Table of Contents

Chapter 1

> Centuries ago a Westerner heard a Hindu say the earth rested on the back of a turtle. When asked what the turtle was standing on the Hindu explained, “It's turtles all **the** way down.”

Turtle Programming

In this chapter you'll learn to make the computer do things using programming. I'll introduce you to some of the basic tools of programming languages—loops, variables, functions, and a few other tricks —using Python's built-in turtle tool.

The Logo Turtle

Python's *turtle* module is a program that gives you control over a small image (a turtle!), like a video game character. You need to use precise instructions to direct it around the screen. The turtle leaves a trail wherever it goes so it can draw shapes as it progresses. The Python turtle is based on the original turtle from the Logo programming language, invented in the 1960s to make computer programming more accessible to both adults and children. The signature turtle environment made interacting with the computer visual and engaging. Seymour Papert's brilliant book *Mindstorms* contains lots of great ideas for learning math using turtles.

The people behind the Python programming language liked the Logo turtles so much they wrote a module called turtle to let us play around with turtle geometry in Python.

Importing the turtle module.py

To use turtles in Python, you have to import the functions from the turtle module first. Open a new Python file in IDLE. There are many ways to import functions from a module, and we'll use a simple one here. Enter the following at the top of the file:

```
from turtle import *
```

The `from` command indicates that we're importing something from outside our file. We then give the name of the filemodule (a file or files) we want to import from, in this case called ***turtle.py***, containing a lot of useful code that was automatically downloaded when you installed Python. We use the import keyword to get the useful code we want in our file`from` the *turtle library*module. The ***asterisk (*)*** is being used here as a wildcard command, meaning "import everythingall the functions from that module." Make sure to put a space between `import` and the asterisk.

This import gets the computer ready to use all the functions from the turtle module*turtle.py*: do a web search for "python turtle"; the first result should be the Python documentation (https://docs.python.org/3.6/library/turtle.html) that lists all the functions in the turtle module*turtle.py*, as in Figure 2-1:

Figure 12-1: The turtle documentation on the Python website

Scroll down this page a little and you should see all the turtle functions (or "*methods*"), as in Figure 12-2.

Figure 12-2: All the turtle methods on python.org

And that's not even the whole list!

All we're going to use at first is `forward`, which can be abbreviated "`fd`," for moving the turtle forward a certain number of steps (leaving a trail behind it) and `right` (abbreviation "`rt`"), for making the turtle turn right a certain number of degrees. Save this file , like name descriptiveawith as ***myturtles.py***; *or ashleyturtle.py (but(do* ***not save the file as*** *turtle.py. This filename already exists and will cause a conflict with the import from the turtle module!)*.

Drawing with Your Turtle

With the turtle module already imported, we can start putting in instructions for moving the turtle. To make the turtle go forward, enterwrite:

```
forward(100)
```

The parentheses are important, because `forward` is a function and it needs to know how many steps to move. Feel free to try other numbers, though. Save and run the program. As soon as you run it, a new window will open with an arrow in the center. The `forward(100)` command makes the arrow move 100 paces to the right , as in Figure 2-3:

Figure 12-3: Running your first line of code!

The turtle started in the middle of the screen and walked forward 100 steps. The default shape is an arrow and the default direction the arrow is facing is to the right. If you want the turtle to look like a turtle, enter this above the forward line”

```
shape('turtle')
```

Save and run your file again. Now your arrow will look like a tiny turtle, as in Figure 2-4:

Figure 12-4: The turtle looks like a turtle!

The turtle can only go in the direction it's facing, so to turn, you must make the turtle turn a specified number of degrees by using the `right` or `left` functions and then go forward. Add the following to the bottom of your program and run it again:

```
right(45)
forward(150)
```

The output should look like Figure 12-5:

Figure 12-5: The turtle turns!

See? The turtle started in the middle of the screen, went forward 100 steps, turned right 45 degrees and then went forward another 150 steps. Every line is run in order.

~~Exercise~~Example 12.1: Square Dance

~~Your~~The first challenge is to get the turtle to make a square. This only requires the `forward` and `right` functions. ~~(~~Solutions to all exercises are XXXXX, and~~)~~ ~~F~~feel free to look back at the earlier code to figure out how to complete the exercise.

All through the book, you'll get these challenges and it'll make you a better programmer and math student if you put in the effort to solve them yourself.

Solution 12.1:

The ~~code that will draw a~~ square is shown in Figure 1~~2~~-6:

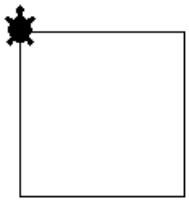

Figure 1~~2~~-6: IDLE editor showing the turtle square code (left) and the output (right)

PROD: CAN WE CROP OUT THE WINDOW HEADER?

Repeating Code with Loops

Your solution to Exercise 1~~2~~.1 has a lot of repeated code. ~~-~~Every programming language has a way to automatically repeat commands a given number of times to save you typing out the same code over and over and cluttering up your program.

The for Loop

In Python we use the `for` loop to repeat code. The ~~`for`~~`range` keyword ~~command~~ creates a ~~list~~sequence a certain number of items long, and the for keyword iterates (or "loops") over it. Open a new program file in IDLE and save it as *for_loop.py*, then enter the following and run the program:

```
for i in range(2):
    print("hello")
```

Be sure to indent all the lines of the code you want to repeat with a tab--one tab is 4 spaces. This tells Python which lines are inside the loop, so `for` knows what code to repeat. And don't forget the colon at the end; it tells the computer what's coming up next is in the loop. When you run the program you should~~'ll~~ get:

```
hello
hello
```

The text `hello` is repeated twice because “range(2)” is creates an iterator, a sequence containing two numbers, 0 and 1.the list “[0,1]”. The for command looped over the two items in the listsequence, printing “hello” each time. Change the number in the parentheses and you change the number of times the code is repeated:

```
for i in range(10):
    print(“hello”)
```

When you run this program, you'll get ten `hello`s:

```
hello
hello
hello
hello
hello
hello
hello
hello
hello
hello
```

To repeat code four times, you'd enter:

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

Simple.

A Simple Square

In our turtle program, to make a square, we want to repeat `forward(100)` and `right(90)` four times. Replace your `forward` and `right` commands in your my*turtles.py* program with the following:

```
for i in range(4):
    forward(100)
    right(90)
```

Run the program and you'll see the square, as in Figure 12-7:

Figure 12-7: A square made with a loop

The way this works is that the variable `i` is an *iterator*—a value that increases each time it is used--and the `range(10)` command gives us a ~~list~~sequence of 10 consecutive numbers. Here I've converted the sequence "range(10)" to a list and printed it out:

```
>>> print(list(range(10)))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

In Python, counting begins at 0 rather than 1, so we get the numbers 0 through 9. You can use the value of i in your code:

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

The first line is saying "for each value in the range 0 to 9, apply the indented code that comes next". The second line uses the `print` command, which just displays its argument to the shell. So in all, this program is saying "for each value in the range 0 to 9, display the current number", and the `for` loop repeats the code until it runs out of numbers in the range. You'll get:

```
0
1
2
3
4
5
6
7
8
9
```

In the future we'll have to remember that in a loop, and in Python in general, `i` starts at 0 and ends before the last number, but for now, if you want something repeated 4 times, it's:

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

Code Shortcuts with Functions

Now that we've written code to make a square, we can save all that code to a magic keyword that we can use anytime to call that square code again. Every programming language has a way to do this, and in Python it's called a *function*. Functions are the most important feature of computer programming.

They make code compact and easier to maintain, and dividing a problem up into functions often allows you to see the best way of solving it.

To define a function you start by giving it a name. This name can be anything you want, ~~but~~as long as it's not already a Python keyword, like "list," "range" and so on. When naming functions it's better to be descriptive so you can remember what it's for when you use it again. We'll call our function `square`:

FILE: MYTURTLES.PY

```
def square():
    for i in range(4):
        forward(100)
        right(90)
```

The `def` command tells Python we're defining a function, and the word we list afterwards will become the function name; here it's **`square`**. Don't forget the parentheses! They're a sign in Python ~~(and other languages)~~ that you're dealing with a function. ~~The parentheses are empty now, but l~~Later we'll put values inside them, ~~so~~but they need to be included for Python to know you mean a function. As you did in the loop, don't forget the colon at the end of the function definition. ~~.~~Note that all the code inside the function is indented, so Python knows which code goes inside it.

If you run this program now, nothing will happen. You've defined a function, but you didn't tell the program to run it yet. You can call it at the end of the my*turtles.py* file after the function definition, as in Figure 1~~2~~-8:

Figure 1~~2~~-8: The square function is called at the end of the file.

When you run the program now, ~~it'll~~the turtle will draw a square. You can use the `square` function at any point later in the program to quickly draw another square.

Once you've defined a function, you can use it in a loop to build something more complicated. Let's say we want to draw a square, turn right a little, make another square, turn right a little, a bunch of times, of course we're going to use a loop. Now we can call the square function and the program will remember how to make a square.

Exercise 1~~2~~.2: A Circle of Squares

Write and run a function to draw 60 squares, turning right 5 degrees after each square. Use a loop! It should end up looking like Figure 12-9.

Figure 12-9: A circle of rotated squares

Interesting shape, and it's all made out of squares! It might take your turtle a while to walk all that way, so you can speed it up by adding the `speed(0)` function to your the *myturtles.py program* after `shape('turtle')`. Speed(0) will make the turtle draw the fastest, speed(1) is the slowest. Try different speeds, like speed(5) and speed(10), if you want.

Solution 12.2:

```
def squareThing():
    for i in range(60):
        square()
        right(5)

squareThing()
```

Changing Sizes with Variables

So far all our squares are the same size. In order to make different sized squares, we'll need to vary the distance the turtle walks forward for each side. Instead of changing the definition for the `square` function every time we want a different size, we can use a variable. A *variable* in Python is a word that represents a value you can change, like how in algebra x can represent ...

In math class, variables are single letters, but in programming you can give a variable any name you want, as long as it's not already a Python keyword. Like functions, I suggest naming variables something descriptive to make it easier to remember.

Variables in Functions

Function definitions can take variables as "parameters" inside the parentheses. Then when you call the function, you can place a value (called an "argument") inside the parentheses that we left blank earlier, and whatever number is put inside the parentheses will be used in place of the variable in your function. For example, change your square function to the following, and you can create squares of any size you want rather than a fixed size:

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

```
        forward(sidelength)
        right(90)
```

The square function takes the parameter "sidelength." sidelength is a variable, and we can enter different numbers (arguments) in the parentheses when calling the `square` function to make the sides different lengths. Now calling `square(50)` and `square(80)` would look like Figure 2-10:

Figure 12-10: A square of size 50 and a square of size 80.

Variable Errors

At the moment, if we forget to put a value in the parentheses for the function we'll get this error:

```
Traceback (most recent call last):
  File "C:/Something/Something/my_turtle.py", line 12, in <module>
    square()
TypeError: square() missing 1 required positional argument: 'sidelength'
```

This tells us that we're missing a value for length, so Python doesn't know how big to make the square. To avoid this, we can give a default value for the length in the first line of the function definition:

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

We place a value of 100 in `length` by default. Now if we put a value in the parentheses, `square` will make a square of that length, but if we leave the parentheses empty, it'll default to a square of sidelength 100. This code will produce the drawing in Figure 2-11:

```
square(50)
square(30)
square()
```

Figure 12-11: A default square of size 100, a square of size 50 and a square of size 30.

This makes it easier to use our function without having to worry about getting errors if we do something wrong, known in programming as making the program more *robust*.

Making Variables Vary

There's more we can do with variables: we can automatically increase the variable by a certain amount so that each time the function is run the square is bigger than the last. With ~~our~~a `length` variable, we

can make a square, then increase the length variable a little before making the next square, by incrementing the variable like this:

```
length = length + 5
```

As a math guy, this line of code didn't make sense to me when I first saw it! How can "length equal length + 5"? It's not possible! But code isn't an equation, and an equals sign doesn't mean "this side equals that side." **The equals sign in programming means assigning a value.**

```
radius = 10
```

means we're creating a variable called radius (if there isn't one already) and assigning it the value 10. You can always assign it a different value later:

```
radius = 20
```

If you need to check whether a variable is equal to something, you **use double equals signs**. To check whether the value of the radius variable is 20, you write this:

```
radius == 20
```

press ENTER and it will print out True:

```
True
```

Now the value of the radius variable is 20. It's often useful to increment variables rather than assigning them number values manually. You can use a variable called "count" to count how many times something happens in a program. It starts at 0 and every loop it goes up by one. How do you make a variable "go up by one"? You have to add 1 to its value and then assign the new value to the variable. It could look like this:

```
count = count + 1
```

or this more compact way:

```
count += 1
```

This means "add 1 to my count variable." You can use addition, subtraction, multiplication and division in this notation. Let's see it in action in this code I ran in the Python shell:

```
>>> x = 12
>>> y = 3
>>> x += y
```

```
>>> x
15
>>> y
3
>>> x += 2
>>> x
17
>>> x -= 1
>>> x
16
>>> x *= 2
>>> x
32
>>> x /= 4
>>> x
8.0
```

That's why we use this line of code to make the length increment by 5 every loop:

```
length += 5
```

Every time the length variable is used, like in a loop, a length of 5 is added to the value and saved into the variable....

Exercise 12.3: A Turtle Spiral

Make a function to draw 60 squares, turning 5 degrees after each square and making each successive square bigger. Start at a `length` of 5 and increment 5 units every square.

It should like Figure 12-12:

Figure 12-12: A spiral of squares.

It's interesting that such simple code produces such complicated-looking designs.

Solution:

```
def spiral():
    length = 5
    for i in range(60):
        square(length)
        rt(5)
```

```
        length += 5

spiral()
```

Drawing Polygons

We've taught the turtle to make a square, so now let's teach it to draw a triangle. Let's write a triangle function that will make the turtle walk in a triangular path.

You probably know that the angle in an equilateral triangle is 60 degrees. But if you change the `right` movement in your square function to `60`, you won't get a triangle:

```
def triangle(sidelength=100):
    for i in range(3):
        forward(sidelength)
        right(60)

triangle()
```

Instead, you'll see the shape in Figure 12-13:

Figure 12-13: A first attempt at a triangle.

That looks like a promising start on a hexagon but it's not a triangle. That's because the turning degrees you enter in your program is the *external* angle, but 60 degrees is the ***internal*** angle of an equilateral triangle. This wasn't a problem with the square because it just so happens the internal angle of a square and the external angle were the same, at 90 degrees. To find the external angle for our triangle, simply subtract the internal angle from 180. So the external angle of an equilateral triangle is 120 degrees. Change the turn in the code above to 120 and you should get a triangle.

Exercise 12.4: Polygon Functions

Write a function called `polygon` that will take an integer as an argument and make the turtle draw a polygon with that integer's number of sides. First you need to figure out how many degrees the turtle has to turn in order to get back to its starting point, facing in the same direction, which is of course 360 degrees. That means that you need to divide 360 degrees by the number of sides we want the shape to

have, to get the angle that will make a complete, closed shape. Call the variable to hold the number of turns num. You'll need to divide 360 by whatever is in num to get the angle for each turn.

Solution:

~~Let's call the numbers of turns variable `num`. The number of degrees in each turn will be 360/num.~~ There are lots of shapes you can make with this function, but the core functionality on the function should be the same. Here's an ~~pentagon~~ example that draws a pentagon:

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

polygon(5)
```

This will produce a 5-sided regular polygon, like the one in Figure 12-14:

Figure 12-14: Using the polygon function to produce a pentagon.

Exercise 1.2-5: All Shapes and Sizes!

Now use what you've learned to produce the following designs.

Hint: to change the color, add a `color()` command after the `speed()` line in your program, with a color as an argument. Don't forget to put the argument in quote marks!

Figure 1-15: Nine designs to recreate for exercise 1.5

The Big Picture

The simple things you've learned in this chapter reflect some of the most important ideas of mathematics I want to get across: In this book you'll keep coming across these big ideas:

Computation – ~~a major application of computers,~~how computers crunch~~ing~~ numbers and find~~ing~~ "solutions."

Measurement – extending simple patterns to analyze complicated ones

Transformation - applying formulas to turning numbers into other numbers using formulas, or changing shapes by moving or rotating them

Storage – storiputting values numbers in variables or lists and get them laterfindgo back you can wheresothat you can callreference to use the values later

Models – makcreating a simplified modelversion of a physical situation so you can play around with itunderstand it better

Reasoning – the ability to e b should'll you, the bookthroughoutall overthinking about whichthe tools you can apply to whicha situation to help solve a problems.

The turtle represents an agent through which you can experience the world of math. As you saw in the documentation, the turtle can perform a lot of functions. We only used `forward` and `right`, but there are dozens more that I encourage you to experiment with before moving on to the next chapter. The turtle object ***stores*** all of that functionality, just like a variable stores a value, such as a number or some text.

We made the turtle walk around the screen, ***transforming*** its position and ***measuring*** distance. This is no small feat! We learned to ***transform*** numbers using loops and functions and we transformed shapes like squares using variables. In future chapters we'll create ***models*** to simulate real situations, like planetary orbits or living cells. This is an important application of math, and of course an important application of computer programming.

All the while the computer was performing a bunch of *computations* (adding, subtracting, multiplying, dividing) behind the scenes. Computation used to be the most important thing in a math class, because it's easy to mark a computation right or wrong, but those computations are what's known as a low-level task that's better left to a computer so we can concentrate on bigger ideas, like how to make beautiful designs like fractals or spirals. We have to be able to test our functions to make sure they will give the right output, but after that we let the computer do the number crunching.

While the computer is doing its thing, we have to do ours, which is ***reasoning***. Whether you're doing arithmetic, art, or science using programming, you have to ***reason about*** the input you're giving to a function and the output you're receiving, meaning you need to think it through and make sure you understand what's going on. What's going into a function, and what's coming out? Is itthe output what we expected? Is there a better way?

These ideas will keep resurfacing throughout the book in many different forms, so keep an eye out!

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

```
def fourCrosses():
    for i in range(4):
        cross()
        rt(90)
```

```
def eightCrosses():
    for i in range(2):
        fourCrosses()
        rt(45)
```

```
def polygon(length,sides):
    for i in range(sides):
        fd(length)
        rt(360/sides)


def hexagonalStar():
    for j in range(6):
        for i in range(2):
            fd(100)
            rt(60)
            fd(100)
            rt(120)
        polygon(100,3)
        rt(60)
```

```
def expandingTriangle():
    length = 5
    for i in range(100):
        fd(length)
        rt(120)
        length += 5
```

```
def expandingSquare():
    length = 5
    for i in range(60):
        for j in
range(2):
            fd(length)
            rt(90)
            length += 5
```

```
def ExpandingSquare2():
    length = 5
    for i in range(60):
        for j in range(2):
            fd(length)
            rt(91)
            length += 5
```

```
def rose1():
    '''It's really a col-
lection of octagons.'''
    for i in range(8):
        polygon(100,8)
        rt(45)
```

```
def rose2():
    '''It's really a
collection of
hexagons.'''
    for i in range(8):
        polygon(100,6)
        rt(45)
```