that the second print statement now raises an error.

So, when we do a `from turtle import *`, we are making the entire set of turtle graphics functions available for use. Note that it is perfectly acceptable to do both `import turtle` and `from turtle import *` within the same session. While this may seem a bit odd, there are several reasons why it might make sense to do this, particularly in an interactive Python session.

Help

One of python's most useful features, particularly when you're starting out, is its built-in help system. Of course we didn't actually use this when we did the examples last chapter, but now is a good time to introduce it. Try this:

```
import turtle
help(turtle.goto)
```

This will give you fairly detailed help on whatever function you call it on. Even more useful, though perhaps a bit overwhelming, is:

```
help(turtle)
```

which will give you help on the entire module, including all of the functions it contains. While I personally much prefer the well formatted documentation available from www.python.org, `help()` can still be very useful when you forget how to use a function. Note, however, there is no absolute requirement for module developers to write the sort of detailed help you find in the *turtle* module, but you'll find that the help for most Python libraries is quite good.

Back to Turtles

While the names are pretty self-explanatory, let's consider each of the turtle functions we used in our example:

- Screen() - creates a turtle graphics window and opens it
- reset() - resets the turtle to the origin (0,0), default orientation, ...
- goto(x,y) - moves the turtle to an absolute location (x,y)
- clear() - erases the current display, but doesn't move the turtle
- forward(d) - moves forward in the direction the turtle is pointing by *d* units (generally pixels)

- left(a) - turns the turtle to the left by *a* degrees

As you can see, these are pretty simple commands. Picture the turtle as sitting on a piece of graph paper. Let's say the center of the paper is at (X,Y) coordinates (0,0). Just like graphing functions in grade-school, moving to the right corresponds to an increase in X, and moving up corresponds to an increase in Y. We can move the turtle in two fundamentally different ways: we can use goto() to move to a specific location on the paper, optionally leaving a line in its wake, or we can use commands like left(), right(), forward() to move relative to the turtle's current location.

Robot Turtles

If the turtle were a physical robot with a pen, then obviously, saying something like goto(10,10) would require Python to know where the turtle currently was, then convert the goto(10,10) into relative move commands, because all the turtle can do in reality is move one of its 3 motors: one to move forward/backwards, one to change direction, and one to raise/lower the pen. All of these subtleties would be handled by Python for you, and give you the flexibility to control the turtle in many different ways.

So, while we're discussing turtle graphics, let's look at a list of the most important turtle commands. Note that many commands have one or more equivalent abbreviations:

Function & aliases	Parameters	Description
left lt	angle - default degrees	Turn the turtle left by a specified amount
right rt	angle - default degrees	Turn the turtle right by a specified amount
forward fd	distance - how far to move	Move forward by the specified amount
back backward bk	distance - how far to move	Move backwards by the specified amount
goto	x,y	move in a straight line to position (x,y)
seth	angle - default degrees	Set the direction the turtle is pointing
penup up pu		Start drawing when moving
pendown down pd		Stop drawing when moving
pencolor	name - "red", "green", ... (r,g,b) - 0-255 for each	Color of the 'pen'
fillcolor	name - "red", "green", ... (r,g,b) - 0-255 for each	When filling, use this color instead

Function & aliases	Parameters	Description
width	pensize - a positive number	Width of the pen
reset		Clears the screen, and resets the turtle to starting parameters.
clear		Clear the current page, but leave the turtle alone
dot	[size] - dot diameter [color] - see pencolor above	Draw a dot
circle	radius - size of circle extent - how much of the circle (default degrees)	Draw a circle. Center is radius to the left of the turtle.
speed	speed - 1 (slow) - 10 (fast) 0 (fastest)	How fast the turtle should move.

This list is not exhaustive, and there are a couple of categories of functions we haven't covered, but we'll get to those later. As you can see, even with this list, there are quite a few things we can have our turtle do.

However, before we get to that, we need to have a look at the one line of our example program we haven't explained yet. Amazingly enough, that one line involves no fewer than four critical concepts in the Python language.

What Lives in One line

The line we need to discuss is:

```
for i in range(61):
```

The first concept we will discuss is embodied in `i`, the second in `range(61)`, the third in `for ... in`, and finally, the last concept is embodied in the humble `:` .

Variables

If you've programmed before, please skip this particular subsection. For everyone else reading this book, think back to grade school, or maybe Jr. High, and remember from math class the simple concept of a variable, like *x* in *5=3+2x*. You may think I'm being condescending here, but I'm not. While variables in programming are similar to variables in math, they aren't exactly the same thing, and this often leads beginners to some confusion. This statement is perfectly valid in Python:

```
x=10*20+30
```

However, this is not:

```
5=3+2*x
```

And this one is valid, but it doesn't mean what you may think:

```
x=5*y
```

What the heck ?

In math, when you say something like *x=5*y*, you are establishing a relationship between the variables *x* and *y*. You are saying that *x* is 5 times the value of *y*, and correspondingly, *y* has a value 1/5 of *x*. In programming when you make this statement, you are saying “give x the current value of y times 5”. If y changes, x does not change. For example, after this:

```
y=3
x=y*5
y=4
```

the value of *x* is 15, NOT 20, and the value of *y* is 4. That is, in programming when you say `x=...`, you are not saying “`x` is equal to” you are saying “make `x` equal to the current value of ..., right now”. This is a somewhat subtle point, but a very important one.

Another less subtle point is that variables in programming aren’t limited to holding numbers. For example:

```
x=”abc”+”def”
```

Is perfectly acceptable, and `print(x)` will produce `abcdef`. Variables can contain many other things as well, but so far we’ve only talked about numbers and strings. However, that leads us directly to the second important concept in that line of code :

Lists

Try this:

```
print(list(range(5)))
```

You should see `[0,1,2,3,4]`. This is a **list** of integers. A list is a single object, which contains an ordered group of other objects. Let’s try this example:

```
a=[0,1,2,3,5,7,9]
print(a[0])
print(a[2])
print(a[5])
```

You should have gotten `0`, `2` and `7` back out. It should be obvious by this point in time that the print function is used to display the results of an expression. In an interactive python session, you could have omitted the print statement, and it would have shown you the corresponding values anyway, though in a slightly different form (try it). Regardless, the key here is that a is a list of numbers, and we are able to extract specific elements from the list using `[]`.

This may be a little confusing at first, after all, you created the list by putting a bunch of comma-separated items inside square brackets. Surely that would mean that we could have said `a=[2]`

and made a list with a single element, and indeed, this is true. The trick is in the ***=***. The statement ***a=[2]*** is assigning the one element list containing the number 2 to the variable *a*. The expression ***a[2]***, however, is retrieving the third element from the existing list a (if the list has at least 3 elements).

Now, at this moment, if you've absorbed the whole '=' thing, you're probably saying, "Hang on a second. You said a[2], but then you said the third element of the list !?!? Must be a typo in the book !" Alas, no. It is not a typographical error. Python, like most programming languages, uses zero-indexed lists. The first element in the list is ***a[0]***, the second element is ***a[1]***, ... Don't worry too much about why this is true for the time being, but there are some good reasons to do things this way. This also explains why, when we said ***range(5)***, we got a list from 0 through 4, not 1 through 5. Almost everything in Python is zero indexed.

Since we're talking about lists anyway, now is probably a good time to introduce a couple of other interesting features about lists: negative indices and slicing. What do you think this would produce (feel free to try it) ?

```
a=[0,1,2,3,5,7,9]
print(a[-1])
print(a[-2])
```

The answer ? ***9*** and ***7***, of course ! Yes, that's right, if you use negative indices to access elements in your list, you start at the end and count backwards, so, a[-1] refers to the last element in the list, however long it happens to be, and a[-2] is the second from last element. Neat trick, right ?

Now I have a really tricky one for you: what are ***a[7]*** and ***a[-7]*** ? If you try this, you will find yourself with a nasty looking error message *"list index out of range"*. No funny tricks here. If you ask for an element of a list that doesn't exist, you will get yelled at.

We're ready to move on to slicing now. Let's say you want a new list which contains elements 2-4 of the old list. We could do it like this: ***b=[a[1],a[2],a[3]]***, but that might get a little unpleasant if we wanted elements 197 to 536 from a larger list. Happily python offers us a bunch of interesting shortcuts using the slicing operator. We could equivalently say ***b=a[1:4]***. Why 4 ? Is Guido van Rossum (the author of Python) just being perverse ? No, again, there are some really good reasons for it. For now, just realize that the first index when slicing is inclusive, and the last index is exclusive. That is, a[1:4] says start with element 1 (the second element) and give me all of the elements up to, but not including element 4 (the fifth element). Trust me, it will take a little getting used to, but in the end everything will fall into place.

There are a number of other clever things we can do with slicing. For example, we can omit either the first or last index in a slice, implying the beginning or end of the list, respectively. Here are some examples:

```
a[:4]       # returns list elements 0 through 3
a[2:]       # returns elements 2 through the end
a[-3:]      # returns the last 3 elements
a[:-3]      # returns all but the last 3 elements
```

Clear ? Hang on, I slipped something in there. The lines up above are valid Python code, even the bit after the `#`. The `#` character in Python begins a **comment**. That is, anything after this character on any line of Python code will be completely ignored. If you typed a line of code that said:

```
# a=[1,2,3]
```

it would do absolutely nothing. This is used to document your code. That is, to explain to others, or yourself 3 years later, what exactly you intended that bit of code to accomplish. It isn't very useful when we're using Python in interactive mode, but if we were writing a program in a text file, it is considered very good form to add comments liberally throughout the code.

The last thing to introduce in dealing with lists is assignment. You can change the contents of a list:

```
a=[1,2,3,4,5]
a[2]=10
print(a)
```

As you see, the third element of the list has been changed to 10. There are many other ways of manipulating lists, which we will cover later, but this should be sufficient for now.

for ... in

On to the third important component of our one line of code. The ***for*** statement is used to **iterate** over the elements of a list. This is called a **loop**. We use it by saying:

```
for variable in list: something
```

variable is the name of any python variable, and list is any python list (or a variable containing a list, which is the same thing). This statement will assign each element in the list to the variable, and then do something before moving on to the next item.

Let's try a simple example to demonstrate this:

```
for i in [1,2,4,6,9]: print(i)
```

You'll need to press enter a couple of times here. You should see:

```
1
2
```

```
4
6
9
```

Additionally, after the loop finishes, ***i*** will still exist and have the value ***9***. So, you can see, it sequentially assigns each value from the list to the variable ***i***, then executes the statement ***print(i)***.

While this is a fairly simple concept, this is one of the most useful and heavily used statements in Python. The next thing we need to consider is the 4th important point from our one line of code, if you still remember that far back:

The all important "***:***"

In the example above, there is a ***:*** character separating the list from the ***print*** statement. The ***:*** separates the ***for*** statement from the code that gets executed inside the loop. In our simple example we just put a single ***print*** statement after the ***:***, and indeed we could have put any one single command there, and been fine. However, what if you want to execute more than one command inside the loop ?

The answer is actually very simple. You do it like this:

```
for i in range(5):
    j=i*2
    print(i,j)
print("loop is done")
```

Now, this isn't a very interesting example, but it demonstrates the idea. Instead of putting the command on the same line immediately after the :, we hit <enter> and start a new line, then put our code there. How then does Python know which code to execute inside the loop, and what code to execute after the loop is complete ?

The trick is the indentation before the second and third lines. Anything indented to the same level will be executed inside the loop. The amount of indentation is arbitrary. You could indent one space, or with a single <tab>, or with 3 spaces. As long as you indent exactly the same way on each line, it will work. When you stop indenting, the code is outside the loop, meaning it won't be executed until the loop completes.

Where are the { } ?

If you've programmed in any other programming languages, particularly C, C++ or Java, you're probably expecting the code after the ***for*** statement to be inside curly braces. Sorry, Python doesn't do things that way. { } are used for a completely different purpose, and indentation is the sole way of denoting blocks of code in Python. This, at least, has the advantage of making Python code more readable than a lot of C++ or Java

That's it, we're done. We've considered all aspects of our simple little turtle example. Let's finish off this section with one more simple example program using the techniques we've learned. You'll have to type this one in if you want to see what it does.

```
from turtle import *
Turtle()
reset()
a=[90,180,90,90,180,90,0,180,-90,90,180]
fdr=[100,50,50,50,100,0,50,25,100,25,50]
fnd=[0,0,0,0,0,25,0,0,0,0,0]
ht()
for i in range(11):
    left(a[i])
    forward(fdr[i])
    up()
    forward(fnd[i])
    down()
```

Random Walk

LOREM IPSUM

1. **The random module provides functions for making random numbers of different sorts.**
2. **Less Random Walks**
3. **Traveling Circle**
4. **Making Decisions**

Totally Random Walks

So far, we've introduced two modules: math and turtle. Let's go ahead and add one more to our repertoire. Try this:

```
import random
for i in range(10): print(random.randint(1,100))
```

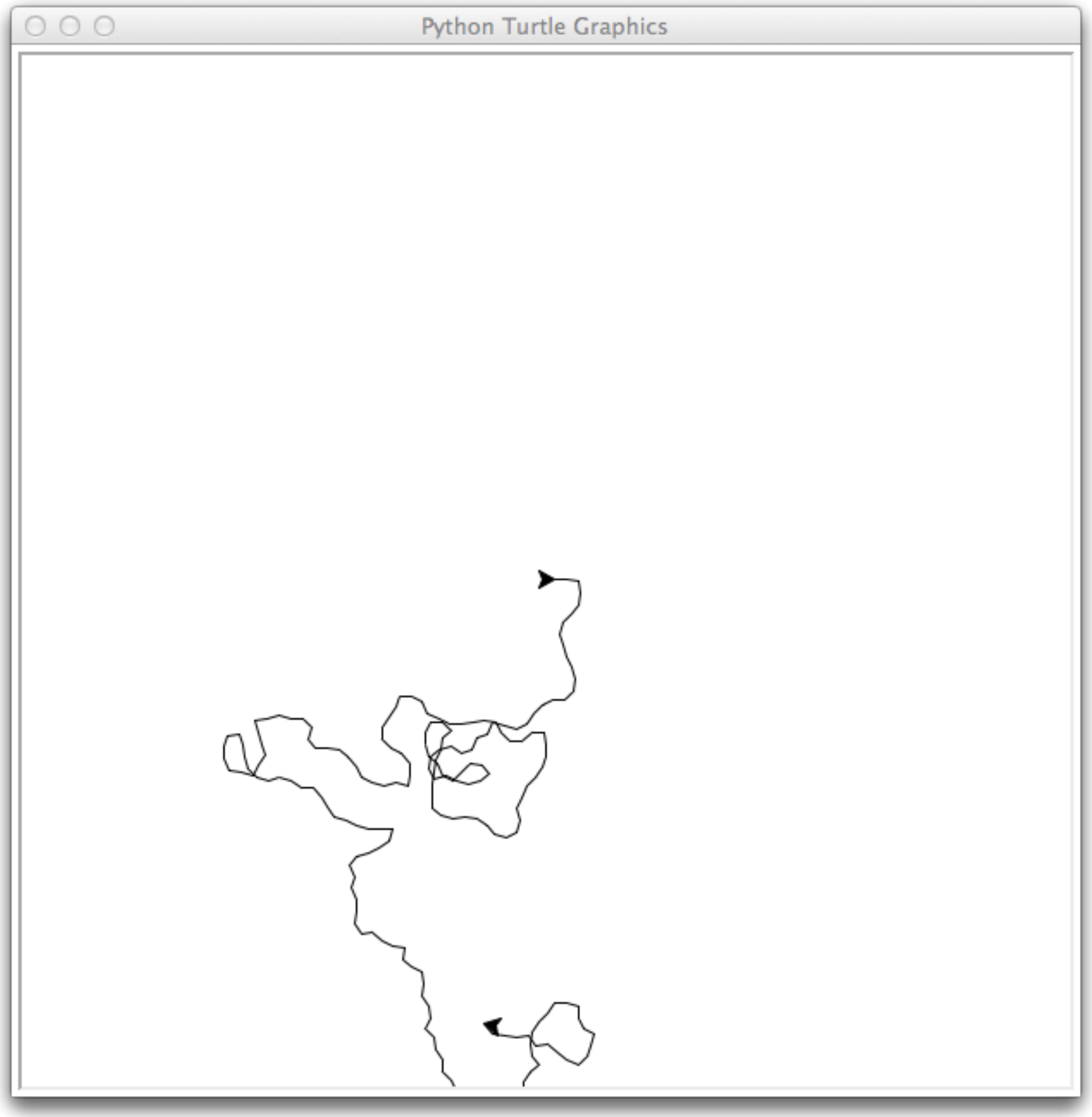
As you'll see, this program will print 10 random numbers between 1 and 100 (possibly including 100). If you run the program again, you'll get a different list of numbers each time. There are a number of other functions available within the random module as well, for example, `random.uniform(1,100)` will return a random floating point number between 1 and 100. `random.gauss(80,10)` will return a 'Gaussian' (a bell-shaped curve) centered at 80, with a width of 10. That is it will be more likely to return values close to 80. The farther you get from 80, the less likely it is to produce that number, but technically it could return 1000. It's simply very unlikely.

Let's try applying this to turtle graphics:

```
from random import *
from turtle import *
a=Turtle()
speed(0)
```

```
for i in range(250):
    forward(10)
    left(gauss(0,40))
```

Example 2.1 Random walks



Doing this, you will see something like (but not exactly) one of these. It can be fun to watch (a couple of times, anyway), so don't just rely on my screenshots. Give it a try :

Less Random Walks

So, we now have a turtle which knows how to wander around randomly on the screen. Not all that useful, though you could learn something about the behavior of random walks by playing around with that program. Let's see if we can make a walk that's random, but not completely random. Start by remembering if we do something like this:

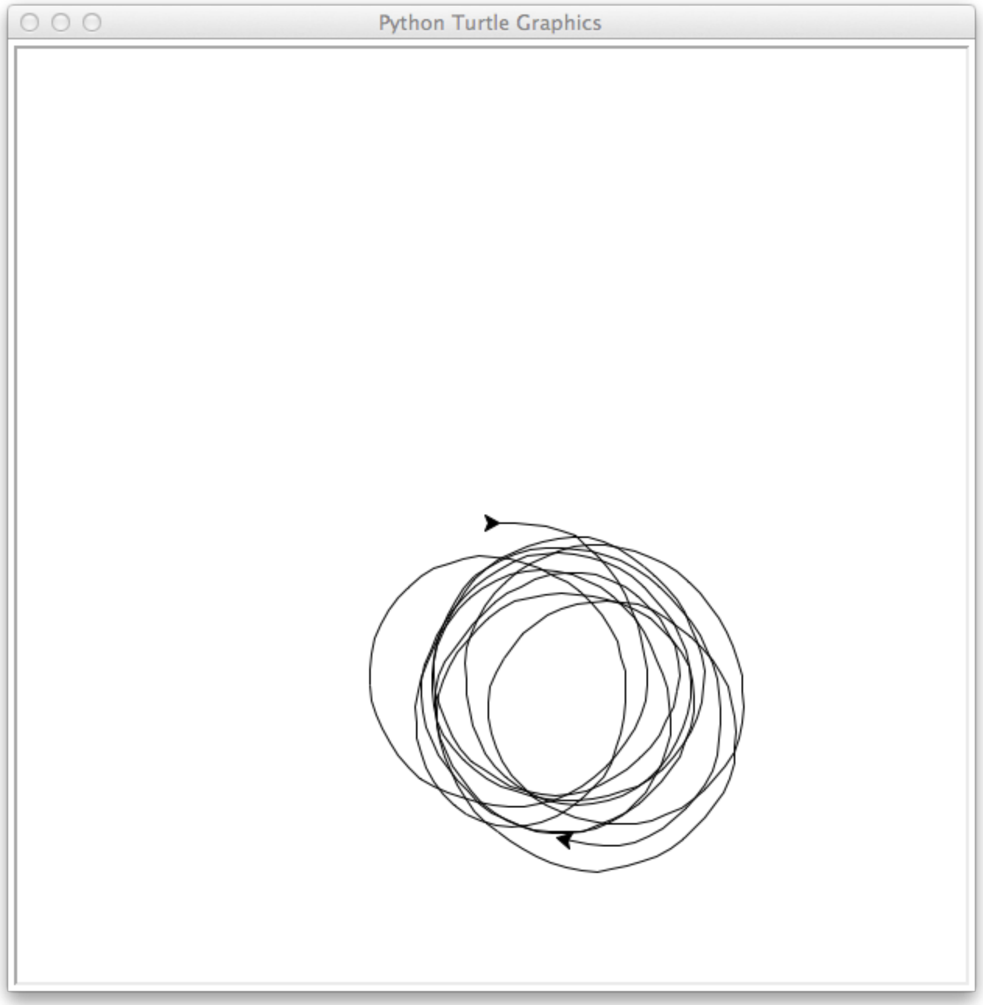
```
reset()
for i in range(36):
    forward(10)
    right(10)
```

we get a circle. Now, let's do the same, but with a little randomness thrown in:

```
reset()
for i in range(500):
    forward(10)
    right(gauss(7,3))
```

Kind of like scribbling circles with a pencil. A little different each time, but vaguely circular. You can play with the parameters inside ***`gauss()`*** and see what effects you can achieve.

Example 2.2 Random circles

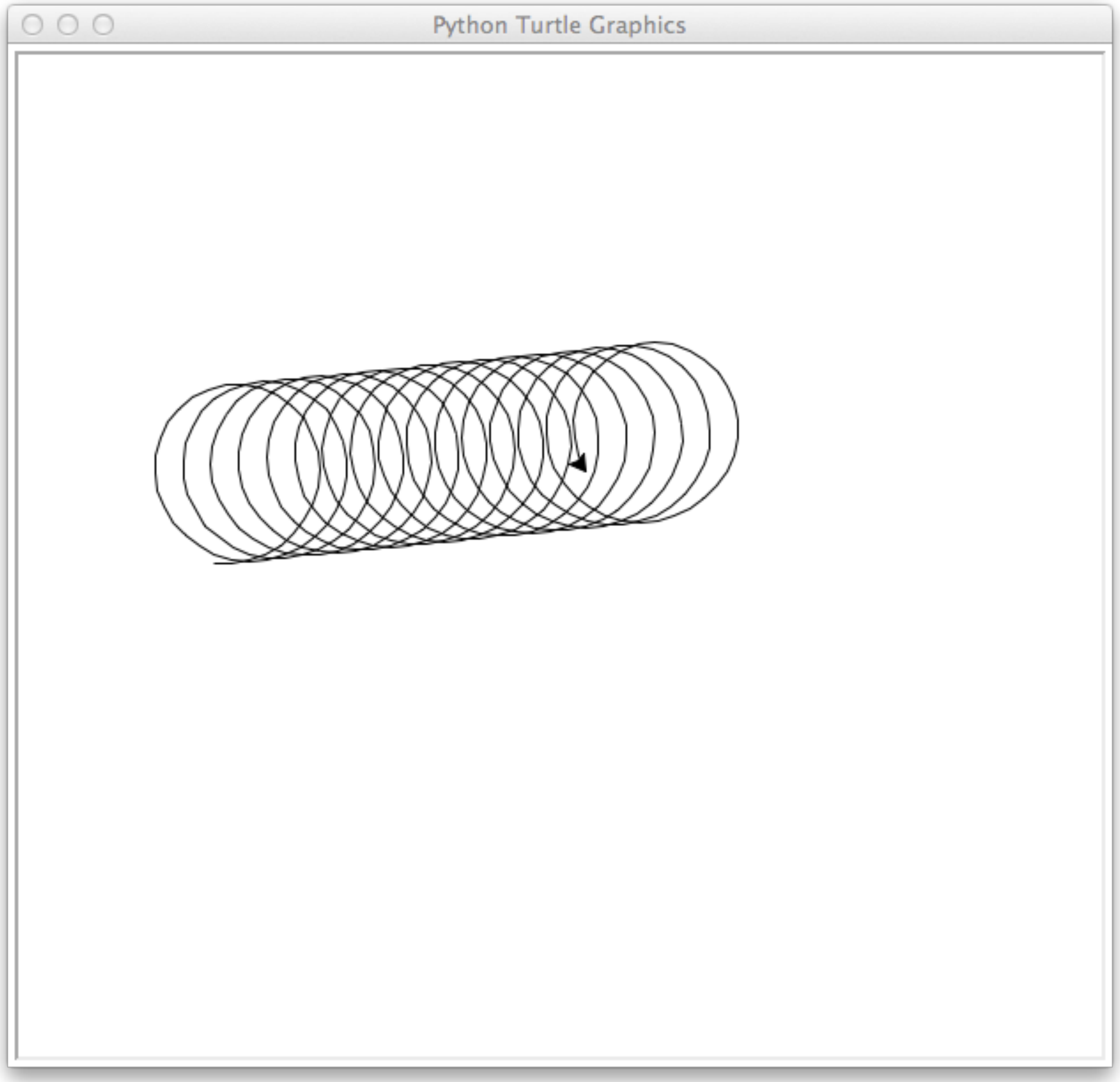


Traveling Circle

Lets take the next step and see if we can get our circle drawing to follow a path. Go ahead and exit your python session (by typing exit() or <ctrl-d>), then start it up from scratch, and give the following program a try:

```
from turtle import *
from math import cos
radians()
goto(-200,0)
clear()
for i in range(500):
    forward(10)
    left(0.2-cos(heading())/50.0)
```

You should see this:



Let's take a closer look at this program. Note that we aren't doing anything with random walks this time. This is basically the same as our circle drawing program with the exception of : ***`left(0.2-cos(heading())/50.0)`***. So, how does this result in drifting in a particular direction ?

The cos() function you may remember from high school. Cos() takes an angle as a parameter and oscillates between -1 and 1. In your math textbook : cos(0)=1.0, cos(90)=0, cos(180)=-1.0 and cos(270)=0. However, things are a little trickier than this. You can measure angles in 3 different ways. Generally in school, you learn to measure angles in degrees, with a right angle being 90 degrees and 360 in a full circle. However, you can also measure angles in radians. In radians, a right angle is π/2, and a full circle is 2*π. In basically all programming languages, sin(), cos() and tan(), take radians as arguments rather than degrees. By default, turtle graphics works with the more familiar degrees. However, since we want to work with cos() from the math library, radians are better. The ***`radians()`*** function tells the turtle to use radians for everything instead of degrees.

You'll also note that we didn't say ***`from math import *`***, but rather just ***`from math import cos`***. As it happens, the math module has a function called radians() too. As you may recall, if there is a conflict between two libraries, whichever one you imported LAST will be the function you see. So, rather than running into potential problems, we can import only the functions we plan to use.

So, we've explained a bit about cos() now, but how do we relate cos() to the direction the turtle is going ? There is another set of turtle functions we haven't talked about yet. The ones we studied in the last section allowed you to control the turtle's actions. The second set of functions allows you to ask the turtle for information about where it is and were it's pointing:

Function & aliases	Returns	Description
position pos	(x,y) - in pixels	Current turtle position
xcor	pixels	X location
ycor	pixels	Y location
heading	angle - radians or degrees depending on settings	Direction the turtle is pointing

In this example program, we use heading() to change how much we turn, depending on which direction we're going. If we're pointing up or down (π/2 or 3 π/2) then cos() is 0, and we draw a normal circle, but if we're pointing right, cos() is 1.0 and we turn a little bit slower than normal, so we go a little farther than we should. If we're facing left, then cos() is -1.0 and we turn a little faster than usual, so we end up going a bit less in that direction than we would for a circle. The result is what you see. We'll get back to this in the problems at the end of the chapter.

Making Decisions

Everything we've done so far has been based on simple sequences of operations. One key concept we're still missing is how to make decisions. Say we want to do a random walk, but we want to try and keep the walk inside some particular region of the screen. We need to be able to tell the program to act differently when certain conditions are met. This is accomplished in Python (and many other programming languages) through the ***if*** statement. Try this:

```
from random import *
from turtle import *
a=Turtle()
speed(0)
for i in range(750):
    forward(10)
    left(gauss(0,40))
    if xcor()>100 : seth(gauss(180,30))
    if xcor()<-100 : seth(gauss(0,30))
    if ycor()>100 : seth(gauss(270,30))
    if ycor()<-100 : seth(gauss(90,30))
```

What's happening here ? If the turtle moves out of a box going from (-100,-100) to (100,100), then it points the mouse generally back towards the center of the box. Technically it could still migrate out of this region, but it's very unlikely. To do this, however, we have to test whether the turtle is outside the box or not.

To make a decision in Python, we simply say:

if expression ***:*** do something

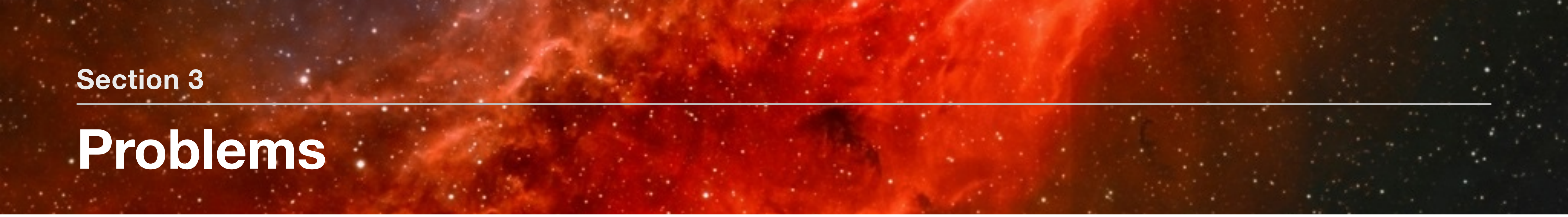
Expression is the (mathematical) question we're asking. If this question is true, then whatever is after the ":" gets executed, otherwise it doesn't. In this case True can also mean "not zero". To make these sorts of decisions, Python provides a number of **Boolean Operators**. These are very much like simple math operations (+ ,- ,/ ,* ,...) but each one returns either ***True*** or ***False***, not a number. Without adieu, here are Python's boolean operators:

>	greater than	<	less than
>=	greater or equal	<=	less or equal
=	equal	!=	not equal
is	identical	is not	not identical
and	both are True	or	either are True
not	T->F, F->T	in	item in list

With these operations you can have Python ask virtually any question you need answered. For example, consider:

```
(x<23 or x>35) and y<17
```

As you can see, parentheses can also be used for grouping terms, just as they can with normal mathematical expressions.


Section 3

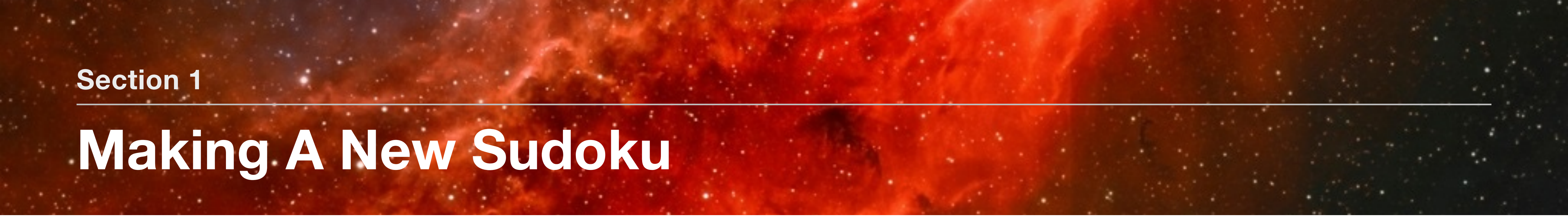
Problems

1)

Chapter 3

Sudoku

In this chapter, we'll learn how to create and solve Sudoku puzzles. Even if you don't know Sudoku or don't like them, this chapter will introduce many useful concepts.



Section 1

Making A New Sudoku

LOREM IPSUM

Sudoku, if you aren't familiar with it, is a popular puzzle game of pattern completion. While the rules are simple, they can be extremely challenging to solve. This chapter will introduce a large number of new concepts, so don’t feel bad if you don’t get everything immediately. Take it easy, and remember to try and have fun doing it!

The Rules

Before we can start thinking about how to write a program to solve or create Sudoku puzzles, we need to understand the rules. If you're already familiar with the rules, you can skip this section.

Sudoku is a pretty simple game. It's played on a 9 x 9 square grid. each square holds a number between 1 and 9. The trick ? Each row, each column and each 3x3 smaller square can only have one of each of the 9 numbers. An example of a solved Sudoku is shown to the right. When you have a book of Sudoku puzzles, some of the numbers are missing. The goal of the game is to fill in all of the missing numbers. The more that are missing, the harder the puzzle.

1	2	3	4	5	6	7	8	9
4	5	6	7	8	9	1	2	3
7	8	9	1	2	3	4	5	6
2	3	4	5	6	7	8	9	1
5	6	7	8	9	1	2	3	4
8	9	1	2	3	4	5	6	7
3	4	5	6	7	8	9	1	2
6	7	8	9	1	2	3	4	5
9	1	2	3	4	5	6	7	8

A Solved Sudoku

How would you go about making a Sudoku from scratch ? One simple approach would be to begin with an empty grid, then start filling in random numbers in squares. Each time you insert a number, you check to see if it's legal. If it isn't, then you try another. You then repeat until the whole puzzle is filled in. Then you can remove a few entries to form the unsolved puzzle. However, this isn't a very efficient approach. It could take a lot of time to compute a new Sudoku this way.

Scrambled Sudoku

A much better approach is to start with a valid solved Sudoku and change it following specific rules to make another valid solved Sudoku. Then remove a few of the numbers. There are some simple rules for changing a solved Sudoku so it remains a legal solution. Take a look and see if you can figure it out.

Consider for a moment what would happen if you took 2 rows of the solved Sudoku and swapped them. Since the whole row stays intact, you know that you haven't broken any rules within the row. Also, from the perspective of the columns, all you've done is swapped the position of two existing numbers. Again, that could never make the solution illegal. So, the only thing to consider then is what would happen to the 3x3 squares. Again, the answer is pretty straightforward, as long as you swap the rows within a 3x3 box rather than in-between boxes, you'll also keep a valid solution. Clearly the same arguments could be made for columns. So, to change an existing Sudoku, you can swap any pair of rows or columns within a 3x3 group. By repeating this swapping process multiple times, you should be able to achieve lots of different solved puzzles.

However, this won't get you every possible solved puzzle. There is one more operation we can perform which should always give a valid solution. We can take any pair of numbers, like 3 and 6, and change every 3 to a 6 and vice-versa. Like the row & column swapping, this operation can never produce an incorrect solution. If you think about it, there is nothing magical about the numbers 1-9. If we wanted, we could just as easily use the letters A-I. That is, when we exchange all 3s for 6s, we're just changing the symbols we're using, not the actual logic of the puzzle.

Functions

We could just take what we know and write a program to generate Sudoku puzzles now, but this is an excellent opportunity to introduce one of the most important concepts in programming: the **function**. You're probably already familiar with the basic idea of a function from math. In math I could say something like $f(x)=5+3*x+x^2$. Then $f(2)$ would be 15. In math this is a way of taking an equation that you will use often and giving it a name, $f(x)$ in this case. The same concept exists in programming, but it's a little more general. Like other concepts in programming the idea of a function doesn't just apply to math but can be used with strings, lists, etc.

Let's start with our example above, but express it in Python, just as a comfortable starting point:

```
def f(x):
    return 5.0+3.0*x+x**2.0
print(f(2))
print(f(5))
```

As you can see, the ***def*** statement is used to tell Python that we want to define a new function. The ***return*** statement is used to say what value the function returns to the caller. In this example the print statement calls the function f, and the ***return*** value is printed. The name of the function doesn't matter. Any name you could use for a variable could also be used for a function. You just need to be careful about existing built-in functions. For example, you could define your own function called ***open()***, but if you did, you'd be in trouble the next time you needed to open a file !

While this is a fine example to start with, functions are far more powerful than this. Let's try another example:

```
def squarelist(x):
    ret=[]
    for i in x: ret.append(i*i)
    return ret

mylist=[1,2,3,4,5]
print(squarelist(mylist))
```

So, we've defined a function called squarelist(), which will return a new list with each element squared. Now, we could do this a completely different way:

```
def square_inplace(x):
    for i in xrange(len(x)):
        x[i]=x[i]*x[i]

mylist=[1,2,3,4,5]
print(square_inplace(mylist))
print("--------")
print(mylist)
```

You may find this example confusing at first. Look at it very carefully. The first print statement doesn't show anything at all. That's because the function we defined doesn't have a return statement, so it doesn't return anything at all. However, it changes the contents of the list we pass into it. So, when we print mylist after calling the function, it doesn't contain the original values any more. This type of function is described as **acting by side-effect**. In the field of *functional programming*, this approach is considered extremely poor style, and in many languages it is impossible. However, it can be a useful method, and in some situations it is a practical necessity. Unfortunately, it can also be confusing. Give this example a try, for instance:

```
def test(x):
    x=x*x
```

```
y=10.0
test(y)
print(y)
```

You may have been expecting to see 100 printed out on the screen, right ? What you actually got, though, was 10. So, what's the difference between this example and the previous example ?

The answer is a little subtle, and may take a while before you completely understand it, if you aren't familiar with programming. In the first example, mylist contained a reference to a specific list that we defined. In our function, we made it so the temporary variable x pointed to the same list as mylist, then changed the list itself. We didn't do anything to change mylist, but rather, we changed the contents of the list that mylist was pointing to.

In the second example, y contains the number 10. When you call test(), it assigns the value of y, that is, 10 to the temporary variable x. You then change x so it's 100, but this has no impact on y. When you exit the function, the temporary variable x ceases to exist, and y is unchanged.

This simple example may help, if you still feel confused:

```
list1=[1,3,5,7,9]
list2=list1
list2[2]=99
print list1
print list2
```

In this example, list1 and list2 both point at the same list, so if you change one, you change the other.

As a closing note, it's also possible to define functions of more than one variable:

```
def hypot3(x,y,z):
    return sqrt(x**2+y**2+z**2)
```

Sudoku Storage

Before we can scramble our Sudoku, we need to decide how we will store the numbers. Clearly we want to use a list of some sort. There are several different ways this can be done, including a list of lists, or an array object from a module called NumPy. However, the simplest organization will be the easiest to explain with what we know right now. So we will work with a single list, 81 (9x9) numbers long, arranged as shown in the table to the right.

0	1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16	17
18	19	20	21	22	23	24	25	26
27	28	29	30	31	32	33	34	35
36	37	38	39	40	41	42	43	44
45	46	47	48	49	50	51	52	53
54	55	56	57	58	59	60	61	62
63	64	65	66	67	68	69	70	71
72	73	74	75	76	77	78	79	80

Storing Sudoku as a List

Now, before we write the overall scrambler program, we'll start by writing functions to do each of the rearrangements we discussed. Given how Sudoku numbers are arranged in the list, we need to figure out a way to swap rows and columns. Rows are more straightforward with this data arrangement, so we'll start with that.

The basic issue we need to address here is how to swap the data in two rows in the simplest possible way. You're probably thinking "Ahh, we're going to use a ***for*** loop !" Nope. Turns out there is a better way.

Advanced Slicing

If you recall our earlier discussions of slicing, we can use expressions like a[3:7] to pull out pieces of a list. What we didn't discuss earlier is that it's also possible to assign to a slice. Try this:

```
a=[0,0,0,0,0,0,0,0]
a[3:6]=[1,2,3]
print a
```

So, it's possible to modify a block of values in an array all at once. Now, here's another puzzler for you to try:

```
a=[1,2,3,1,2,3,1,2,3]
print a[::3]
print a[1::3]
```

We learned about slicing with ***a[from:to]*** previously, but what we didn't learn is that you can put a second ***:*** in. This second ***:*** lets us do ***a[from:to:step]***. The third value lets us take every ***step*** value from the list. If you still find this a bit confusing, try this one:

```
a=[1,2,3,4,5,6,7,8,9]
print a[::3]
print a[1::3]
```

Now think about this in terms of our big Sudoku list. Lets say we want all of the numbers from the 3rd column. Nothing simpler: a[2::9] !

Now we need one final trick. In most programming languages if you want to swap the values of two variables, you have to go through a three-step process:

```
a="abc"
b="def"
print(a,b)
x=a
a=b
b=x
print(a,b)
```

However, in Python you can do two assignments with one statement, making swapping a breeze:

```
a,b=b,a
```

If we put these three concepts together, we could swap row 'n' with row 'm' in a Sudoku list 'a', like this:

```
a[n*9:n*9+9],a[m*9,m*9+9]=a[m*9,m*9+9],a[n*9:n*9+9]
```

We can swap columns as well:

```
a[n::9],a[m::9]=a[m::9],a[n::9]
```

lots of nines...

We're now ready to start constructing our first serious program!

Sudoku Scrambler

Note: Don't immediately start typing all this code in yet. Just read it and try and understand what it does. In the next section we will discuss packing all of these functions into a usable program

We're now ready to write the three functions we need for the three scrambling operations :

```
def swap_row(sod,n,m):
    if n==m : return
    sod[n*9:n*9+9],sod[m*9:m*9+9]=sod[m*9:m*9+9],
sod[n*9:n*9+9]

def swap_col(sod,n,m):
    if n==m: return
    sod[n::9],sod[m::9]=sod[m::9],sod[n::9]
```

```
def swap_numbers(sod,a,b):
    if a==b: return
    for i in range(81):
        if sod[i]==a : sod[i]=b
        elif sod[i]==b : sod[i]=a
```

Let's also write a function to print a Sudoku out on the screen:

```
def show(sod):
    "Print a Sudoku list nicely on the screen.
Any zero entries will be shown as a blank"
    for a in range(0,81,9):
        if a in (0,27,54) : print("-"*25)
        for b in range(a,a+9):
            if b%3==0: print("|",end="")
            if sod[b]==0 : print(" ",end="")
            else : print(sod[b],end="")
        print("|")
    print("-"*25)
```

Note two new additions in this function. First, some of our ***print*** calls include ***end=""***. The default is for ***end*** to start a new line. That is, ***end=""*** causes the next print statement to continue on the same line of text on the screen.

The other new addition is that there is a string by itself on the line right after the ***def*** statement. This is a documentation string. You remember Python has a built-in ***help()*** interface, right ? This string is how you make your own functions work with ***help()***. After entering the ***show()*** function, try typing ***help(show)***.

We can now make use of these four functions to randomize a good starting Soduku and produce a brand new puzzle:

```
from random import randint,choice
sud=[1,2,3,4,5,6,7,8,9, 4,5,6,7,8,9,1,2,3,
7,8,9,1,2,3,4,5,6, 2,3,4,5,6,7,8,9,1,
5,6,7,8,9,1,2,3,4, 8,9,1,2,3,4,5,6,7,
3,4,5,6,7,8,9,1,2, 6,7,8,9,1,2,3,4,5,
9,1,2,3,4,5,6,7,8]

# we do one of each type of swap 20 times
# some will be skipped (if e1==e2)
for i in range(20):
    tri=choice([0,3,6])
    e1=randint(0,2)
    e2=randint(0,2)

    swap_row(sud,tri+e1,tri+e2)

    tri=choice([0,3,6])
    e1=randint(0,2)
    e2=randint(0,2)
    swap_col(sud,tri+e1,tri+e2)

    e1=randint(1,9)
    e2=randint(1,9)
    swap_numbers(sud,e1,e2)

ans=sud[:]  # This copies the list
n=int(raw_input("How many numbers to exclude? "))

# set a random number of elements to 0
for i in range(n): sud[randint(0,80)]=0

show(sud)

if raw_input("\nDo you want the answer ? (y/
n)")=="y" :
    show(ans)
```

The only new concept that we've used here are the ***choice()*** function from the ***random*** module. As is probably pretty obvious, ***choice()*** randomly selects one item from a list.

Writing Actual Programs

You now have all of the pieces you need for a program you can run to produce Sudoku puzzles, but how do we take all that code, put it together and run it as a program ? Clearly we don't want to type it all in by hand every time we want a new puzzle.

The basic idea is pretty simple. You create a plain text file, for example "myfile.py" then instead of just saying ***python***, you say ***python myfile.py***, and it will execute whatever you put into the file instead of giving you an interactive prompt. How we handle this, though, depends on the type of computer you're using.

Anaconda

If you have opted to install Anaconda (regardless of platform). It includes two different built-in editor and execution environments. One (spyder) is a traditional editor with a debugging environment, and the other (ipython notebook) is a web-based interactive notebook environment, sort of like Matlab or Mathematica. Both are excellent choices!

It is critical to understand that plain text files are not just .doc files or .rtf files which have been renamed with a .py extension. These software specific file formats contain a lot of information other than the text itself, and Python (nor any other programming language) will read these. We need files containing plain text.

Windows Users

While there are word processors on Windows machines that edit plain text files, such as the built in notepad.exe, the easier solution is to use the editor that comes with the Windows Python installation (or see the Anaconda box above). Instead of typing ***`python`*** in a terminal window, you need to launch the *`IDLE`* development environment, which includes both an interactive Python session as well as a text editor. When you launch *`IDLE`*, it opens the interactive session window, but not the editor window. You will need to go to the *`File`* menu and select *`New Window`* to get an editor.

IDLE has many features beyond simple editing, but let's start with the basics. Try typing in any of the simple examples you've done, then go up to the *`File`* menu and select *`Save`* or *`Save As`*, and give your program a name (doesn't matter what you call it really, but use a *`.py`* extension). Once you've done this, you can go to the *`Run`* menu and select *`Run Module`*.

This will cause your program to start in the interactive Python session window. When it finishes, either with an error, or because it's finishes normally, you'll get a Python prompt back again. If you typed something wrong, just go back to the editor window, fix it, then try running the program again.

Also note that once the file is saved with a *`.py`* extension, if you find the file on your desktop or with Windows explorer, you can just double-click on your program to run it inside Python. It will open a terminal window, run the program, and when it finishes, close the window again. Be forewarned, if your program doesn't ask for user input (like our last little program does), the window will pop open and then pop closed again before you can even see what it's doing. One solution is to put a ***`raw_input()`*** at the end of your program to stop it from exiting until you press <enter>.

Linux Users

On Linux machines, there are many different options. Most Linux systems will have Python preinstalled, but many won't come with the IDLE editor described in the Windows section installed. It's a separate package on Linux. However, it IS an available package on virtually every Linux distribution, so you can search for it using your package manager, and install it. There are also dozens of other (free) editing options for Linux. I personally like a text editor called `kate`, but there are many many other choices as well. Hard-core programmers will usually also learn a non-graphical text-editor such as *emacs* or *vi*, but this isn't necessary for our purposes. For the most part, I will simply assume that you are using IDLE, and you can read through the instructions in the Windows section.

There is one other major difference between Windows and all of the other platforms. On Windows, if you save the file with a .py extension, it will know that it's supposed to run that file using Python. On Linux or other UNIX systems, the file extension isn't used this way. The easiest solution, from the command-line is to simply type:

```
python mycommand.py
```

This will run your program within the existing terminal window. When the program completes, it will exit python and give you another system command prompt.

The Harder Way

It sure would be nice if we could skip the **`python`** part, and just type **`myprogram.py`** at the command prompt, or equivalently, double click on it from a browser. So, what would that take ? To accomplish this, you need to do two things. First, the first line of your python program must be this:

```
#!/usr/bin/env python
```

This tells Linux that the rest of the file should be run through the Python interpreter. Second, we need to tell Linux that this is a program we can run. We do this with this command:

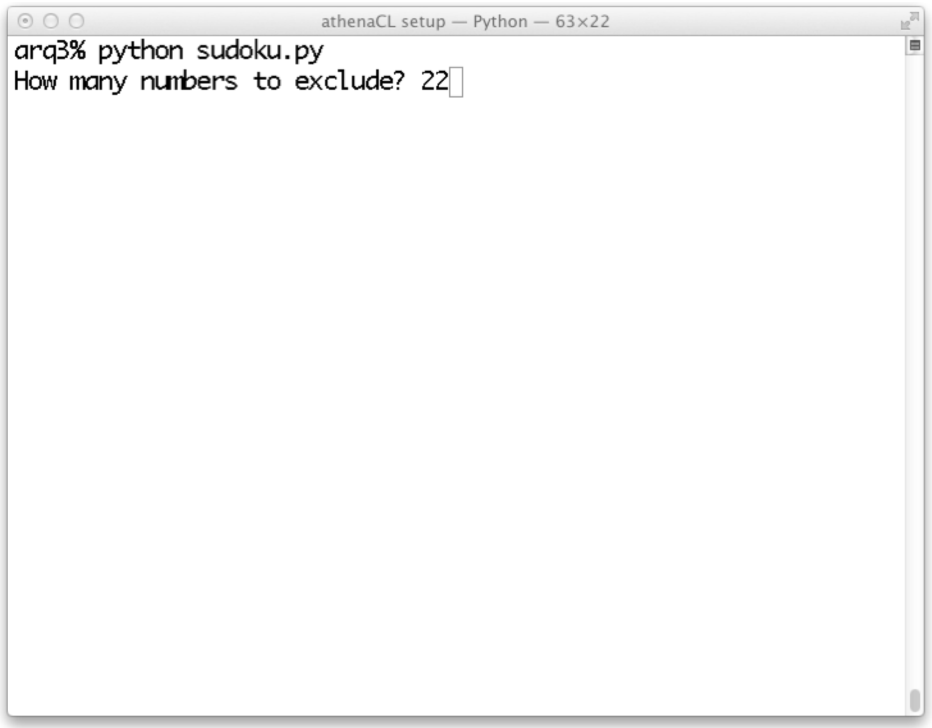
```
chmod a+x myprogram.py
```

After doing this, you can just type **`myprogram.py`** to run it.

Mac users

Like Linux users, Mac users will already have Python installed on their computers when they get them. In addition, they will find IDLE installed as well. Great, right ? Well, that depends. If you are using MacOS Lion (10.7) or newer (and you should be), then yes. Just read through the Windows instructions.

If you find yourself in that situation, the Mac does come with a simple text editor called *`TextEdit.app`*. You can use this to edit your files, then run them using *`python`* from the command prompt, but you must remember to save the files as *`Plain Text`*



```
arq3% python sudoku.py
How many numbers to exclude? 22
```

! The default for *`TextEdit`* is to save as *`Rich Text`*, which will not work at all.

There are also any number of other decent choices of text-editors on the Mac that you can use as well. Some even have integrated interpreters like IDLE does. The Mac App Store has a number of good choices that only cost a few dollars. Just make sure whatever you use can edit plain text files. Again, the Anaconda solution mentioned above is also an excellent choice.

Back to Sudoku

To try the Sudoku generating program out, you'll need to type in the 4 functions: ***`swap_row()`***, ***`swap_col()`***, ***`swap_numbers()`***, ***`show()`*** as well as the main program into your editor, in that order, into a single text file. Once entered, you can run it as discussed earlier. If you aren't big on typing, remember that you can download the full set of of source code from this book at:

One little bug in this program is that, given the way we randomly select the elements to exclude, it's possible that we may not exclude exactly the requested number of values every time. Other than this, however, this program should produce valid Soduku puzzles every time.

Nesting and Recursion

SUMMARY

1. **A nested loop is a loop inside another loop. It is used to loop over more than one dimension. For example, all of the rows and columns in a table.**
2. **A recursive function is a function that calls itself. Recursion has many uses, and while it is a bit difficult to absorb, in some cases there is virtually no other good way of achieving the same results.**

So far we've gotten by with simple `for` loops and lists. Before we move on, we're going to introduce not one, but two core concepts in programming: nested loops and recursive functions.

Nested Loops

Nested loops shouldn't really be a very difficult concept to grasp, but sometimes it can take a little practice to understand when to use them, and some people develop a bit of a mental block. The `for` loops we've already used operate on a single list. Now consider a table (like the 9 x 9 elements in our Sudoku puzzle). While it's absolutely possible to do what we did with the Sudoku puzzle, and 'unwrap' our table into a single list, it can also make our programs a lot more complicated.

To think about the alternative, let's think about a simple problem: printing a multiplication table. We'll do the same thing two different ways, one using a single `for` loop, and one using a nested loop.

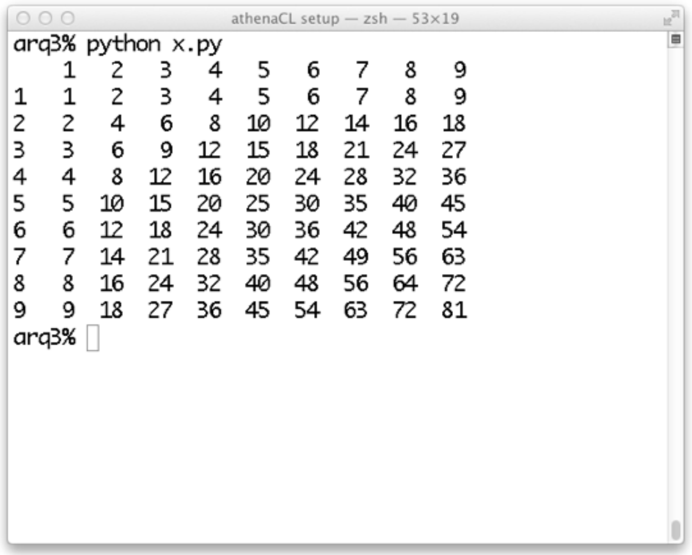
The Old Approach

If we want to make a multiplication table with a single `for` loop, we need to know the total number of entries in the table, and then figure out which row and column we're in for each item in the loop. We also have to jump through a few hoops to decide when to start a new line. For simplicity, we'll do a 9x9 multiplication table. That means (like the Sudoku) we will have 81 elements in our table. We can use division and the modulus (remainder) operator, `%`, to compute the row and column for each iteration of the ***`for`*** loop.

```
print("   ",end="")
for i in range(1,10):
    print("{:^3}".format(i),end="")
print("")

for i in xrange(81):
    r=i//9+1        # rounds down
    c=i%9+1
    if c==1 : print("{:<2}".format(r),end="")
    print("{:^3}".format(r*c),end="")
    if c==9 : print("")
```

So, we have to use ***`r=i//9+1`*** and ***`c=i%9+1`*** to compute where we are in the table. Then we have to use ***`if c==1`*** and ***`if c==9`*** to detect the beginning and ending of each row. This works perfectly well, but it's a bit awkward.



```
arq3% python x.py
     1   2   3   4   5   6   7   8   9
1    1   2   3   4   5   6   7   8   9
2    2   4   6   8  10  12  14  16  18
3    3   6   9  12  15  18  21  24  27
4    4   8  12  16  20  24  28  32  36
5    5  10  15  20  25  30  35  40  45
6    6  12  18  24  30  36  42  48  54
7    7  14  21  28  35  42  49  56  63
8    8  16  24  32  40  48  56  64  72
9    9  18  27  36  45  54  63  72  81
arq3%
```

One other thing we need to talk about is the statements like:

`print("{:^3}".format(i),end="")`

The issue we ran into with this program is the need to keep the rows and columns nice and neat. We didn't have this issue with Sudoku puzzles because all of the numbers are 1 digit. In a multiplication table, we have some one and some two digit numbers. The ***`format()`*** method, allows us to change how numbers and other variables are represented when we print them. The method used here was a very simple, but not very obvious, statement. ***`{:^3}`*** says that the number should occupy 3 spaces and should be centered. We will get back to the details of formatting in a later chapter.

Let's get back to nested loops. Here is an alternate version of the same program:

```
print("   ",end="")
for i in range(1,10):
print("{:^3}".format(i),end="")
print("")

for r in range(1,10):
    print("{:<2}".format(r),end="")
    for c in range(1,10):
        print("{:^3}".format(r*c),end="")
    print("")
```

This program has exactly the same output as the previous program. While it is only 1 line shorter than the previous program, conceptually it's much simpler. No trying to compute indices, no tricky if statements. Also, if we want to expand our multiplication table to go to 15 instead of 9, for example, we just need to change the upper limit of 3 range statements.

It should be pretty clear how this program works. The first ***for*** loops over the variable r. This is called the **outer loop**. The second ***for*** loops over the variable c, and is called the **inner loop**. The inner loop covers each value of c for each value of r. This process is called nesting. The second ***for*** is nested inside the first ***for***.

As we'll see in some later examples, this isn't limited to one loop inside another. There is no reason you can't have a loop inside a loop inside a loop, if you like. Of course, say each loop was covering 10 different values. If you made a nested loop 5 deep, that means anything inside the innermost loop would be executed 100,000 times ! Nonetheless, there are some very good reasons for doing this sort of thing.

Recursion

The other concept we need to discuss is **recursion**. This is one of only a handful of basic programming concepts which is universally taught. There are some problems which can only be effectively solved using recursion. The definition of recursion sounds very simple. A recursive function is a function which calls itself. While the definition is simple, actually using recursion is a bit trickier, and it's pretty easy to wind up in a situation where your recursive function goes forever. On some older computers, you could even crash the entire computer this way. Take a look at the following example:

```
def f(x):
    return f(x-1)*x

print(f(4))
```

You can see what would happen if you did run this program. It would compute:

```
f(4) which is
f(3)*4 which is
f(2)*3*4 which is
```

```
f(1)*2*3*4 which is
f(0)*1*2*3*4 which is
f(-1)*0*1*2*3*4 ... forever
```

Now it turns out if you actually run this example, it won't run forever. Python has a safety mechanism to prevent infinite recursion, but the program will crash. If we want recursion to do something useful, we have to have some test that makes it stop.

```
def f(x):
    if x<=1 : return 1
    return f(x-1)*x
```

With this if statement, `f(x)` will keep getting smaller until it it reaches 1, then it will stop, and the chain of function calls collapses, and a number comes out. The reason we said `x<=1` in our condition rather than `x==1` was an error prevention measure. If we said `x==1` and someone called f(-1), it would never reach the termination condition.

So, what is this function ? If you look at the example above, you will see that it's basically doing 1*2*3*4*5*...*x. In other words, this is the factorial function. Of course, this isn't really the most efficient way to write a factorial function. We could have just done this:

```
def f(x):
    r=1
    for i in range(2,x+1): r*=i
    return r
```

So, what then is the use of recursion ? We'll see some good uses for it in upcoming sections, but here is a simple example of a program which is easy to write recursively, but difficult to write as a loop:

```
def numbers(digits,base=""):
    ret=[base]
    if digits<=0: return ret
    for i in range(digits+1):
        ret.extend(numbers(i-1,base+str(i)))
    return ret

print(numbers(6))
```

If you can't figure out what it does, give it a try. Then see if you can come up with a program that does the same thing without recursion.

Section 3

Solving Sudoku

SUMMARY

Let's turn the problem around now. Let's say we have a Sudoku with missing numbers. How would we go about filling in the missing values (correctly) ?

Regardless of how we try to fill the numbers in, we will need to define a function that checks to see if a given 'Sudoku list' is a legal solution or not. This is pretty straightforward conceptually. We just need to loop over all of the rows, columns and 3x3 regions, and check if all of them have exactly 1 of each number 1-9. If any fail the test, then we return False. Otherwise we return True.

Sets

To do this efficiently, we need to introduce yet another type of Python object: the `set`. Like a `list`, a `set` contains other objects, such as numbers or strings. Unlike a `list`, a `set` has no order, and the items in the `set` are unique. That is, if you have a set containing 1,3 and 5, then add 3 to the set, the set will still have 1,3 and 5 in it. With a `list`, if you had `[1,3,5]`, and appended 3 to the `list`, you would have `[1,3,5,3]`. Give this a try:

```
a=[1,2,3,4,3,2,5,7,9,12,3]
b=set(a)
print a,b
```

Sometime in grade school, you probably learned about sets (and most likely thought they were the most useless things you'd ever heard of). Sets in Python are pretty much the same thing, and you will soon learn they are a lot more useful than you may have thought in grade school. You can do all of the normal set operations with them: union, intersection, difference, ...

So, why use a set ? Say one of the rows in our possible Sudoku contains the numbers [1,3,7,6,9,8,5,4,3]. We need to know if this is a valid set of numbers for the row. To be valid, there needs to be exactly 1 of each number somewhere in the list. How do we do it ? Here is one simple approach:

```
r=[1,3,7,6,9,8,5,4,3]

def check(x):
    for i in range(9):
        if z.count(i)!=1 : return False
    return True

print(check(r))
```

This works, but it's slooow. If we have to check many thousands of possible solutions, this will make our program painful to use.

What else could we try? How about this:

```
r=[1,3,7,6,9,8,5,4,3]
def check(x):
    x.sort()
    if x!=[1,2,3,4,5,6,7,8,9] : return False
    return True

print(check(r))
```

This is very slightly more efficient than the previous program, and is certainly easier to read, but it still isn't really a very optimal approach. It also has the side-effect of changing the ordering of r, which may not be desirable.

Sets give us a much easier solution. If we know that our list is only going to contain numbers from 1 to 9, then if there are 9 unique values in the list then we know the solution is valid (for that list). If there are less than 9 unique values, then the solution isn't valid:

```
def check(x):
    if len(set(x))!=9 : return False
    return True
```

Sudoku Checker

Based on this concept, we can go ahead and write a complete Sudoku checker:

```
def check(x):
    """This function checks a Sudoku list for
correctness. x must be a list of exactly 81
integers from 1-9."""
    # rows
    for r in range(0,81,9):
        if len(set(x[r:r+9]))!=9 : return False

    # columns
    for c in range(9):
        if len(set(x[c::9]))!=9 : return False

    # 3x3 blocks
    for b in [0,3,6,27,30,33,54,57,60]:
        bl=x[b:b+3]+x[b+9:b+12]+x[b+18:b+21]
        if len(set(bl))!=9: return False

    return True
```

How does this work ? It simply takes each possible row, column, and 3x3 square, makes a set for each one, and makes sure the set has 9 values in it. If it has fewer than 9 values, we can assume that there is a duplicate.

With this function, we could write a program that tries many different possible solutions, and checks to see if each one is ok. We only need one (and in most cases there can be only one).

Solving a Sudoku

We're ready to try and write our full sudoku problem solver now. There are several different ways we could try and do this. Say we have this (extremely) simple unsolved puzzle:

	2		4	5	6	7	8	9
4	5	6	7	8	9	1	2	3
7	8	9	1	2	3	4	5	6
2	3	4	5	6	7	8		1
	6	7	8	9	1	2	3	
8	9		2	3	4	5		7
3	4	5	6	7	8	9	1	2
6	7	8	9	1	2	3	4	5
9	1	2	3	4	5	6	7	8

This particular Sudoku can be solved using simple rules. One of the missing squares is in a row where only one number is missing. Filling that number in produces another blank with only one possible answer, and so on. However, not all Sudokus are this simple. We can't just write a program that will assume that one answer will just fall out.

Dumb but Persistent

It may sound a little intimidating to try and write a program to "look at" the puzzle and figure out what each number is and fill them in. While doing that isn't really as bad as it sounds, there is a very simple solution to the problem, which is guaranteed to work. It just isn't guaranteed to be very quick about it. There are 7 missing numbers in our Sudoku. We can simply try every possible number (1-9) in each empty slot, then check if each solution is legal. With 7 missing values, that basically means we to look at every possible 7 digit number containing no zeroes. This will be 9^7 = 4,782,969 different solutions to test.

When writing programs, novice programmers used to be taught a technique called "flowcharting" a method for drawing a diagram indicating the sequence of operations in a program. The flowchart would then be converted into a program. We are going to take a simpler approach and simply outline the steps required in our program. This is an excellent way to start when writing your own programs. For our simple Sudoku solver, we need to:

1. Define a function for the user to enter a Sudoku
2. Define a function for testing a Sudoku
3. Define a function to print a sudoku
4. Define a recursive function to give us numbers to fill in
5. Call function 1
6. Make a list of where all of the missing numbers are
7. Use function 4 to generate possible Sudokus
    1. Test each answer for correctness
    2. If we get a valid answer, print it and exit

We'll start with a function that asks the user to enter the puzzle, putting a 0 in wherever there is a missing value. To do this, let's introduce the `raw_input()` function. This function will display a string, then wait for the user to type something and press enter. Whatever the user typed is returned by the function. so:

```
name=input("What's your name: ")
print "Hi ",name
```

will ask for your name, wait for you to enter something, then say hi. Now, back to Sudoku.

It would be annoying for the user to have to type a lot of spaces or commas, so the user is asked to simply enter the Sudoku as a string of numbers for each row, such as `020456789` for the first row of our unsolved puzzle:

```
def enter_puzzle():
    print "Enter the puzzle using 1-9 and 0 for
unknowns. 9 numbers per line:"
    ret=[]
    for r in range(9):
        ln=input(": ")
        for c in ln: ret.append(int(c))
    return ret
```

This doesn't do any error checking, so the user will have to enter the puzzle exactly correctly. The next two steps, a function to check Sudoku and one to print Sudoku we've already written. So, all that's left is to write function 4, and the main program that calls the functions.

So, how do we write a function to generate the values to fill in ? One of the examples in the recursion section did something very like what we need in this case. We basically need a list of all possible N digit numbers containing the values from 1-9, then we can replace the N 0's in the Sudoku with the digits from the number. While this idea is fine for our current 7 missing number puzzle, if we had more missing values than this, we would start running into problems having a list of hundreds of millions of numbers.

If we think about it, however, we don't really need to make an actual list of all possible numbers. We could check each possibility as it gets created instead, and when we find a good answer, print it and exit:

```
def find_answer(puzzle):
    z=-1
    for i in range(81):
        if puzzle[i]==0:
            z=i
            break

    #this means there are no zeroes
    if z==-1:
        if check(puzzle):
            show(puzzle)
            sys.exit(0)
    # fill in the first zero and recurse
    for i in range(1,10):
            puzzle[z]=i
            find_answer(puzzle)
        puzzle[z]=0
```

There are three parts to this function. First we look for the first zero in the array. If we don't find one, then the puzzle is full, and we need to check it. If we do find one, we fill it in with each possible value, and iterate to look for the next one.

We also used a new command in the first for loop: ***break***. This command has a simple purpose. It exits a loop before it's complete. That is, say you have a for loop looping over a list of 10 numbers, but you execute the break command while the 5th number is being processed. The loop will stop immediately, and the last five numbers won't get looped over.

To actually run this program, you're going to need to put the four functions: **find_answer()**, **check()**, **show()** and **enter_puzzle()** into a single file, followed by:

```
puzzle=enter_puzzle()
find_answer(puzzle)
```

Then you can run the program. I'd suggest starting with the simple example puzzle shown at the beginning of the section. This example puzzle should only take a couple of seconds to run, but if you try a puzzle with, say, 15 missing values, you may sit waiting for a VERY long time.

A Little Smarter

Since most Sudokus will have quite a few missing values, clearly we want to be a little more intelligent about how we look for solutions so we don't sit around for a month waiting for the answer. So, rather than blindly trying every value in every empty slot, let's take the next step, and for each open slot, we'll compute which values are possible given the rest of the values that are already available in the puzzle.

The actual change to the find_answer() function is pretty simple. we're just going to change:

```
for i in range(1,10):
```
to
```
for i in choices(puzzle,z):
```

Now all we have to do is write the choices() function, which will return a list of available values for a specific zero position, z. Even this function isn't very hard to write, since it will be based on the Sudoku checking function:

1. Make a set containing 1-9
2. Make a set containing the values from the same row
3. Union with a set containing the values from the same column
4. Union with a set containing the values from the same 3x3
5. subtract set 4 from set 1

If we used lists, this would be really messy, but with sets making sure we never have duplicates, it's quite straightforward:

```
def choices(puz,z):
    """This function finds permitted values for
location z in a Sudoku."""
    row=z//9
    col=z%9
    # This is the first element in each block
    blocks=[0,3,6,27,30,33,54,57,60]
    sq=blocks[col//3+(row//3)*3]

    bad=set(puz[row*9:row*9+9])     # row
    bad.update(puz[col::9])         # column
    # the 9 values from the local 3x3

bad.update(puz[sq:sq+3]+puz[sq+9:sq+12]+puz[sq+18
:sq+21])

    good=set(range(1,10))-bad

    return good
```

The math here may not be completely obvious. If you find it confusing reference back to the figure where we outlined how sudokus are stored as a list. Still there are a few items that might seem odd, like `(row//3)*3`. Mathematically this seems idiotic, right ? The trick is that `row//3` is not the same as `row/3.0`. `row//3` is integer math, which always rounds down, so `(row//`

3)*3 has the result of rounding row down to the nearest multiple of 3. We could also have done the same thing with ***(row-row%3)***.

The only new element we introduced in the ***choices()*** function was the ***update()*** method. Try this:

```
a=set(1,2,4)
b=set(2,5,6)
a.update(b)
print a
```

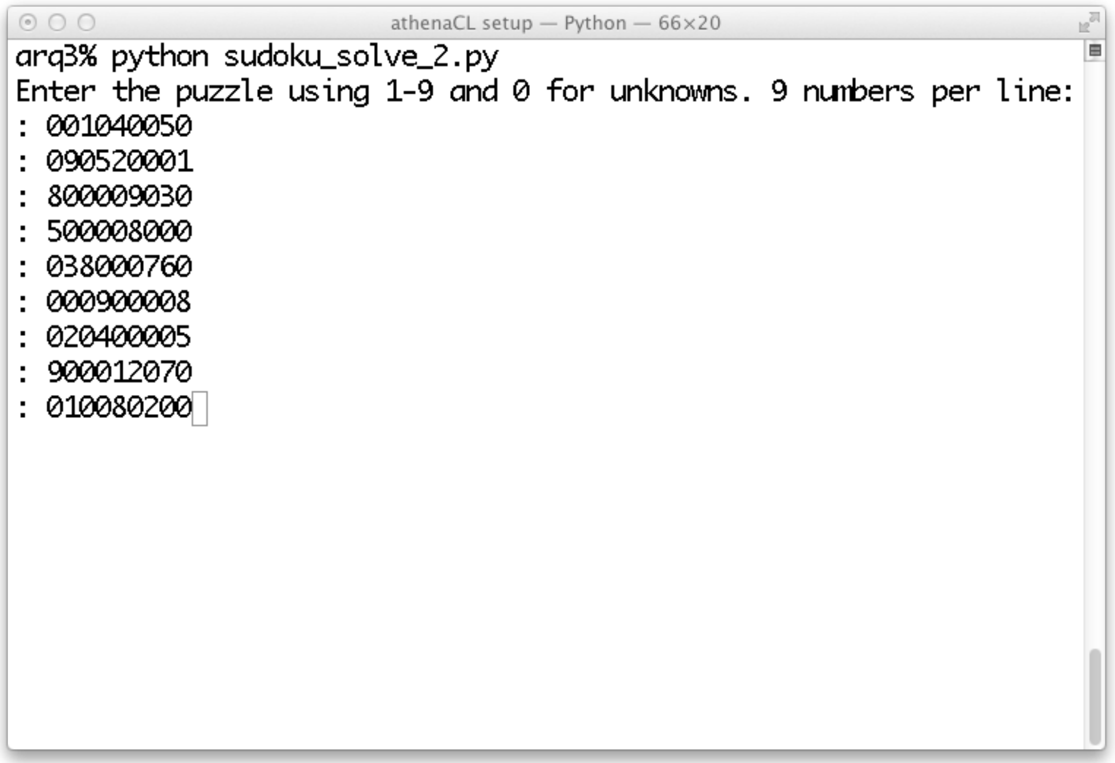
As you can see, ***update*** allows you to merge the contents of one set with another.

If we consider the overall program, the simple replacement of ***range(1,10)*** with ***choices()*** makes a massive difference. With the old program, even a puzzle with only 12 missing numbers would take hours to finish. This seemingly small change can solve the most difficult class of Sudoku puzzles, with 53 missing numbers, in under a second.

You can find the source for the entire program in the usual place at:

Here's an example of the entire program running:

Example 3.1 An example of the sudoku solver running



```
arq3% python sudoku_solve_2.py
Enter the puzzle using 1-9 and 0 for unknowns. 9 numbers per line:
: 001040050
: 090520001
: 800009030
: 500008000
: 038000760
: 000900008
: 020400005
: 900012070
: 010080200
```

This demonstrates the solver running on a very challenging puzzle with 53 of 81 values missing.

Section 4

Problems

Lorem ipsum dolor sit amet, ligula suspendisse nulla pretium, rhoncus tempor placerat fermentum, enim integer ad vestibulum volutpat. Nisl rhoncus turpis est, vel elit, congue wisi enim nunc ultricies sit, magna tincidunt. Maecenas aliquam maecenas ligula nostra, accumsan taciti. Sociis mauris in integer, a dolor netus non dui aliquet, sagittis felis sodales, dolor sociis mauris, vel eu est libero cras. Interdum at. Eget habitasse elementum est, ipsum purus pede porttitor class, ut lorem adipiscing, aliquet sed auctor, imperdiet arcu per diam dapibus libero duis. Enim eros in vel, volutpat nec pellentesque leo, temporibus scelerisque nec.

Ac dolor ac adipiscing amet bibendum nullam, massa lacus molestie ut libero nec, diam et, pharetra sodales eget, feugiat ullamcorper id tempor eget id vitae. Mauris pretium eget aliquet, lectus tincidunt. Porttitor mollis imperdiet libero senectus pulvinar. Etiam molestie mauris ligula eget laoreet, vehicula eleifend. Repellat orci eget erat et, sem cum, ultricies sollicitudin amet eleifend dolor nullam erat, malesuada est leo ac.

Varius natoque turpis elementum est. Duis montes, tellus lobortis lacus amet arcu et. In vitae vel, wisi at, id praesent bibendum libero faucibus porta egestas, quisque praesent ipsum fermentum placerat tempor. Curabitur auctor, erat mollis sed fusce, turpis vivamus a dictumst congue magnis. Aliquam amet ullamcorper dignissim molestie, sed mollis. Tortor vitae tortor eros wisi facilisis. Consectetuer arcu ipsum ornare pellentesque vehicula, in vehicula diam, ornare magna erat felis wisi a risus. Justo fermentum id. Malesuada eleifend, tortor molestie, a fusce a vel et. Mauris at suspendisse, neque aliquam faucibus adipiscing, vivamus in.

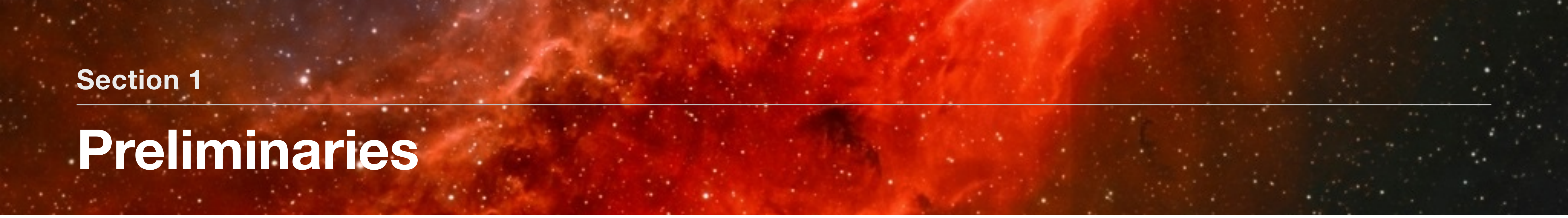
Wisi mattis leo suscipit nec amet, nisl fermentum tempor ac a, augue in eleifend in ipsum venenatis, cras sit id in vestibulum felis in, sed ligula. In sodales suspendisse mauris quam etiam erat, quia tellus convallis eros rhoncus diam orci, porta lectus esse adipiscing posuere et, nisl arcu vitae laoreet. Morbi integer molestie, amet suspendisse morbi, amet, a maecenas mauris neque proin nisl mollis.

Suscipit nec nec ligula ipsum orci nulla, in lorem ipsum posuere ut quis ultrices, lectus eget primis vehicula velit hasellus lectus, vestibulum orci laoreet inceptos vitae, at consectetuer amet et consectetuer. Congue porta scelerisque praesent at, lacus vestibulum et at dignissim cras urna.

Chapter 4

Wordgames

In this chapter, we'll use strings in Python to provide solutions for some common word-games, or perhaps, even create some new puzzles.



Section 1

Preliminaries

LOREM IPSUM

More Ify Statements

While we could just jump right in and start writing programs again. Let's take a break and introduce a few more concepts first.

Let's start by going back briefly to our discussion of the ***if*** statement. Actually, there's even more to it than we've seen. The full description of ***if*** is:

```
if expression : do something
elif expression : do something else
else : do something completely different
```

This sequence of commands lets you ask a sequence of linked questions. If the first question is *True*, then it executes **do something** and skips the rest. If the first question is *False*, then it asks the second question. If that's *True*, then it executes **do something else**, and skips the *else*. If that one isn't *True* either, it executes **do something completely different**. You can put as many *elif* statements in as you like, but there can only be a single *else* at the end, which happens only when everything else in the list is false.

This sequence is often used to do things like ask the users questions. Let's try using ***input()*** with our new extended ***if*** statement as an example:

```
s=raw_input("What do you say ? ")
if s=="hi" or s=="hello" : print "Hi to you too"
elif s=="bye" : print "Adios"
elif s=="huh" : print "You COULD say hello"
else: print "Sorry, I didn't understand"
```

If you type this little program interactively, you'll find that it asks you "What do you say ?" before you even have a chance to finish entering everything. You'll have to actually answer the question before you can finish typing the program in. This is rather inconvenient, so you may want to start putting some of these new examples into **.py** files and running them, as we did in the last chapter. Alternatively, if you are using the ***ipython notebook*** system, you can paste in an entire block of code into one *in[]* section before pressing <shift-enter>.

While Not ?

We've finished with ***if*** now, and already learned about ***for***, but there are two more commands that it would be useful to know for the programs to come: ***while*** and ***continue***.

We'll start out with ***while***, which is very much like ***for*** in many ways. It repeats a block of code over and over again. However, ***for*** executes the block once for each item in a list, whereas ***while*** lets you decide how long the loop should continue using an expression like the one used with an ***if*** statement. Give this a try:

```
a=1.7
while a<3.0 :
    a=a+0.1
    print(a)
```

Even someone who didn't know any programming could probably figure out what this is supposed to do. As long as the condition ***a<3.0*** is met, it will execute the code inside the loop over and over again. Based on that, if you changed ***a=a+0.1*** to ***a=a-0.1***, you would be in real trouble, as the program would keep running forever (the condition would always be true).

Just as a little side-note, if you do find yourself in this situation (a program that's running forever), on most machines <ctrl-c> in the terminal window is the best way to force it to stop. Alternatively, closing the window it's running in will generally also kill the program.

Let's get back to ***while*** for a while. As we've seen, ***while*** continues as long as a specific condition is true. Consider this example:

```
a=1.7
while True:
    a=a+0.1
    if a>=3.0 : break
    print(a)
```

Seems like it would run forever, right ? Well, not quite. If you recall from an earlier chapter, the ***break*** statement causes a loop to exit. It works for ***while*** as well as for ***for*** loops, and gives a little more flexibility. Specifically, it allows you to exit a loop in the middle rather than having to wait until the ***while*** condition is tested again.

The ***continue*** statement also allows us to manipulate how the loop executes. Here's another example:

```
a=0
while a<20:
    a+=1
    if a%3==0: continue
    print(a)
```

Give this a try and see what happens. You should see numbers from 1 to 20, however, if you look carefully, you'll notice that any number divisible by 3 is missing. If you didn't catch it earlier, the `%` operator is the ***modulus*** operator. It returns the remainder after integer division. For example 7/3 is 2R1, and ***7%3*** is thus 1.

Of course, what we're actually interested in here is the ***continue*** statement. While ***break*** causes loops to end prematurely, ***continue*** skips only the current cycle of the loop. That is, ***continue*** jumps immediately to the beginning of the next cycle.

Strings

We already introduced strings briefly in Chapter 1, but we need to know a lot more about them if we want to play word-games. Previously we created a simple string with an expression like "ABCDEFG". Somewhat unusually, Python includes no fewer than 4 ways to define a string:

```
"abcdef"
'abcdef'
"""abcdef"""
'''abcdef'''
```

All four of these lines produce exactly the same string. So, why then would Python have 3 different ways to do the same thing ? For the first two methods, it solves one problem, which isn't really a problem. Let's say you wanted a string: *`I can't make it.`* How would you do that. If you used a single quote string:

```
'I can't make it'
```

then the apostrophe would end the string, and you'd get an error, but if you use double quotes, then everything is fine. This is more of a rationalization than a reason for adding a new feature to a language, but it's all I've got.

The difference between single quotes and triple quotes (of both types) is a bit more practical. If you want to put an entire paragraph of text into a string, including multiple lines, triple quotes let you do it. for example:

```
"""This is an example of a string
that spans
three lines of text"""
```

While this may not seem very useful at first, later on when we learn how to add documentation to our programs, you'll see that it can be extremely convenient.

Slicing Strings

You remember when we discussed lists, you learned that you could pull out pieces of lists using ***[n:m]*** ? Well, the same things you can do with lists can also be done with strings. Here are a couple of examples:

```
a="A test string"
print(a[2])
print(a[2:8])
```

You can even use ***for*** to iterate over the letters in a string:

```
for L in a:
    print(L)
```

Changing Strings

You can't.

What ? What do you mean, I can't ? Sorry. It's true. Once you create a string, you can't change it. What you can do is make a new string, which is a changed version of the old one. Unfortunately, that means that you cannot do something like ***a[4]="X"***. Instead, you would have to do the, much more complicated :

a=a[:4]+"X"+a[5:].

This has exactly the same effect as ***a[4]="X"*** would have, but creates a new string rather than changing an existing one. On the bright side, there are plenty of convenience functions provided which allow you to create modified strings in useful ways. For example ***"A Test String".lower()*** will return a new string, `"a test string"`. You can get a list of all of these useful functions with ***help(str)***.

Mutable Strings

Ok, I just said strings weren't mutable, and it's true, they aren't. You can't change them. However, for the more advanced programmer who really needs a string-like object they can modify, there is the ***bytearray***. This object is very string-like, but retains full mutability, much like a ***list***. You'll have to make a foray into the manual for more details, though.

Lists: They Aren't Just for Numbers any More

When we talked about lists, we were dealing with numbers. We talked about ***range(n)***, which gave us a list of *n* numbers. However, if we want to check a word to see if it's acceptable, we'll need to put the 'good' words somewhere. As it happens, lists can store pretty much anything, not just numbers. So, this is completely acceptable:

```
a=["this","is","a","list","of","words"]
```

Now, since we have a list of words, we can do things like ***a[0]***, which will return *"this"*, or ***a[-2]*** which is *"of"*. So, what do you think ***a[0][3]*** would be ? If it isn't obvious to you that it's *"s"*, try thinking about it as ***(a[0])[3]***. In other words, ***[3]*** is applied to ***a[0]***, which is *"this"*, so we get *"s"*.

It's also worth mentioning that while you can't modify a string, you can replace any string within a list with another string, such as, ***a[1]="was"*** which will replace *"is"* with *"was"* in the list above, or a[0]=a[0].upper() which will replace "this" with "THIS". In this situation, it is the list that's being changed, not the string.

Finally, let's say you want to take a list of strings and combine them into a single string. For example, let's say we want to turn ***a*** into a sentence with a space between each word. You already know enough to do it like this:

```
sentence=""
for s in a: sentence+=s+" "
print sentence
```

However, while this will work, it's really inefficient, since it creates a bunch of partial sentences before it's done, and it leaves a " " at the end, which we don't really need. Since this is something that you may need to do fairly often in certain types of programs, Python has a shortcut for this operation: the ***join()*** method. With this, the solution is much easier, and doesn't have the spurious " " at the end:

```
sentence=" ".join(a)
print(sentence)
```

join() will combine all of the strings in a list to make a new string. The string you call the method on (the " " at the beginning) is the spacer that's put in between words.

Section 2

What Words Can You Make ?

WORDGAMES

1. **Building words programmatically**
2. **How to read and write information to files on disk**

There are a lot of games which involve making words from random letters, some involve dice, some involve tiles. What they all share in common is a set of random letters.

Once again, when settling down to write a program to do something, the first thing we need to do is describe exactly what the program would do, step by step. Think of it as explaining the steps of the process to a six year old. You need to cover every step in detail.

So, in this section, the problem we want to solve is to take a list of N (could be 5 or 7 or whatever you like) letters, and rearrange them to produce any possible word. We would also like to know that what we have produced are real words, not just random strings of letters.

Break it Down

Now that we have a broad definition of the problem we want to solve we need to go into more detail. We'll start out with a simplified version of the problem. Say, we have 5 letters: A,B,C,D,E. In most games, the rule is that you can use each letter only once when you make words. We'll deal with that eventually, but let's start out without that restriction, just to make things easier. That is, we have an "A", so we can use "A" as many times as we like. Let's summarize the rules of our little "game":

We have 5 letters

Each letter can be used as many times as we like

Words will be exactly 5 letters long

Words must be real dictionary words

Now, let's consider how we would write such a program. We'll stick to a simple outline-style plan for our program:

```
1. Get a list of letters from the user
2. For each possible output word
    2.1. For each letter in the word
        2.1.1. Pick one of the letters from the list
    2.2.   Check to see if the created word is a real word
    2.3.   If it is, print it out
```

This is pretty straightforward, and we should already know all of the Python commands we need to accomplish it with one exception: task 2.2, "check to see if the created word is a real word". How are we going to accomplish this ?

Finding a Dictionary of Words

Obviously, if we want to tell if a word is a "real" word, we need to define what "real" means ! This isn't as obvious as it might seem. For example, most games exclude things like abbreviations and proper names, but some may not. Is "OK" a valid word ? How about "okay" ?

In the end, we need to separate the decision about what is and isn't a word from the program itself, and simply say that the user will provide a dictionary containing a list of all valid words. Where the dictionary comes from and what's in it is the user's problem. In writing the program we just read the words from the dictionary, and check each word we make against it.

Now, as a user, where would we get a dictionary from ? There is a popular web site which contains free, open-source software projects called SourceForge. One of the projects they host is a freely available dictionary useful for word games. Unfortunately, such things cannot always be relied on to remain around forever, but at the time I'm writing this, you can get a free dictionary file at: http://sourceforge.net/projects/scrabbledict. Go ahead and download this, or search the web for some other text file containing a dictionary. We'll need it in just a little while.

Reading and Writing Files

Now that we have our dictionary, we need to convince Python to take that information from the file on disk and turn it into a list of strings in Python. Dealing with files containing text isn't really all that difficult. It is important to reiterate that we're talking about text files. The same sort of files that contain your Python program. If you create a .doc file with a word processor, you don't have a text file, you have a word processor "document" file, which contains all sorts of other information in addition to the letters and numbers you typed. Such files are quite difficult to do anything with. As long as we stick with plain text files, our task is pretty straightforward.

We'll introduce three basic operations for now: opening files, reading from files and writing to files. There are a few more advanced things we can do, but we'll get to those in a later chapter. Even though all we need to do here is read from a file, we'll go ahead and introduce writing as well.

Before you can do anything with a file, you need to open the file. This is done with the aptly named ***open()*** function, which is built-in (no need to ***import*** anything). To open a file, we need the **path** to the file.

Here is an example:

```
myfile=open("twl.txt","rU")
words=[]
for word in myfile:
    words.append(word.strip())
```

This example, in fact, does everything we need for the wordgame programs we're going to write. The first line calls the ***open()*** function, which returns a file object (or prints an error if it can't fine or open the file). ***"twl.txt"*** is the name of the file we want to read from. In this case, the file needs to be in the same directory where you ran ***python***. If you want to open a file some other place on your computer, you'll need to provide the full path to the file, such as "/home/stevel/myfile.txt" or (on Windows) "C:/myfile.txt". Finally we have "rU". "r" means you want to open the file for reading rather than "w", which would open for writing. The "U" is a bit tricky to explain, but basically it deals with a subtle difference between text files created on Windows machines and Linux machines. If you're really interested, the Python manual will explain in more detail. Otherwise, just go ahead and add "U" when reading text files.

Once we have the file open, we create an empty list to fill with the words as we read them in. Then we use a ***for*** loop. While myfile is a file object, not a list, the ***for*** statement is smart enough to know that we want it to loop over the lines of text in our file rather than the items in a list. Each pass through the ***for*** loop will return the next line of the open file. Unfortunately, when it reads the line from the file, it also reads the character at the end of each line, which marks the end of the line. Once in a while you'll see this character, "\n" in a string. Try this:

```
print("This is\na test\nwith 3 lines")
```

So, if the first line in the file is "test", what we'll get in ***word*** is "test\n". To deal with this, we introduce the ***strip()*** method. This method will remove any whitespace (spaces, linefeeds, ...) from the beginning or end of a string.

The only other method we need to consider is ***append()***. This method, when applied to a list, will add a single new item to the end of the list.

So, after we run this program we will have a list of strings, each of which is one line from the file. Since any dictionary file you're likely to run into will have one word on each line, this will do exactly what we need for our wordgames.

Since we've learned how to read files, we might as well learn basic file writing as well. Writing to a file is exactly like reading from a file, but in reverse. Here's our example:

```
words=["a","list","of","words","to","file"]
outfile=open("words.txt","w")
for word in words:
    outfile.write(word+"\n")
```

As you might guess, this will create a new file (we opened it in "w") mode, and write a list of words, one per line. One important thing to notice here is the ***+"\n"***. When you ***print*** a string, it puts a "\n" at the end of the line for you. When you ***write()*** to a file, you have to do it yourself.

Danger Will Robinson!

When you open a file for writing with "w", if the file already exists, it completely erases the existing file and starts from scratch. There is no way to undo this if you do it by mistake. Your file will be gone forever. There are other options you can use which will allow you to modify an existing file without erasing it first. Very briefly (again the details are in the manual). If you use "a" instead of "w", then it will append anything you write to the end of an existing file. If you use "r+" it will open the file for reading (at the beginning of the file), but you can also ***write()***. If you do, it will overwrite the contents of the file starting at the current location for as many bytes of data as you ***write()***. The rest of the file will be unchanged. There are more advanced methods for moving around within files and reading and writing at arbitrary locations, but we're getting ahead of ourselves a bit here. If you forget the rest, just remember the warning about "w" erasing your files!

Back to the Word Game

Now that we know how to read words from a file, let's write the first simple version of our wordgame program as outlined at the beginning of the chapter. You can skip typing the '#' comment lines if you like, but remember to indent :

```
# Ask the user for letters until we get 5
letters=[]
while len(letters)!=5 :
    letters=input("Enter 5 letters :")

# Read the file containing good words
# remember twl.txt must be in the same folder
myfile=open("twl.txt","rU")
```

```
goodwords=[]
for word in myfile:
    goodwords.append(word.strip())

# one loop for each letter
words=[]
for l1 in letters:
    for l2 in letters:
        for l3 in letters:
            for l4 in letters:
                for l5 in letters:
                    word=l1+l2+l3+l4+l5
                    if word in goodwords :
                        words.append(word)

print(words)
```

Go ahead and try running this program, and give it a couple of different 5 letter sequences to start with. Assuming you used a decent sized file containing "good words", you may have noticed that this program can take a little while to run. We'll talk about that in a minute. First, though, let's look at one other little peculiarity. If you enter a word like "alpha", and look at the output "good" words, you'll see that each good word is replicated four times! Why did this happen ?

(Think about it yourself before reading on)

The reason isn't really very complicated. Since we have two a's in the list of letters, each a can be used once in each position, so any word with two a's in it will come out of our program four times. So, what can we do about this ?

One simple solution would be to simply make sure that we don't add words to the list if they're already there. If we change:

```
if word in goodwords :
```

to

```
if word in goodwords and word not in words :
```

we will immediately see that our duplicates are gone. Is this the best solution ? Well, that opens up a whole new area of discussion.

A World of Possibilities

We're going to take a little diversion from learning how to program, and talk a bit about philosophy.

In programming, the first priority is obviously having the program work properly. That is, accept input from the user and provide the correct output. Clearly a program that doesn't meet this requirement, be it by crashing, giving the incorrect answer, or simply being so confusing that the user cannot operate it, is not a useful program. Within that limitation, however, there is still a lot of flexibility. This raises a number of other concerns we can consider.

Speed

One important consideration for users is, of course, speed. If a program gives the correct answer, but takes two years to finish running, it's probably out of the 'useful' range. So, we would like our programs to be reasonably efficient. The process of taking a slow program and turning it into a fast program is known as optimization.

When optimizing a program, it is critical that our need for speed not obscure priority #1 (getting the correct answer). For this reason, in general, you should always write a working program first, using very straightforward logic. Only after you have the program working should you go back and consider ways to make it run faster.

Readability

Another concern when writing software is making the program easily readable. This doesn't only mean that someone else should be able to pick up your code and figure out how it works, but it also means that if you look at your own code again 2 years later, you should be able to figure out how it works. Again, this depends a bit on the purpose. If you are writing a program that you will use exactly one time, right now, to do something, then you're going to throw it away, you may not need to pay as much attention to this point. Then again, except for truly trivial programs, I generally hang on to any little bit of code that I write, in case I happen to need to do something similar in the future.

Elegance and Beauty

My, how dainty! Seriously, though, if you talk to seasoned programmers, they will often refer to code as "elegant" (or not). Unfortunately, as in art, this is a difficult term to define precisely. Or perhaps a better way to phrase it is "you'll know it when you see it". Most of the time, "elegant" code is code that is space-efficient, without being unreadable; having the shortest readable program that can accomplish the given task. However, even this is not a complete description. Consider, for a moment, this example from our program:

```
#Start by opening the file
myfile=open("twl.txt","rU")

#create an empty list to store the results
goodwords=[]

# loop over the lines in the file
for word in myfile:
    # add each word with whitespace removed
    goodwords.append(word.strip())
```

In Python there are many different ways to accomplish the same goal. This program is functionally equivalent to the code above:

```
wordfile=open("twl.txt","rU")
goodwords=[word.strip() for word in wordfile]
```

Now, how about even more dense (this is one line):

```
goodwords=[word.strip() for word in
    open("twl.txt","rU")]
```

Now, take a look at how far you can take this concept: http://preshing.com/20110926/high-resolution-mandelbrot-in-obfuscated-python/.

While such efforts as this final Mandelbrot set are amazing, and certainly qualify as art, "elegant" is really no longer the correct term. These have entered the realm of intentional obfuscation.

Balance in All Things

In the end, you must decide which of these three secondary factors should dominate your code. In scientific programming, the focus is often on speed, and that's probably appropriate in most cases. If elegance makes your program slower, it's probably best to skip it, even if it's clever. Readability is most critical in cases where you or someone else is likely to have to change the code down the road. You may not always be able to predict this, so it's probably a good idea not to obfuscate things too much.

Back to the Code !

After our brief philosophical detour, let's go back to our example program again (if you still remember it). As you may recall, the program was working when we left it, but it took a fairly long time to run. Let's take a look at some possible things we could do to to make the program run faster.

As it turns out, we've already learned everything we need to make the most productive speedup, and it's very simple. Simply change:

```
goodwords=[]
```

to

```
goodwords=set()
```

and

```
goodwords.append(word.strip())
```

to

```
goodwords.add(word.strip())
```

Now try running the script again. You should find that it runs about 50 times faster. Why does this happen ? The problem is (was) this line:

```
if word in goodwords:
```

When goodwords is a list, every time you ask whether word is in the list, it has to loop over all 178,000 words from the twl.txt file. Sets on the other hand don't work this way, and looking for a member in a set takes dramatically less time than looking for a member in a list. The longer the list/set is, the more profound the effect.

Once we've managed to achieve a 50x speedup and brought the time a program takes to run down to below 1 second, it probably isn't really necessary to keep going, unless you plan to run the program for 50,000 different 5 letter combinations or somesuch.

However, even though it may not really be worthwhile this is an excellent place to look at all of the different ways in which you can accomplish the same task.

If you recall a few pages ago we changed our program to avoid duplicates in the output list by adding an extra clause to our if statement. If you look carefully you'll notice that our output in this case is a list, when, once again, it would make a lot of sense if it were a set instead. So, we could change these three lines (they aren't together in the source):

```
words=[]
if word in goodwords and word not in words :
    words.append(word)
```

to

```
words=set()
if word in goodwords:
    words.add(word)
```

While this change should be satisfying from the perspective of elegance of coding style, if you measure the time the program takes to run, you will find that it made no significant difference, since there simply aren't enough items in ***words*** to have an effect.

If you'll recall back to chapter 3, when we introduced the concept of nested loops, we also introduced the concept of recursion. A recursive function was something that you could use to avoid nested loops, particularly in situations where the number of levels of nesting wasn't known in advance. In our simple little example, we know we have 5 letters in our word, so we know that we have to have one nested level for each letter. However, what if we wanted to allow the user to specify the number of letters in the output word? Now we have a problem. If we want to continue to write the program with nested loops, we'd have to write each possible set of loops up to some reasonable limit, then call the correct routine based on the user's selection. That would be the exact opposite of elegant coding style!

For the sake of completeness I will mention one other solution to this problem, without invoking the concept of recursion. **Self-modifying code**! This is one of the dirtiest phrases in the programmer's encyclopedia. Of all of the programming practices that you could come up with, that's the one that most seriously violates the readability criterion, and in some professional programming environments using it might be enough to get you fired. The basic idea is simple. You write a program which, itself writes a program, which then gets executed. There are a few situations where this practice may actually be worth the stigma associated with it, and I must confess, I did actually use it once (without getting fired) in the days when I programmed professionally. We'll introduce this concept briefly in Chapter 5.

A good solution to this problem involves recursive programming, but for our simple example here, it isn't really well motivated. So, let's go ahead and take the next step to write a program with some more reasonable game-playing uses.

The Scrabble Problem

We'll call this "the Scrabble problem", but it is basically the same for a whole host of other word games, like "Words with Friends" and other games of that ilk. The idea is that you have a set of letters (tiles) to work with, and you need to combine those letters to make valid words. In the case of Scrabble, you'd normally have 7 letters, and you might try and make words of any length with them (if you don't know how to play Scrabble, it might be worth looking it up). There is also the possibility that you're working off other letters on the board, so you might need to consider combinations of more than 7 letters at a time.

This is more complicated than our previous problem in a couple of ways: first, the words can be of any length, and second, we're only allowed to use each letter one time. If we want to make a word with two a's in it we have to have two a's in our list.

The code for reading in the list of valid words is going to be the same in this new program, so we can just leave that alone. However, the rest of the program will have to be written a little differently. If we wanted to do it the way we wrote the previous program, not only would we have to have our nested loop 7 layers deep, we would also have to add code at each layer of nesting to check to see if we had a valid (shorter than 7 letter) word. Very messy and definitely not elegant.

From a programming perspective, it is actually easier to write this program to simply go through the list of words and check to see if each one can be made from the list of available letters. So, let's write a function which checks to see if a single word can be made out of a provided list of letters:

```
def goodword(word,letters):
  tmp=list(letters)
  for letter in word:
    if letter in tmp:
       tmp.remove(letter)
       continue
    return False
  return True
```

This is actually a lot simpler than it might seem like the task should be. `tmp=list(letters)` makes a copy of the list of letters, so we can modify it without hurting the original list. We then go through each of the letters in the word to be tested, and see if we have the necessary "letter tile" available to make it. If we do, we remove that letter from the list so we don't re-use it. If we make it to the end of the word without a failure, then we have successfully formed the word, and we return `True`.

With this function, the rest of the program is pretty trivial. We just loop over all of the words in our list and check to see if each one is valid:

```
letters=input("Enter letters: ")
```

```
words=open("twl.txt","r")
for word in words:
  if goodword(word.strip(),letters) :
    print(word,end="")
```

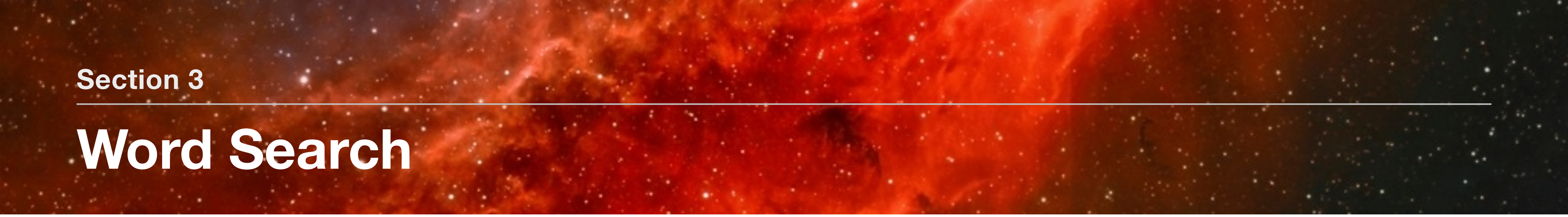
That's it, a fully fledged, "what words can I make from this set of letters" program!

Now, generally, longer words are better than short words, game-wise, so how about sorting our output so the longer words come out first. We'll use the same "goodword" function, but replace the body of the program:

```
words=open("twl.txt","r")
goodwords=[]
for word in words:
  if goodword(word.strip(),letters) :
    goodwords.append((len(word)-1,word.strip()))

for word in reversed(sorted(goodwords)):
 print(word[1])
```

This introduces a couple of new functions. Instead of just printing out any word we find is good, we add the word to a list of *goodwords*. However, rather than just making a list of the words, we make a list containing tuples. The first element of the tuple is the length of the word, and the second element is the word itself. Once we've done this, we can use the ***sorted()*** function to return a copy of the list sorted first by the length of the word, and then alphabetically by the word itself (when the length is the same). The behavior of ***sorted()*** is to return a sorted version of a list based on a number of common-sense rules. If you gave it a simple list of strings or numbers, it would sort the list alphabetically or numerically respectively. If the list contains tuples, it first sorts based on the first element of the tuple, then the second element as a tiebreaker, and so on. The behavior of ***reversed()*** should be pretty obvious.



Section 3

Word Search

LOREM IPSUM

1. **Lorem ipsum dolor sit amet**
2. **Consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.**
3. **Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.**
4. **Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur.**

Lorem ipsum dolor sit amet, ligula suspendisse nulla pretium, rhoncus tempor placerat fermentum, enim integer ad vestibulum volutpat. Nisl rhoncus turpis est, vel elit, congue wisi enim nunc ultricies sit, magna tincidunt. Maecenas aliquam maecenas ligula nostra, accumsan taciti. Sociis mauris in integer, a dolor netus non dui aliquet, sagittis felis sodales, dolor sociis mauris, vel eu libero cras. Interdum at. Eget habitasse elementum est, ipsum purus pede porttitor class, ut adipiscing, aliquet sed auctor, imperdiet arcu per diam dapibus libero duis. Enim eros in vel, lorem ispum volutpat nec pellentesque leo, temporibus scelerisque nec. Ac dolor ac adipiscing amet bibendum nullam, massa lacus molestie ut libero nec, diam et, pharetra sodales eget, feugiat ullamcorper id tempor eget id vitae. Mauris pretium eget aliquet, lectus tincidunt. Porttitor mollis imperdiet lorem ipsum libero senectus pulvinar.

Etiam molestie mauris ligula eget laoreet, vehicula eleifend. Repellat orci eget erat et, sem cum, ultricies sollicitudin amet eleifend dolor nullam erat, malesuada est leo ac. Varius natoque turpis elementum est. Massa lacus molestie ut libero nec, diam et, pharetra sodales eget, feugiat ullamcorper id tempor eget id vitae. Mauris pretium eget aliquet, lectus tincidunt. Porttitor mollis imperdiet libero senectus pulvinar. Etiam molestie mauris ligula eget laoreet, vehicula eleifend. Repellat orci eget erat et, sem cum.

Chapter 5

MATH

Lorem ipsum dolor sit amet, ligula suspendisse nulla pretium, rhoncus tempor placerat fermentum, enim integer ad vestibulum volutpat. Nisl rhoncus turpis est, vel elit, congue wisi enim nunc ultricies sit, magna tincidunt. Maecenas aliquam maecenas ligula nostra.

Section 1

Untitled

LOREM IPSUM

1. **Lorem ipsum dolor sit amet**
2. **Consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.**
3. **Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.**
4. **Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur.**

Integers, as you probably know, are numbers without any fractional part, ranging from -∞ to ∞. Floating point numbers on the other hand are numbers with fractional values, expressed on computers using decimal notation rather than fractions (in most cases). In Python and most other languages, if you do math with only integers, the result is also an integer, but if either value is floating point, the result is floating point.

Chapter 6

Making!

Lorem ipsum dolor sit amet, ligula suspendisse nulla pretium, rhoncus tempor placerat fermentum, enim integer ad vestibulum volutpat. Nisl rhoncus turpis est, vel elit, congue wisi enim nunc ultricies sit, magna tincidunt. Maecenas aliquam maecenas ligula nostra.

Back in the 80's, the term "hacker" was all over the place, and at the time it wasn't associated with the "black hats" who break into your computer to steal your identity, or the other ne're-do-wells who try to break into your computer. It was associated with creative kids who were trying to learn about technology and build new things. In the 2000's the term "hacker" developed SUCH negative press that the community of people who wanted to build things and revel in technology started calling themselves "makers". This was associated with the rise of the home 3-D printer market as well, so the transition was natural.

Ok, sorry for the historical interlude. What does this have to do with programming? One large branch of the Maker community likes to computerize things. This may be anything from little robots, to programmable light displays to building a remote control system for your home theater. How do you do this ? With "embedded computers". These are tiny little computers which use only a small amount of power, and are designed to attach to things in the real world. The two most popular types

Chapter 7

A Compilation of Concepts

This chapter reviews all of the concepts introduced in the earlier chapters. It's a useful reference, and can be referred to when you're learning the concepts to see additional examples.

Section 1

A Summary of Common Data Types

MAIN DATA TYPES IN PYTHON

1. **Integer**
2. **Floating Point Number**
3. **String**
4. **List / Tuple (immutable)**
5. **Dictionary**
6. **Set**

	Conversion Function	Mutable	Main Operators, Functions & Methods
Integer	int()	-	+, -, *, /, **, %
Floating Point Number	float()	-	+, -, *, /, **, % import math

String	str()	No	+, *, [a:b]
			len()
			strip(), split(), find(), rfind(), replace()

List	list()	Yes	+, *, [a:b]
			len(), sorted()
			append(), remove(), count(), sort(), reverse()

Tuple	tuple()	No	+, *, [a:b]
			len(), sorted()
			count(), index()

Dictionary	dict()	Yes	[key]
			len()
			keys(), values(), items()

Set	set()	Yes	I, &, -, ^
			len()
			add(), remove(), clear(), union(), intersection(), difference()

Integers

The normal mathematical operators: +, -, * can be used and behave as you would expect.

Division with fractional results is always rounded down to the nearest integer, so `199/100` = 1 and `-199/100` = -2.

The modulus operator, %, will produce the remainder from an integer division, such as `199%100` = 99. This is often used with cyclic behaviours, such as converting minutes to hours and minutes: hours=bigminutes/60, minutes=bigminutes%60

When mixing integers and floating point numbers, the result is floating point, but watch order of operations, such as: `3/2+1.0` = 2.0, NOT 2.5.

** is the exponent operator. ie `5**2` is 5*5 and `5**3` is 5*5*5.

Floating Point Numbers

Again, the usual mathematical operators behave as expected: +, -, *, /, **.

Floating point numbers must have a decimal point. `2.0` and `2.` (no trailing 0) are both floating point numbers, but `2` is an integer. Numbers can also be specified in scientific notation using e: `12.34e2` = 12.34×10^2 = 1234.0

Floating point numbers in Python are 'double precision', meaning they have roughly 15 digits of precision. This is enough for most purposes, but you could run into a case where this produces unexpected behavior. For example, `1.0+1.0e-16` is exactly `1.0`.

The transcendental functions such as `sqrt()`, `pow()`, `sin()`, `cos()`, etc. exist in the math library. To use them either `import math`, and use the functions as `math.sqrt()`, etc. or `from math import *` and make them all available without the math prefix. `help(math)` will give a complete list of available functions.

Strings

Strings are surrounded either by double-quotes: " or single quotes: '. For strings that span multiple lines, """ or ''' can be used.

Examples of common methods:

```
"alphabet"[0] → "a"
"alphabet"[:5] → "alpha"
len("abracadabra") → 11
" test ".strip() → "test"
"abc,def,ghi".split(",") → ["abc","def","ghi"]
"abcdef".find("de") → 3
"abcdefabc".rfind("ab") → 6
"this is".replace("i","0") → 'th0s 0s'
"alphabet".count("a") → 2
```

Prefixing a string with u makes the string Unicode (a way of encoding foreign characters and special symbols). In Python 3, all strings are Unicode by default.

Prefixing a string with r makes a raw string: `"abc\n"` has a newline at the end, `r"abc\n"` has a literal \n (2 characters) at the end.

Lists & Tuples

Lists are defined with comma separated values inside square brackets: `[ 1, 2, 3 ]`. Tuples are identical, except use parentheses: `( 1, 2, 3 )` and are immutable (cannot be changed once created). List methods which modify the list will not work on tuples. Addition and multiplication can be used on lists and tuples.

Examples of common methods:

```
[4,5,6][1] → 5
[1,2,"abc",2.0][2:3] → ["abc",2.0]
sorted([2,1,4,3]) → [1,2,3,4]
len((1,2,3)) → 3
x=[1,2,3]
x.append(4) → x=[1,2,3,4]
x.extend([1,2]) → x=[1,2,3,4,1,2]
x.sort() → [1,1,2,2,3,4]
x.index(2) → 2
del x[2] → x=[1,1,2,3,4]
list("abcd") → ["a","b","c","d"]
```

Dictionaries

A dictionary defines relationships between pairs of objects, referred to as keys and values. Values may be of any type. Keys must be immutable. This is one reason why strings are immutable in Python. While seldom useful it also means that tuples may be used as keys. The keys and values in a dictionary are unordered.

Examples of common methods:

```
x={1:2,"a":"b",4:"q"}
x[1] → 2
x["a"] → "b"
x["cc"]=5.5 → x={'a':'b',1:2,4:'q','cc':5.5}
x.keys() → ["a",1,4,"cc"]
x.values() → ["b",2,"q",5.5]
x.items() → [('a','b'),(1,2),(4,'q'),('cc',5.5)]
x.setdefault(3,5) → 5 and sets x[3]=5
x.setdefault(1,5) → 2
```

Sets

A set is an unordered group of unique items. Adding an item to a set that's already in the set is legal but has no effect. Operations exist for computing unions, intersections, etc. very efficiently. Sets are usually initialized from a list or tuple.

Examples of common methods:

```
x=set((1,2,4,7,9,4)) → x=set([1,2,4,7,9])
y=set((2,3,5,7,10)) → y=set([2,4,5,7,10])
x.union(y) → set([1, 2, 3, 4, 5, 7, 9, 10])
x.intersection(y) → set([2, 7])
x.difference(y) → set([1, 4, 9])
x.add(3) → x=set([1,2,3,4,7,9])
x-set((2,3,4)) → x=set([1,7,9])
```

Miscellany

“HI” program

```
from turtle import *

Turtle()

a=[90,180,90,90,180,90,0,180,-90,90,180]

fdr=[100,50,50,50,100,40,50,25,100,25,50]

ht()

clear()

for i in range(11):

    left(a[i])

    forward(fdr[i])
```

Acting by side-effect

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.

Related Glossary Terms

Drag related terms here

Index

Find Term

Chapter 3 - Making A New Sudoku

Boolean Operators

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.

Related Glossary Terms

Drag related terms here

Index

Find Term

Chapter 2 - Random Walk

Comment

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.

Related Glossary Terms

Drag related terms here

Index

Find Term

Chapter 2 - Spirograph Example

Compiler

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.

Related Glossary Terms

Drag related terms here

Index

Find Term

Chapter 1 - Installing Python

Complex math

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.

Related Glossary Terms

Drag related terms here

Index

Find Term

Chapter 2 - Spirograph Example

CPU

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.

Related Glossary Terms

Drag related terms here

Index

Find Term

Chapter 1 - Installing Python

Executable

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.

Related Glossary Terms

Drag related terms here

Index Find Term

Chapter 1 - Installing Python

Floating point

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.

Related Glossary Terms

Drag related terms here

Index Find Term

Flowcharting

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.

Related Glossary Terms

Drag related terms here

Index Find Term

Function

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.

Related Glossary Terms

Drag related terms here

Index Find Term

Chapter 3 - Making A New Sudoku

Inner loop

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.

Related Glossary Terms

Drag related terms here

Index

Find Term

Chapter 3 - Nesting and Recursion

Interactive mode

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.

Related Glossary Terms

Drag related terms here

Index Find Term

Interpreted language

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.

Related Glossary Terms

Drag related terms here

Index Find Term

Iterate

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.

Related Glossary Terms

Drag related terms here

Index

Find Term

Chapter 2 - Spirograph Example

List

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.

Related Glossary Terms

Drag related terms here

Index Find Term

Chapter 2 - Spirograph Example

Loop

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.

Related Glossary Terms

Drag related terms here

Index

Find Term

Chapter 2 - Spirograph Example

Modules

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.

Related Glossary Terms

Drag related terms here

Index

Find Term

Chapter 1 - Taking Python Out for a Spin

Nested loops

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.

Related Glossary Terms

Drag related terms here

Index Find Term

Chapter 3 - Nesting and Recursion

Operators

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.

Related Glossary Terms

Drag related terms here

Index

Find Term

Chapter 1 - Taking Python Out for a Spin

Outer loop

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.

Related Glossary Terms

Drag related terms here

Index

Find Term

Chapter 3 - Nesting and Recursion

Path

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.

Related Glossary Terms

Drag related terms here

Index

Find Term

Chapter 4 - What Words Can You Make ?

Recursion

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.

Related Glossary Terms

Drag related terms here

Index Find Term

Chapter 3 - Nesting and Recursion

Scripting language

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.

Related Glossary Terms

Drag related terms here

Index

Find Term

Chapter 1 - Installing Python

Self-modifying code

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.

Related Glossary Terms

Drag related terms here

Index Find Term

String

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.

Related Glossary Terms

Drag related terms here

Index Find Term

Terminal window

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.

Related Glossary Terms

Drag related terms here

Index

Find Term

Chapter 1 - Installing Python