Computer Science 112
Art & Science of Computer Graphics
The College of Saint Rose
Fall 2015

Topic Notes: Functions

As programmers, we often find ourselves writing the same or similar code over and over. We learned that we can place code inside of loop constructs to have the same code executed multiple times. There are other situations where we wish to execute some of the same code repeatedly but not all in the same place in our program's execution.

Consider a simple Python program (that does not use Ambrosia), that prints the lyrics to *Baa Baa Black Sheep*.

On the Wiki: Baabaabad

Notice that we have 4 printouts repeated – these are the refrain of the song. To accomplish this, we either need some copying and pasting or we need to re-type some lines.

A loop wouldn't help us here, as the lines we need to repeat are separated by 4 other lines that do not repeat.

Fortunately, all modern programming languages, including Python, allow us to group sets of statements together into units that can be executed "on demand" by inserting other statements. These constructs go by many names: *functions*, *methods*, *functions*, *subroutines* or *subprograms*.

In Python, the preferred term is function.

Here is our example, using a Python function to group our repeated statements.

On the Wiki: Baabaabetter

Our function for printing the refrain might look a little like a `for` loop to you (at least, that's the closest thing we've seen so far). It is a statement that is followed by a colon, and then a series of lines, a *suite* of statements, that form the *body* of our function.

That first line:

```
def refrain():
```

is the *function header*, and it informs Python that we are about to "define" (`def`) a function. Everything that follows that is indented is treated as the function's definition – the set of statements that will execute when someone *calls* this function. We see two examples of calls to `refrain`:

```
refrain()
```

An important thing to notice here is that when we run our program through Python, the function gets *defined* but not *executed*. If we commented our our calls to `refrain`, we'd see that those lines never got printed, even though the `print` statements are still there in the function body. That suite of statements inside the function definition only execute (resulting in the printing of those four lines) when the function is called.

As our programs grow more complex, we will have more function definitions at the start of our programs. The first things that will actually be executed by Python are those unindented statements that do not start a function definition.

Note that we used a function in this case primarily because it allowed us to reduce the amount of repeated code. This is in itself a worthy goal. If we had misspelled one of the words in the refrain, we can change it in one place and it will be corrected in both printings of the refrain.

However, there is another advantage in readability. By having 2 calls to the `refrain` function, it is more clear what we are doing there. With this in mind, we can consider moving the remaining printouts in the program into a separate function just for clarity.

On the Wiki: Baabaa2functions

Functions with Parameters

Writing functions that do exactly the same thing each time they're called can only get us so far. Here's a program that uses a function to print out a countdown:

On the Wiki: Countdownfrom10

This is much like our previous function examples, except that we have a loop inside of our function, which requires the suite of statements inside the loop to be indented even further than the rest of the function definition.

We also see here that a range can have a negative increment, which allows us to count backwards!

But what if we sometimes want to count down from different starting values? We can augment our function to take a *formal parameter* that tells our function where to start counting:

On the Wiki: Countdown

The idea here is much like the familiar mathematical definition of a function. If we're given this function in math:

$$f(x) = x^2 + 3x - 7$$

we can evaluate $f(4)$ by replacing all of the x instances with 4, getting $4^2+3\cdot 4-7 = 16+12-7 = 21$.

A parameter to a Python function can be thought of in the same way. When we execute

```
countdown(10)
```

that tells Python to use 10 everywhere it sees the name `start` in the function definition for `countdown`. The value passed to our function is called an *actual parameter*. Then later, when we call

```
countdown(20)
```

the name `start` takes on the value of 20.

This idea of *passing parameters* to a function allows for us to write more generic code segments that can then be used in a variety of ways. The value we pass as the actual parameter when we call the function is assigned to the name given as the formal parameter for use inside the function's body.

Another example, printing some information about various numbers:

On the Wiki: Numberinfo

There are a number of things here we have not seen before:

- Python lets us print out a mixture of text and numbers. You will not be responsible for being able to do this, but in case you wish to do so, the `printNumberInfo` function does it.
- We can pass numbers directly to our function, as we do in

  ```
  printNumberInfo(2)
  printNumberInfo(7)
  ```

- We can pass a number stored in a name:

  ```
  printNumberInfo(num)
  ```

- We can pass a number returned from another method call:

  ```
  printNumberInfo(random.randint(1,100))
  ```

Functions that Return Values

The other thing functions can do is to *return values* to their caller. This idea again is analogous to mathematical functions. By writing "$f(4)$" we are expecting that function to compute and give us back some value. Same with many Python functions.

As an example, let's look at a program that includes a function to compute the sum of the first n integers for any number n.

On the Wiki: Sum1ton

Much of our `sum1ToN` function works like the ones we've just seen. The formal parameter `n` gets its value from the actual parameter. The statements within the function are executed when the method is called. But in this case, we also have a *return statement* at the end of the function that transmits information back to the caller. So the assignment statement

```
sumTo10 = sum1ToN(10)
```

results in the name `sumTo10` taking on the value in the return statement inside this call to the `sum1ToN` function.

We can see immediately that we have some similar advantages to our earlier functions that did not return any values. The program becomes shorter, and we avoid potentially having to repeat sections of code when we want to compute such a sum in multiple places in our program.

In this case, there is an additional advantage. Some of you may remember that there is a much easier (computationally speaking) algorithm for computing this sum. Rather than looping through all of the numbers and adding each to a running total, we could apply this formula:

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

This is a more efficient operation, at least for larger numbers. Here, we do one addition, one multiplication, and one division. (Moreover, the division is a division by 2 - something computers are very good at.)

So if we discover this formula and want to change our program to use it, we need only change our method. We don't need to change anything outside of our function definition!

On the Wiki: Sum1tonbetter

So to summarize the basics of functions in Python:

1. A function definition (the `def` line) provides a pattern for calling the function, using formal parameters to transmit information needed by the function.
2. The function's body (the indented statements following the `def` line) provides a suite of statements to be executed, cast in terms of the formal parameters.
3. The function's return value can be used as the result of calling the function.

Using Functions with Ambrosia

We will be defining functions to use with Ambrosia to help to organize and simplify our programs.

Many of you have been defining collections of materials where all have identical material properties other than color. For example, suppose you wanted to create some plaster materials in colors other than those built in to Ambrosia. You would write code such as:

```
orangePlasterMat = Material()
orangePlasterMat.type('plaster')
orangePlasterMat.color((1, .5, 0))
```

and

```
pinkPlasterMat = Material()
pinkPlasterMat.type('plaster')
pinkPlasterMat.color((1, .5, .5))
```

and so on. We can avoid this repeated code by defining a function that construct and returns a plaster `Material` with a given color.

On the Wiki: Plastermat

Or perhaps what you really want is to create plastic materials by specifying only the hue of an HSV color, with full saturation and value. This is another great job for a function.

On the Wiki: Plastichue

Functions with Lists/Tuples

Before we get to our next few function examples, let's recall a few Python constructs we have seen and been using.

First, lists. We have used lists in a couple of contexts. When we define profile polygons for extrudes, prisms, and spindles, we specify a list of points that make up the polygon, *e.g.*:

```
starFront2D = [(0,50), (10,10), (50,0), (10,-10), (0,-50), (-10,-10),
               (-50,0), (-10,10)]
```

and we could then use the `raisePoly` function to append a z=0 coordinate to each;

```
starFront = raisePoly(starFront2D)
```

Then in one of our early examples with `for` loops, we defined a list of numbers:

```
nums = [ 12, 23, 8, 17 ]
```

It turns out there are many more things we can do with lists in Python, and we will see a few of them in later examples.

We have also been using a similar construct, called a *tuple*, in our examples, without using the term until now.

Each coordinate pair in the `starFront2D` list above is a tuple. The triples we have used to represent RGB and HSV colors are also tuples:

```
>>> red
(1, 0, 0)
```

When we created the parenthesized triples to pass to the `plasterMat` function, we were also creating tuples:

```
scene.add(cylinder, plasterMat((.1, .1, .1)))
```

More importantly, the parameter to the `plasterMat` function is a tuple:

```
def plasterMat(c):
    mat = Material()
    mat.type('plaster')
    mat.color(c)
    return mat
```

In this function, we simply passed along the tuple as the color for our material. But in our next example, we see how we can access the values within the tuple and use them individually or to create another tuple.

On the Wiki: Lighterdarkerrgb

We can use this idea to write a function that computes the midpoint between two given points in three-dimensional space:

```
def midpoint(a,b):
    (x1, y1, z1) = a
    (x2, y2, z2) = b
    midX = (x1 + x2)/2
    midY = (y1 + y2)/2
    midZ = (z1 + z2)/2
    return (midX, midY, midZ)
```

The only thing different here is that this function takes two formal parameters, meaning that when we call it, we will need to send it two tuples as actual parameters:

```
mid = midpoint((0,10,25),(-10,30,125))
```

which would return the tuple `(-5,20,75)`.