Introduction: You have super powers!

Welcome to a world where you have super powers. Here you can create, control and change the things around you, solve problems, and invent things no one has seen before. This is the world of code, and all you need to be its master is a bit of knowledge.

First you need to know what code is. Code is the set of instructions which tell a computer what to do. These can be as simple as “write ‘hello’ 5 times” or as complex as creating a website, or the latest Star Wars movie, or the latest hit song, or playing Minecraft, or even exploring Mars in virtual reality.

Different situations require different kinds of code. The code for a web site looks much different from the code to create a song, and uses different sets of words and instructions to tell the computer how to do what we need it to do.   Over time, people have organized these sets of instructions into “languages” the same way that human speech is organized into languages, each having its own specific set of rules about what words can be used, and in what order.

The rules that tell us how to use a language (whether it be a human language or a computer language) are called ‘syntax’, and before we can use a language we need to understand its syntax. In human languages, this means knowing what letters and symbols we need to use in order to communicate with other people who speak the same language. In computer languages, this means knowing what commands to give the computer to tell it to work the way we want it to.

Some computer languages have a rather complicated syntax, which can be very difficult for a beginner to understand and use. Additionally, some languages are only useful for specific purposes, or on specific kinds of computers. Other languages, however, are designed to be quick and easy to learn, and useful for a wide variety of purposes.

Python is one such simple language, and you will find that it is useful for just about any purpose you can imagine. In our case, we are going to use Python to create Art. To do this, we will enlist the help of a *module* (a piece of code saved in a separate place, which can be used from other programs) called Turtle. Turtle allows us to draw on and control the screen, and what we will use if for here is only the beginning of what it can do. See the “Useful Links” section for a link to a site that describes everything turtle can do. if you are curious.
In the meantime, let’s start coding!

Section 1: The Python Shell

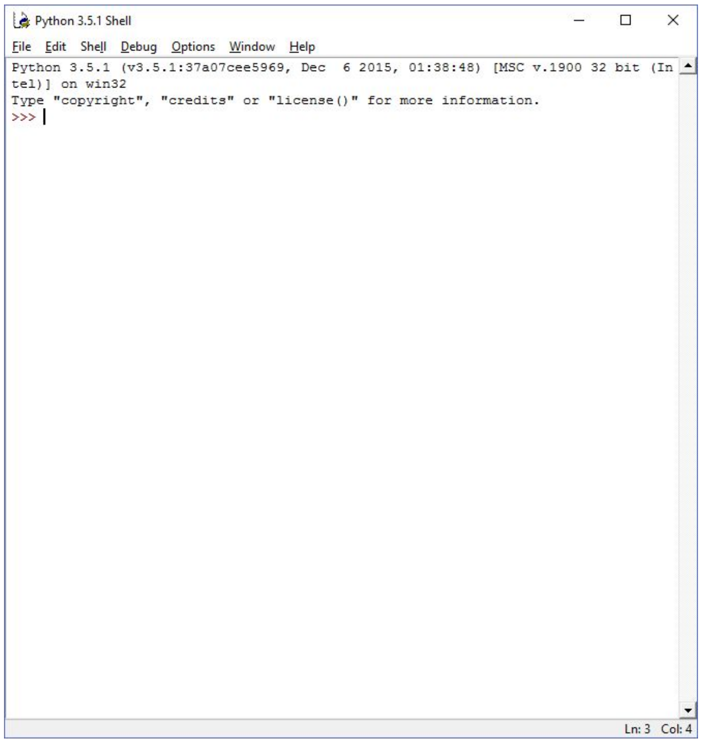
As we have already discussed, we write code to make things happen on the computer. When we do this successfully, we end up with a computer program which we can run (or “execute”). Since the code we write is the *source* of our program, it is called “Source Code”. How each Computer Language handles this source code can be very different. In many languages, the Source Code and the finished Program are two separate things. You need to run your source code through a process called “Compiling”, during which the instructions in your source code are evaluated and used to build a new program.

In other languages (such as Python), however, the source code and the finished program are the same thing. Instead of being compiled, the source code itself is executed inside another program. Since during this process the instructions in your source code are interpreted by another program, that program is called an Interpreter. This interpreter gives your code a place to run, and usually has tools built in to show you the results of your code and allow you to interact with it in some way.

Python is one such interpreted language, and it includes a special program to interpret your code called the Python Shell. In the Python Shell, you can type your code, one line at a time, and the instructions are interpreted by Python. This gives you an excellent way to try out code and see what works. We will be using the Python Shell for this course, and on every kind of desktop computer, the Python Shell comes as part of a larger program called IDLE.

First thing to do is to find IDLE on your computer. If you don’t have it, please see the Links section at the end of this book, or simply go to http://python.org and click “Download Python”, then find the one for your computer. If you’re not sure what to do, ask someone who knows how to install software to help you.

Once it is installed, run Python IDLE. You will be presented with a window like the one on the right.



```
Python 3.5.1 (v3.5.1:37a07cee5969, Dec  6 2015, 01:38:48) [MSC v.1900 32 bit (In
tel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> 
```

Section 2: Prompts, Calculators, and Variables

In the Python shell, you will see a window with a bunch of text (“writing”) in it. Somewhere in this window, there will be a line called a *prompt* which says **>>>** and has a cursor blinking next to it. Wherever this cursor shows up is where the text will go when you type. Try it out and press **Enter** a few times. You should see a few more lines of **>>>** show up.

So what can you do with this Python Shell? A lot. For starters, it can be a calculator. To do this, simply type in a math problem like **1+1** and press **Enter**. As you will see, Python helpfully responds by printing “2” on the next line. Now you can try to type in a bigger problem. Remember that / is “divided by” and “*” is “multiplied by”, so 346*147/23 is “Three hundred forty-six times one hundred forty-seven divided by twenty-three”. If you type this in, Python will show how fast it is at math and instantly reply with 2211.391304347826.

Code	Output
346*147/23	2211.391304347826

You will find that Python is a VERY fast and powerful calculator which can compute numbers MUCH larger than a normal calculator, and MUCH MUCH larger than most people can compute in their head. If you learned nothing else about Python than how to use it as an awesome calculator, you’d still be better off for learning it. Thankfully, though, Python can do WAY more than just be a calculator, and even as a calculator you will find it has plenty of tricks up its sleeve.

The first trick to learn with Python, and probably one of the most useful ones, is *variables.* You can think of a variable as a sort of storage place for information, which goes by an easy nickname. In real life we actually do this all the time without realizing it. For example, you probably own a backpack. You don’t call it “black backpack with blue trim which I bought 2 years ago while school shopping with Mom”. You call it “backpack”. If you put something inside it, all you need to know in order to get that thing back out is that it’s in your backpack. You can then get the backpack, open it, and take out that thing. Also, one of your friends also probably owns a backpack, and also just calls it “backpack”, but neither of you ever gets confused about which backpack your things are in.

On a computer, a variable works much like a backpack. You can put bits of information in (or “assign them to”) it using an equals sign (“=”). You can then use this variable’s name in place of that bit of information any time in your code. Variables can be called anything you want (even whole words), but often we use just a single letter (usually “x” or “y”) just like in Algebra.

For example, instead of saying “1+1”, we could type the following code (remembering to always press Enter at the end of each line):

Code	Output
x=1 **x+x**	2

And python would still respond with 2. You can even use more than one variable at a time. For example you could type:

Code	Output
x=1 **y=3** **z=4** **x+y+z+12**	20

And python would convert x, y and z to their numerical values, add them together , add 12 to that, and respond with an answer of 20.

Section 3: More Variables and Functions

When it comes to using variables in Python, you can store any bit of information in them the same way, whether it is numbers or groups of letters and/or words or other text (called "strings"). The main thing to remember is that if you start using a variable by putting a number in it, that variable can only have numbers in it for the rest of your program. Likewise, if you start using a variable (declare it) by putting in a string (like the letter "a" or the phrase "peter piper picked a pack of pickled peppers"), it can only have strings in it for the rest of your program.

One interesting thing is that you can still "do math" with variables no matter what you put in them, only you may get results different from what you expect. For example:

Code	Output
x = 17 **y = 4** **x+y**	21

You will get a response of "21", but if instead you use words, Python will *put them together into one new string if you add two string variables together*:

Code	Output
x = "foot" **y = "ball"** **x+y**	football

What you can't do is add strings to numbers by just using a plus sign ('+'). For example:

Code	Output
x = 23 **y = "apple"** **x+y**	TypeError: unsupported operand type(s) for +: 'int' and 'str'

Python will, instead of saying "23apple", respond with a scary looking red error message about "unsupported operand type". As we mentioned earlier, Python has a way to show numbers as strings. To do this we need to use a *function.*

Functions are bits of code that do something, and we can get that thing to happen by just typing the name of the function. If we once again compare a variable to a backpack, then a function is more like a task or action we remember how to do. Usually these tasks are made up of several steps.

For example, a common action is to take a book out of your backpack and open it. This task is made up of the following steps:

1. Open backpack
2. Take out the book
3. Open the book

Maybe we call this task “open book”. When we perform this task, we do each action, one at a time, until the task is complete. In Python, we can name functions anything, so we could make a function called “openbook”, but *not* “open book”, since you can’t have spaces in function or variable names.

But what if we want to open the book to a certain page? We would say “open the book to page 37”. For our Python function, we would write it as openbook(37). The (37) at the end is called an “*argument*” and is the piece of information that the function acts upon, in this case, the specific page number to which the book is opened.

Getting back to our previous problem about strings and numbers, once again we have the following code:
x = 23
y = “apple”

If we want to put those things together into “23apple”, we would need to tell Python to show the variable x as a string instead of a number. The function which does this is called **str** and its argument is the number or variable which it shows as a string. For example:

Code	Output
str(x)+y	23apple

Although you might expect it to work the other way, since Python wouldn’t know what number to show “apple” as, we cannot do this quite as easily. There are ways to do this in Python, but they will be left as an exercise for the reader.

There are many (many many) other functions in Python, but perhaps one of the most useful functions is called **print.** This function lets us show something (or "print" it) on the screen. Its simplest argument is the thing you want printed. So for example:

Code	Output
x=36 **print(x)**	36

You can print strings the same way, just make sure you put them in quotation marks:

Code	Output
x =”Hello There” **print(x)**	Hello There

You can also write other code inside print's argument, and it will show the code's output. For example if x = 9, you could write x+3 as:

Code	Output
x=9 **print(x+3)**	12

You can also write strings and numbers right inside the print argument, and even mix them by separating them with a comma. For example, if you have a variable age that equals 17, and you want to say "Bobby is 17 years old", you could write:

Code	Output
age=17 **print(“Bobby is”, age, “years old”)**	Bobby is 17 years old

We can even make our own functions. Let's say we want a function that takes whatever argument we give it and tells us what it is. We could call our function *showarg.* To make a function, or "define" it, you use **def**. You can also tell the function it has an argument (or more than one!), by giving it a variable to use as that argument. Let's call our variable *argument*. At the end of our **def** line, we need to put a colon (":") so that Python knows "everything after this is the code inside the function". The **def** ends when you leave a blank line. So for example:

```
def showarg(argument):
        print("You typed",argument)
```

Would define a "showarg" function. We would use it by typing the following on a new line:

Code	Output
showarg(46)	You typed 46

You can also make functions with no argument, but you still need the parenthesis () so Python knows it's a function. So if you had a function called *hello*, you'd make it with **def hello():** and use it with **hello()**.

You can even have a function give information back (or **return()** it) to your main program. To do this, you use the **return()** function, and its argument is whatever (a number, a string, a variable, whatever) you want the function's output to be. In this way, you can assign a function's output to a variable. For example:

Code	Output

def addthree(n):
  y=n+3
  return(y)
 	
x=addthree(17)
print(x)
	20

In this way, you can pass information to a function (as its argument), do something to it inside the function, and then pass back the results (return() them) to your main program. This makes functions very very useful indeed.

Section 4: Modules and the Turtle

There are lots of functions for Python, and lots more that people have made on their own. A lot of the functions people have made are included with Python so that you can use them too. These functions are saved in special files called *modules* which each are named after what kind of functions are inside them. You then can *import* (bring in) that file into your code so you can use the functions from it.

For example, suppose we want to make a program that picks a number between one and ten randomly. There is no built-in way in Python to generate a random number, but there *is* a module called *random* that has a way. We import it with:
import random

Now we can use the functions inside the *random* module. Since in our program we want to pick a number between one and ten, we probably want it to only be a whole number (not a negative number or fraction). These kinds of numbers are called "integer" or (in Python, *int)*, and there is a function called **randint** (short for *random integer)* that we can use for this. The functions that come from modules are called *methods*, and you use them by writing it like *module.method(arguments)***.** The **randint** method accepts two arguments (the lowest and highest number to make a number between), so for our example, we'd write:
random.randint(1,10)

Code	Output

import random
random.randint(1,10)
	3

When we press Enter it would randomly come up with and show us a number between one and ten. You can even assign the number it comes up with to a variable, like:

Code	Output
x=random.randint(1,10) **print(x)**	7

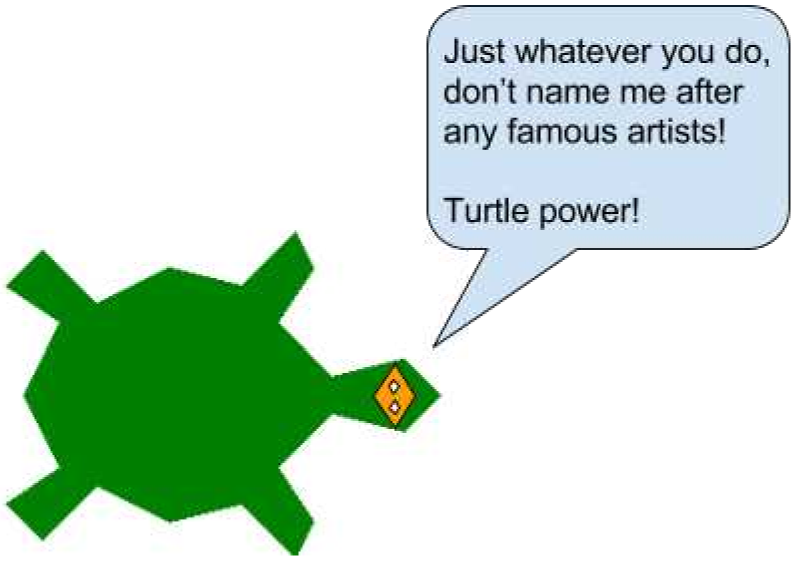
Now we are ready to make our first picture. In order to do this we need a different module called *turtle.* Just like before, we first import it with **import turtle**. Then we are ready to use its functions.

The Turtle module is special because when we use it, it opens a whole different window on our computer. This window starts out blank, and since it is where we will make our picture, it is called a *canvas*. When use the methods found in the Turtle module, a little shape (the Turtle!) moves around the canvas and draws.

One interesting thing about modules and methods is that you can give them a new name. There are several reasons to do this. The first is that you can save yourself some typing by using a shorter name. The second is that maybe you want to have more than one turtle. If you give them different names, you can use more than one turtle in your code.

To give the turtle a new name (let's call our turtle "tim"), you use the name like a variable, and assign a special method to it. The method is *turtle.Turtle(),* and we assign it the name 'tim' with:
tim=turtle.Turtle()

Now you can move the turtle around by using the name tim, which is not only shorter than 'turtle', but cuter too. If you prefer, you can call your turtle tina or tyler or tootsie or even a name that doesn't start with 't', like SimonJamesAlexanderRagsdaleTheThird, though this is not recommended since you'd have to type that really long name a LOT in your code.


Just whatever you do, don't name me after any famous artists!

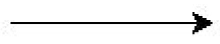
Turtle power!

Section 5: Our First Picture

So now that we have some basic knowledge of Python, and we have imported our turtle and named it, we can draw a picture! Please note that we will use 'tim' in our examples, but if you used a different name, you'll need to type that in place of 'tim' wherever you see it in our examples.

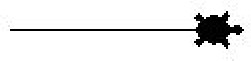
To draw with a pen on a piece of paper, you put the tip of the pen down on the paper and then move it around. The pen then leaves ink lines on the paper, and when you get done, you either have a work of art or a scribbled mess, or some stick people, or whatever shapes you moved the pen around in. That is exactly how the turtle draws on the canvas. The only difference is that you have to tell the turtle which direction you want it to move in, and how far.

If you want tim to move forward 100 spaces, you say:

Code	Output
tim.forward(100)	

Congratulations! You have just drawn your first picture using code! It is maybe not the most exciting picture yet, and you are probably left after this exercise with a few questions. For one, why does it look like an arrow and not a turtle? Also, why did we pick 100 as the number to move?

The "turtle" is an arrow because it is easy for the computer to draw, and clearly shows which direction the 'turtle' is facing. We can make it look like a turtle though! To do this, type:

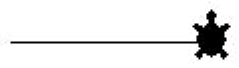
Code	Output
tim.shape('turtle')	

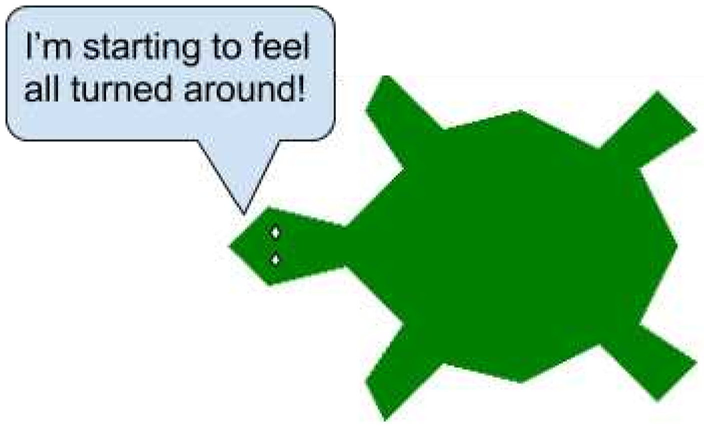
Now we have a cute little turtle!

You can make the turtle look like other shapes too, but that will be left as an exercise for the reader.

As for the second question, why did we move 100?   We chose 100 because it is a big enough motion to draw a decently sized line, but not so big that it won't fit on the canvas. The canvas is **800 wide by 600 tall,** but the turtle starts in the middle of it. It can go 400 forward before it gets to the right edge. To go to the left side of the canvas, you move backward instead of forward. So, from where the turtle starts, you can move forward(400) to get to the right edge, or backward(400) to get to the left edge.

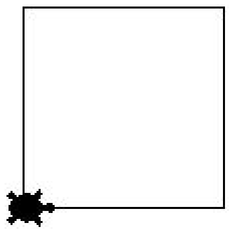
Now that our turtle has moved forward 100, let's keep going and draw more of a picture. To do this, we next need to turn our turtle. This is accomplished with the *left* and *right* methods.   For example, to turn a 90 degree left turn:

Code	Output
tim.left(90)	



So now that we can move and turn, let's make some cool shapes! First let's make a box. A box is just four lines and four corners, and we've already drawn one line and turned, so next we need to draw three more lines and three more turns:

Code	Output

tim.forward(100)
tim.left(90)
tim.forward(100)
tim.left(90)
tim.forward(90)
tim.left(90)
	

Cool, right? But doesn't this seem like a LOT of typing just to make a box? What if there is an easier way? There is. We can use a loop. A loop is a kind of function that repeats the instructions inside it a certain number of times. This way you can "re-use" code and save a lot of work (and make your code easier to read).

The two most common kinds of loops are "while loops" and "for loops". We can use either one to make a box. First, let's try a while loop. The 'while' function has as its argument a description of a certain situation, called a *condition*. While that condition is true, the code inside the loop runs over and over (it 'loops') until the condition stops being true.

A simple example is a loop that counts to four. This will use several of the things we've already learned:

Code	Output

x=0
while(x<4):
    x=x+1
    print('X is:',x)
	X is: 1
	X is: 2
	X is: 3
	X is: 4

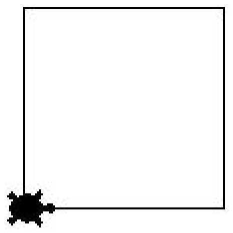
So what's going on here? First, we set x to 0. Then we have a 'while' function that says "while x is less than 4…", followed by a colon (:) that says "everything after this is in this function until a blank line". Inside the function we have a special variable assignment **x=x+1** (set x to whatever x+1 would be) called an increment, followed by a print statement that shows us what x is currently set to. As you can see, the contents of a loop look just like the contents of a function (because they *are* the contents of a function). As for the condition, this can be anything that makes sense in your code, even just the word True, which (since it's always true) runs the loop forever.

So how do we use this to make a box? First (since we already have a box), let's reset our canvas back to being empty so we can draw a new picture. For this, the code is just:
tim.reset()

As you can see, the reset() method accepts no arguments (kind of like your parents), and simply clears the canvas and puts the turtle back in the middle facing to the right. Now we can draw a new picture.

Let's put together everything we've learned, and use a while loop to make a square. First, we need to declare x as 0 again. After that, we use our while loop to go forward and then left, counting up as we do so:

Code	Output

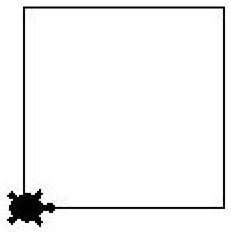
x=0
while(x<4):
  tim.forward(100)
  tim.left(90)
  x=x+1
	

And we get a box just like before. But what if this is STILL not the easiest way to do it. There is another kind of loop, the 'for' loop, that can do this with two less lines of code. A for loop takes as its arguments a variable and a *range*.   Here's how it works:

If we use the statement **for i in range(5):**, what we are saying is "set i to 0, then add one to it each time you go through the loop, until you get to 5, then stop". We can use any variable in our for loop, but most people use *i* (maybe because it's short for *increment).*

So to make our square with a for loop, first we do a reset() to clear our canvas, then we type:

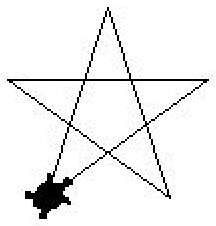
Code	Output

for i in range(4):
  tim.forward(100)
  tim.left(90)
	

And we get the same exact square. Of course we can make any other shape we want, and not just a square.

Next, let's make a five-pointed star. This one is really just one more line, and a few different angles. First, we reset(), then here's the code:

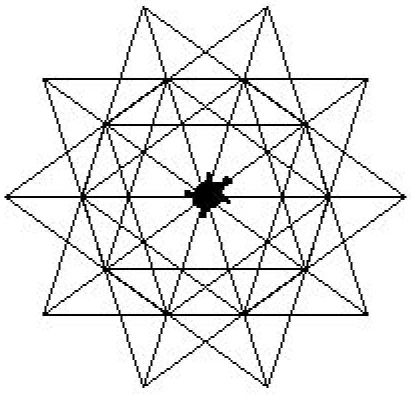
Code	Output

tim.left(36)
for i in range(5):
  tim.forward(100)
  tim.left(144)
	

One thing that may stand out if you are observant is that there is an extra **tim.left(36)** at the beginning. This is simply to position the turtle so that the star it draws is right-side-up (with the point facing up). In making art in Python, you'll find plenty of times when your turtle ends up facing the wrong direction. In these cases a right() or left() is all that is needed to position it so that your picture turns out looking the way you intend.

We can even define our own function, and then put a for loop inside it. Let's make a *star()* function with the above for loop inside it, and then put THAT inside a loop to make a cool design:

Code	Output

```
def star():
  for i in range(5):
    tim.forward(100)
    tim.left(144)

tim.left(36)
for i in range(10):
  star()
  tim.left(36)
```
	

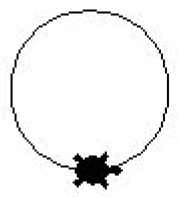
```
def star():
  for i in range(5):
    tim.forward(100)
    tim.left(144)

tim.left(36)
for i in range(10):
  star()
  tim.left(36)
```

Section 7: Circles

So we've learned one way to draw geometric shapes like squares. These are fairly easy because they have a very low number of straight sides. What if we want to draw a circle, though? Thankfully there is a helpful method for this, simply called **circle()**.

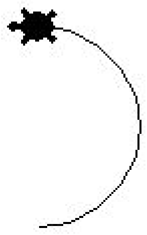
This new **circle()** method is interesting in a few ways. For one, it can take different numbers of arguments depending on how you use it. Second, it can be used to make a lot more than just circles, as you'll soon see. Here's the syntax:
circle(*size, extent, steps)*

So what does this mean? This is showing that the **circle** method can take up to three arguments (size, extent, and steps). In Python, when there are multiple arguments, it starts on the left and uses the arguments it is given one at a time until it runs out of arguments. This means, in the case of the **circle** method, if you only give it one argument, it will use that number as the *size.* If you give it two arguments, it will use them as *size* and *extent*. If you have three, it will use them as *size* then *extent* then *steps*. If you have four or more, it will just use the first three, since the **circle** method only can take three arguments. Here are a few examples to show how **circle** works.

Code	Output
tim.circle(40)	

You can see that because it is a circle, the turtle ended up where it started again. This is one of the other features that makes the **circle()** method so useful - you always know that your turtle will end up facing the way it started again.

Now let's **reset()** the canvas and then make a semicircle:

Code	Output
tim.circle(50, 180)	

Wait. A semicircle?! This is what the "Extent" argument does - it specifies how much of a circle it is. Since a full circle is 360 degrees, half a circle is 180 degrees. Smaller numbers mean a smaller part of a circle. This allows you to to draw all kinds of complex shapes by arranging semicircles and circles together.

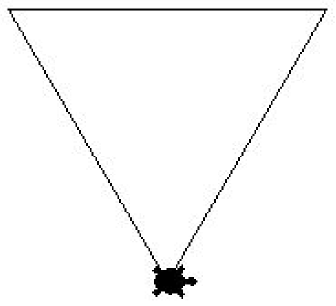
As you can tell already, the **circle()** method is surprisingly useful. Now we will show what is perhaps its greatest trick: to make shapes that are not circles at all. This is what the third argument (*steps)* does - it specifies the number of steps used to draw the circle.

If this doesn't make sense, consider that an actual circle is a very complex shape for a computer to draw. Instead what the computer does is draw an *approximation* of a circle. That's to say, it uses very short lines, each one turned slightly and attached to the end of the previous line, until it forms a shape that looks like a circle. If you've ever made a circle on the ground out of sticks or rocks, that's basically what **circle()** does too. Each one of those lines is a *step*, and the more of them there are, the smaller they can be, and the more the shape looks like an actual circle.

You can use this to your advantage by specifying a really small number of steps. It will then go around the shape using only that many lines.   Because of this, you can use **circle()** to make any simple geometric shape. For example, let's return to our very first example: the square. You can actually make a square with ONE line of code, by doing **circle(100, None, 4).** This would create an even-sided "circle" using only 4 lines, each 100 spaces long. We call a shape with four even sides a square.   The "None" has to be in there because Python looks at arguments from left-to-right, and since "Steps" is the third argument, you need a second argument too.

Because of this, it has built in that you can specify "None" for extent instead of an actual number, so it creates a full circle instead of a semicircle.

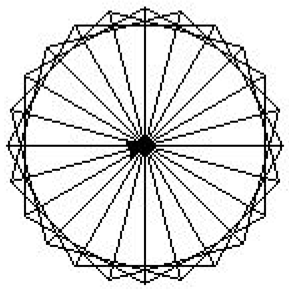
You can specify a different number of steps to create a different shape. For example, just change the 4 to a 3:

Code	Output
circle(100, None, 3)	

And you get a triangle! Take a moment to experiment with putting different numbers in. What if you use a 1? How about a 5? How about 50?

And now here's a new trick: remember how, in our earlier example of the 5 pointed star, we used **tim.left(36)** to rotate our turtle so the star was upright? Rotating in this way using **left()** or **right()** can allow you to make all kinds of complex shapes. You can even layer shapes by going through a loop and just rotating a little bit each time. For example, let's make a spoked wheel using **circle()** (to make a triangle), **left(),** and a for loop. Here's the code:

Code	Output

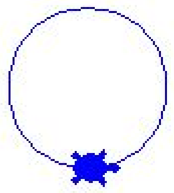
for i in range(25):
  tim.circle(40, None, 3)
  tim.left(15)
	

That's wheely pretty cool, right? Now we're almost ready to make some bigger pictures, so let's keep rolling.

So far we've been able to make some pretty neat pictures, but they're all just black and white. Now let's add some color! To do this, we simply tell python what color we want it to use with the **color()** method. Python then draws with that color until we tell it to use a different color. We can tell Python what color to use in several different ways, but the easiest way is simply using words.

For example, let's do a **reset()** and then draw a blue circle:

Code	Output

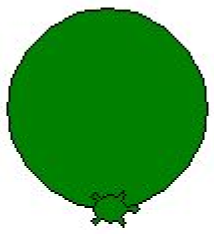
tim.color('blue')
tim.circle(40)
	

That's it. Python knows lots of colors, and you can just call them all by name. But you may notice that the circle's LINE is blue, but the circle itself is still empty. What if we want a solid blue circle? For this we need to add two more methods: **fillcolor()** and **begin_fill()**

The **fillcolor()** method tells Python which color it will be filling in the shape with, and it otherwise works exactly the same as **color().** The **begin_fill()** method has no arguments. Just using it tells Python to use whatever color you told it in **fillcolor()** to fill in whatever stuff you draw after you use **begin_fill()**.

Here's an example of a filled-in green circle (remember to first do a **reset()**):

Code	Output

tim.fillcolor('green')
tim.begin_fill()
tim.circle(50)
tim.end_fill()
	

If you were watching the turtle as you wrote your code, you may have noticed that the circle didn't actually fill in until you put in the final method **end_fill().** This way you can do **begin_fill()**, draw a whole bunch of things, and then fill them all in at once.

So now that we know how to use colors, we can learn some ways to use them in our code. For starters, since the names of colors (like ‘red’, and ‘blue’) are just strings, you can do anything with them that you can do with a string. For example you can assign them to variables, and you can store them in *lists*.

Wait.. what are lists?!   Lists are just what they sound like: they are lists of information stored in an easily-accessible way. We use these all the time in real life, for everything from grocery shopping to keeping track of how much of something we have.   Typically we write lists either as one entry per line, or as a bunch of things separated by commas. Once again, Python works similarly to how you already use lists.

In Python a list is like a combination between a module and a special kind of variable. Just like how Python knows you’re talking about a function because you use parenthesis at the end (like *function()*), it knows you’re talking about a list because you use square brackets at the end (like *colors[]*). But here’s where they’re different: for functions (and modules), you put the argument you’re passing to it inside the parenthesis. For lists, on the other hand, you put the *index*.

Think of indexes like this: imagine writing a shopping list on a piece of paper, and then writing numbers next to each item in the list. Later on, instead of having to remember everything on the list, you can just refer to items by their number. For technical reasons, Python starts its numbering at zero instead of one, so the first item is numbered 0, the second item is numbered 1, the third item is 2, and so on.

Let’s say we have our shopping list in Python. Let’s call it *groceries[].* In the list are *milk, eggs, butter, and bread*.   Since lists are accessed like variables, you can *print* them to see their values. But how do we get items into the list? We assign them like variables, sort of. The difference again is that we assign multiple things at once, separated by commas, inside square brackets. So let’s make our shopping list, and then print out the third item:

Code	Output
groceries=[‘milk’,’eggs’,’butter’,bread’] **print(groceries[2])**	butter

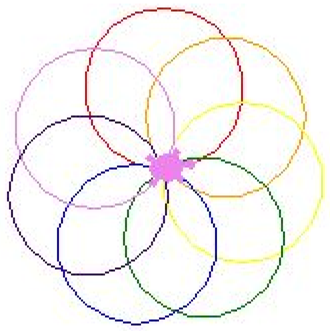
So in this way we can access specific items in the list. You can also use a colon (:) with the number to say that it’s a range. If there are 5 things in the list, :5 would say “up to 5”, 3: would say
“3 and up”, and 3:5 would mean “3 to 5”.

If we want to see all of the items, we can just use **print()** on the list itself:

Code	Output
print(groceries)	[‘milk’, ‘eggs’, ‘butter’, ‘bread’]

But what if we want to do something to each item in the list? You can use a *for loop*. Since this section started talking about colors, let’s make a for loop that makes circles of different colors. First we need a list of colors, then our *for loop* will cycle through the list, one color at a time, making a circle of each color. Here’s the code:

Code	Output

colors=[‘red’,’orange’,’yellow’,’green’,’blue’,’indigo’,’violet’]
for c in colors:
   tim.color(c)
   tim.circle(40)
   tim.right(50)
	

Now we’re making art! (okay so we were before too, but colors are nice, aren’t they?) Already you can really see how all of the different parts of Python can get put together in interesting ways.

One last thing to mention about colors is that we’ve so far only talked about setting the drawing (or “foreground”) color, but we can also set the color of the canvas (or “background”) itself. This makes it easier to make colorful pictures, since we don’t need to manually make a rectangle the exact size of the canvas if we want to have a blue sky, for example.

To set the background color, the method is **bgcolor(*color*)**. This means, if we want a light blue window (to look like the sky?), we can do **turtle.bgcolor(‘light blue’).**

Do you notice that it says **turtle.bgcolor** instead of **tim.bgcolor**? That’s not a typo! It’s because the name of our turtle only applies to the turtle. If we want to deal with the screen itself, we either use the generic name ‘turtle’, or we can name the screen in very much the same way as we named our turtle. For example, suppose we want the screen to be called *scr*, then we can use **scr=turtle.Screen()** to name it. After that, we can do **scr.bgcolor(‘light blue’)** instead, which is perhaps easier to read and understand.

Section 9: Movement

Way earlier in the book, we talked about drawing with a pen on paper. We said that, to draw a picture with a pen, first you put the tip of the pen on the paper, and move the pen around to make shapes. In Python, the “pen” (turtle) always starts down on the “paper” (canvas), so when you use **circle()** or **forward()** or any other drawing method, the pen (turtle) just draws the shape. But what happens if you want to move the pen without making a line? We need a way to lift the pen up.

Well, as it turns out, there is actually a method called **up()** that does exactly this! If you do **tim.up()** it looks like nothing happens. But now do **tim.forward(100)** and your turtle will move without drawing a line! When you want to draw again, you use the matching **down()** method. You could use this technique to go **forward()**, **back()**, **left()**, and **right()** and move all over the canvas, drawing shapes as you go. You could even make some very complex pictures like this, except for one “problem”: you always need to be concerned which way your turtle is facing.

For example, suppose you want to draw a simple scene with green “ground”, blue “sky” and a yellow “sun”. You’d need to first draw a big rectangle, then lift your pen, turn, use forward() and left()/right() to get to where the sky is, put your pen down, draw a bigger blue rectangle, then lift your pen up, use several more lines of code to position yourself again, then put your pen down and draw a yellow circle. Seems like a lot doesn’t it?

What if you could just tell your turtle to go to whatever part of the canvas you want using one line of code? Not only would it be less typing, but it would make your code much easier to read. Well, as it turns out, there IS a way to do this, and the method is simply called **goto()**.

The **goto()** method relies on something we mentioned earlier: how big your canvas is. You need to know this because this tells you what numbers to use with **goto()** in order to position the turtle where you want it.

The canvas in Python turtle starts out **800 wide by 600 tall**, with your turtle in the middle. Numbers going *up and right are positive,* and *numbers going down and left are negative,* and *all numbers are relative to the middle of the canvas.* Whenever we set a position with height and width, the numbers are called *coordinates*, and they are always written as two numbers in the form *width,height*. For example, the middle of the canvas is 0,0. The upper left is -600,600. If you want your turtle to use these coordinates to go to the upper left, you can just do **tim.goto(-600,600)** and your turtle will go there. Make sure to use **tim.up()** first if you don’t want to leave a line when you move.

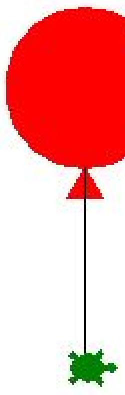
One final note is about setting the canvas size. You can set the canvas size using **turtle.screensize(*width,height,bgcolor*)**. Notice that there’s even the ability to set the bgcolor right from there. Often this is the first line included in a Python turtle program (after **import turtle** of course!). Doing this ensures that you know just how wide and tall your window is.

This is it. We have gone over a small selection of the basics of Python. The goal of this text was never to be a complete guide to Python (there are many excellent Python guides already on the internet!), but instead to teach beginners the basics of the language with a focus on making cool pictures using the Turtle Graphics Module (which few other books seem to cover). We have done this, and now we can provide several examples of programs and what their output should look like.

Note: These examples will be provided without much explanation, in order to fit in more examples. If you need more assistance, or see a new function or method that doesn't make sense to you, please refer to the complete Turtle Graphics Module document which is linked in the *Appendix: Useful Links* section following this one.

Example 1: A Red Balloon (our turtle learns to fly)!

Code	Output

tim.color('red')
tim.fillcolor('red')
tim.begin_fill()
tim.circle(40)
tim.left(180)
tim.circle(10,None,3)
tim.end_fill()
tim.left(90)
tim.color('black')
tim.forward(100)
tim.left(90)
tim.color('green')
	

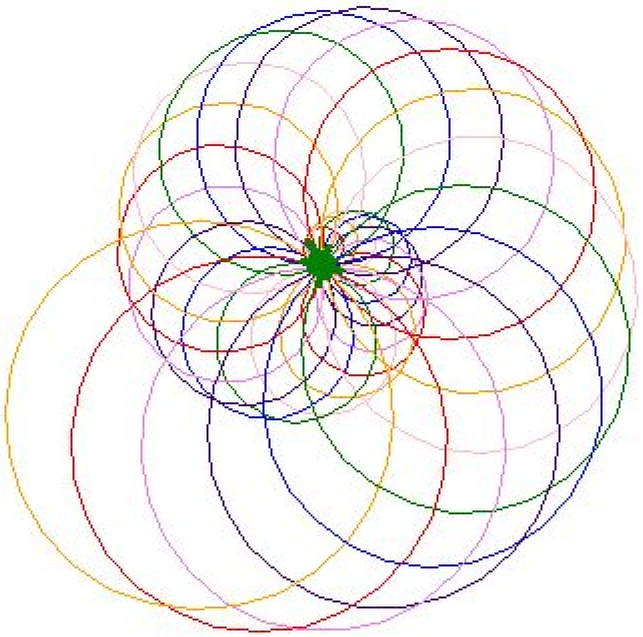
Example 2: Comments *(because they are very useful)*

Code	Output
# These lines starting with the # sign **# sign are comments. They are not** **# processed by Python. Programmers** **# use comments so that people reading** **# their code can know what the parts do** **print('This is not a comment') # but this is**	This is not a comment

Example 3: The Growing Color Spiral ***(from the front cover of this book!)***

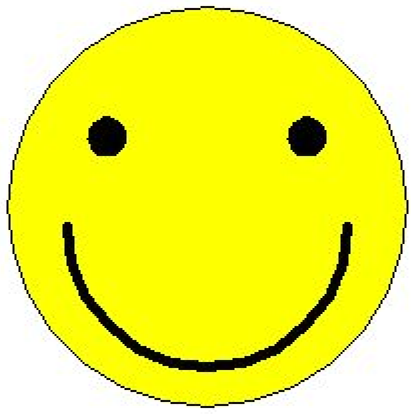
Code

`# Here's our list of colors.  Note that we used pink instead of yellow`
`# Because yellow doesn't show up very well on a white background`
`colors=['red','orange','pink','green','blue','indigo','violet']`
`c=0`
`size=10`
 	
`# The for loop that actually does the spiral`
`# Note the if() statement that makes sure the`
`# c variable for color stays within the 0-6 range`
`for i in range(30):`
`  if(c>6):`
`     c=0`
`  tim.color(colors[c])`
`  tim.circle(size)`
`  tim.right(20)`
`  size=size+3`
`  c=c+1`
`tim.color('green')`

Output


Example 4: Smiley Face!

Code	Output

**# First draw the 'Head' circle 	
tim.up()
tim.goto(0,-100)
tim.down()
tim.fillcolor('yellow')
tim.begin_fill()
tim.circle(100)
tim.end_fill()
 	
# Next draw the eyes 	
tim.up()
tim.goto(-50,25)
tim.fillcolor('black')
tim.begin_fill()
tim.circle(10)
tim.end_fill()
tim.up()
tim.goto(50,25)
tim.begin_fill()
tim.circle(10)
tim.end_fill()
tim.up()
 	
# Finally draw the mouth 	
tim.goto(-70,-10)
tim.right(90)
tim.width(5)
tim.down()
tim.circle(70,180)
 	
# HIDE THE TURTLE 	
tim.hideturtle()**
	

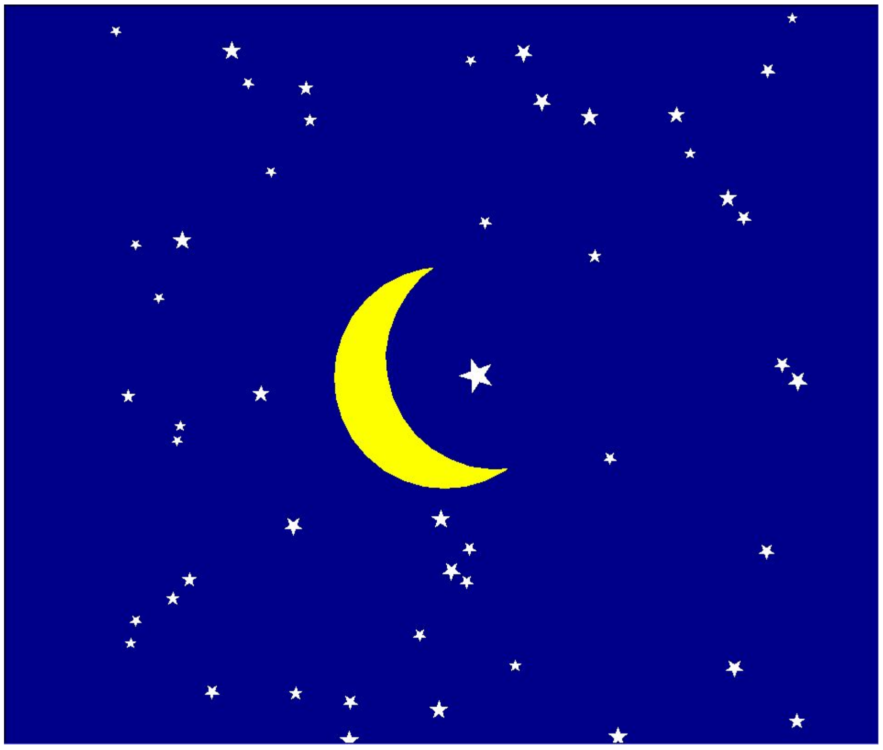
*Note: this example introduces **hideturtle()** so you can not show the turtle in your picture. If you hide the turtle and want to show it again, the method is **showturtle()***

Example 4: Starry Sky *(code is two pages!)*

Code *(Note: This is complete code. See appendix 1 for more info!)*

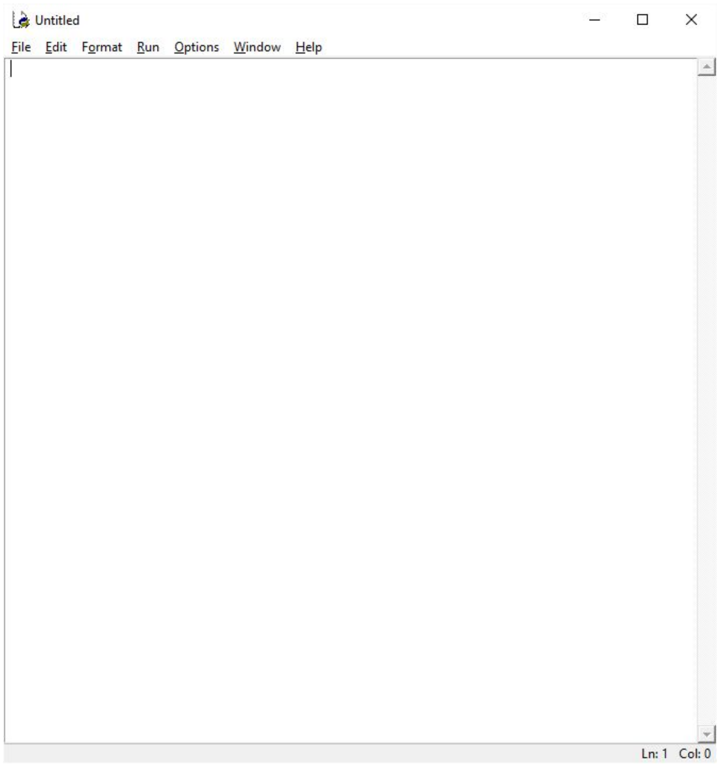
```
import turtle
import random

# Set up our program
tim=turtle.Turtle()
scr=turtle.Screen()
scr.screensize(400,400,'dark blue')

# speed things up
tim.speed(10)

# Create our Star function
def star(x, y, size):
    tim.up()
    tim.goto(x,y)
    tim.down()
    tim.color('white')
    tim.fillcolor('white')
    tim.begin_fill()
    tim.left(36)
    for i in range(5):
        tim.forward(size)
        tim.left(144)
    tim.end_fill()

# Draw 50 random stars
for i in range(50):
    starX = random.randint(-400,400)
    starY = random.randint(-400,400)
    starSize = random.randint(10,20)
    star(starX,starY,starSize)


# Draw the moon
tim.up()
tim.goto(0,-120)
tim.down()
tim.color('yellow')
tim.fillcolor('yellow')
tim.begin_fill()
tim.circle(120)
tim.end_fill()
tim.circle(120,20)
tim.up()
tim.forward(60)
tim.down()
tim.color('dark blue')
tim.fillcolor('dark blue')
tim.begin_fill()
tim.circle(120)
tim.end_fill()

# One more star to fill in the blank
star(30,-16,36)

# Hide the turtle
tim.hideturtle()
```

Output *(scaled to fit! Actual image is 400x400!)*



Appendix 1: Saving Your Work

In this book we have focused on using the Python shell to write short programs. This is great when you're learning how to code, because if there is an error in your code, Python can tell you right away, and you can instantly see results. The problem with this is that you can't re-use your code once you write it. If you want to run a program again later, you need to type in the code again.



Of course there is a different way. You can use the *Edit Window* instead of the Shell window. To do this, open Python like you always have, but then click **File** and then **New File**. You will be presented with a blank window like the one shown on the right.

In this window you can write your code like you always have. Just remember that every program needs to start with **import turtle** if you want to use turtle methods! Also remember to name your turtle (we use 'tim' in our examples).

One big difference you will notice right away is that when you type code and press Enter, nothing happens!! This is because the code in this window is saved into a file, and the WHOLE FILE is interpreted by Python and run at once. This allows you to put in all your code and edit it as much as you want, and go back and edit it again if you find that it doesn't work when you run it.

To run the code in the edit window, you either click **Run** and then **Run Module**, or **Press F5 on your keyboard**. Either way, if you haven't saved your file, it will ask you to save it before it runs the code. Just give it a name that describes what your code does and it will be saved as filename.py. For example, the starry sky example picture at the end of the last section is saved as *starry-sky.py* on my computer.

Once you get used to doing things this way, you will find that it is much nicer to code in the edit window, as it allows you to work on code little bits at a time and come back to it later to keep working, meaning that you can write much longer and more interesting programs, and even share your code with other people, or run code that other people wrote without having to type it all in!

Appendix 2: Useful Links

As you have no doubt discovered, Python is a fun, FREE, and relatively straightforward programming language designed to be picked up by anyone and used for nearly any purpose. Because of this, there are countless examples and how-to sites on the internet that will teach you everything from Graphics to Game Design to Web Application development in Python. Below you will find some links that I have found useful as I have learned the little bit I know of this fun language:

The Official Python Documentation Page for Turtle

URL/Link	https://docs.python.org/2/library/turtle.html
Author's Comment	This page lists **every** turtle command. **This page is THE Essential Reference for anyone using Turtle!** Python's docs are not always super easy to read, and tend to go WAY more in depth than a beginner would like, but try to get used to it and you'll find it to be worth its weight in gold.

How to Think Like A Computer Scientist: Learning with Python 3

URL/Link	http://openbookproject.net/thinkcs/python/english3e/index.html
Author's Comment	A great introduction to programming with Python which is WAY more thorough (and probably way more accurate) than my book. If my book leaves you wanting to learn more, this is where I recommend going next.

Intro to Python Turtles

URL/Link	http://interactivepython.org/runestone/static/IntroPythonTurtles/Summary/summary.html
Author's Comment	A really well done Python Turtle tutorial with interactive coding and quizzes, and definitions of tons of programming terms.

Python Turtle Graphics pt.1 (Youtube)

URL/Link	https://www.youtube.com/watch?v=6lyHpNjXqMI
Author's Comment	A nice youtube video that shows you how to get started using Python Turtle. There are many more such videos on youtube, so check them out!.