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 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, 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 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, 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. If you're feeling really ambitious and happen to have (or want to buy) a LEGO Mindstorms set, there is even a project to let the Python turtle module control a LEGO turtle (http://code.google.com/p/nxturtle).

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 either, 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 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 statement 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 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's up to you. 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 Pythc code more readable than a lot of C++ or Java code.

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()
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()
```

Section 2

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)
```

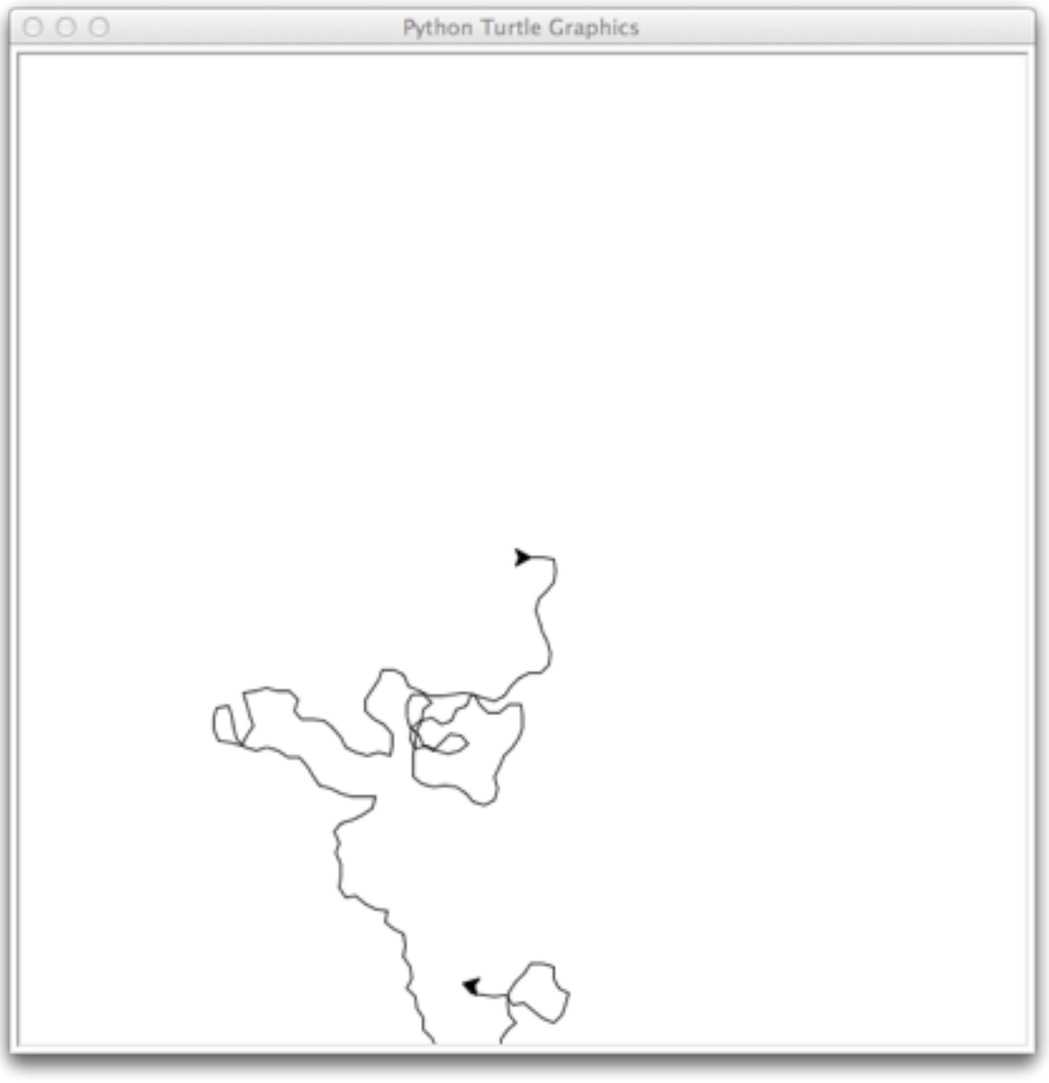
As you'll see, this program will print 10 random numbers between 1 and 100 (possibly including 100). If you run the program again, you'll get a different list of numbers each time. There are a number of other functions available within the random module as well, for example, `random.uniform(1,100)` will 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:

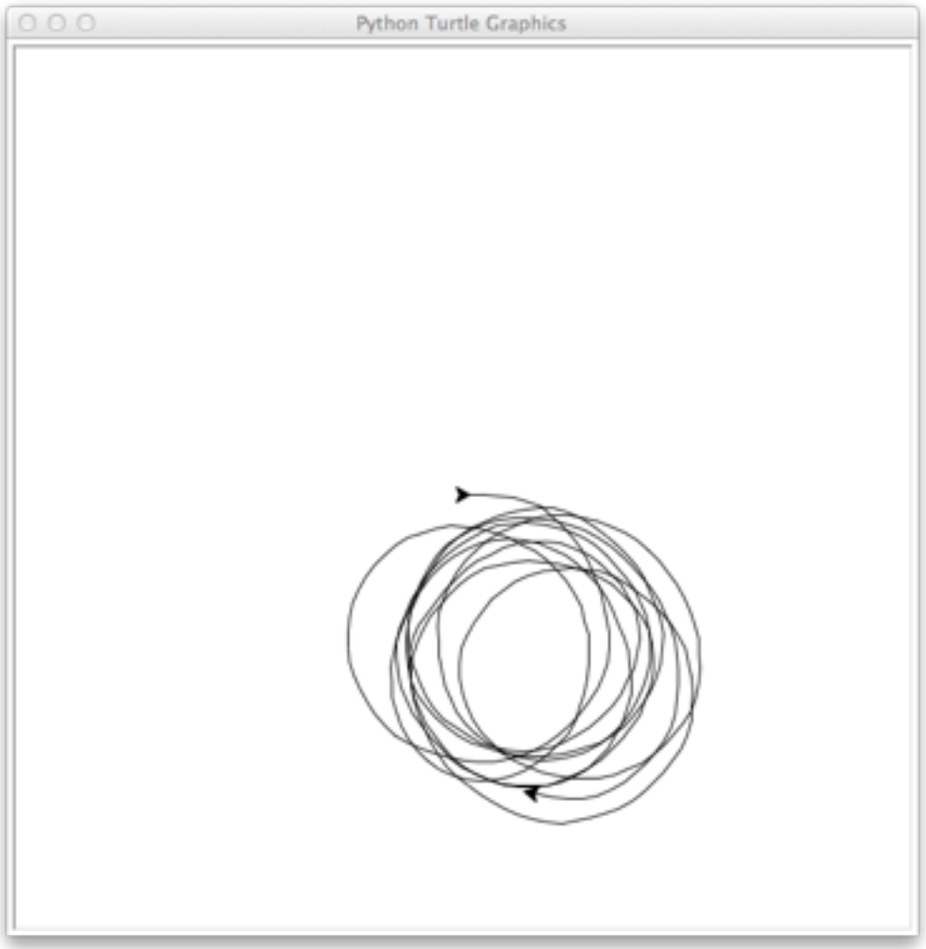
```
reset()
for i in range(36):
    forward(10)
    right(10)
```

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))
```

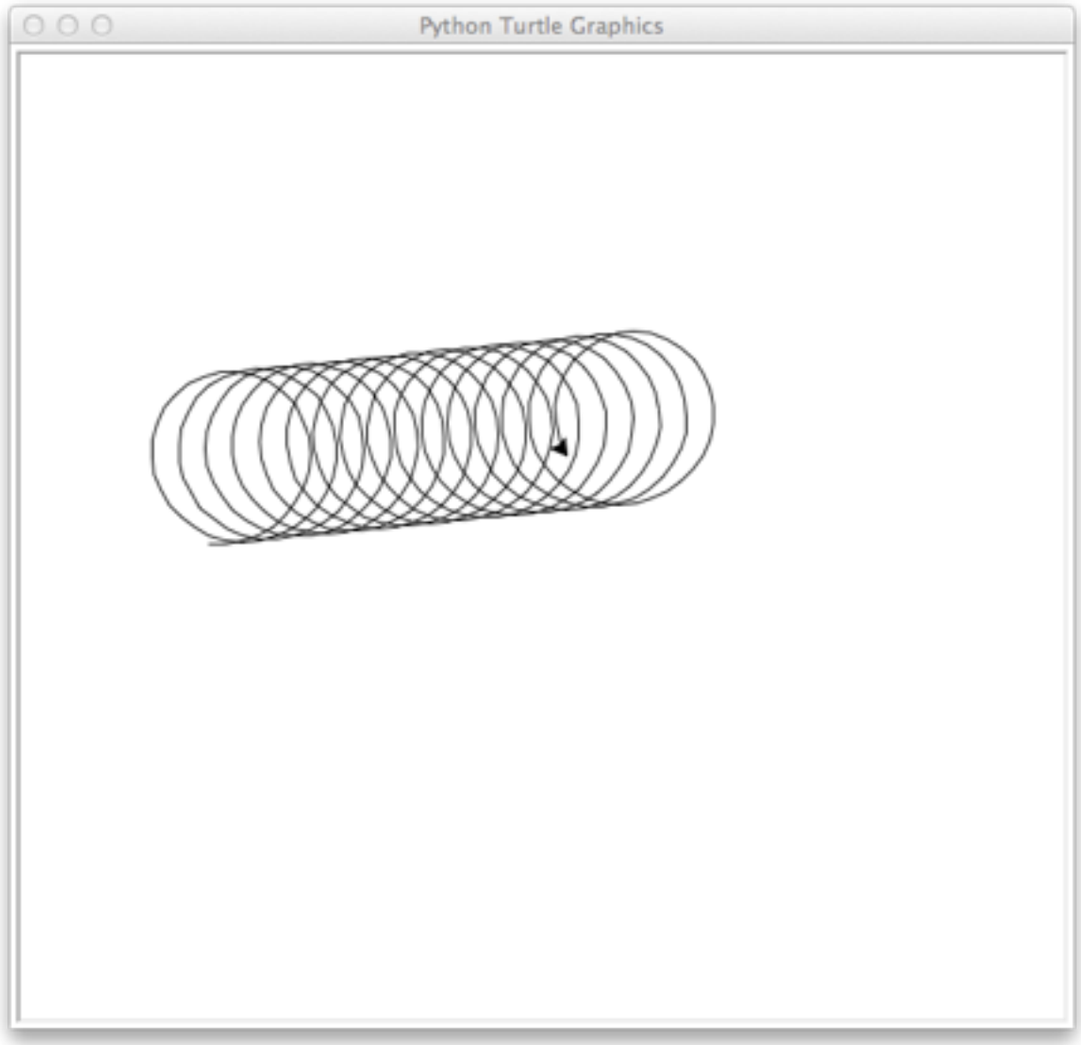
Kind of like scribbling circles with a pencil. A little different each time, but vaguely circular. You can play with the parameters inside `gauss()` and see what effects you can achieve.

Example 2.2 Random circles




```
goto(-200,0)
clear()
for i in range(500):
    forward(10)
    left(0.2-cos(heading())/50.0)
```

You should see this:



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()
```

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)