Chapter 1, Python Boss Statements

John M. Morrison

June 21, 2021

Contents

1 Introduction

So far, we have dealt with Python statements that carry out simple commands. All of these statements constitute grammatically complete, imperative sentence. Let's look at some examples. The statement

```
x = 5
```

translates to "`x` gets 5." or "bind the name '`x`' to the integer object 5." The statement

```
print(x)
```

translates to "print the object `x` points to." We refer to these sorts of statements as *worker statements.*

Any program we can now write is just a list of worker statements that execute *in seriatum.* This is a frustratingly limited palette with little creative potential. All we can do are the following.

1. We can make assignments to variables such as `x = 5*6`
2. We can evaluate expressions, such as 5*4 + x*6, where we carry `x` over from the last line. Note how Python fetches `x`'s value from the symbol table.
3. We can end messages to objects to have them do work, as in `"fooment"[3:]`, or `"wowser".capitalize()`.
4. We do can one or more of these in a single statement.

Grammatically, we are going to need more. For Python to do real work, we need some subordinating clauses, as well as complete sentences. Here are some very legit questions.

- For every item in a list or tuple, can we execute a segment of code?
- If some condition is true, can we execute a segment code in response?
- Is there are way to make more complex decisions?
- As long as some condition is true, can we keep executing the same code over and over again?
- If there is a chunk of code we use over and over again, can we assign it a name and call it by that name?
- Can we make a program "jump" from one piece of code to another in an orderly fashion, so statements do not have to execute in seriatum?
- Can we go through a sequence and do the same thing for each element?
- Can we open a file and process its contents?

2 Introducing Boss Statements

A *boss statement* is a statement that owns a chunk of code. Python is organized into segments of code called *blocks*. A block is simply a consecutive set of lines that are indented. So, what we expect to see is something like this.

```
boss statement:
    worker1
    worker2

        .
        .
        .

    lastWorker
```

Python, like every other language, has conventions with punctuation. We have met one already, the comma. Commas separate items in lists and tuples. Quote marks (single or double) delimit string literals. Lists are delimited by `[]` and tuples by `()`.

All boss statements end in a colon in Python. Right after a boss statement there must be a block (sequence of indented lines) of code with at least one line in it. These lines are the "property" of the boss statement. So a proper boss statement with a block has this appearance.

```
boss statement:
    worker1
    worker2

        .
        .
        .

    lastWorker
```

Note If you want a boss statement to do nothing, use the Python keyword `pass`, which means "do nothing." Now we shall introduce our first boss statement.

3 Introducing the Simple `if` Statement

The two fundamental boss statements in Python you will meet in this chapter are `def`, which means "to define" and `if`, which means well... "if." The `def` statement will allow us to store a procedure (a sequence of Python statements) under a name. The *def* statement means “To define.”

The `if` statement will allow us to make decisions. We will begin by introducing the simple `if` statement, and introduce `def` a little later in the chapter.

Create this program `decision.py`

```python
x = input("Enter a number:  ")
x = int(x)   ##input comes in as a string, cast to an int
if x > 0:
    print(f"The number you entered, {x}, is a positive number.")
if x == 0:
    print("You entered zero.")
if x < 0:
    print(f"The number you entered, {x}, is a negative number.")
```

Now we run this in a command window.

```
$ python3 decision.py
Enter a number:  5
The number you entered, 5, is a positive number
$ python3 decision.py
Enter a number:  0
You entered zero.
$ python3 decision.py
Enter a number:  -5
The number you entered, -5, is a negative number
$
```

Note how this program lived up to its name. It made a decision based upon the variable the user typed in.

Look at the grammar of the boss statement `if x > 0:`. This translates into “if `x > 0`,” a grammatically incomplete sentence, which by itself, makes no sense to Python. The same thing applies to humans. We cannot extract meaning from grammatically incomplete sentences. We are left hanging by them. The same is true of Python. Following a boss statement must come one or more worker statements. These complete the sentence.

Consider these lines of code.

```python
if x > 0:
    print("The number you entered", x, "is a positive number")
```

They translate into the complete sentence, “If `x` is positive, print the statement ‘The number you entered, `x`, is a positive number.’ ” Since this is a complete sentence, you can run it in its own separate Python program. Said program would only print something if `x` were positive; otherwise, it would be mute.

What you have seen here is an example of the *simple if* statement. This statement requires a *predicate*, or boolean-valued expression, to do its job. This is natural. If you see the word "if" the first thing that comes to mind is "*if what?*"

So, in a nutshell, the simple `if` statement allows you to jump over a block of code if its predicate is false. We are no longer simply executing worker statements *in seriatum*. We are able to omit or include code based on the truth or falsity of a predicate.

4 `if` with `else`

The simple `if` statement skips code if its predicate is false. The `else` boss statement will enable us to have code execute if an `if` statement's predicate is false.

Here is a simple example showing `else` at work. Create this program, `abby.py`.

```
x = int(input("Enter a number:  "))
if x >= 0:
    print(f"The absolute value of {x} is {x}")
else:
    print(f"The absolute value of {x} is {-x}")
```

Now run the program.

```
$ python3 abby.py
Enter a number:  5
The absolute value of 5 is 5
$ python3 abby.py
Enter a number:  0
The absolute value of 0 is 0
$ python3 abby.py
Enter a number:  -5
The absolute value of -5 is 5
$
```

Notice that the `else` is linked to the `if`. The `else` makes no sense on its own. It is also a boss statement. It ends in a colon. It is not only grammatically incomplete as a sentence, it is nonsensical in the absence of a governing if statement.

Grammar Point! If you are using `if` with `else`, do not put other code between the two statements, or python will hiss. Consider this code fragment.

```
1.    if x > 0:
2.        print x
3.    print("goombah")
4.    else:
5.        print "negative"
```

Python will pitch a fit because the code on line 3 is illegally placed. The `else` must immediately follow the `if` that is belongs to. You cannot have an isolated `else` statement, for this makes no sense.

5 Another Linked Boss Statement: `elif`

You can see that if statements and if-else progressions create a two way fork in the road of execution. What if we want a multiway fork, or multiple selection?

This is the job of `elif`. The `elif` statement, like the `if` statement, will require a predicate to do its job. Multiple selection is achieved easily. It looks like this.

```
if predicate1:
    block1
elif predicate2:
    block2
elif predicate3:
    block3
.
.

elif predicateN:
    blockN
else:
    blockWithDefaultCode
```

The `else` block at the end is optional; it is smart to use it if you want a default case. Default cases tend to prevent errors, so you are strongly encouraged to use them. The `elif` keyword can be read "else if."

Just as with `if-else`, you cannot have any code between the `else`, `elifs`, or the `else`.

The behavior of this progression is simple. If the first Boolean fails, you move on to the successors, As soon as one of the predicates evaluates to `True`, its block of code executes and you drop out of the progression. If none of the `elif`s execute and the `if` fails to execute, the `else`, if present, will execute. To wit: the progression keeps trying until it succeeds. When it succeeds the rest of the progression is ignored.

The presence of the `else` or `elif` implies that there is an `if` statement to which this item is glued. It is an error to have code between an `if` or `elif` statement and any other `elif` or `else` to which it is glued. Python will grind to a halt in such a place and it will hiss an error message.

Let's go for a trip to Dusty's Pub and see how he uses multiple selection. You will notice Dusty is not very PC. Place this code in the program `dusty.py`.

```
age = int(input("Enter your age:  "))
if age < 21:
    print("Scram, punk!")
elif age < 65:
    print("Name yer poison.")
else:
    print("Shall I cash yer soshal, geezer??")
```

Now run this. Notice that the joker entering a negative number gets the boot, just as he deserves.

```
$ python3 dusty.py
Enter your age:  -6
Scram, punk!
$ python3 dusty.py
Enter your age:  17
Scram, punk!
$ python3 dusty.py
Enter your age:  25
Name yer poison.
$ python3 dusty.py
Enter your age:  75
Shall I cash yer soshal, geezer??
$
```

6 Boolean Operations

There are three operations that combine Boolean objects: `and`, `or` and `not`. All three of these are language keywords. Type them into a Python file and watch them turn color.

The `not` operator is a unary prefix operator; it has just one operand which occurs after `not`. Its action on its operand is to its truth–value from `True` to

`False` or `False` to `True`. This definition can be displayed in a *truth-table*, which we see here

P	not P
T	F
F	T

This table is just a verbose way of saying, "not P is true precisely when P is false;" `not`ting just toggles the truth value of a boolean. This operation is called *negation*.

Mathematically, the symbols for boolean operators are written as follows. We begin by discussing them in a mathematical context, and will then look at their applications in Python.

1. $\sim$ for `not`
2. $\vee$ for `or`
3. $\wedge$ for `and`

Make sure you understand both the mathematical and the Python notation for boolean operators. This makes other references on the subject accessible to you. We shall use the mathematical notation in mathematical context and the Python keywords when coding.

Note that there is an order of operations here; the `not` operator binds before `and` and `or`. You can override this order of operations using parentheses.

We next show a truth table for defining `and` and `or`. Since these are binary operators, we need to show all possible combinations for `True` and `False` in the truth table.

P	Q	$P \wedge Q$	$P \vee Q$
T	T	T	T
T	F	F	T
F	T	F	T
F	F	F	F

The expression $P \wedge Q$ evaluates to true precisely when *both* P and Q are true. The expression $P \vee Q$ evaluates to true.

Two predicates are *logically equivalent* if they always both true or untrue together. We use the symbol

$$P \Leftrightarrow Q$$

to say that P and Q are logically equivalent.

We prove logical equivalence by constructing a truth table. Here we shall prove $\sim\sim P \iff P$

P	$\sim P$	$\sim\sim P$
T	F	T
F	T	F

Notice that the exact same truth values live in the P and $\sim\sim P$ columns. We have proven logical equivalence. We shall now prove one of *DeMorgan's laws*,

$$\sim (P \vee Q) \iff (\sim P) \wedge (\sim Q).$$

Now we make the needed truth table. Notice there are two Booleans so we need four lines.

P	Q	$P \wedge Q$	$\sim (P \wedge Q)$	$\sim P$	$\sim Q$	$(\sim P) \vee (\sim Q)$
T	T	T	F	F	F	F
T	F	F	T	F	F	T
F	T	F	T	T	F	T
F	F	F	T	T	T	T

DeMorgan's laws can often simplify the appearance of predicates. The other DeMorgan law is stated and addressed in the exercises. Judicious use of the DeMorgan laws can make predicates simpler, and your code easier to read.

Exercises

1. Prove the second DeMorgan's law
$$\sim (P \wedge Q) \iff (\sim P) \vee (\sim Q).$$
2. We define the *exclusive or* operator $\oplus$ by
$$P \oplus Q \iff (P \vee \ Q) \oplus (Q \vee \ P).$$
Show that
$$P \oplus Q \iff (P \vee Q) \wedge (\sim (P \wedge Q).$$
3. Make an eight-line truth table to prove the distributive laws
$$P \vee (Q \wedge R) \iff (P \vee Q) \wedge (P \vee R)$$
and
$$P \wedge (Q \wedge R) \iff (P \wedge Q) \vee (P \wedge R).$$

6.1 Short Circuiting

Suppose that `P` and `Q` are predicates. If `P` is true and you are evaluating `P or Q`, then the predicate `Q` is ignored. Look at this piece of code. Will it crash and burn because we are accessing something out of bounds in the string `x`? If you can't stand the suspense, run it!

```
x = "string of some small length"
n = 3
if (n < 5) or x[1000] == "cows":
    print ("Moooooo!!!")
```

Now run this. Here is the result

```
i$ python shortCircuit.py
Moooooo!!!
~/book>
```

Short circuiting "protected" the bad call. Since the first predicate `x < 5` evaluated to `True`, the second one was never seen.

Short circuiting can also appear with `and`. Here is a simple example.

```
s = input("Enter a string:  ")
if( len(s) >= 3 and s[2] == "a"):
    print("The second index has the character a")
else:
    print("The second index does not have the character a")
```

Now run it. The "short string" will not produce an exception because of short-circuiting on `and`.

```
$ python ssAnd.py
Enter a string:  q
The second index does not have the character a
$ python ssAnd.py
Enter a string:  maaaaa
The second index has the character a
```

So here is a summary.

- In the evaluation of `P or Q`, if `P` evaluates to `True`, then `Q` is never seen.
- In the evaluation of `P or Q`, if `P` evaluates to `False`, then `Q` is never seen.

You can use this to "guard" a statement that might produce an error from being executed. Make use of this nice tool!

Programming Exercises

1. Write a program that asks the user to input a string. If the string contains fewer than five words, print out, "The string is a short string." Otherwise, print out, "The fifth word of the string is (fifth word)."

6.2 Truthiness

An object is said to be `truthy` if, when it is cast to a boolean, the result is `True`. Otherwise it is said to be `falsy`. Knowing about truthiness makes your code more idiomatic. This brings us to a little exploration. Write down a rule for each of these situations.

Programming Exercises

1. When is an integer truthy? falsy?
2. When is a float truthy? falsy?
3. When is a string truthy? falsy?
4. When is a list truthy? falsy?
5. When is a tuple truthy? falsy?
6. There is a graveyard object called `None`. What is the type of `None`. Is it truthy or falsy?

7 Introducing `def`

As we said before, the keyword `def` means "to define." You can try it interactively as follows.

```
>>> def square(x): return x*x
...
>>> square(10)
100
>>> square(5)
25
>>> square(-3)
9
>>>
```

The code `def square(x):` translates to "to define `square(x)`," You will notice that this is a grammatically incomplete sentence. The worker statement `return x*x` completes it.

Let us show its usage in a program.

```
def square(x):
    return x*x
print(square(10))
print(square(5))
print(square(-3))
```

Now run it.

```
$ python squareEx.py
100
25
9
```

Under the name `square`, we have stored the squaring procedure. The *return value* is the square of the number we *passed* to `square`.

Let us look at what happens in each line

`def square(x):` This means "to define `square(x)`." It says that this procedure will depend on one object, whose name the function will see as `x`.

The purpose of `return x*x` is twofold. First `x*x` is evaluated, based on the value passed to `square`. Then, the result of this evaluation is sent back to the *caller*. The last three lines are all callers of this function. Once a function encounters a `return` statement, its execution is over.

What actually happens in the first two lines is that Python creates a new variable named `square` and has it point at the instructions you placed in the body; to wit, these instruction know how to accept an object and square it. Of course, this only makes sense for numerical objects.

You will notice that, if you remove the three `print` statements, the program will run but no output will go to the screen. Did nothing happen? The answer to this is "no." Python actually created a `square` procedure and stored the code necessary to make it go. Our program just ended before we used, or called, it.

Now we arrive at the third line, `print square(10)`. This is where it gets interesting. Remember, variables know where to find their objects in memory. A copy of the location of 10 in memory is sent to `square` and now the `x` in `square` is pointing at this same object.

Inside this function, it is now known that `x` is pointing at the integer object 10. The next line is `return x*x`. Whatever expression is returned, is first evaluated. When the evaluation is complete (Python evaluates this to 100), the value 100 is sent back to the line that called it, `print square(10)`. The value 100 is substituted for `square(10)`, so we now have the intermediate, unseen step, `print 100`, which Python cheerfully does.

The process repeats for `print(square(5))` and `print(square(-3))`. Once this happens, our program runs out of code and its execution ends. It leaves behind the stuff it put to `stdout`.

The object `square` we created is called a Python `function` object. The name of a function is a variable just like any other. It can be assigned to another value. Remember, variables have no type. Only objects harbor type. Bear witness to this.

```
>>> def square(x): return x*x
...
>>> type(square)
<class 'function'>
>>>
```

We see that the name `square` points at a function object. Function objects remember procedures for you, so *functions are objects, too.*

Remember we said at the outset that the `def` statement is a grammatically incomplete sentence. It is actually a form of assignment. So it's grammatically incomplete, just as $x =$ would be. Python functions are what are called *callable* objects. That means you can put parentheses immediately after them with experessions in a comma-separated list inside, and they will act on those things you pass, provided the right number and types of objects are passed. A function's name is just a variable, which points at the code with the instructions for the function in it.

Let us now go on a brief digression and discuss mathematical functions, so we are all speaking the same language.

8 Mathematical Functions

It will be very useful for us to review the concept of a function in the mathematical sense and to expand this definition to realms not usually thought about in a mathematics class. Let us review the *mathematical* definition of function.

A *function* is an object consisting of three parts.

- A set A called the *domain* of the function
- A set B called the *codomain* of the function
- A rule f tying each element of A to a unique element of B. If $x \in A$, we will write $f(x)$ for the rule f applied to the element x.

To fully define a function we must know all three of these things. When we see this we will write $f : A \to B$. Note that for any given input the same output must be generated each time.

The `len` function we described for sequences is a function of this type. Its domain is the set of all sequences. Its codomain is the non-negative integers. The rule associating a string to an integer is: *count the number of characters in the string.* For example, we see that

```
len("eat ice cream")
```

returns the value `13`. Note that the spaces count as characters. This is indeed a function in the mathematical sense, but it is a safe bet you didn't discuss this example in your math class.

When you hear someone say "the function $f(x) = x^3 + 4x$" in a mathematical context, what is really being said? In math class you adopt a convention that dictates what is *really* being said is that the domain and codomain are the set of real numbers and the rule tying each element x of the domain to an element of the codomain is described by

$$f(x) = x^3 + 4 * x.$$

The variable name x used in our description here is immaterial. What is important is the rule: *cube a number and add it to four times the number.* That rule could just as well be embodied by

$$f(\mathrm{cow}) = \mathrm{cow}^3 + 4 * \mathrm{cow}.$$

The names x and cow are called *arguments* of the function. The choice of an argument name does not affect the action of the function. The argument name is really just a placeholder; it is a convenience that helps us to describe the action of the rule.

9 Why Create Python Functions?

When you use an application such as a web browser, you can begin to imagine how complex the code that goes into such a program might be. Menus, buttons, and other appurtenances all require code to handle their input. Getting things to display in the content pane must be complex. Do programmers write such things in giant monolithic blobs of code? Of course not! Smart programmers *modularize*, which means they break the tasks down they wish their program to accomplish and they write code to handle each specific task.

In this way we use abstraction to our advantage. We produce pieces of code with desired behavior. Then, we test these pieces of code; once they are tested and debugged, they can be deployed. We can then concern ourselves with what these pieces of code *do* and forget about `how` they do it. This underlies the modern philosophies of procedural and object-oriented programming.

In Python, programmers can create new types of objects with behaviors suited to the tasks they wish to accomplish. A blueprint for such a new type is created using a *class*. You will meet classes in Chapter 8, but you are already using objects that classes generate. The basic Python types are all created with classes.

We shall see that a class specifies the data which comprise the state of an object, and it can specify methods, which constitute object behavior.

A recurring theme in computer science is answering the question: *Can we break this complex problem into smaller sub-problems and solve each of these?* We will spend a great deal of time and energy thinking about ways of taking an problem and breaking it into smaller ones. Python functions provide us with an excellent means of modularization. Functions provide a first level of abstraction in Python.

9.1 Are all Python Functions Mathematical Functions?

No. We show a very simple example. Make this program, `noinput.py`.

```
def justDoIt():
    print("Here is a function requiring no input")
    print("To add insult to injury, it has no ")
    print("return statement.  This function just ")
    print("'does stuff' by putting things to stdout.")
justDoIt();
```

Now let's run this program.

```
python noinput.py
Here is a function requiring no input
To add insult to injury, it has no
return statement.  This function just
'does stuff' by putting things to stdout.
```

We created the function `justDoIt()`, which simply prints stuff to `stdout` and then it ends. Notice it has no `return` statement, so, and it is not a mathematical function, owing to its lack of input.

This function simply runs out of code. When that happens, we say the function has a *tacit return*. Functions lacking a return statement actually return the Python *graveyard object* `None`, which you met in an earlier exercise.

The printing it does is called a *side effect* of the function. So, to describe a Python function properly, we must specify its allowable inputs, its output, if any, and all side-effects it has.

The purpose of such a function is to record a procedure under a name, so we can use this name instead of re-coding the procedure every time we want it carried out. This brings us to something important.

In contrast to mathematical functions Python functions can lack input, outputs or both. They can also violate another specification of a mathematical function. We have seen this already. Consider this statement.

```
x = input("Enter your name")
```

The function `input` is actually a built-in Python function. Its input is the prompt string shown the user. Its output *depends upon what the user types*, so it is likely to be different every time it runs. This is about as far away as you can get from a mathematical function. You will often hear programmers refer to functions that are mathematical functions as *pure functions*. Quite a few Python functions are impure in this sense.

The Eleventh Commandment Moses busted the bottom off of his tablets, or you would see emblazoned on the bottom of them, "Thou shalt not maintain duplicate code." Why not? If you make a change in a procedure, you then must hunt down every instance of that procedure and change it, too. That is not good. So, if you find yourself doing something again and again, it's time to create a function! This will make your code shorter, easier for humans to understand, and easier to update and maintain.

You will sometimes find yourself in this situation. When you do you will *refactor* your code by eliminating the duplicate segments and replacing them will appropriate function calls.

Refactoring can make your code simpler and easier to maintain and modify. Be on the lookout for opportunities to do this.

When we discuss a function, we must be aware of three important items.

- We should describe the return value or indicate there is none. A mathematical function describes its rule completely via its return value, and has the same output for any given set of inputs.
- We should describe the correct types for the function's arguments.
- We should describe any side-effects the function has. Mathematical functions have no side-effects.

10 More Examples of Defining Functions

Functions can have as many arguments as you like. Here is a quick example. This function takes two legs for a right triangle and returns the length of the triangle's hypotenuse.

```
>>> def hypot(a,b):
...     return (a*a + b*b)**(.5)
...
>>> hypot(5,12)
13.0
>>> hypot(1,1)
1.4142135623730951
>>>
```

Let's create a function that computes the median of a numerical list. Just to remind you, here is how the rule works.

1. If your list has odd length, the median is the middle element, in order.
2. if your list has an even length, you average the middle two elements.

Lets open an interactive session to experiment. We begin with a list of odd length.

```
>>> x = [6,2,9,8,5]
>>> x
[6, 2, 9, 8, 5]
```

Now sort the list.

```
>>> x.sort()
>>> x
[2, 5, 6, 8, 9]
```

Find the middle index and fetch the element there.

```
>>> n = len(x)
>>> x[n//2]
6
```

Next, try the even case. Make a list and sort it.

```
>>> x = [3,6,8,2,9,4]
>>> x.sort()
>>> x
[2, 3, 4, 6, 8, 9]
>>>
```

We need the two middle elements, which we name `left` and `right`.

```
>>> left = x[n//2]
>>> right = x[n//2 + 1]
>>>
```

Now average them.

```
>>> (left + right)/2
5.0
>>>
```

We now have a pretty good idea as to what to do. First, sort the list. If the list has even length, return the middle element. If it has odd length, obtain the two middle elements and average them.

To determine if an integer `n` is even (divisible by 2), just check to see if `n % 2 == 0`.

So, let's start writing code. We can sort the list, and calculate its length to get started.

```
x.sort()
n = len(x)
```

Now let us handle the case of where the length is odd.

```
x.sort()
n = len(x)
if n % 2 == 1:
    return x[n//2]
```

If the list's length is even, the `return` statement ends the function's execution right there. The rest of the code in the function is only seen if the length is odd. We comment on that.

```
x.sort()
n = len(x)
if n % 2 == 0:
    return x[n//2]
#The even length case is handled.  The length must be odd
#if we make it here.
```

Now calculate `left` and `right` using the interactive session as the boneyard. Going fishing, we see that

```
x.sort()
n = len(x)
if n % 2 == 0:
    return x[n//2]
#The even length case is handled.  The length must be odd
#if we make it here.
left = x[n//2]
right = x[n//2 + 1]
return (left + right)/2
```

completes the calculation. Now just indent it all an put a function header on it.

```
def median(x):
    x.sort()
    n = len(x)
    if n % 2 == 0:
        return x[n//2]
    #The even length case is handled.  The length must be odd
    #if we make it here.
    left = x[n//2]
    right = x[n//2 + 1]
    return (left + right)/2
```

Now create some numerical lists and test to see if this works to your liking.

11 Functions with Keyword and Default Arguments

So far we have seen *positional* arguments in Python functions. A typical header looks like this

```
def f(foo, bar, meh):
    return foo + 2*bar + 3*meh
```

The arguments are positional because the order in which they appear is important. Notice that, in this example, `f(1,2,3)` is different from `f(3,2,1)`. If you try to pass fewer than three arguments to this function, Python will hiss, and your program will crash on the spot.

Python functions can also have *keyword arguments*. These *must* appear after any positional arguments. Here we show an example. Let us modify the function we just created to support this.

```
def f(foo, bar = 0, meh = 0):
    return foo + 2*bar + 3*meh
```

You can call this function in any of these ways

```
x = 1
y = 2
z = 3
print("f({x}) = {f(x}")
    print("f({x}, {y}) = {f(x,y)}")
    print("f({x}, {y}, {z}) = {f(x, y, z)}")
    print("f({x}, meh = 6) = {f(x, meh = 6)}")
```

Let us look at the reason each works as it does. Notice, firstly that there is one positional argument, so you must pass at least one argument to `f` or Python will send you a nastygram.

Place the function and the code above into file `keyword.py` and run it. This will be the result.

```
$ python3 keyword.py
f(1) = 1
f(1, 2) = 5
f(1, 2, 3) = 14
f(1, meh = 6) = 19
```

You get the result `f(1) = 1` because Python uses the default arguments `bar = 0` and `meh = 0` in evaluating the expression to be returned. Hence, `foo + 2*bar + 3*meh = 1 + 2*0 + 2*0 = 1`.

You get the result `f(1, 2) = 5` because Python uses the default argument `meh = 0` and `bar = 2` in evaluating the returned expression. Hence, `foo + 2*2 + 3*meh = 1 + 2*2 + 2*0 = 5`.

In the case of `f(1, 2, 3) = 14`, all three arguments are treated as positional arguments and no defaults are used.

But we can also force the use of one default argument as a passed argument. We get `f(1, meh = 6) = 19` because Python uses `foo = 1`, `bar = 0` and `meh = 6` to evaluate the returned expression.

12 Arguments, Mutability, Call by Value and Pink Houses

Once again, we will see that mutability has its dangers. Consider the following interactive session. Begin by defining a list.

```
>>> x = [1,2,3,4]
```

Next we define a function called smitch as follows. Note the use of simultaneous assignment.

```
>>> def smitch(a,b,x):
...
x[a],x[b] = x[b],x[a]
...
```

We now call the function. The objects at indices 0 and 1 are switched. Do you see any black clouds yet??

```
>>> smitch(0,1,x)
>>> x
[2, 1, 3, 4]
```

No? Then let's do this.

```
>>> def swatch(x):
...
x = [0,0,0,0]
>>> x
[2, 1, 3, 4]
>>> swatch(x)
>>> x
[2, 1, 3, 4]
```

A seeming inconsistency looms here. What happened? Here is what we learned about functions. Each argument is evaluated, the result is placed on the heap, and a copy of the result's memory address is sent on to the function.

```
x = [1, 2, 3, 4]
```

get modified? Why did `swatch` not set the list to `[0, 0, 0, 0]`?

The key to understanding is to ask: "What is a variable storing, again?" A variable stores the location of an object. So the call `smitch(x)` gets a copy of the location of `x`. This would be akin to your going to a painter, making a copy of your address and saying, "Paint the house at this address pink." You will come home to a pink house. If you pass a variable pointing at a mutable object to a function and do something that changes that object's state, the object will be changed. Such a thing cannot happen to an immutable object passed a function. You should infer that immutability provides a measure of safety. Mutability provides convenience and an economy of memory.

Now let us look at `swatch`. Here we give our painter a copy of the address and tell him, "Change the copy of the address of this house to (some other address)". Surprise. Your house's address does not change!

What happens in swatch is that a copy of the heap address of the object x got changed. That will not change x. If you remember that variables store the location of their objects, you can rest assured: all arguments to functions are copied when they are passed. We repeat that Python is a *pass by value* language.

Programming Exercises

1. Consider the following three functions.

   ```
   def quack(x, y):
       x.append(y)
   def moo(x,y):
       x = x + [y]
   def baa(x,y):
       return x + [y]
   ```

   If `x` is a list and `y` is an integer, what are the actions of these three functions. What if `x` is a tuple? Can you explain what is happening? Does casting `[y]` to a tuple change things? In each case, tell what happens to `x`.