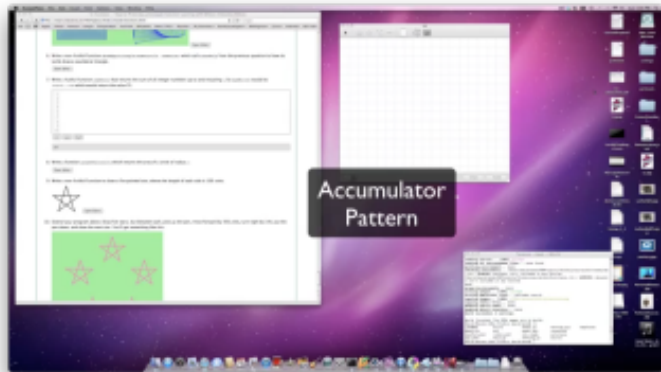


The Accumulator Pattern



In the previous example, we wrote a function that computes the square of a number. The algorithm we used in the function was simple: multiply the number by itself. In this section we will reimplement the square function and use a different algorithm, one that relies on addition instead of multiplication.

If you want to multiply two numbers together, the most basic approach is to think of it as repeating the process of adding one number to itself. The number of repetitions is where the second number comes into play. For example, if we wanted to multiply three and five, we could think about it as adding three to itself five times. Three plus three is six, plus three is nine, plus three is 12, and finally plus three is 15. Generalizing this, if we want to implement the idea of squaring a number, call it n , we would add n to itself n times.

Do this by hand first and try to isolate exactly what steps you take. You'll find you need to keep some "running total" of the sum so far, either on a piece of paper, or in your head. Remembering things from one step to the next is precisely why we have variables in a program. This means that we will need some variable to remember the "running total". It should be initialized with a value of zero. Then, we need to **update** the "running total" the correct number of times. For each repetition, we'll want to update the running total by adding the number to it.

In words we could say it this way. To square the value of n , we will repeat the process of updating a running total n times. To update the running total, we take the old value of the "running total" and add n . That sum becomes the new value of the "running total".

Here is the program in activecode. Note that the function definition is the same as it was before. All that has changed is the details of how the squaring is done. This is a great example of "black box" design. We can change out the details inside of the box and still use the function exactly as we did before.

```
1 def square(x):
2     runningtotal = 0
3     for counter in range(x):
4         runningtotal = runningtotal + x
5
6     return runningtotal
7
8 toSquare = 10
9 squareResult = square(toSquare)
10 print("The result of", toSquare, "squared is", squareResult)
11
```

ActiveCode: 1 (sq_accum1)

Run

The result of 10 squared is 100

In the program above, notice that the variable `runningtotal` starts out with a value of 0. Next, the iteration is performed `x` times. Inside the for loop, the update occurs. `runningtotal` is reassigned a new value which is the old value plus the value of `x`.

This pattern of iterating the updating of a variable is commonly referred to as the **accumulator pattern**. We refer to the variable as the **accumulator**. This pattern will come up over and over again. Remember that the key to making it work successfully is to be sure to initialize the variable before you start the iteration. Once inside the iteration, it is required that you update the accumulator.

Note

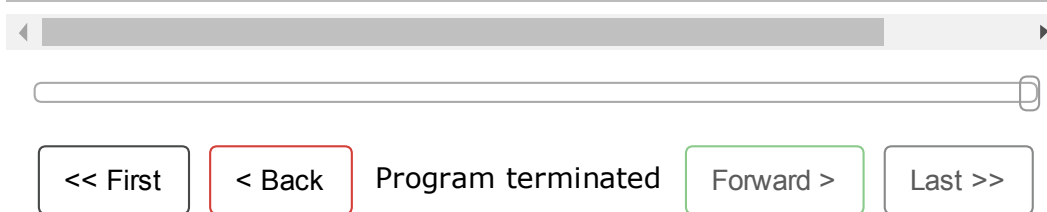
What would happen if we put the assignment `runningTotal = 0` inside the for statement? Not sure? Try it and find out.

Here is the same program in codeLens. Step thru the function and watch the “running total” accumulate the result.

```

1 def square(x):
2     runningtotal = 0
3     for counter in range(x):
4         runningtotal = runningtotal + x
5
6     return runningtotal
7
8 toSquare = 10
9 squareResult = square(toSquare)
→ 10 print("The result of", toSquare, "squared is", squ

```



→ line that has just executed

→ next line to execute



Program output:

```
('The result of', 10, 'squared is', 100)
```

Frames

Objects

Global variables

square	
toSquare	10
squareResult	100

function
square(x)

CodeLens: 1 (sq_accum3)**Note**

This workspace is provided for your convenience. You can use this activecode window to try out anything you like.

1	
2	

ActiveCode: 2 (scratch_05_04)

Run

Check your understanding

func-13: Consider the following code:

```
def square(x):  
    runningtotal = 0  
    for counter in range(x):  
        runningtotal = runningtotal + x  
    return runningtotal
```

What happens if you put the initialization of runningtotal (the line runningtotal = 0) inside the for loop as the first instruction in the loop?

- ☒ a) The square function will return x instead of x*x
- ☐ b) The square function will cause an error
- ☐ c) The square function will work as expected and return x*x
- ☐ d) The square function will return 0 instead of x*x

Check Me

Compare Me

func-14: Rearrange the code statements so that the program will add up the first n odd numbers where n is provided by the user.

Drag from here

Drop blocks here

```
n = int(input('How many even numbers would you like to add together?'))  
thesum = 0  
odddnumber = 1
```

```
for counter in range(n):
```

```
    thesum = thesum + oddnumber  
    oddnumber = oddnumber + 2
```

```
print(thesum)
```

Check Me

Reset

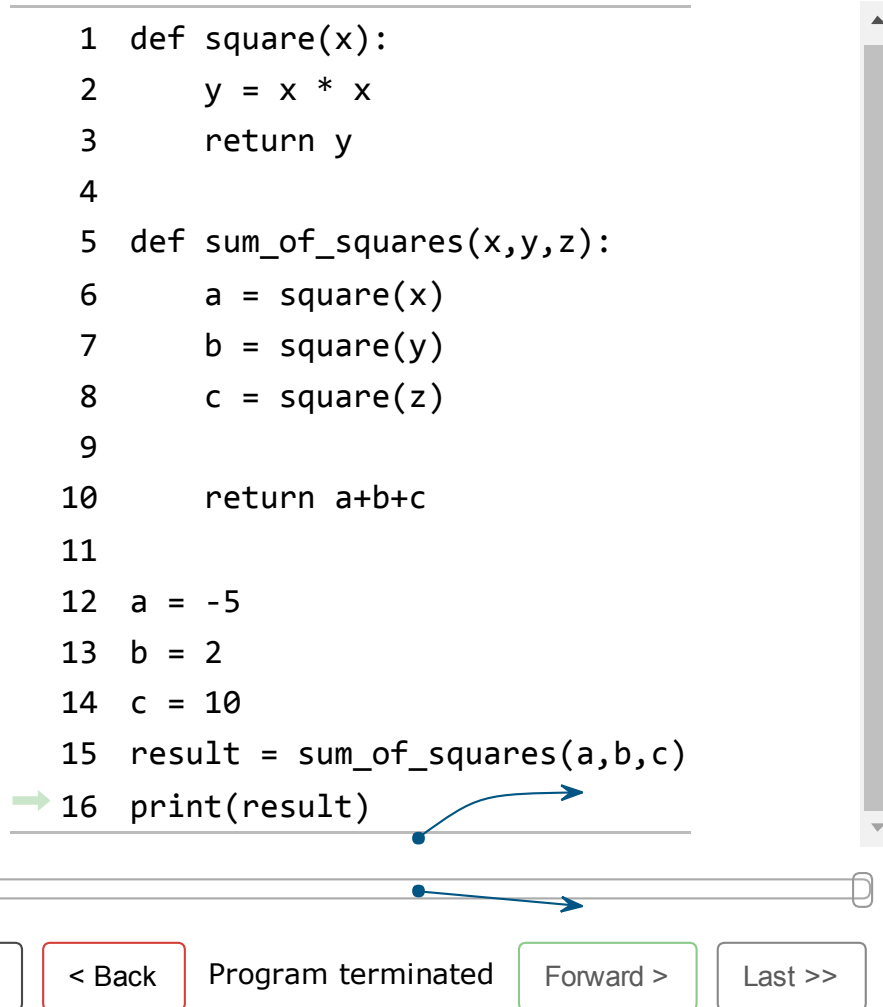
Perfect!

Functions can call other functions

It is important to understand that each of the functions we write can be used and called from other functions we write. This is one of the most important ways that computer scientists take a large problem and break it down into a group of smaller problems. This process of breaking a problem into smaller subproblems is called **functional decomposition**.

Here's a simple example of functional decomposition using two functions. The first function called `square` simply computes the square of a given number. The second function called `sum_of_squares` makes use of `square` to compute the sum of three numbers that have been squared.

```
1 def square(x):
2     y = x * x
3     return y
4
5 def sum_of_squares(x,y,z):
6     a = square(x)
7     b = square(y)
8     c = square(z)
9
10    return a+b+c
11
12 a = -5
13 b = 2
14 c = 10
15 result = sum_of_squares(a,b,c)
→ 16 print(result)
```



→ line that has just executed

→ next line to execute

Program output:

129

Frames

Objects

Global variables

square	
sum_of_squares	
a	-5
b	2
c	10
result	129

function

square(x)

function

sum_of_squares(x, y, z)

CodeLens: 2 (sumofsquares)

Even though this is a pretty simple idea, in practice this example illustrates many very important Python concepts, including local and global variables along with parameter passing. Note that when you step through this example, codelens bolds line 1 and line 5 as the functions are defined. The body of square is not executed until it is called from the `sum_of_squares` function for the first time on line 6. Also notice that when `square` is called there are two groups of local variables, one for `square` and one for `sum_of_squares`. As you step through you will notice that `x`, and `y` are local variables in both functions and may even have different values. This illustrates that even though they are named the same, they are in fact, very different.

Now we will look at another example that uses two functions. This example illustrates an important computer science problem solving technique called **generalization**. Assume we want to write a function to draw a square. The generalization step is to realize that a square is just a special kind of rectangle.

To draw a rectangle we need to be able to call a function with different arguments for width and height. Unlike the case of the square, we cannot repeat the same thing 4 times, because the four sides are not equal. However, it is the case that drawing the bottom and right sides are the same sequence as drawing the top and left sides. So we eventually come up with this rather nice code that can draw a rectangle.

```
def drawRectangle(t, w, h):  
    """Get turtle t to draw a rectangle of width w and height h."""  
    for i in range(2):  
        t.forward(w)  
        t.left(90)  
        t.forward(h)  
        t.left(90)
```

The parameter names are deliberately chosen as single letters to ensure they're not misunderstood. In real programs, once you've had more experience, we will insist on better variable names than this. The point is that the program doesn't "understand" that you're drawing a rectangle or that the parameters represent the width and the height. Concepts like rectangle, width, and height are meaningful for humans. They are not concepts that the program or the computer understands.

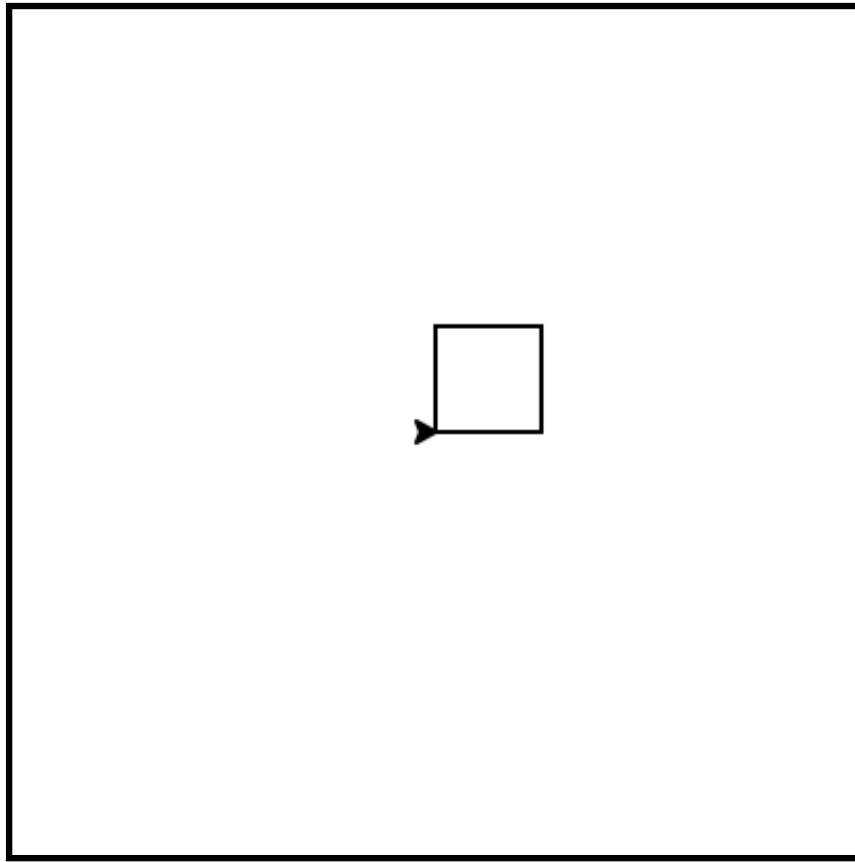
Thinking like a computer scientist involves looking for patterns and relationships. In the code above, we've done that to some extent. We did not just draw four sides. Instead, we spotted that we could draw the rectangle as two halves and used a loop to repeat that pattern twice.

But now we might spot that a square is a special kind of rectangle. A square simply uses the same value for both the height and the width. We already have a function that draws a rectangle, so we can use that to draw our square.

```
def drawSquare(tx, sz):          # a new version of drawSquare  
    drawRectangle(tx, sz, sz)
```

Here is the entire example with the necessary set up code.

```
8         t.forward(h)  
9         t.left(90)  
10  
11 def drawSquare(tx, sz):          # a new version of drawSquare  
12     drawRectangle(tx, sz, sz)  
13  
14 wn = turtle.Screen()            # Set up the window  
15 wn.bgcolor("lightgreen")  
16  
17 tess = turtle.Turtle()          # create tess  
18  
19 drawSquare(tess, 50)  
20  
21 wn.exitonclick()  
22
```


ActiveCode: 3 (ch04_3)[Run](#)

There are some points worth noting here:

- Functions can call other functions.
- Rewriting `drawSquare` like this captures the relationship that we've spotted.
- A caller of this function might say `drawSquare(tess, 50)`. The parameters of this function, `tx` and `sz`, are assigned the values of the `tess` object, and the integer 50 respectively.
- In the body of the function, `tz` and `sz` are just like any other variable.
- When the call is made to `drawRectangle`, the values in variables `tx` and `sz` are fetched first, then the call happens. So as we enter the top of function `drawRectangle`, its variable `t` is assigned the `tess` object, and `w` and `h` in that function are both given the value 50.

So far, it may not be clear why it is worth the trouble to create all of these new functions. Actually, there are a lot of reasons, but this example demonstrates two:

1. Creating a new function gives you an opportunity to name a group of statements. Functions can simplify a program by hiding a complex computation behind a single command. The function (including its name) can capture your mental chunking, or *abstraction*, of the problem.

2. Creating a new function can make a program smaller by eliminating repetitive code.
3. Sometimes you can write functions that allow you to solve a specific problem using a more general solution.

Flow of Execution Summary

When you are working with functions it is really important to know the order in which statements are executed. This is called the **flow of execution** and we've already talked about it a number of times in this chapter.

Execution always begins at the first statement of the program. Statements are executed one at a time, in order, from top to bottom. Function definitions do not alter the flow of execution of the program, but remember that statements inside the function are not executed until the function is called. Function calls are like a detour in the flow of execution. Instead of going to the next statement, the flow jumps to the first line of the called function, executes all the statements there, and then comes back to pick up where it left off.

That sounds simple enough, until you remember that one function can call another. While in the middle of one function, the program might have to execute the statements in another function. But while executing that new function, the program might have to execute yet another function!

Fortunately, Python is adept at keeping track of where it is, so each time a function completes, the program picks up where it left off in the function that called it. When it gets to the end of the program, it terminates.

What's the moral of this sordid tale? When you read a program, don't read from top to bottom. Instead, follow the flow of execution. This means that you will read the `def` statements as you are scanning from top to bottom, but you should skip the body of the function until you reach a point where that function is called.

Check your understanding

func-15: Consider the following Python code. Note that line numbers are included on the left.

```
1  def pow(b, p):
2      y = b ** p
3      return y
4
5  def square(x):
6      a = pow(x, 2)
7      return a
8
9  n = 5
10 result = square(n)
11 print(result)
```

Which of the following best reflects the order in which these lines of code are processed in Python?

- ☐ a) 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11
- ☐ b) 1, 2, 3, 5, 6, 7, 9, 10, 11

- ☐ c) 9, 10, 11, 1, 2, 3, 5, 6, 7
- ☐ d) 9, 10, 5, 6, 7, 1, 2, 3, 11
- ☒ e) 1, 5, 9, 10, 6, 2, 3, 7, 11

Check MeCompare Me

Correct!! Python starts at line 1, notices that it is a function definition and skips over all of the lines in the function definition until it finds a line that it no longer included in the function (line 5). It then notices line 5 is also a function definition and again skips over the function body to line 9. On line 10 it notices it has a function to execute, so it goes back and executes the body of that function. Notice that that function includes another function call. Finally, it will return to line 11 after the function square is complete.

func-16: Consider the following Python code. Note that line numbers are included on the left.

```
1  def pow(b, p):
2      y = b ** p
3      return y
4
5  def square(x):
6      a = pow(x, 2)
7      return a
8
9  n = 5
10 result = square(n)
11 print(result)
```

What does this function print?

- ☒ a) 25
- ☐ b) 5
- ☐ c) 125
- ☐ d) 32

Check MeCompare Me

Correct!! The function square returns the square of its input (via a call to pow)

A Turtle Bar Chart

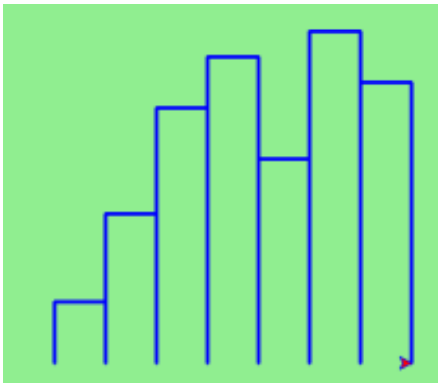
Recall from our discussion of modules that there were a number of things that turtles can do. Here are a couple more tricks (remember that they are all described in the module documentation).

- We can get a turtle to display text on the canvas at the turtle's current position. The method is called `write`. For example, `alex.write("Hello")` would write the string `hello` at the current position.
- One can fill a shape (circle, semicircle, triangle, etc.) with a fill color. It is a two-step process. First you call the method `begin_fill`, for example `alex.begin_fill()`. Then you draw the shape. Finally, you call `end_fill` (`alex.end_fill()`).
- We've previously set the color of our turtle - we can now also set it's fill color, which need not be the same as the turtle and the pen color. To do this, we use a method called `fillcolor`, for example, `alex.fillcolor("red")`.

Ok, so can we get tess to draw a bar chart? Let us start with some data to be charted,

```
xs = [48, 117, 200, 240, 160, 260, 220]
```

Corresponding to each data measurement, we'll draw a simple rectangle of that height, with a fixed width. Here is a simplified version of what we would like to create.



We can quickly see that drawing a bar will be similar to drawing a rectangle or a square. Since we will need to do it a number of times, it makes sense to create a function, `drawBar`, that will need a turtle and the height of the bar. We will assume that the width of the bar will be 40 units. Once we have the function, we can use a basic for loop to process the list of data values.

```
def drawBar(t, height):  
    """ Get turtle t to draw one bar, of height. """  
    t.left(90)          # Point up  
    t.forward(height)   # Draw up the left side  
    t.right(90)  
    t.forward(40)       # width of bar, along the top  
    t.right(90)  
    t.forward(height)   # And down again!  
    t.left(90)          # put the turtle facing the way we found it.  
  
    ...  
for v in xs:            # assume xs and tess are ready  
    drawBar(tess, v)
```

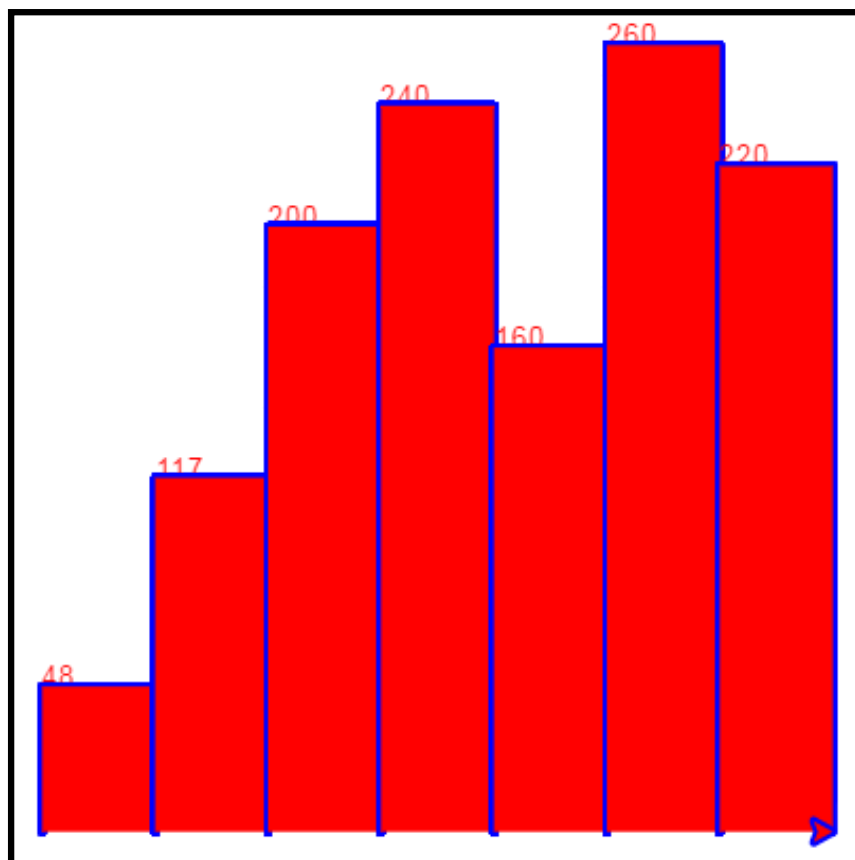
It is a nice start! The important thing here was the mental chunking. To solve the problem we first broke it into smaller pieces. In particular, our chunk is to draw one bar. We then implemented that chunk with a function. Then, for the whole chart, we repeatedly called our function.

Next, at the top of each bar, we'll print the value of the data. We will do this in the body of `drawBar` by adding `t.write(str(height))` as the new fourth line of the body. Note that we had to turn the number into a string. Finally, we'll add the two methods needed to fill each bar.

The one remaining problem is related the fact that our turtle lives in a world where position (0,0) is at the center of the drawing canvas. In this problem, it would help if (0,0) were in the lower left hand corner. To solve this we can use our `setworldcoordinates` method to rescale the window. While we are at it, we should make the window fit the data. The tallest bar will correspond to the maximum data value. The width of the window will need to be proportional to the number of bars (the number of data values) where each has a width of 40. Using this information, we can compute the coordinate system that makes sense for the data set. To make it look nice, we'll add a 10 unit border around the bars.

Here is the complete program. Try it and then change the data to see that it can adapt to the new values. Note also that we have stored the data values in a list and used a few list functions. We will have much more to say about lists in a later chapter.

```
23 tess = turtle.Turtle()           # create tess and set some attributes
24 tess.color("blue")
25 tess.fillcolor("red")
26 tess.pensize(3)
27
28 wn = turtle.Screen()             # Set up the window and its attributes
29 wn.bgcolor("lightgreen")
30 wn.setworldcoordinates(0-border,0-border,40*numbars+border,maxheight+border)
31
32
33 for a in xs:
34     drawBar(tess, a)
35
36 wn.exitonclick()
37
```

ActiveCode: 4 (ch05_barchart)[Run](#)

Note

This workspace is provided for your convenience. You can use this activecode window to try out anything you like.

1	
2	

ActiveCode: 5 (scratch_05_06)

Run

© Copyright 2013 Brad Miller, David Ranum, Created using Runestone Interactive.