Defining New Functions in Python

We have identified in Python some of the elements that must appear in any powerful programming language:

1. Numbers and arithmetic operations are primitive built-in data and functions.
2. Nested function application provides a means of *combining* operations.
3. Binding names to values provides a limited means of *abstraction*.

Now we will learn about *function definitions*, a much more powerful abstraction technique by which a name can be bound to compound operation, which can then be referred to as a unit.

We begin by examining how to express the idea of *squaring*. We might say, "To square something, multiply it by itself." This is expressed in Python as

```
>>> def square(x):
        return mul(x, x)
```

which defines a new function that has been given the name `square`. This user-defined function is not built into the interpreter. It represents the compound operation of multiplying something by itself. The `x` in this definition is called a *formal parameter*, which provides a name for the thing to be multiplied. The definition creates this user-defined function and associates it with the name`square`.

Function definitions consist of a `def` statement that indicates a `<name>` and a list of named `<formal parameters>`, then a`return` statement, called the function body, that specifies the `<return expression>` of the function, which is an expression to be evaluated whenever the function is applied.

```
def <name>(<formal parameters>):
            return <return expression>
```

The second line *must* be indented! Convention dictates that we indent with four spaces. The return expression is not evaluated right away; it is stored as part of the newly defined function and evaluated only when the function is eventually applied. (Soon, we will see that the indented region can span multiple lines.)

Having defined `square`, we can apply it with a call expression:

```
>>> square(21)
441
>>> square(add(2, 5))
49
>>> square(square(3))
81
```

We can also use `square` as a building block in defining other functions. For example, we can easily define a function`sum_squares` that, given any two numbers as arguments, returns the sum of their squares:

```
>>> def sum_squares(x, y):
        return add(square(x), square(y))
>>> sum_squares(3, 4)
25
```

User-defined functions are used in exactly the same way as built-in functions. Indeed, one cannot tell from the definition of`sum_squares` whether `square` is built into the interpreter, imported from a module, or defined by the user.

Both `def` statements and assignment statements set the binding of names to values, and any existing bindings are lost. For example, `g` below first refers to a function of no arguments, then a number, and then a different function of two arguments.

```
>>> def g():
        return 1
>>> g()
1
>>> g = 2
>>> g
2
```

```
>>> def g(h, i):
        return h + i
>>> g(1, 2)
3
```

Source : http://inst.eecs.berkeley.edu/~cs61A/book/chapters/functions.html#defining-new-functions