

Exploring Math Through Computer Programming

By Peter Farrell

Copyright © 2015 Peter Farrell

All rights reserved.

ISBN-10: 1508656940

ISBN-13: 978-1508656944

Table of Contents




1. Introduction to Programming.....	5
Tools.....	5
Python	5
Installation:.....	6
The Turtle Module.....	7
Using the Turtle Module.....	8
Using Loops.....	10
Defining Functions	12
Using Variables.....	13
Loops and Printing	18
While loops	19
2. Arithmetic	22
Functions.....	22
Conditionals and Input.....	23
Conditionals.....	23
User Input.....	24
The random module	25
Lists.....	27
The Modulo Operator	29
3. Algebra.....	Error! Bookmark not defined.
Solving Higher-Degree Equations.....	Error! Bookmark not defined.
Major Math Tool: Create Your Own Grapher.....	Error! Bookmark not defined.
Synthetic Division.....	Error! Bookmark not defined.
Exploring Prime Numbers	Error! Bookmark not defined.
Binary Numbers.....	Error! Bookmark not defined.
4. Geometry.....	Error! Bookmark not defined.
Finding intersections of lines	Error! Bookmark not defined.
Finding Area of a Triangle Using Heron's Formula	Error! Bookmark not defined.
Finding Lines Through Points.....	Error! Bookmark not defined.
Perpendicular Bisector.....	Error! Bookmark not defined.
Distance from a point to a line	Error! Bookmark not defined.
Circumcircle	Error! Bookmark not defined.
Centroid.....	Error! Bookmark not defined.
5. Trigonometry.....	Error! Bookmark not defined.

Sines and Cosines.....	Error! Bookmark not defined.
Harmonographs	Error! Bookmark not defined.
Spirograph	Error! Bookmark not defined.
Radians	Error! Bookmark not defined.
Visual Python	Error! Bookmark not defined.
Vectors	Error! Bookmark not defined.
The Solar System Model	Error! Bookmark not defined.
6. Recursion and Fractals.....	Error! Bookmark not defined.
Fractal Trees.....	Error! Bookmark not defined.
Pythagorean Tree	Error! Bookmark not defined.
Koch Snowflake	Error! Bookmark not defined.
Sierpinski Triangle.....	Error! Bookmark not defined.
7. Matrices.....	Error! Bookmark not defined.
Entering Matrices as Lists	Error! Bookmark not defined.
Transforming Points using Matrices.....	Error! Bookmark not defined.
Multiplying Matrices	Error! Bookmark not defined.
Rotation matrices	Error! Bookmark not defined.
8. Series.....	Error! Bookmark not defined.
Iteration.....	Error! Bookmark not defined.
9. Complex Numbers	Error! Bookmark not defined.
Polar Form	Error! Bookmark not defined.
DeMoivre's Theorem	Error! Bookmark not defined.
Graphing Complex Multiplication	Error! Bookmark not defined.
Mandelbrot Set.....	Error! Bookmark not defined.
10. Calculus	Error! Bookmark not defined.
Derivatives	Error! Bookmark not defined.
Newton's Method	Error! Bookmark not defined.
Integrals.....	Error! Bookmark not defined.
Numerical Integration	Error! Bookmark not defined.
The Trapezoidal Method.....	Error! Bookmark not defined.
Differential Equations.....	Error! Bookmark not defined.
The Runge Kutta Method.....	Error! Bookmark not defined.
Conclusion	Error! Bookmark not defined.
Answers to Exercises	Error! Bookmark not defined.

1. Introduction to Programming

Tools

In the popular video game Minecraft, you're "spawned" into a random world with no instructions. You need to make yourself a shelter and some tools. First you gather wood blocks to make a crafting table and wood tools. With the wood tools you can gather stone to craft harder stone tools. With stone tools you can mine iron ore to craft even harder iron tools.

		
Wooden Block and pickaxe	Stone Block and Pickaxe	Iron Ore and Pickaxe

It goes on like this: you use the tools you have to make bigger, more effective tools. Once you've made stone tools, there's no reason to go back to wooden tools!

Math is like that, too. You start out learning the tools of arithmetic and build on those to create algebraic tools and so on. For centuries this "tool-making" was simply figurative or psychological, but with the availability of free computer applications, we can literally (or virtually, anyway) craft tools that will help us explore higher and higher math topics.

I'm not talking about using calculators or computers to avoid doing math. I'm talking about writing programs to avoid excessive repetition of tasks. These can be simple two-line functions to return the average of two numbers, or much more complicated programs to draw graphs or model 3D situations.

Python

The folks who developed the Python programming language have given us a bunch of pre-loaded tools we can use to create our personal tools:

- Loops
- Conditionals
- Variables
- Functions
- Lists
- Classes

The difference between this and Minecraft is that there are plenty of real-world applications of learning to program, even if you start by making turtles walk around.

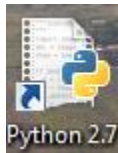
Another difference is that there are programmers and developers out there right now creating even newer tools you can use if you find them, download them and read their documentation. Once you master the Python tools listed above you can simply “import” other packages and use all the tools they’ve made for you!

Installation:

(If you already have Python on your computer, skip this section)

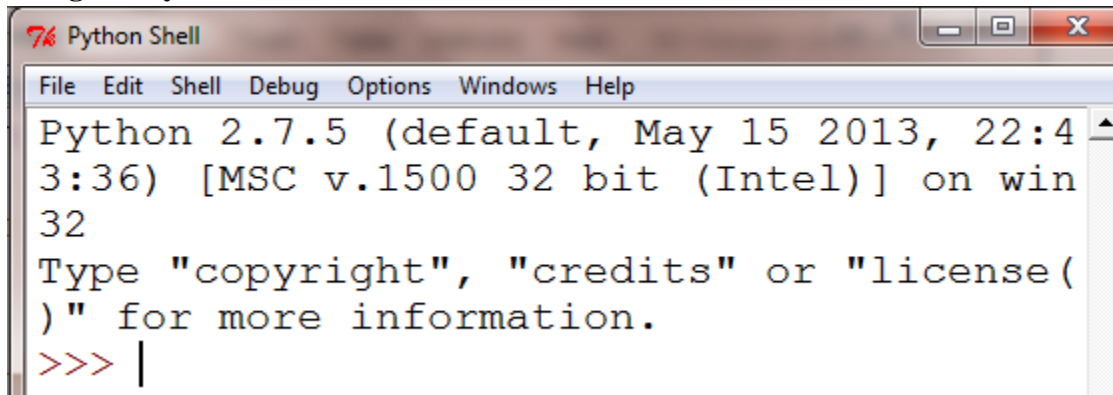
To download Python, go to python.org and select the kind of operating system you’re running. I’m going to use Python 2.7. It gives you an installer file you can run and just follow the directions to install it. Pretty easy!

Now when you click the icon for Python



or IDLE its editor, you get an interactive “shell.”

Using the Python Shell

A screenshot of a Windows-style window titled "Python Shell". The window has a menu bar with "File", "Edit", "Shell", "Debug", "Options", "Windows", and "Help". The main content area shows the following text: "Python 2.7.5 (default, May 15 2013, 22:43:36) [MSC v.1500 32 bit (Intel)] on win32", followed by "Type 'copyright', 'credits' or 'license()' for more information.", and finally the prompt ">>> |". The prompt consists of three red greater-than signs followed by a vertical bar.

Notice the little carrots (>>>). They’re called the “prompt.” You can use the shell as a calculator:

```
>>> 12 + 45
57
>>> 97 - 53
44
```

Multiplication is an asterisk.

```
>>> 65 * 33
2145
```

Two asterisks mean an exponent:

```
>>> 5**3
125
```

Forward slash means division:

```
>>> 100 / 12
8
```

Wait a minute! 12 doesn't go evenly into 100. One thing to remember about Python 2 is it assumes you want only integers unless you tell it otherwise. One way to tell it is by using a decimal point somewhere in your sum:

```
>>> 100./12
8.333333333333334
```

We'll need to remember that when we're dividing in our programs. Anyway, you can even write code in the shell. The first thing most people learn is how to **print** some text: in Python 2 it couldn't be easier. Type:

```
>>> print "hello world!" (Type along with all of these examples)
```

and press enter. It prints what you told it to print! Congratulations. You just ran your first line of code!

The Turtle Module

In the 1960s MIT was interested in creating a programming language that would be easy to use, even for children. This eventually culminated in the Logo computer language, as discussed in Seymour Papert's book *Mindstorms*. I've used the StarLogo and NetLogo applications. The creators of Python thought the Logo turtles were important enough to include a turtle module in Python.

A **module** in Python is a file with code in it. The "turtle module" is a file called "**turtle.py**" (all Python files need the .py extension), and it's somewhere in the Python folders when you

download the language. turtle.py contains all the functions you can use to make your turtles move around and make cool graphics!

Using the Turtle Module

Importing the turtle module is easy. If you type

```
import turtle
```

at the beginning of a program, you can call the functions in the module by typing “turtle” before every command, like “turtle.fd(10)”. You can shorten this by importing the module this way:

```
import turtle as t
```

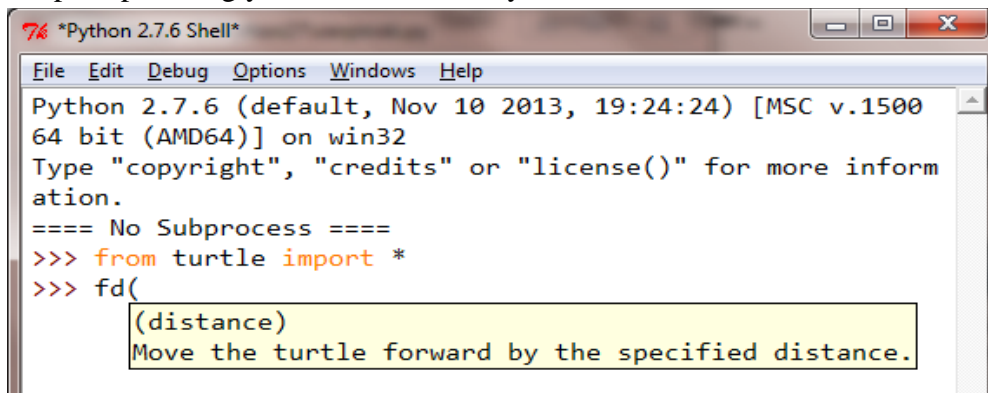
Then you just have to type t before every command, like “t.fd(10).”

Personally, I prefer typing less rather than more. **My recommendation** is to import the turtle module this way:

```
from turtle import *
```

and then you don’t have to type anything else before your turtle commands.

The great thing about Python is that it gives you prompts: messages to show you’ve typed the command in the right way. For example, in the Python shell, when you type this code, you see the prompt asking you for the distance you want the turtle to move forward.

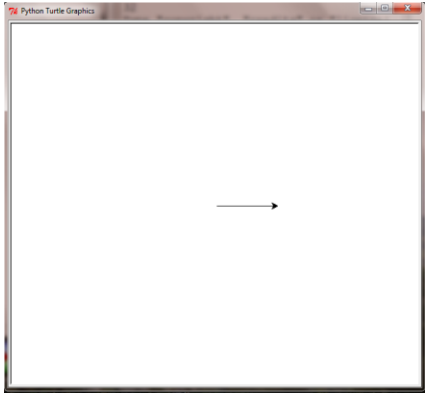


```
*Python 2.7.6 Shell*
File Edit Debug Options Windows Help
Python 2.7.6 (default, Nov 10 2013, 19:24:24) [MSC v.1500
64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more inform
ation.
==== No Subprocess ====
>>> from turtle import *
>>> fd(
(distance)
Move the turtle forward by the specified distance.
```

Type in a value and press ENTER.

```
>>> fd(100)
```

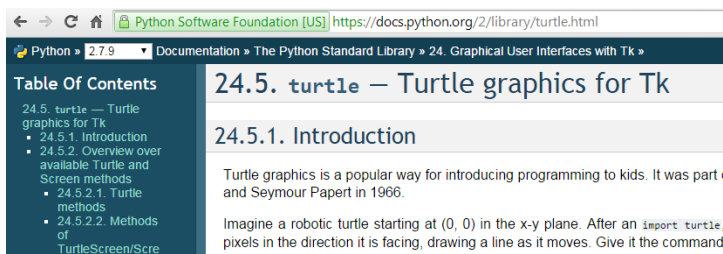
The Turtle Graphics window should pop up with your turtle and the path it leaves from walking forward 100 steps.



If you want to clear your turtle graphics, simply type

```
>>> clear()
```

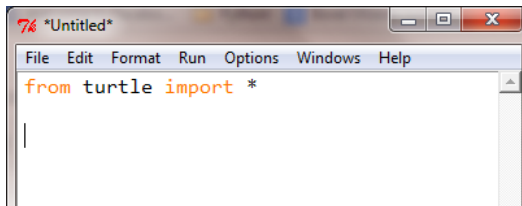
into the shell and the turtle path will be erased (but the turtle will still be standing there).



All the commands you'll need can be found in the Python.org docs, under the turtle module. Just Google "Python turtle" and it should be the first one that pops up.

Making Turtles Move

If we're going to do any serious programming we have to leave the interactive shell and create permanent Python files we can save.



Open a **new window** by pressing the CTRL key and N at the same time. Notice there's no prompt or carrots. This is called a **module window** and we can save modules and run them again later or share them with other people.

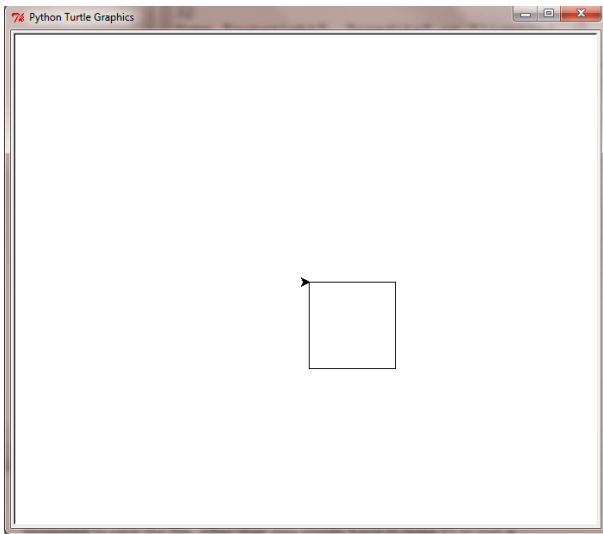
The first time you run a file (using the "Run" dropdown at the top of IDLE), you'll be prompted to save the file. Name it something like "turtle1.py" because there is already a "turtle.py." After that, you simply have to press F5 to run a program. And you can store all the programs we write for our turtles in "turtle1.py."

How to Make a Square

```
from turtle import *
```

```
fd(100)
rt(90)
fd(100)
rt(90)
fd(100)
rt(90)
fd(100)
```

Press F5 or click “Run” and you’ll see a square in the turtle window:



Using Loops

You could do the same thing with less typing by using a **loop**, a very important tool that will show up in nearly every computer program. Every language has its own way of making the computer repeat a block of code a certain number of times, and in Python it’s

```
for i in range(4):
```

to make it repeat 4 times. *i* is a variable you can use in the code (and we will use it soon) but for right now all you need to know is that’s how to create a repeat loop.

To repeat a block of code 5 times, the code is

```
for i in range(5):
```

To repeat a block of code 10 times, it’s

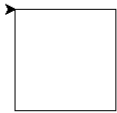
```
for i in range(10):
```

and so on.

When you type a colon and press ENTER, Python automatically indents the next line (or lines) for you. The indented block will be what's repeated four times. The indenting is really important! If your code isn't indented correctly, it won't work properly. So here's our loop:

```
for i in range(4):  
    fd(100)  
    rt(90)
```

The outcome (when we run it) is a square.



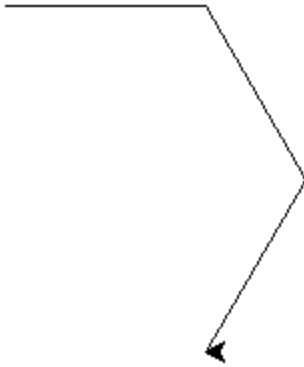
Congratulations! Your first Python program! Play around with the length to make different squares, and change the angle to make other shapes.

Making a Triangle

Changing your code from drawing a square to drawing a triangle is a good exercise in geometry. Obviously instead of repeating the code 4 times you repeat it 3 times for a triangle, since it only has 3 sides. But how many degrees do you turn to make a triangle? The simple answer is 60, but that's the internal angle. If you adapt your square code like this:

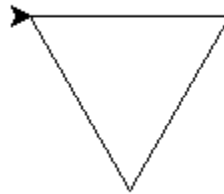
```
for i in range(3):  
    fd(100)  
    rt(60)
```

here's what it looks like in the Turtle Graphics Window:



What the turtle does is turn the **external** angle, not the internal angle. So to draw a triangle, you have to make the turtle turn 120 degrees each time.

```
for i in range(3):  
    fd(100)  
    rt(120)
```



Defining Functions

A good way to organize your code is to name a set of commands, so you can call them easily:

```
def square():  
    for i in range(4):  
        fd(100)  
        rt(90)
```

“def” means **define** the function

Notice that everything is indented inside the `square` function, and everything inside the loop is indented again. The good news is IDLE automatically indents for you after you type a colon.

Now we can just type
>>> `square()`

in the shell and the turtle will draw a square. **Functions** are another important programming tool you'll use in every program, no matter what language you use! We can even call the `square`

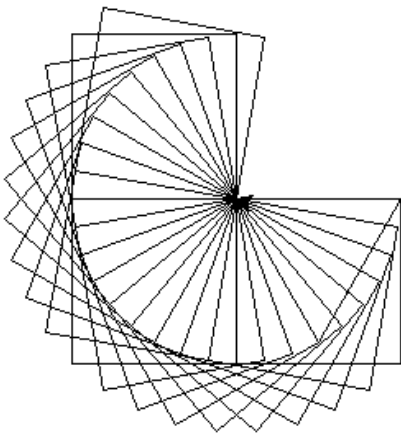
function inside another function, like this:

```
def square_thing():  
    for i in range(20):  
        square()  
        rt(10)
```

all the “square” code will automatically be run in the new function. In this case, the turtle will make a square, then turn right 10 degrees, and repeat that 20 times. Run it and type

```
>>> square_thing()
```

into the Python shell and press Enter. The figure should look like this:



How would you make a circle?

```
def circle():  
    for i in range(360):  
        fd(1)  
        rt(1)
```

Using Variables

Variables are another very useful tool. You replace numbers with a letter or a word and then you can change every instance of the variable at once.

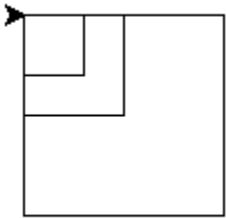
Program: square with side length side

Change your “square” function to:

```
def square(length):  
    for i in range(4):
```

```
fd(length)
rt(90)
```

Now you can draw squares of any length you want:

<pre>>>> square(30) >>> square(50) >>> square(100)</pre>	
---	--

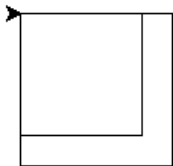
Whatever you put in the parentheses will be saved to the “length” variable and applied wherever that variable is in the function. Unfortunately, if you don’t specify a length from now on, you’ll get an error message because the program is looking for a length.

```
>>> square()
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    square()
TypeError: square() takes exactly 1 argument (0 given)
```

I read the last line of the errors first. It’s saying the square function expects to be told the length. In programming lingo, it “takes 1 argument.” We’ll write functions that take more than one argument later. There’s a way to tell Python, “if a user doesn’t tell you a length, just use 100,” for example. Just change the parameter (what’s in the parentheses) to:

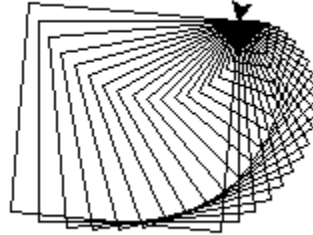
```
def square(length = 100):
```

Now if you specify a length, it will draw a square with that length. If you leave the parentheses empty, then the length will default to 100.

<pre>>>> clear() >>> square() >>> square(80)</pre>	
---	--

Program: Spiral

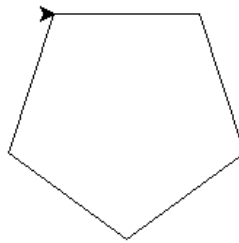
```
def spiral(times):  
    length = 10  
    for i in range(times):  
        square(length)  
        rt(5)  
        length = length + 5
```



Polygon Function

What about a polygon with more sides, like a pentagon or hexagon? We have to think like a turtle. We start at the center, facing a certain direction. We turn a certain number of times and end up at the same spot, facing our original direction. How many degrees have we turned? 360. So each turn will be 360 divided by the number of turns. The number of degrees we turn for a pentagon will be $360 / 5$, which is 72. So our pentagon program will look like this:

```
for i in range(5):  
    fd(100)  
    rt(72)
```



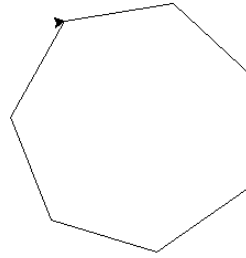
To draw a regular polygon with n sides, how many degrees should the turtle turn?

$$360/n$$

Telling the turtle how to draw a polygon depends on how many sides you want. This is called the “parameter” of a function. Let’s return to our pentagon program and let the user choose how many sides they want. I’m changing the name to “polygon”:

```
def polygon(sides):
    for i in range(sides):
        fd(100)
        rt(360./sides)
```

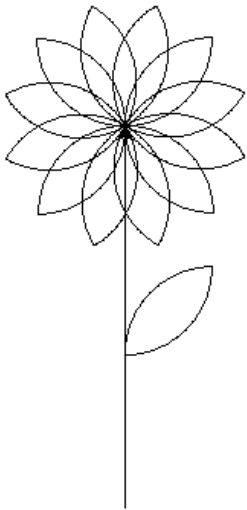
And at the prompt:
>>> polygon(7)



Using subroutines

Program: Flower

How would you make a complicated design like this flower?



First you learn to make a petal by putting together two quartercircles. Do you remember how to make a circle?

```
def circle():
    for i in range(360):
        fd(1)
        rt(1)
```

How would you change that code to make a quartercircle? Think about it.

```
def quartercircle():
    for i in range(90):
        fd(1)
        rt(1)
```

Then you can make a “petal” function composed of two quartercircles. Experiment with the angle between the quartercircles before peeking at the code!

```
def petal():
    for i in range(2):
        quartercircle()
        rt(90)
```

The flowerhead is made up of twelve petals. So you would make a flowerhead function and then finally put them all together into a flower function. The great news is all you have to do is execute the flower function and all the subroutines will run automatically!


```

def flowerhead():
    for i in range(12):
        petal()
        rt(30)

def flower():
    speed(0)
    setpos(0,-200) #lower on screen
    setheading(90) #straight up
    fd(100)
    petal()
    fd(150)
    flowerhead()

```

By the way, did you notice I wrote little notes in the program? It's a good idea to write "**comments**" to explain your code, and that's why every programming language has a syntax for comments. In Python, it's the hashtag (#). Anything on the line after the hashtag will be ignored by the computer. It's only for humans to read!

```

>>> print 3
3
>>> #print 3
(Nothing was printed)

>>> print 3,'hello'
3 hello
>>> print 3#,'hello'
3

```

By now you've worked with 3 of the major tools of Programming:

Loops
Variables
Functions

In a future section we'll be doing much more with turtle graphics including graphing, fractals and differential equations!

Let's take a break from turtles for a moment and learn some more tools to help explore the world of math.

Loops and Printing

You might not always be working with turtles, but very often you'll be printing something, even to check your programs. Here's how to print.

Program: simple print

```
>>> print "Hello World!"  
Hello World!
```

In Python 3, you have to put everything you're printing in parentheses:

```
>>> print ("Hello World!")  
Hello World!
```

Program: print strings using variables

<pre>def print3(): for i in range(10): print i</pre>	<pre>>>> print3() 0 1 2 3 4 5 6 7 8 9</pre>
--	--

It printed 10 numbers, but it starts at 0. If you want to print starting with 1, you need to start the range with 1, and end it with 11:

<pre>def print3(): for i in range(1,11): print i</pre>	<pre>>>> print3() 1 2 3 4 5 6 7 8 9 10</pre>
--	---

Program: print strings using loops

<pre>def print3(): for i in range(4): print 2*i + 1</pre>	<pre>>>> print3() 1 3 5 7</pre>
---	--

Program: print numbers up to maxnum

<pre>def print1(maxnum): for i in range(maxnum): print i</pre>	<pre>>>> print1(10) 0 1 2 3 4 5 6 7 8 9</pre>
--	--

Remember Python (and computers in general) starts counting with 0, not 1. To print the numbers from 1 to maxnum, including maxnum, you'd have to change the range:

<pre>def print1(maxnum): for i in range(1,maxnum+1): print i</pre>	<pre>>>> print1(10) 1 2 3 4 5 6 7 8 9 10</pre>
--	---

While loops

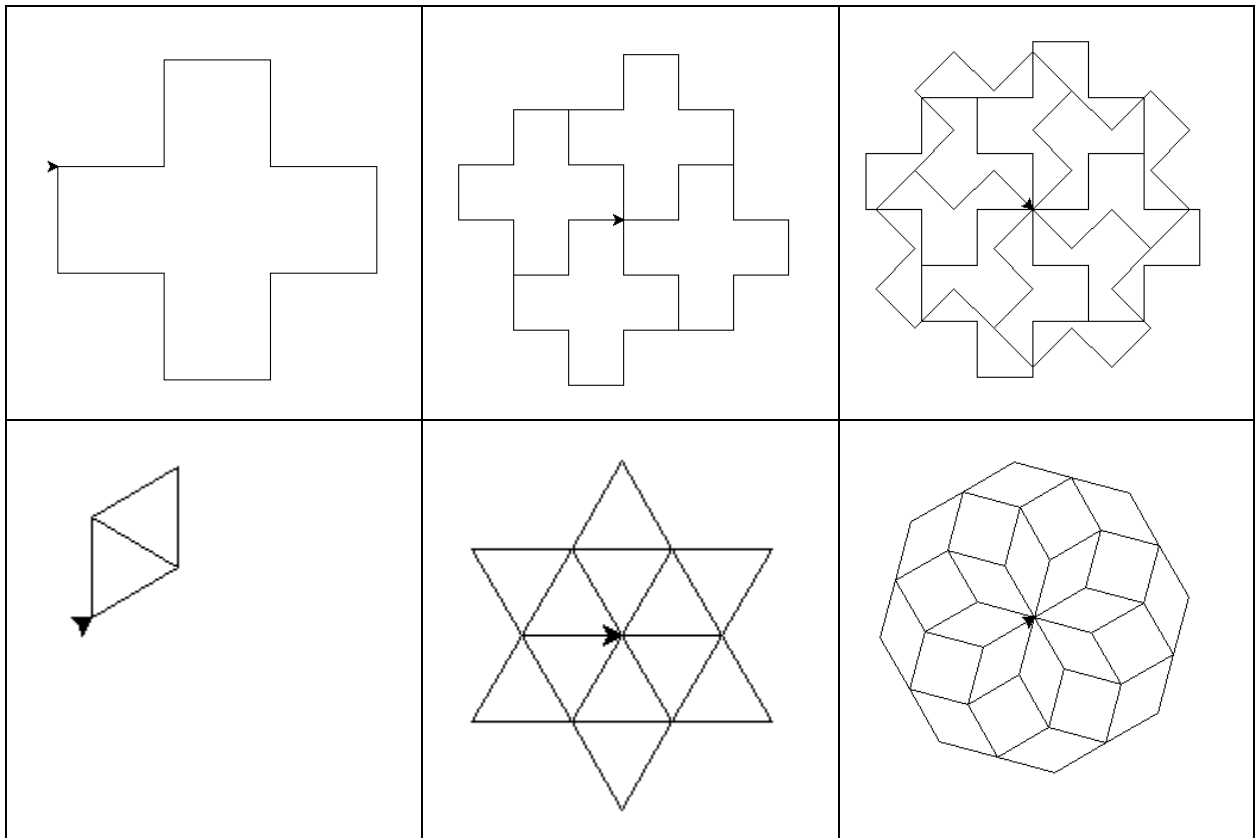
There's also a way to make the program keep doing something while a certain condition is true. For example, keep printing numbers while they're less than or equal to maxnum. You could use a **while loop**:

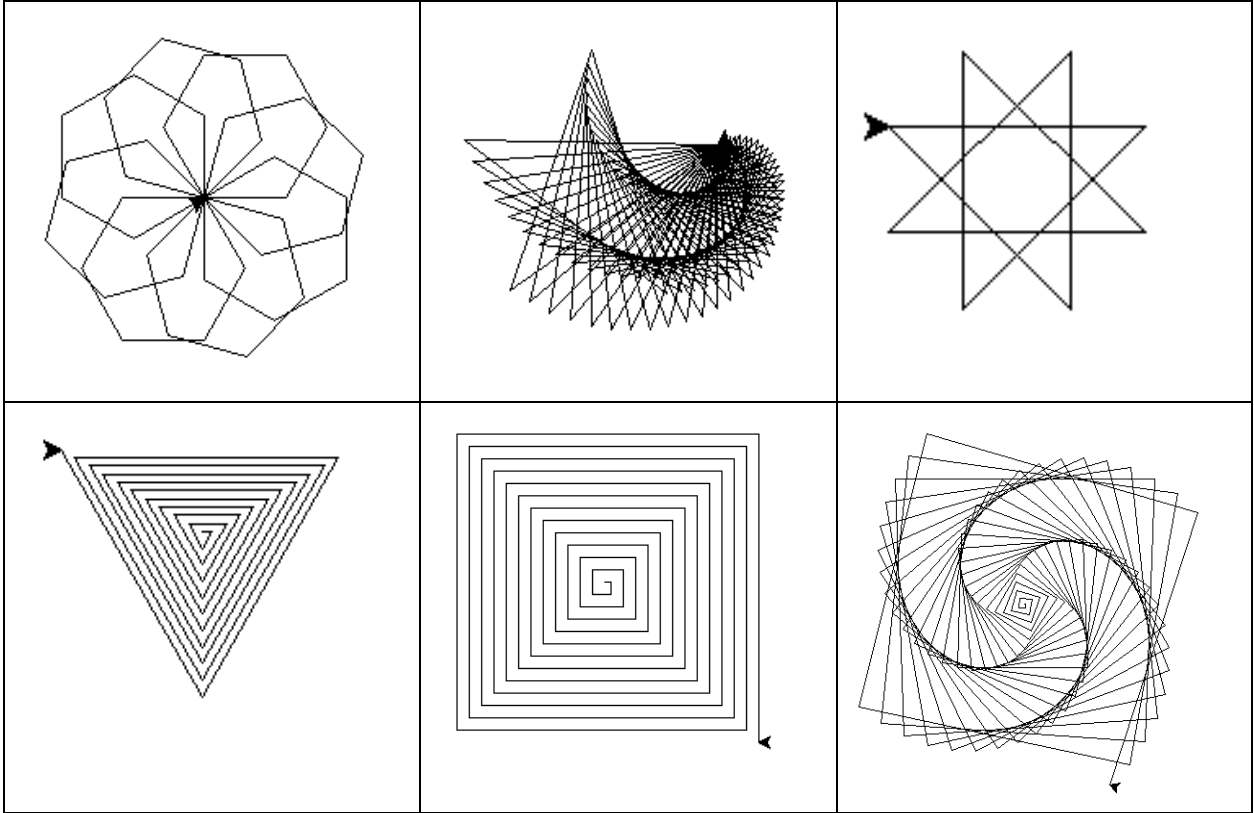
```
def printUpTo(maxnum):  
    num = 0  
    while num <= maxnum:  
        print num  
        num += 1
```

```
printUpTo(10)  
0  
1  
2  
3  
4  
5  
6  
7  
8  
9  
10
```

Turtle Exercises:

Using these tools, you should be able to create a bunch of complicated looking shapes. Try these:





2. Arithmetic

Functions

So far we've used functions to draw things and print things out. The real use of functions in math is to **return** values. Think input and output:

```
def function(input):  
    return output
```

That's not a real function, of course. Let's create some real ones that do something to our input and return the output:

```
def double(x):  
    return 2 * x
```

```
def one_less(x):  
    return x - 1
```

```
def special_square(x):  
    return x**2 + 3
```

It's up to you what you name a function. You can't use spaces or most punctuation but you can use capital letters and underscores. It's a good idea to make your function names descriptive.

Now we can either use the functions one by one. I'll put 4 into the first function and put the output into the 2nd function and so on.

```
>>> double(4)  
8
```

We get an output of 8. Let's put that into the next function:

```
>>> one_less(8)  
7
```

The output is 7. Let's put that into the third function:

```
>>> special_square(7)  
52
```

The final output is 52. We could just have Python do all that work: make the functions return values and automatically put them in to the next function. The innermost parentheses are

evaluated first.

```
>>> special_square(one_less(double(4)))  
52
```

That might seem like a lot of parentheses to keep track of right now, but I hope it gives you a taste of what you can do with Python functions!

Program: Average

Here's a useful tool to craft. We'll need it in Geometry for finding midpoints. Remember, Python 2 needs to be "told" you don't want it to round off to whole numbers by putting a decimal point somewhere in your function.

<pre>def average(a,b): return(a+b)/2. #needs decimal point</pre>	<pre>>>> average(10,15) 12.5</pre>
--	---

Program: Convert Fahrenheit to Celsius

Another common task is converting numbers according to a formula. Here are two functions that illustrate conversion tools we'll see later on.

<pre>#Converts Celsius to Fahrenheit def ctof(celsius): print (9./5)*celsius + 32,"F" #Converts Fahrenheit to Celsius def ftoc(fahrenheit): print (5./9)*(fahrenheit - 32),"C"</pre>	<pre>>>> ctof(20) 68.0 F >>> ftoc(212) 100.0 C</pre>
---	--

Conditionals and Input

There's a number-guessing game in which you try to guess the number I'm thinking of. If you don't guess it I have to tell you "higher" or "lower." How many guesses would it take to guess correctly if my number is between 1 and 10? 1 and 100? 1 and 1000? You get the idea. Before we can find out we need to learn a few more programming tools.

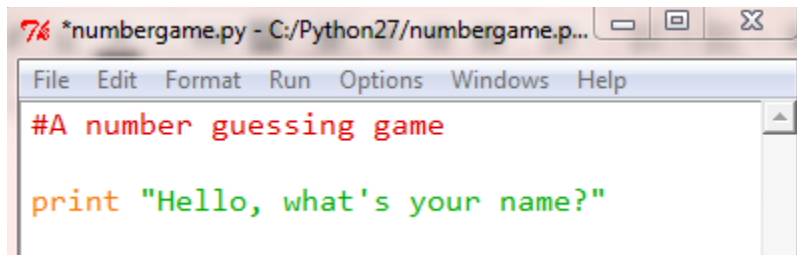
Conditionals

Another common task when programming is telling the program to check whether something is true, and if so, do something. It's sometimes called an "if-then" statement, but in Python, there's no "then." For example:

```
>>> if name == "Peter":
    print "That's my name, too!"
```

The Number Game

Let's start a new module called `numbergame.py` and start by saying hello:



```
*numbergame.py - C:/Python27/numbergame.p...
File Edit Format Run Options Windows Help
#A number guessing game
print "Hello, what's your name?"
```

Notice that the first line of code is a **comment**. When you run this program, you'll get

```
>>>
Hello, what's your name?
>>>
```

But if you try to enter your name, you'll get an error message:

```
Hello, what's your name?
>>> Peter
Traceback (most recent call last):
  File "<pyshell#20>", line 1, in <module>
    Peter
NameError: name 'Peter' is not defined
>>>
```

You might see this error message in the future, if Python doesn't recognize a variable or some other code. Right now Python doesn't know what to do with what I typed. It printed the line it was supposed to, and that's all we told it to do.

User Input

In order to take in user input, we have to use Python's "raw_input" function. In Python 3, it's just "input." Change the code to this:

```
name = raw_input("What's your name? ")
```


Now whatever the user types in, it will save that string to a variable called `name`.

```
What's your name? Peter
```

```
>>> print name
```

```
Peter
```

Input differences in Python 2 & 3

If you're using Python 3, you only have to type "input."

Python 2: <code>raw_input("...")</code>	Python 3: <code>input("...")</code>
--	--

Add this line to your code:

```
print "Hi,", name
```

Now, whatever the user types in, the program will greet them.

```
What's your name? Peter
```

```
Hi, Peter
```

To play our number guessing game, we could choose a number for the user to guess, but what if they want to play again and again? It'll be easier to just let the computer choose a random number for us. That will require the `random` module.

The random module

You can use **the random module** to generate random numbers and choose randomly from a range or a list. Here's how you use it in our number guessing game.

```
import random
number = random.randint(1,10)
```

It will save a random number between 1 and 10 to a variable called "number." Now you can use **Conditional statements** to respond differently according to the input the user enters:

```
guess = int(raw_input("What's your first guess? "))
if guess == number: #notice the double equals sign!
    print "That's it!"
elif guess < number: # "elif" means "otherwise"
    guess = int(raw_input("Nope. Higher. "))
elif guess > number:
    guess = int(raw_input("Nope. Lower. "))
```

You need the “int” before the raw_input because user input is always in the form of a string, not a number. “int” changes the string to an integer.

“elif” means “otherwise, if...” and you can have numerous “elif”s in a conditional, if there are a lot of possible conditions.

Checking for equality requires **double equals signs** (==). A single equals sign is how you assign a value to a variable.

Put together some loops and you have your number guessing game:

Program: Number Guessing

```
import random

def numbergame(upperlimit):
    guesses = 1
    win = 0
    name = raw_input("Hi! What's your name? ")
    print "Hi,",name
    print("Let's play a game.")
    while True: #infinite loop
        print "I'm thinking of a number between 1 and ", upperlimit
        guess = int(raw_input("What's your first guess? "))
        number = random.randint(1,upperlimit)
        playing = True
        while playing:
            if guess == number:
                print "That's it!"
                print "You guessed it in", guesses, "guesses!"
                win = 1
                break
            elif guess < number:
                guess = int(raw_input("Nope. Higher."))
            elif guess > number:
                guess = int(raw_input("Nope. Lower."))
            else: #if the input isn't a number
                guess = int(raw_input("Error."))
            guesses += 1

    replay = raw_input("Play again? y/n")
```

```

while replay != "y" and replay != "n":
    replay = int(raw_input("Error."))
if replay == "n":
    break
elif replay == 'y':
    guesses = 0    #reset the number of guesses
    win = 0       #reset the win to 0

```

numbergame(100)

Here's the output:

```

Hi! What's your name? Peter
Hi, Peter
Let's play a game.
I'm thinking of a number between 1 and 100
What's your first guess? 50
Nope. Higher.75
Nope. Higher.87
Nope. Higher.92
Nope. Lower.90
Nope. Higher.91
That's it!
You guessed it in 5 guesses!
Play again? y/n

```

Back to our original question. How many tries would it take in the above program to guess a number between 1 and 100? What if the number is between 1 and 1000? Make a conjecture and try it out!

Lists

A very useful way to group items together is using **lists**. Every programming language has lists, and in Python they're easy to use. Group the items inside square brackets, separated by commas:

```

>>> a = [1,2,3]
>>> print a
[1, 2, 3]
>>> type(a)
<type 'list'>

```

You can add lists (called “concatenation”) to get a list with all the items combined.

```
>>> b = ['hello', 'there']
>>> a + b
[1, 2, 3, 'hello', 'there']
```

You can add items and remove them using the `append()` and `remove()` functions:

```
>>> a.append(10)
>>> print a
[1, 2, 3, 10]
>>> a.remove(2)
>>> a
[1, 3, 10]
```

Items have a position in a list called an index. Here’s how to call the first item in a list:

```
>>> a = [6,3,5,2]
>>> a[0]
6
```

In most programming languages, numbering starts at 0. The second item in the list will have index 1.

```
>>> a = [6,3,5,2]
>>> a[1]
3
```

List indices can be negative, too. That means “starting from the last item in the list.”

```
>>> a = [6,3,5,2]
>>> a[-1]
2
>>> a[-2]
5
```

Program: Median

In math class you’re often asked to find the median of a list of numbers. It’s easy to put them in a list and sort them using Python’s “`sort()`” function but a little tricky to choose which one is the median. If the length of the list is an odd number n , the median is the term with the index that’s the whole number part of $n/2$. The Python syntax for “just the whole number part of a division” is “`//`”.

```

#Returns the median of a list of numbers:
def median(listA):
    #first put the list in order:
    listA.sort()
    n = len(listA) #n is the length of the list
    #If there's an even number of items in the list:
    if n % 2 == 0:
        #The median is the average of the middle two
        return average(listA[n//2-1],listA[n//2])
    else:
        #The median is the middle one
        return listA[n//2]

```

Program: Factoring a number

A common task in arithmetic is factoring a number. This program will come in handy when we do algebra.

```

#Returns a list of factors of "number":
def factors(number):
    factor_list = [1.0] #1 is definitely a factor
    for i in range(2, number+1):
        if number % i == 0:
            i = float(i) #convert the factor to a decimal
            factor_list.append(i) #add it to the list
    return factor_list

```

The Modulo Operator

The “Modulo Operator” sounds scarier than it really is. In Python (and some other languages, too) the percent sign (%) is used to divide two numbers and only return the **remainder**. We’re using it exclusively to check if the remainder is zero.

```

>>> 10%5
0
>>> 12%5
2
>>> 20%7
6

```

The remainder when you divide 10 by 5 is zero. When you divide 12 by 5 the remainder is 2, and when you divide 20 by 7 the remainder is 6.

In the “factors” function we convert the factor to a “float” because we’ll be doing some division with it soon. Let’s check to see if it works:

```
>>> factors(24)
[1.0, 2.0, 3.0, 4.0, 6.0, 8.0, 12.0, 24.0]
```

Yep, those are all the factors of 24.

Generating the Fibonacci Sequence

Here’s a good illustration of the use of negative list indices. The first two Fibonacci numbers are 1 and 1. To get the next Fibonacci number, you add the previous two together.

```
#Returns n Fibonacci numbers
def fibo(n):
    fibos = [1,1] #The first two Fibonacci
    for i in range(n-2):
        fibos.append(fibos[-1] + fibos[-2])
    return fibos
```

Now to get the next Fibonacci number you just add the **last two in the list**. To get 10 Fibonacci numbers, enter

```
>>> fibo(10)
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

Lists are necessary if you want to make a random choice from a number of items. First import the random module:

```
>>> import random
>>> a = [6,3,5,2]
>>> random.choice(a)
2
>>> random.choice(a)
3
>>> random.choice(a)
5
```

We’ll make extensive use of lists in all the math topics that follow. We’ll get a lot of practice declaring lists, adding to them, using their index numbers and iterating over their elements. They’re a very useful tool!

Arithmetic Exercises (solutions on page 128):

Use the functions you created in this chapter to answer these questions:

1. What is the average of 225 and 723?
2. What is the average of 1,412 and 36,877?
3. Convert 212 degrees Fahrenheit to Celsius.
4. My friend was telling me about the weather in Brazil, saying, "It's 25 degrees!" I realized he meant Celsius. What's that in Fahrenheit?
5. Find the median of these numbers: 18,12,11.5,14,9,21,8,15,3,25,10,18,6
6. Find the median of these numbers: 76,59,64,23,11,98,56,77,91,89,48,101,55,37
7. What is the 20th Fibonacci number?