2 [ChapterStart]

Turtles

In this chapter you'll learn to make the computer do things using programming. You'll also learn why programming languages use all the tools they do: loops, variables, functions and so on.

The Logo Turtle

In the 1960s the Logo programming language was invented to make computer programming more accessible, even to children. Its 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 file so we can do turtle geometry using Python. It's called turtle.py and all you have to do at the beginning of your program is import the functions from the turtle module. There are many ways to do this, but for starting out we can just type

```
from turtle import *
```

The asterisk just means “wildcard,” meaning “import all the functions from that module.” there's a space between “import” and the asterisk. This gets the computer ready to use all the functions from turtle.py: they're all listed in the Python documentation. Do a web search for “python turtle” and it'll be the first thing that pops up, as you can see in Figure 2-1:

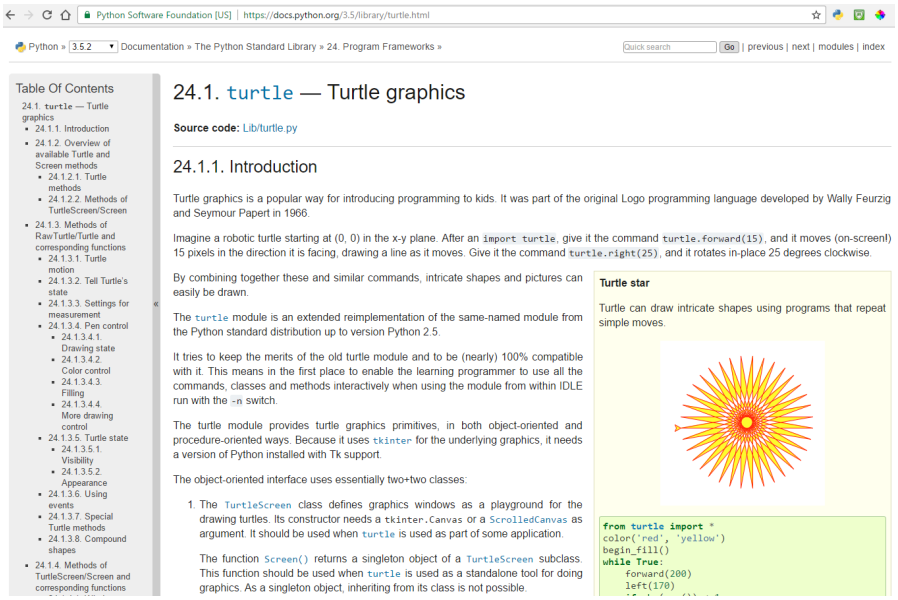


Figure 2-1: The turtle documentation on the Python website

Down the page it lists all the turtle functions you can use:

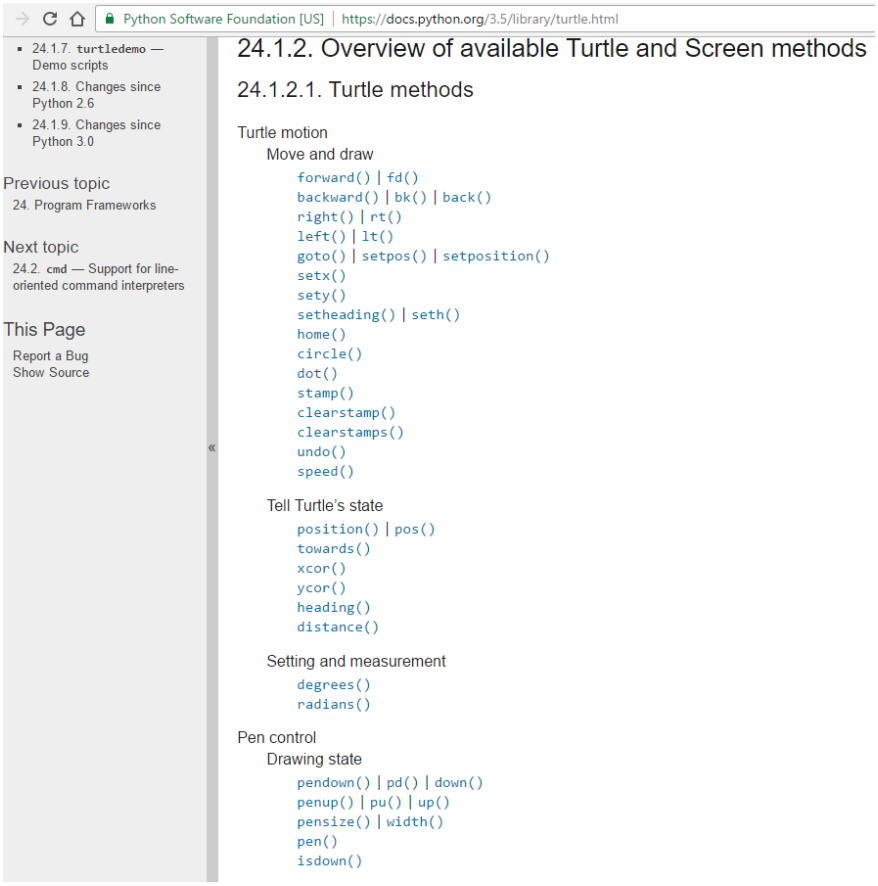


Figure 2-2: All the turtle methods on python.org

And that's not even the whole list! All we're going to use at first is “forward,” for going forward a certain number of steps and “right,” for turning right a certain number of degrees.

Open a new Python file in IDLE or Trinket and import the turtle module at the top. Now you can make the turtle go forward by adding

```
forward(100)
```

and running the program. You'll see what's in Figure 2-3:

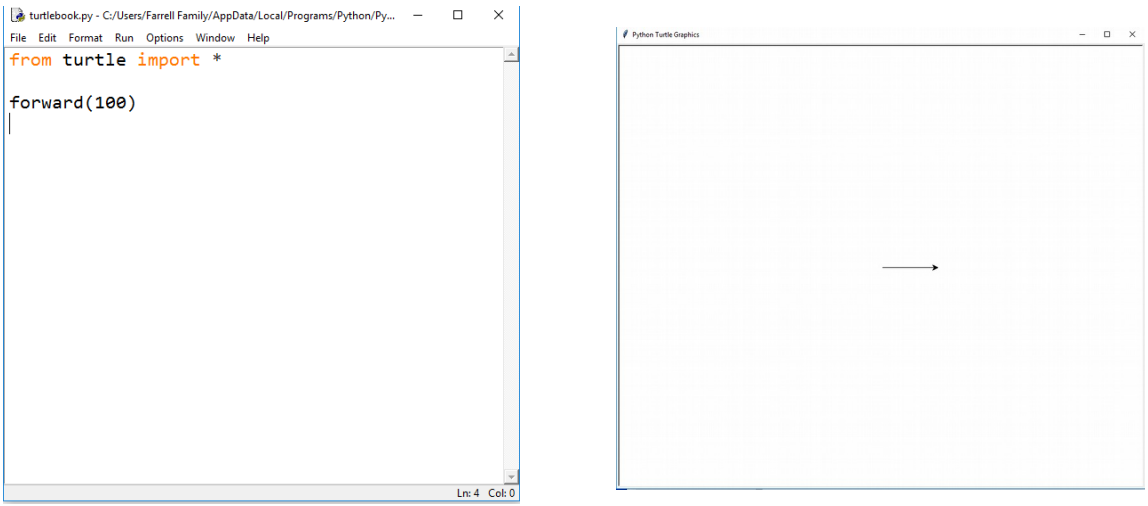


Figure 2.3: Running your first line of code!

The turtle started in the middle of the screen and walked forward 100 steps. The default direction is to the right and the default shape is an arrow. If you want the turtle to look like a turtle, type this before "forward:"

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

Now it'll look like a turtle, like Figure 2-4:

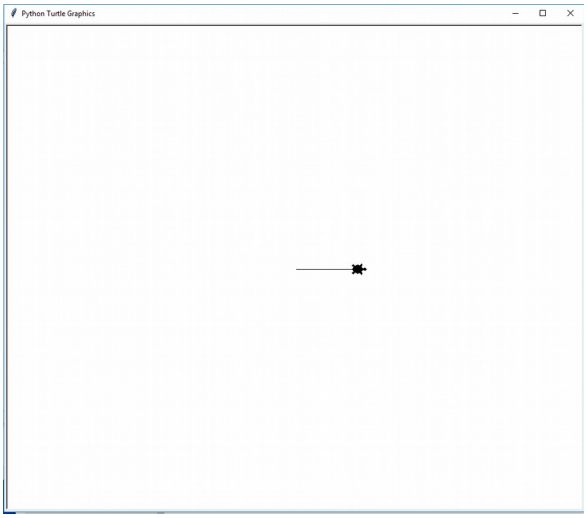


Figure 2-4: The turtle looks like a turtle!

You can make the turtle turn a specified number of degrees by using the "right" or "left" functions:

```
forward(100)
right(45)
forward(150)
```

The output will look like Figure 2-5:

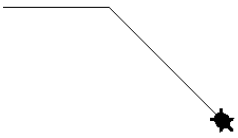


Figure 2-5: The turtle turns!

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 2.1

Your first challenge is to make the turtle walk so its path makes a square. This only requires the "forward" and "right" functions. **Try this before looking ahead at the code!** Of course, you can look back at the old code.

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:

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

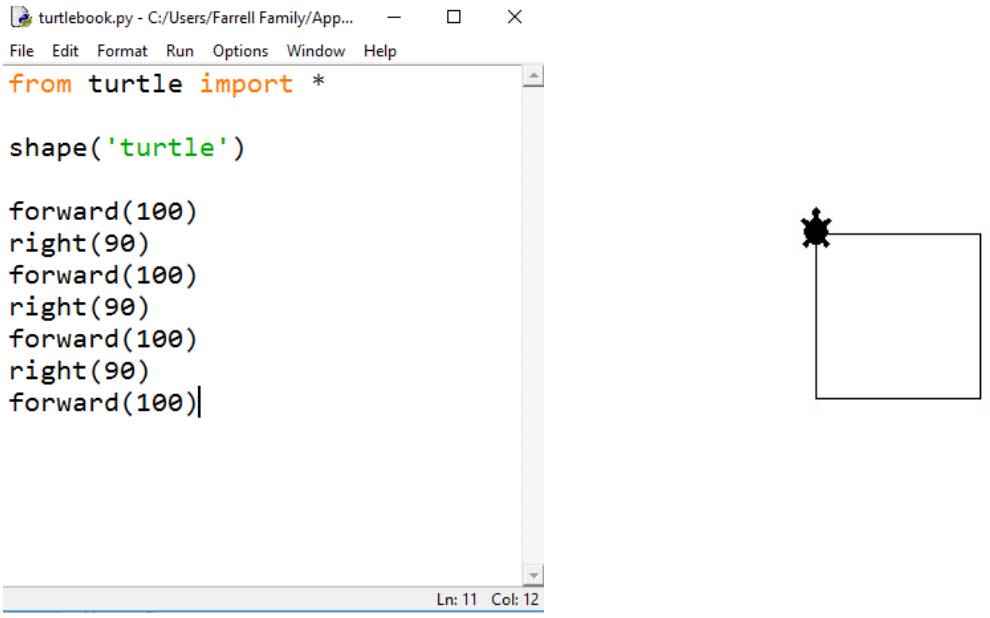


Figure 2-6: IDLE editor showing the turtle code (left) and the output (right)

Loops

As you can see in Figure 2-6, there's a lot of repeated code. It's a bunch of "forward(100)" and "right(90)." in every programming language there's a way to make the computer repeat commands a given number of times.

In Python the way to repeat something twice is to type:

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

Be sure to indent all the lines of the code you want to repeat. One tab is 4 spaces. And don't forget the colon at the end. You'll get

```
hello
hello
```

Change the number in the parentheses and you change the number of times the code is repeated:

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

You'll get ten hello's:

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

To repeat code four times, it's

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

So in our turtle program, to make a square, we want to repeat `forward(100)` and `right(90)` four times. Here's how:

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

Run this and you'll see the square, as in Figure 2-7:

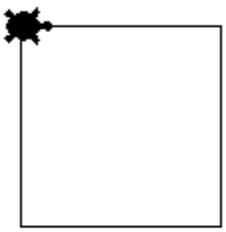


Figure 2-7: A square made with a loop

How does this work?

The variable i is an iterator, and “range(10)” is a list of 10 numbers:

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

You can use the value of i in your code:

```
for i in range(10):
    print(i, end=" ")
```

The `end=" "` part means not to do a line break. Everything will be on the same line. You'll get

```
0123456789
```

In the future we'll have to remember in a loop, 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):
```

Functions

Now that we've made a square, we can save all that code to a magic word so we can use it anytime. Every programming language has a way to do this, and in Python it's called a Function. The way to define a function is by giving it a name. You can name it anything you want, but it's better to be descriptive. We'll call our function “square”:

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

Important: all the code inside the function is indented again. As you did in the loop, don't forget the colon at the end of the function definition. And the parentheses are important. They're empty now, but later we'll put values inside them. But if you run this, nothing will happen. You've defined a function, but you didn't tell the program to run it. You can call the function from the shell:

```
>>> square()
```

Or you can call it at the end of the file, as in Figure 2-8:

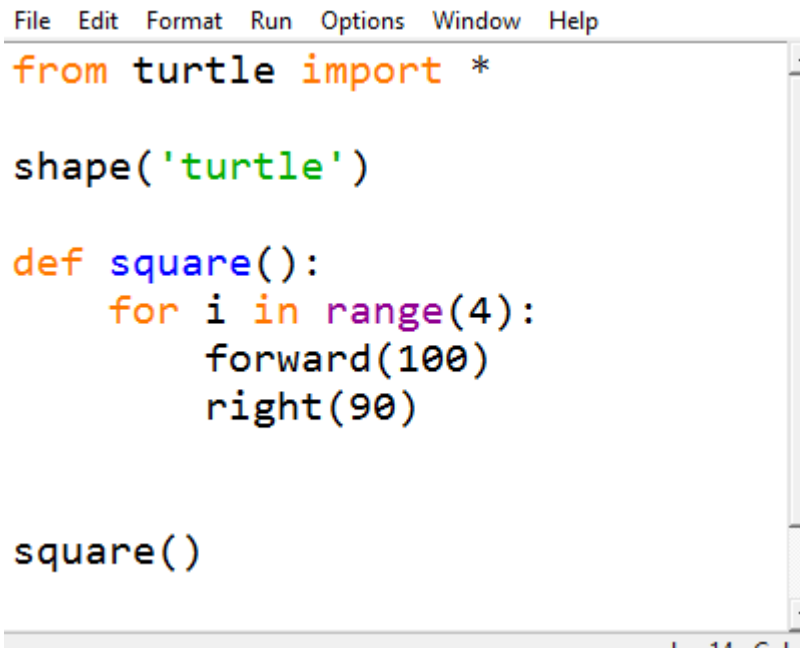


Figure 2-8: The square function is called at the end of the file.

Now it'll draw a square.

Using Functions

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 2.2

Draw 60 squares, turning right 5 degrees after each square. Use a loop!

Solution:

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

squareThing()
```

Run it and you'll see what's in Figure 2-9:

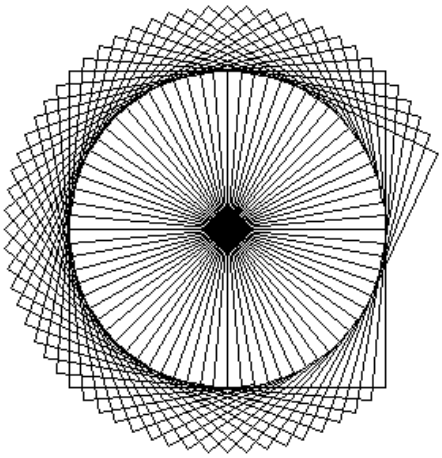


Figure 2-9: A circle of rotated squares

Interesting shape, and it's all made out of squares! Now that we're making the turtle walk more, add "`speed(0)`" to the beginning of your code (after `shape('turtle')`) to make it walk faster.

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. Instead of changing the square function definition every time we want a different size, we can use a variable.

In math class, variables are single letters, but in programming you can call a variable anything you want. Like functions, I suggest naming variables something descriptive.

Function definitions can take variables as "arguments" or "parameters" inside the parentheses. Then whatever number is put inside the parentheses will be used where you want it in the function. If we change our square function to this, we can create squares of any size we want:

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

Now calling square(50) and square(80) would look like Figure 2-10:

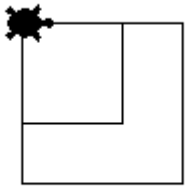


Figure 2-10: A square of size 50 and a square of size 80.

But if we ever forget to put a value in the parentheses 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: 'length'
```

If we give a default value for the length in the function definition we won't get that error:

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

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

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

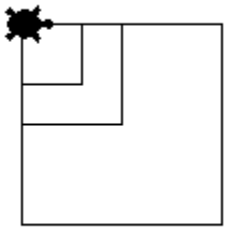


Figure 2-11: A default square of size 100, a square of size 50 and a square of size 30.

Making Variables Vary

What would it look like if we made a square, then increased the length variable a little before making the next square? You can increment a variable this way:

```
length = length + 5
```

or this more compact way:

```
length += 5
```

Exercise 2.3

Make 60 squares, turning 5 degrees after each square and making each successive square bigger. Start at a sidelength of 5 and increment 5 units every square.

Solution:

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

spiral()
```

It'll look like Figure 2-12:

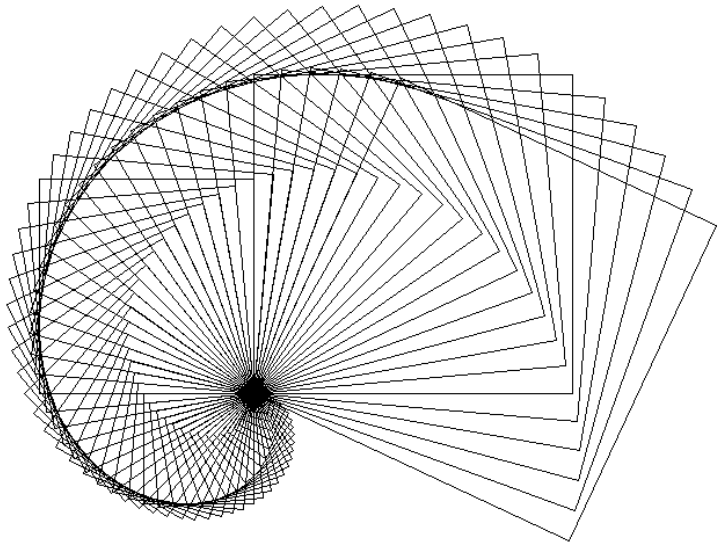


Figure 2-12: A spiral of squares.

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

Polygons

We've taught the turtle to make a square, so now let's teach it to draw a triangle.

Exercise 2-4:

Write a triangle function that will make the turtle walk in a triangular path.

Solution:

Many people know that the angle in an equilateral triangle is 60 degrees. But if you change your square function to this, you won't get a triangle:

```
def triangle(length=100):
    for i in range(3):
        fd(length)
        rt(60)

triangle()
```

You'll get what's in Figure 2-13:

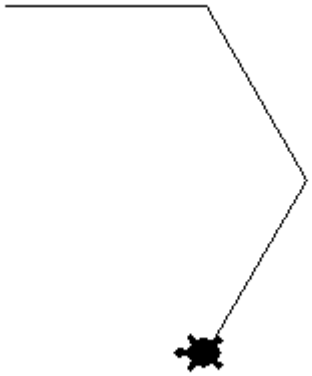


Figure 2-13: A first attempt at a triangle.

That looks like a promising start on a hexagon but it's not a triangle. That's because 60 degrees is the **internal** angle of an equilateral triangle, and the turtle turns the **external** angle. It just so happens the internal angle of a square and the external angle were the same, 90 degrees. To find the external angle 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.

Polygon Function

Exercise 2.5

Write a function called “polygon” that will take an integer as an argument and make the turtle draw a polygon with that number of sides.

Solution:

How many degrees does the turtle have to turn in order to get back to its starting point, facing in the same direction? 360 degrees, of course. How many turns will it make? Let's call the variable num. The number of degrees in each turn will be 360/num.

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

polygon(5)
```

This will produce a pentagon (5-sided regular polygon), like the one in Figure 2-14:

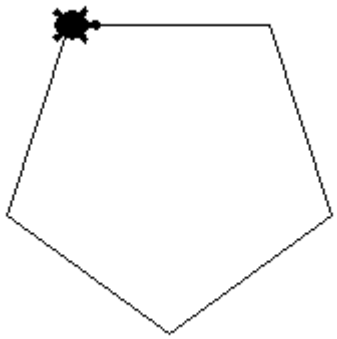
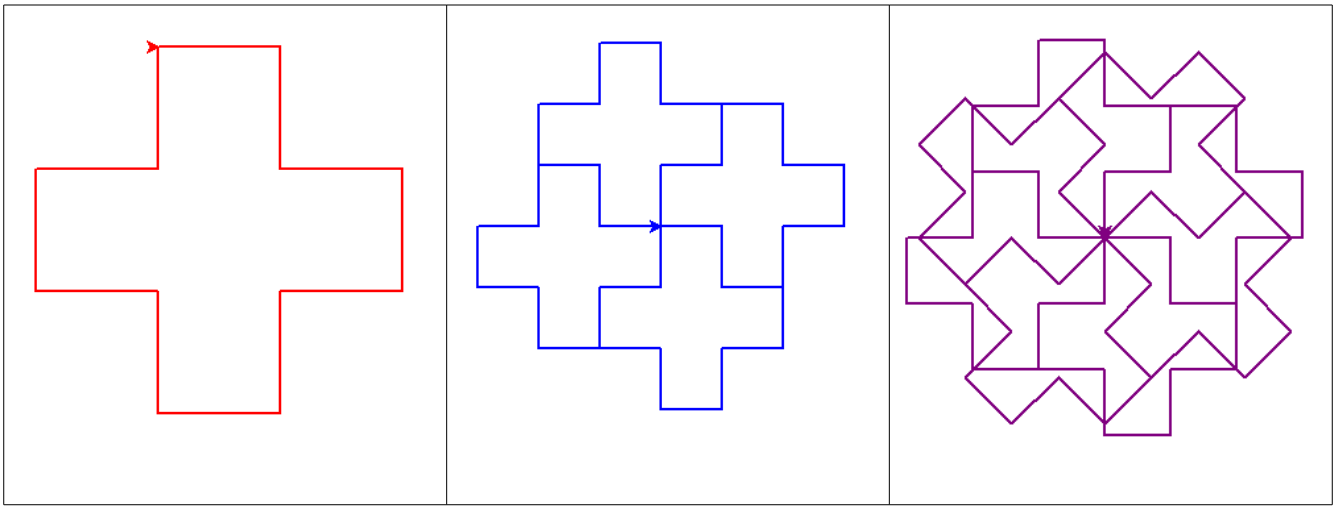
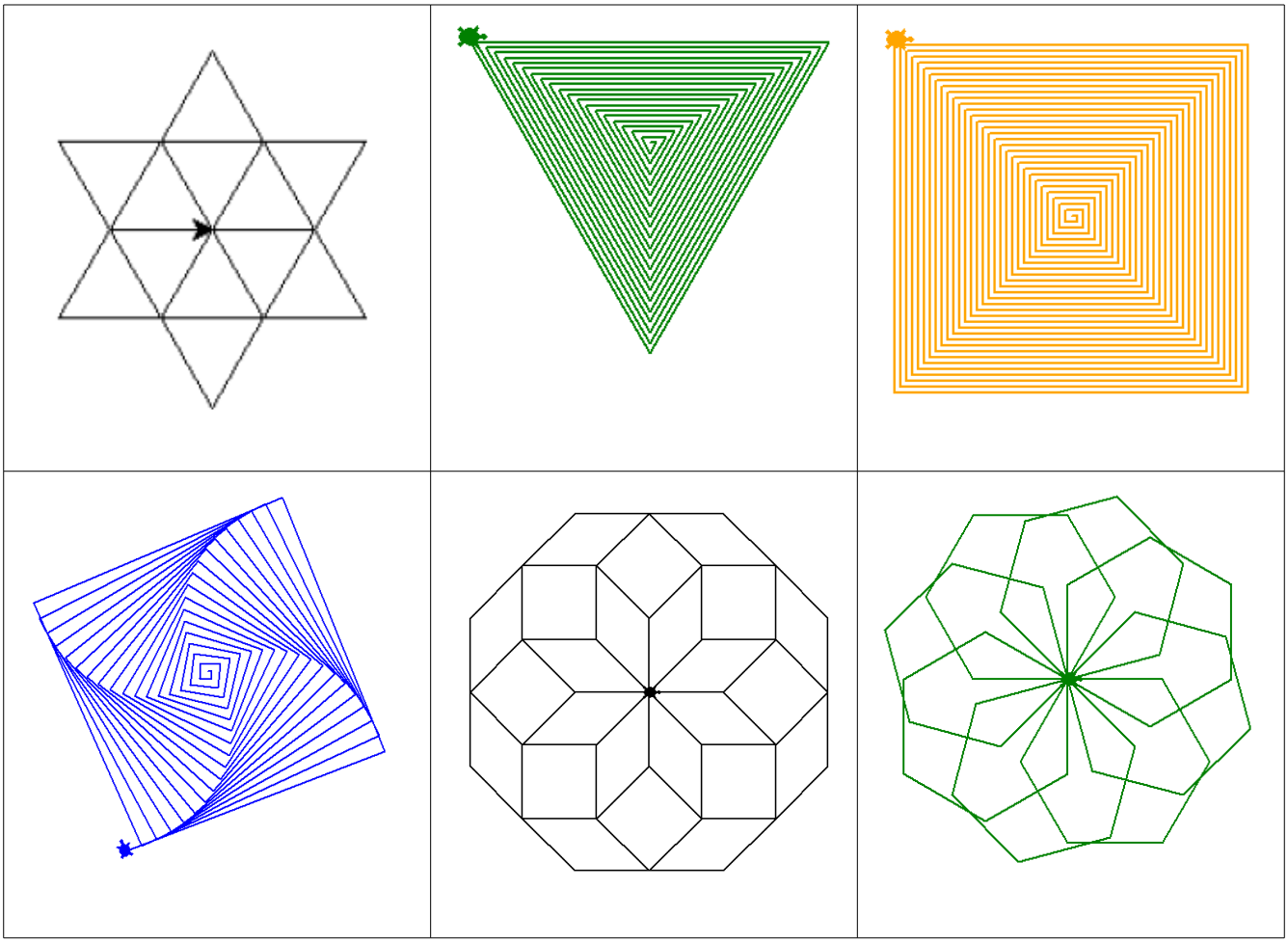


Figure 2-14: Using the polygon function to produce a pentagon.

Now use what you've learned to produce these designs:





The Big Picture

The simple things you've learned in this chapter reflect the most important ideas of Mathematics I want to get across. I've boiled down the whole picture of Math to a few big ideas:

Computation

Measurement

Transformation

Storage

Models

Reasoning

The Turtle represents an agent through which you can experience the world of math. As you saw in the turtle documentation, the Turtle can perform a lot of functions. We only used “forward” and “right,” but there are dozens more. The Turtle object **stores** that functionality. A variable can be seen as another **storage** device; it **stores** a value, such as a number or a string. 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 well create **models** to simulate real situations, which 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 a “low-level” task that's better left to a computer. We have to be able to test our functions to make sure they will give the right output, but after that we let them do the number crunching. So 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.

These ideas will keep resurfacing throughout the book, so I hope they're easy to remember.