6. Functions

6.1. Functions

6.2. Functions that Return Values

6.3. Unit Testing

6.3.1. Choosing Good Unit Tests

6.4. Variables and Parameters are Local

6.5. The Accumulator Pattern

6.5.1. The General Accumulator Pattern

6.5.2. A Variation on the Accumulator Pattern

6.6. Functions can Call Other Functions

6.7. Flow of Execution Summary

6.8. Using a Main Function

6.9. Program Development

6.10. Composition

6.11. A Turtle Bar Chart

6.12. Glossary

6.13. Exercises

6.2. Functions that Return Values

Most functions require arguments, values that control how the function does its job. For example, if you want to find the absolute value of a number, you have to indicate what the number is. Python has a built-in function for computing the absolute value:

1 print(abs(5))

2

​3 print(abs(-5))

4

​

In this example, the arguments to the abs function are 5 and -5.

Some functions take more than one argument. For example the math module contains a function called pow which takes two arguments, the base and the exponent.

1 import math

2 print(math.pow(2, 3))

3

​4 print(math.pow(7, 4))

5

​

Of course, we have already seen that raising a base to an exponent can be done with the \*\* operator.

Another built-in function that takes more than one argument is max.

1 print(max(7, 11))

2 print(max(4, 1, 17, 2, 12))

3 print(max(3 \* 11, 5 \*\* 3, 512 - 9, 1024 \*\* 0))

4

​

max can be sent any number of arguments, separated by commas, and will return the maximum value sent. The arguments can be either simple values or expressions. In the last example, 503 is returned, since it is larger than 33, 125, and 1. Note that max also works on lists of values.

Furthermore, functions like range, int, abs all return values that can be used to build more complex expressions.

So an important difference between these functions and one like drawSquare is that drawSquare was not executed because we wanted it to compute a value — on the contrary, we wrote drawSquare because we wanted it to execute a sequence of steps that caused the turtle to draw a specific shape.

Functions that return values are sometimes called fruitful functions. In many other languages, a chunk that doesn’t return a value is called a procedure, but we will stick here with the Python way of also calling it a function, or if we want to stress it, a non-fruitful function.

Fruitful functions still allow the user to provide information (arguments). However there is now an additional piece of data that is returned from the function.

How do we write our own fruitful function? Let’s start by creating a very simple mathematical function that we will call square. The square function will take one number as a parameter and return the result of squaring that number. Here is the black-box diagram with the Python code following.

1 def square(x):

2 y = x \* x

3 return y

4

​5 toSquare = 10

6 result = square(toSquare)

7 print("The result of", toSquare, "squared is", result)

8

​

Activity: 6.2.4 ActiveCode (ch04\_square)

The return statement is followed by an expression which is evaluated. Its result is returned to the caller as the “fruit” of calling this function. Because the return statement can contain any Python expression we could have avoided creating the temporary variable y and simply used return x\*x. Try modifying the square function above to see that this works just the same. On the other hand, using temporary variables like y in the program above makes debugging easier. These temporary variables are examples of local variables, pursued further in the next section.

Note

The call to a function terminates after the execution of a return statement. This is fairly obvious if the return statement is the last statement in the function, but we will see later where it makes sense to have a return statement even when other statements follow, and the further statements are not executed.

1 def square(x):

2 y = x \* x

3 return y

4

5 toSquare = 10

6 squareResult = square(toSquare)

7 print("The result of", toSquare, "squared is", squareResult)

All Python functions return the value None unless there is an explicit return statement with a value other than None. Consider the following common mistake made by beginning Python programmers. As you step through this example, pay very close attention to the return value in the local variables listing. Then look at what is printed when the function returns.

1 def square(x):

2 y = x \* x

3 print(y) # Bad! should use return instead!

4

5 toSquare = 10

6 squareResult = square(toSquare)

7 print("The result of", toSquare, "squared is", squareResult)

The problem with this function is that even though it prints the value of the square, that value will not be returned to the place where the call was made. Since line 6 uses the return value as the right hand side of an assignment statement, the evaluation of the function will be None. In this case, squareResult will refer to that value after the assignment statement and therefore the result printed in line 7 is incorrect. Typically, functions will return values that can be printed or processed in some other way by the caller.

Check your understanding

func-2-7: What is wrong with the following function definition:

def addEm(x, y, z):

return x + y + z

print('the answer is', x + y + z)

A. You should never use a print statement in a function definition.

B. You should not have any statements in a function after the return statement. Once the function gets to the return statement it will immediately stop executing the function.

C. You must calculate the value of x+y+z before you return it.

D. A function cannot return a number.

func-2-8: What will the following function return?

def addEm(x, y, z):

print(x + y + z)

A. None

B. The value of x + y + z

C. The string 'x + y + z'

6.3. Unit Testing

When we write functions that return values, we intend to use them over and over again. However, we want to be certain that they return the correct result. To be more certain these functions work correctly we write unit tests.

To write a unit test, we must know the correct result when calling the function with a specific input.

1 def square(x):

2 '''raise x to the second power'''

3 return x \* x

4

​5 import test

6 print('testing square function')

7 test.testEqual(square(10), 100)

8

​

testEqual (from the test module) is a function that allows us to perform a unit test. It takes two parameters. The first is a call to the function we want to test (square in this example) with a particular input (10 in this example). The second parameter is the correct result that should be produced (100 in this example). test.testEqual compares what the function returns with the correct result and displays whether the unit test passes or fails.

Extend the program …

On line 8, write another unit test (that should pass) for the square function.

Note

The test module is not a standard Python module. Instead, there are other more powerful and more modern modules. However, the test module offers a simple introduction to testing that is appropriate at this stage in the interactive text. The test module source code is included as an appendix in this interactive text.

6.3.1. Choosing Good Unit Tests

When we write unit tests, we should consider significantly different valid inputs to the function.

For example, the input to the square function could be either a positive or negative value. These two different kinds of inputs give us two equivalence classes of inputs. We then choose an input from each of these classes. It is important to have at least one test for each equivalence class of inputs.

Semantic errors are often caused by improperly handling the boundaries between equivalence classes. The boundary for this problem is zero. It is important to have a test at each boundary.

Extend the program …

Starting on line 9, write two more unit tests (that should pass) so that all input equivalence classes and boundaries are covered.

6.4. Variables and Parameters are Local

An assignment statement in a function creates a local variable for the variable on the left hand side of the assignment operator. It is called local because this variable only exists inside the function and you cannot use it outside. For example, consider again the square function:

1 def square(x):

2 y = x \* x

3 return y

4

5 z = square(10)

6 print(y)

The variable y only exists while the function is being executed — we call this its lifetime. When the execution of the function terminates (returns), the local variables are destroyed.

Formal parameters are also local and act like local variables. For example, the lifetime of x begins when square is called, and its lifetime ends when the function completes its execution.

So it is not possible for a function to set some local variable to a value, complete its execution, and then when it is called again next time, recover the local variable. Each call of the function creates new local variables, and their lifetimes expire when the function returns to the caller.

On the other hand, it is legal for a function to access a global variable. However, this is considered bad form by nearly all programmers and should be avoided. Look at the following, nonsensical variation of the square function.

1 def badsquare(x):

2 y = x \*\* power

3 return y

4

​5 power = 2

6 result = badsquare(10)

7bprint(result)

8

Although the badsquare function works, it is silly and poorly written. We have done it here to illustrate an important rule about how variables are looked up in Python. First, Python looks at the variables that are defined as local variables in the function. We call this the local scope. If the variable name is not found in the local scope, then Python looks at the global variables, or global scope. This is exactly the case illustrated in the code above. power is not found locally in badsquare but it does exist globally. The appropriate way to write this function would be to pass power as a parameter. For practice, you should rewrite the badsquare example to have a second parameter called power.

There is another variation on this theme of local versus global variables. Assignment statements in the local function cannot change variables defined outside the function, without further (discouraged) special syntax. Consider the following codelens example:

1 def powerof(x, p):

2 power = p # Another dumb mistake

3 y = x \*\* power

4 return y

5

6 power = 3

7 result = powerof(10, 2)

8 print(result)

Now step through the code. What do you notice about the values of variable power in the local scope compared to the variable power in the global scope?

The value of power in the local scope was different than the global scope. That is because in this example power was used on the left hand side of the assignment statement power = p. When a variable name is used on the left hand side of an assignment statement Python creates a local variable. When a local variable has the same name as a global variable we say that the local shadows the global. A shadow means that the global variable cannot be accessed by Python because the local variable will be found first. This is another good reason not to use global variables. As you can see, it makes your code confusing and difficult to understand.

To cement all of these ideas even further lets look at one final example. Inside the square function we are going to make an assignment to the parameter x There’s no good reason to do this other than to emphasize the fact that the parameter x is a local variable. If you step through the example in codelens you will see that although x is 0 in the local variables for square, the x in the global scope remains 2. This is confusing to many beginning programmers who think that an assignment to a formal parameter will cause a change to the value of the variable that was used as the actual parameter, especially when the two share the same name. But this example demonstrates that that is clearly not how Python operates.

1 def square(x):

2 y = x \* x

3 x = 0 # assign a new value to the parameter x

4 return y

5

6 x = 2

7 z = square(x)

8 print(z)

Check your understanding

func-3-5: What is a variable’s scope?

A. Its value

B. The range of statements in the code where a variable can be accessed.

C. Its name

func-3-6: What is a local variable?

A. A temporary variable that is only used inside a function

B. The same as a parameter

C. Another name for any variable

func-3-7: Can you use the same name for a local variable as a global variable?

A. Yes, and there is no reason not to.

B. Yes, but it is considered bad form.

C. No, it will cause an error.

6.5. 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. Note that the heading of 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

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.

Check your understanding

func-4-5: Consider the following code:

def square(x):

for counter in range(x):

runningtotal = 0

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

6.5.2. A Variation on the Accumulator Pattern

1 def square(x):

2 '''raise x to the second power'''

3 runningtotal = 0

4 for counter in range(x):

5 runningtotal = runningtotal + x

6

​7 return runningtotal

8

​9 toSquare = 10

10 squareResult = square(toSquare)

11 print("The result of", toSquare, "squared is", squareResult)

12

​

Modify the program …

Change the value of toSquare in line 9 to -10 and run.

We now see that our function has a semantic error. Remember when we first introduced the square function, unit testing and equivalence classes?

Change the value of toSquare in line 9 back to 10 and run.

What would happen if we change runningtotal = runningtotal + x to use multiplication instead of addition? Make this change to the program and look at the output.

It is very important to properly initialize the accumulator variable. Do a web search on additive identity and multiplicative identity. Properly initialize the accumulator variable and run the program.

Now we get an answer other than 0. However, the answer is not the square of of x. It is also important that the loop repeat the proper number of times. How many times do we need to execute line 5 to get the square of x? Change line 4 to repeat the correct number of times. Now the program should produce the correct result.

Change the value of toSquare in line 9 to -10 and run. Now negative inputs also work!

Remember that the boundary between our equivalence classes is 0. Try that value for toSquare also.

6.6. 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)

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.

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 chosen as single letters for conciseness. In real programs, 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.

1import turtle

2

​3 def drawRectangle(t, w, h):

4 """Get turtle t to draw a rectangle of width w and height h."""

5 for i in range(2):

6 t.forward(w)

7 t.left(90)

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

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:

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.

Creating a new function can make a program smaller by eliminating repetitive code.

Sometimes you can write functions that allow you to solve a specific problem using a more general solution.

6.7. 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-6-2: Consider the following Python code. Note that line numbers are included on the left.

def pow(b, p):

y = b \*\* p

return y

def square(x):

a = pow(x, 2)

return a

n = 5

result = square(n)

print(result)

What does this function print?

A. 25

B. 5

C. 125

D. 32

6.8. Using a Main Function

Using functions is a good idea. It helps us to modularize our code by breaking a program into logical parts where each part is responsible for a specific task. For example, in one of our first programs there was a function called drawSquare that was responsible for having some turtle draw a square of some size. The actual turtle and the actual size of the square were defined to be provided as parameters. Here is that original program.

import turtle

def drawSquare(t, sz):

"""Make turtle t draw a square of with side sz."""

for i in range(4):

t.forward(sz)

t.left(90)

wn = turtle.Screen() # Set up the window and its attributes

wn.bgcolor("lightgreen")

alex = turtle.Turtle() # create alex

drawSquare(alex, 50) # Call the function to draw the square

wn.exitonclick()

If you look closely at the structure of this program, you will notice that we first perform all of our necessary import statements, in this case to be able to use the turtle module. Next, we define the function drawSquare. At this point, we could have defined as many functions as were needed. Finally, there are five statements that set up the window, create the turtle, perform the function invocation, and wait for a user click to terminate the program.

These final five statements perform the main processing that the program will do. Notice that much of the detail has been pushed inside the drawSquare function. However, there are still these five lines of code that are needed to get things done.

In many programming languages (e.g. Java and C++), it is not possible to simply have statements sitting alone like this at the bottom of the program. They are required to be part of a special function that is automatically invoked by the operating system when the program is executed. This special function is called main. Although this is not required by the Python programming language, it is actually a good idea that we can incorporate into the logical structure of our program. In other words, these five lines are logically related to one another in that they provide the main tasks that the program will perform. Since functions are designed to allow us to break up a program into logical pieces, it makes sense to call this piece main.

The following code shows this idea. In line 11 we have defined a new function called main that doesn’t need any parameters. The five lines of main processing are now placed inside this function. Finally, in order to execute that main processing code, we need to invoke the main function (line 20). When you push run, you will see that the program works the same as it did before.

1 import turtle

2

​3 def drawSquare(t, sz):

4 """Make turtle t draw a square of with side sz."""

5

​6 for i in range(4):

7 t.forward(sz)

8 t.left(90)

9

​10

​11 def main(): # Define the main function

12 wn = turtle.Screen() # Set up the window and its attributes

13 wn.bgcolor("lightgreen")

14

​15 alex = turtle.Turtle() # create alex

16 drawSquare(alex, 50) # Call the function to draw the square

17

​18 wn.exitonclick()

19

​20 main() # Invoke the main function

21

Now our program structure is as follows. First, import any modules that will be required. Second, define any functions that will be needed. Third, define a main function that will get the process started. And finally, invoke the main function (which will in turn call the other functions as needed).

Note

In Python there is nothing special about the name main. We could have called this function anything we wanted. We chose main just to be consistent with some of the other languages.

Advanced Topic

Before the Python interpreter executes your program, it defines a few special variables. One of those variables is called \_\_name\_\_ and it is automatically set to the string value "\_\_main\_\_" when the program is being executed by itself in a standalone fashion. On the other hand, if the program is being imported by another program, then the \_\_name\_\_ variable is set to the name of that module. This means that we can know whether the program is being run by itself or whether it is being used by another program and based on that observation, we may or may not choose to execute some of the code that we have written.

For example, assume that we have written a collection of functions to do some simple math. We can include a main function to invoke these math functions. It is much more likely, however, that these functions will be imported by another program for some other purpose. In that case, we would not want to execute our main function.

The code below defines two simple functions and a main.

1 def squareit(n):

2 return n \* n

3

​4 def cubeit(n):

5 return n\*n\*n

6

​

7 def main():

8 anum = int(input("Please enter a number"))

9 print(squareit(anum))

10 print(cubeit(anum))

11

​12 if \_\_name\_\_ == "\_\_main\_\_":

13 main()

14

6.9. Program Development

At this point, you should be able to look at complete functions and tell what they do. Also, if you have been doing the exercises, you have written some small functions. As you write larger functions, you might start to have more difficulty, especially with runtime and semantic errors.

To deal with increasingly complex programs, we are going to suggest a technique called incremental development. The goal of incremental development is to avoid long debugging sessions by adding and testing only a small amount of code at a time.

As an example, suppose you want to find the distance between two points, given by the coordinates (x1, y1) and (x2, y2). By the Pythagorean theorem, the distance is:

Distance formula

The first step is to consider what a distance function should look like in Python. In other words, what are the inputs (parameters) and what is the output (return value)?

In this case, the two points are the inputs, which we can represent using four parameters. The return value is the distance, which is a floating-point value.

Already we can write an outline of the function that captures our thinking so far.

def distance(x1, y1, x2, y2):

return 0.0

Obviously, this version of the function doesn’t compute distances; it always returns zero. But it is syntactically correct, and it will run, which means that we can test it before we make it more complicated.

We import the test module to enable us to write a unit test for the function.

1 import test

2 def distance(x1, y1, x2, y2):

3 return 0.0

4

​5 test.testEqual(distance(1, 2, 1, 2), 0)

6

The testEqual function from the test module calls the distance function with sample inputs: (1,2, 1,2). The first 1,2 are the coordinates of the first point and the second 1,2 are the coordinates of the second point. What is the distance between these two points? Zero. testEqual compares what is returned by the distance function and the 0 (the correct answer).

Extend the program …

On line 6, write another unit test. Use (1,2, 4,6) as the parameters to the distance function. How far apart are these two points? Use that value (instead of 0) as the correct answer for this unit test.

On line 7, write another unit test. Use (0,0, 1,1) as the parameters to the distance function. How far apart are these two points? Use that value as the correct answer for this unit test.

The first test passes but the others fail since the distance function does not yet contain all the necessary steps.

When testing a function, it is essential to know the right answer.

For the second test the horizontal distance equals 3 and the vertical distance equals 4; that way, the result is 5 (the hypotenuse of a 3-4-5 triangle). For the third test, we have a 1-1-sqrt(2) triangle.

At this point we have confirmed that the function is syntactically correct, and we can start adding lines of code. After each incremental change, we test the function again. If an error occurs at any point, we know where it must be — in the last line we added.

A logical first step in the computation is to find the differences x2- x1 and y2- y1. We will store those values in temporary variables named dx and dy.

def distance(x1, y1, x2, y2):

dx = x2 - x1

dy = y2 - y1

return 0.0

Next we compute the sum of squares of dx and dy.

def distance(x1, y1, x2, y2):

dx = x2 - x1

dy = y2 - y1

dsquared = dx\*\*2 + dy\*\*2

return 0.0

Again, we could run the program at this stage and check the value of dsquared (which should be 25).

Finally, using the fractional exponent 0.5 to find the square root, we compute and return the result.

1 import test

2 def distance(x1, y1, x2, y2):

3 dx = x2 - x1

4 dy = y2 - y1

5 dsquared = dx\*\*2 + dy\*\*2

6 result = dsquared\*\*0.5

7 return result

8

​9 test.testEqual(distance(1,2, 1,2), 0)

10 test.testEqual(distance(1,2, 4,6), 5)

11 test.testEqual(distance(0,0, 1,1), 1.41)

12

Fix the error …

Two of the tests pass but the last one fails. Is there still an error in the function?

Frequently we discover errors in the functions that we are writing. However, it is possible that there is an error in a test. Here the error is in the precision of the correct answer.

The third test fails because by default testEqual checks 5 digits to the right of the decimal point.

Change 1.41 to 1.41421 and run. The test will pass.

There are circumstances where 2 digits to the right of the decimal point is sufficiently precise.

Copy line 11 on to line 12. On line 12, change 1.41421 to 1.41. Run. The test fails.

Type , 2 after 1.41. (The 2 represents the precision of the test – how many digits to the right of the decimal that must be correct.) Run.

Now all four the tests pass! Wonderful! However, you may still need to perform additional tests.

When you start out, you might add only a line or two of code at a time. As you gain more experience, you might find yourself writing and debugging bigger conceptual chunks. As you improve your programming skills you should find yourself managing bigger and bigger chunks: this is very similar to the way we learned to read letters, syllables, words, phrases, sentences, paragraphs, etc., or the way we learn to chunk music — from indvidual notes to chords, bars, phrases, and so on.

The key aspects of the process are:

Make sure you know what you are trying to accomplish. Then you can write appropriate unit tests.

Start with a working skeleton program and make small incremental changes. At any point, if there is an error, you will know exactly where it is.

Use temporary variables to hold intermediate values so that you can easily inspect and check them.

Once the program is working, you might want to consolidate multiple statements into compound expressions, but only do this if it does not make the program more difficult to read.

6.10. Composition

As we have already seen, you can call one function from within another. This ability to build functions by using other functions is called composition.

As an example, we’ll write a function that takes two points, the center of the circle and a point on the perimeter, and computes the area of the circle.

Assume that the center point is stored in the variables xc and yc, and the perimeter point is in xp and yp. The first step is to find the radius of the circle, which is the distance between the two points. Fortunately, we’ve just written a function, distance, that does just that, so now all we have to do is use it:

radius = distance(xc, yc, xp, yp)

The second step is to find the area of a circle with that radius and return it. Again we will use one of our earlier functions:

result = area(radius)

return result

Wrapping that up in a function, we get:

1 def distance(x1, y1, x2, y2):

2 dx = x2 - x1

3 dy = y2 - y1

4 dsquared = dx\*\*2 + dy\*\*2

5 result = dsquared\*\*0.5

6 return result

7

​8 def area(radius):

9 b = 3.14159 \* radius\*\*2

10 return b

11

​12 def area2(xc, yc, xp, yp):

13 radius = distance(xc, yc, xp, yp)

14 result = area(radius)

15 return result

16

​17 print(area2(0,0,1,1))

18

​

We called this function area2 to distinguish it from the area function defined earlier. There can only be one function with a given name within a module.

Note that we could have written the composition without storing the intermediate results.

def area2(xc, yc, xp, yp):

return area(distance(xc, yc, xp, yp))

6.11. 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.

../\_images/tess\_bar\_1.png

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 to 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.

1 import turtle

2

​3 def drawBar(t, height):

4 """ Get turtle t to draw one bar, of height. """

5 t.begin\_fill() # start filling this shape

6 t.left(90)

7 t.forward(height)

8 t.write(str(height))

9 t.right(90)

10 t.forward(40)

11 t.right(90)

12 t.forward(height)

13 t.left(90)

14 t.end\_fill() # stop filling this shape

15

​16

​17

​18 xs = [48, 117, 200, 240, 160, 260, 220] # here is the data

19 maxheight = max(xs)

20 numbars = len(xs)

21 border = 10

22

​23 wn = turtle.Screen() # Set up the window and its attributes

24 wn.setworldcoordinates(0-border, 0-border, 40\*numbars+border, maxheight+border)

This code is quite concise, but each height label is partly covered by the top segment of its bar. Can you modify the drawBar code, moving the label up slightly but not changing the bar? Hint: The label cannot be drawn during the polygon fill sequence.

6.12. Glossary

chatterbox function

A function which interacts with the user (using input or print) when it should not. Silent functions that just convert their input arguments into their output results are usually the most useful ones.

composition (of functions)

Calling one function from within the body of another, or using the return value of one function as an argument to the call of another.

dead code

Part of a program that can never be executed, often because it appears after a return statement.

fruitful function

A function that yields a return value instead of None.

incremental development

A program development plan intended to simplify debugging by adding and testing only a small amount of code at a time.

None

A special Python value. One use in Python is that it is returned by functions that do not execute a return statement with a return argument.

return value

The value provided as the result of a function call.

scaffolding

Code that is used during program development to assist with development and debugging. The unit test code that we added in this chapter are examples of scaffolding.

temporary variable

A variable used to store an intermediate value in a complex calculation.

6.13. Exercises

Write a fruitful function sumTo(n) that returns the sum of all integer numbers up to and including n. So sumTo(10) would be 1+2+3...+10 which would return the value 55. Use the equation (n \* (n + 1)) / 2.

Write a function areaOfCircle(r) which returns the area of a circle of radius r. Make sure you use the math module in your solution.

Write a non-fruitful function to draw a five pointed star, where the length of each side is 100 units.

Extend your program above. Draw five stars, but between each, pick up the pen, move forward by 350 units, turn right by 144, put the pen down, and draw the next star. You’ll get something like this (note that you will need to move to the left before drawing your first star in order to fit everything in the window):

What would it look like if you didn’t pick up the pen?

Extend the star function to draw an n pointed star. (Hint: n must be an odd number greater or equal to 3).

Write a function called drawSprite that will draw a sprite. The function will need parameters for the turtle, the number of legs, and the length of the legs. Invoke the function to create a sprite with 15 legs of length 120.

Rewrite the function sumTo(n) that returns the sum of all integer numbers up to and including n. This time use the accumulator pattern.

Write a function called mySqrt that will approximate the square root of a number, call it n, by using Newton’s algorithm. Newton’s approach is an iterative guessing algorithm where the initial guess is n/2 and each subsequent guess is computed using the formula: newguess = (1/2) \* (oldguess + (n/oldguess)).

Write a function called myPi that will return an approximation of PI (3.14159…). Use the Leibniz approximation.

Write a function called myPi that will return an approximation of PI (3.14159…). Use the Madhava approximation.