```
x = 20
y = 30
formatStr = '{0} + {1} = {2}; {0} * {1} = {3}.'
equations = formatStr.format(x, y, x+y, x*y)
print(equations)
```

Try the program.

EXERCISE 1.10.4.1. * Write a version of Exercise 1.10.3.1, `add3f.py`, that uses the string format method to construct the final string.

EXERCISE 1.10.4.2. * Write a version of Exercise 1.10.3.2, `quotientformat.py`, that uses the string format method to construct the final string.

1.11. Defining Functions of your Own

1.11.1. Syntax Template Typography. When new Python syntax is introduced, the usual approach will be to give both specific examples and general templates. In general templates for Python syntax the typeface indicates the the category of each part:

Typeface	Meaning
`Typewriter font`	Text to be written verbatim
Emphasized	A place where you can use an arbitrary identifier. The emphasized text attempts to be descriptive of the meaning of the identifier in the current context.
Normal text	A description of what goes in that position, without giving explicit syntax

We will use these conventions shortly in the discussion of function syntax, and will continue to use the conventions throughout the tutorial.

1.11.2. A First Function Definition. If you know it is the birthday of a friend, Emily, you might tell those gathered with you to sing "Happy Birthday to Emily".

We can make Python display the song. *Read*, and run if you like, the example program `birthday1.py`:

```
print("Happy Birthday to you!")
print("Happy Birthday to you!")
print("Happy Birthday, dear Emily.")
print("Happy Birthday to you!")
```

You would probably not repeat the whole song to let others know what to sing. You would give a request to sing via a descriptive name like "Happy Birthday to Emily".

In Python we can also give a name like `happyBirthdayEmily`, and associate the name with whole song by using a *function definition*. We use the Python `def` keyword, short for *define*.

Read for now:

```
def happyBirthdayEmily():
    print("Happy Birthday to you!")
    print("Happy Birthday to you!")
    print("Happy Birthday, dear Emily.")
    print("Happy Birthday to you!")
```

There are several parts of the syntax for a function definition to notice:

The *heading* contains `def`, the name of the function, parentheses, and finally a colon.

```
def function_name():
```

The remaining lines form the function *body* and are indented by a consistent amount. (The exact amount is not important to the interpreter, though 2 or 4 spaces are common conventions.)

The whole definition does just that: *defines* the meaning of the name `happyBirthdayEmily`, but it does not do anything else yet – for example, the definition itself does not make anything be printed yet. This is our first example of altering the order of execution of statements from the normal sequential order. This is important: the statements in the function *definition* are *not* executed as Python first passes over the lines.

The code above is in example file `birthday2.py`. Load it in Idle and execute it from there. *Nothing* should happen visibly. This is just like defining a variable: Python just remembers the function definition

for future reference. After Idle finished executing a program, however, its version of the Shell remembers function definitions from the program.

In the Idle *Shell* (not the editor), enter

```
happyBirthdayEmily
```

The result probably surprises you! When you give the Shell an identifier, it tells you its *value*. Above, without parentheses, it identifies the function code as the value (and gives a location in memory of the code). Now try the name in the Idle Shell with *parentheses* added:

```
happyBirthdayEmily()
```

The parentheses tell Python to *execute* the named function rather than just *refer* to the function. Python goes back and looks up the definition, and only then, executes the code inside the function definition. The term for this action is a *function call* or function *invocation*. Note, in the function *call* there is no `def`, but there is the function name followed by parentheses.

```
function_name()
```

In many cases we will use a feature of program execution in Idle: that after program execution is completed, the Idle Shell still remembers functions defined in the program. This is not true if you run a program by selecting it directly in the operating system. *The general assumption in this Tutorial will be that programs are run in Idle and the Idle Shell is the Shell referred to.* It will be explicitly stated when you should run a program directly from the operating system. (With most of the examples in the tutorial, running from the operating system is OK – the execution method will not actually matter.)

Look at the example program `birthday3.py`. See it just adds two more lines, *not* indented. Can you guess what it does? Try it:

```
def happyBirthdayEmily():                #1
    print("Happy Birthday to you!")      #2
    print("Happy Birthday to you!")      #3
    print("Happy Birthday, dear Emily.") #4
    print("Happy Birthday to you!")      #5

happyBirthdayEmily()                     #6
happyBirthdayEmily()                     #7
```

The *execution* sequence is different from the *textual* sequence:

(1) Lines 1-5: Python starts from the top, reading and remembering the definition. The definition ends where the indentation ends. (The code also shows a blank line there, but that is only for humans, to emphasize the end of the definition.)
(2) Line 6: this is not indented inside any definition, so the interpreter executes it directly, calling `happyBirthdayEmily()` while remembering where to return.
(3) Lines 1-5: The code of the function is executed for the first time, printing out the song.
(4) End of line 6: Back from the function call. continue on.
(5) Line 7: the function is called again while this location is remembered.
(6) Lines 1-5: The function is executed again, printing out the song again.
(7) End of line 7: Back from the function call, but at this point there is nothing more in the program, and execution stops.

Functions alter execution order in several ways: by statements not being executed as the definition is first read, and then when the function is called during execution, jumping to the function code, and back at the the end of the function execution.

If it also happens to be Andre's birthday, we might define a function `happyBirthdayAndre`, too. Think how to do that before going on ....

1.11.3. Multiple Function Definitions. Here is example program `birthday4.py` where we add a function `happyBirthdayAndre`, and call them both. Guess what happens, and then try it:

```
def happyBirthdayEmily():  # same old function
    print("Happy Birthday to you!")
    print("Happy Birthday to you!")
```

```
    print("Happy Birthday, dear Emily.")
    print("Happy Birthday to you!")

def happyBirthdayAndre():
    print("Happy Birthday to you!")
    print("Happy Birthday to you!")
    print("Happy Birthday, dear Andre.")
    print("Happy Birthday to you!")

happyBirthdayEmily()
happyBirthdayAndre()
```

Again, everything is definitions except the last two lines. They are the only lines executed directly. The calls to the functions *happen* to be in the same order as their definitions, but that is arbitrary. If the last two lines were swapped, the order of operations would change. Do swap the last two lines so they appear as below, and see what happens when you execute the program:

```
happyBirthdayAndre()
happyBirthdayEmily()
```

Functions that you write can also call other functions you write. It is a good convention to have the main action of a program be in a function for easy reference. The example program `birthday5.py` has the two Happy Birthday calls inside a final function, `main`. Do you see that this version accomplishes the same thing as the last version? Run it.

```
def happyBirthdayEmily():                  #1
    print("Happy Birthday to you!")        #2
    print("Happy Birthday to you!")        #3
    print("Happy Birthday, dear Emily.")   #4
    print("Happy Birthday to you!")        #5

def happyBirthdayAndre():                  #6
    print("Happy Birthday to you!")        #7
    print("Happy Birthday to you!")        #8
    print("Happy Birthday, dear Andre.")   #9
    print("Happy Birthday to you!")       #10

def main():                               #11
    happyBirthdayAndre()                  #12
    happyBirthdayEmily()                  #13

main()                                    #14
```

If we want the program to do anything automatically when it is runs, we need one line outside of definitions! The final line is the only one directly executed, and it calls the code in `main`, which in turn calls the code in the other two functions.

Detailed order of execution:

(1) Lines 1-13: Definitions are read and remembered
(2) Line 14: The only line outside definitions, is executed directly. This location is remembered as main is executed.
(3) Line 11: Start on main
(4) Line 12. This location is remembered as execution jumps to happyBirthdayAndre
(5) Lines 6-10 are executed and Andre is sung to.
(6) Return to the end of Line 12: Back from happyBirthdayAndre function call
(7) Line 13: Now happyBirthdayEmily is called as this location is remembered.
(8) Lines 1-5: Sing to Emily
(9) Return to the end of line 13: Back from happyBirthdayEmily function call, done with main
(10) Return to the end of line 14: Back from main; at the end of the program

There is one practical difference from the previous version. After execution, if we want to give another round of Happy Birthday to *both* persons, we only need to enter one further call in the *Shell* to:

```
main()
```

As a simple example emphasizing the significance of a line being indented, guess what the the example file `order.py` does, and run it to check:

```
def f():
    print('In function f')
    print('When does this print?')
f()
```

Modify the file so the second print function is *out*dented like below. What should happen now? Try it:

```
def f():
    print('In function f')
print('When does this print?')
f()
```

The lines indented inside the function definition are *remembered* first, and only executed when the function f is invoked at the end. The lines outside any function definition (not indented) are executed in order of appearance.

EXERCISE 1.11.3.1. * Write a program, `poem.py`, that defines a function that prints a *short* poem or song verse. Give a meaningful name to the function. Have the program end by calling the function three times, so the poem or verse is repeated three times.

1.11.4. Function Parameters. As a young child, you probably heard Happy Birthday sung to a couple of people, and then you could sing to a new person, say Maria, without needing to hear the whole special version with Maria's name in it word for word. You had the power of *abstraction*. With examples like the versions for Emily and Andre, you could figure out what change to make it so the song could be sung to Maria!

Unfortunately, Python is not that smart. It needs explicit rules. If you needed to explain *explicitly* to someone how Happy Birthday worked in general, rather than just by example, you might say something like this:

First you have to be *given* a person's name. Then you sing the song with the person's name inserted at the end of the third line.

Python works something like that, but with its own syntax. The term "person's name" serves as a stand-in for the actual data that will be used, "Emily", "Andre", or "Maria". This is just like the association with a variable name in Python. "person's name" is not a legal Python identifier, so we will use just `person` as this stand-in.

The function definition indicates that the variable name `person` will be used inside the function by inserting it between the parentheses of the definition. Then in the body of the definition of the function, person is used in place of the real data for any specific person's name. Read and then run example program `birthday6.py`:

```
def happyBirthday(person):                          #1
    print("Happy Birthday to you!")                 #2
    print("Happy Birthday to you!")                 #3
    print("Happy Birthday, dear " + person + ".") #4
    print("Happy Birthday to you!")                 #5

happyBirthday('Emily')                              #6
happyBirthday('Andre')                              #7
```

In the definition heading for `happyBirthday`, `person` is referred to as a *parameter*, or a *formal parameter*. This variable name is a placeholder for the real name of the person being sung to.

The last two lines of the program, again, are the only ones outside of definitions, so they are the only ones executed directly. There is now an actual name between the parentheses in the function calls. The value between the parentheses here in the function call is referred to as an *argument* or *actual parameter* of the function call. The argument supplies the actual data to be used in the function execution. When the call is

made, Python does this by associating the formal parameter name `person` with the actual parameter data, as in an assignment statement. In the first call, this actual data is `'Emily'`. We say the actual parameter value is *passed* to the function.

The execution in greater detail:

(1) Lines 1-5: Definition remembered
(2) Line 6: Call to `happyBirthday`, with actual parameter `'Emily'`.
(3) Line 1: `'Emily'` is passed to the function, so `person = 'Emily'`
(4) Lines 2-5: The song is printed, with `'Emily'` used as the value of `person` in line 4: printing `'Happy birthday, dear Emily.'`
(5) End of line 6: Return from the function call and continue
(6) Line 7: Call to `happyBirthday`, this time with actual parameter `'Andre'`
(7) Line 1: `'Andre'` is passed to the function, so `person = 'Andre'`
(8) Lines 2-5: The song is printed, with `'Andre'` used as the value of `person` in line 4: printing `'Happy birthday, dear Andre.'`
(9) End of line 7: Return from the function call, and the program is over.

The beauty of this system is that the same function definition can be used for a call with a different actual parameter variable, and then have a different effect. The value of the variable person is used in the third line of `happyBirthday`, to put in whatever actual parameter value was given.

This is the power of *abstraction*. It is one application of the most important principal in programming. Rather than have a number of separately coded parts with only slight variations, see where it is appropriate to combine them using a function whose parameters refer to the parts that are different in different situations. Then the code is written to be simultaneously appropriate for the separate specific situations, with the substitutions of the right parameter values.

You can go back to having a main function again, and everything works. Run `birthday7.py`:

```
def happyBirthday(person):
    print("Happy Birthday to you!")
    print("Happy Birthday to you!")
    print("Happy Birthday, dear " + person + ".")
    print("Happy Birthday to you!")

def main():
    happyBirthday('Emily')
    happyBirthday('Andre')

main()
```

Exercise 1.11.4.1. * Make your own further change to the file and save it as `birthdayMany.py`: Add a function call, so Maria gets a verse, in addition to Emily and Andre. Also print a blank line between verses. (You may either do this by adding a print line to the function definition, or by adding a print line between all calls to the function.)

We can combine function parameters with user input, and have the program be able to print Happy Birthday for anyone. Check out the main method and run `birthday_who.py`:

```
def happyBirthday(person):
    print("Happy Birthday to you!")
    print("Happy Birthday to you!")
    print("Happy Birthday, dear " + person + ".")
    print("Happy Birthday to you!")

def main():
    userName = input("Enter the Birthday person's name: ")
    happyBirthday(userName)

main()
```

This last version illustrates several important ideas:

(1) There are more than one way to get information into a function:
   (a) Have a value passed in through a parameter.
   (b) Prompt the user, and obtain data from the keyboard.
(2) It is a good idea to separate the *internal* processing of data from the *external* input from the user by the use of distinct functions. Here the user interaction is in `main`, and the data is manipulated in `happyBirthday`.
(3) In the first examples of actual parameters, we used literal values. In general an actual parameter can be an expression. The expression is evaluated before it is passed in the function call. One of the simplest expressions is a plain variable name, which is evaluated by replacing it with its associated value. Since it is only the value of the actual parameter that is passed, not any variable name, there is no need to have an actual parameter variable name match a formal parameter name. (Here we have the value of `userName` in `main` becoming the value of `person` in `happyBirthday`.)

1.11.5. Multiple Function Parameters. A function can have more than one parameter in a parameter list separated by commas. Here the example program `addition5.py` uses a function to make it easy to display many sum problems. Read and follow the code, and then run:

```
def sumProblem(x, y):
    sum = x + y
    print('The sum of ', x, ' and ', y, ' is ', sum, '.', sep='')

def main():
    sumProblem(2, 3)
    sumProblem(1234567890123, 535790269358)
    a = int(input("Enter an integer: "))
    b = int(input("Enter another integer: "))
    sumProblem(a, b)

main()
```

The actual parameters in the function call are evaluated left to right, and then these values are associated with the formal parameter names in the function definition, also left to right. For example the function call with actual parameters, `f(actual1, actual2, actual3)`, calling the function f with definition heading
`def f(formal1, formal2, formal3):`
acts approximately as if the first lines executed inside the called function were

```
formal1 = actual1
formal2 = actual2
formal3 = actual3
```

Functions provide extremely important functionality to programs, allowing task to be defined once and performed repeatedly with different data. It is essential to see the difference between the formal parameters used to describe what is done inside the function definition (like x and y in the definition of sumProblem) and the actual parameters (like 2 and 3 or 1234567890123 and 535790269358) which *substitute* for the formal parameters when the function is actually executed. The main method above uses three different sets of actual parameters in the three calls to sumProblem.

Exercise 1.11.5.1. '* Modify the program above and save it as `quotientProb.py`. The new program should have a quotientProblem function, printing as in the Exercise 1.10.3.2. The main method should test the function on several sets of literal values, and also test the function with input from the user.

1.11.6. Returned Function Values. You probably have used mathematical functions in algebra class, but they all had calculated values associated with them. For instance if you defined $f(x) = x^2$, then it follows that $f(3)$ is $3^2 = 9$, and $f(3)+f(4)$ is $3^2+4^2 = 25$. Function calls in expressions get replaced during evaluation by the value of the function.

The corresponding definition and examples in Python would be the following, also in the example program `return1.py`. *Read and run*:

```
def f(x):
    return x*x

print(f(3))
print(f(3) + f(4))
```

The new Python syntax is the *return statement*, with the word `return` followed by an expression. Functions that return values can be used in expressions, just like in math class. When an expression with a function call is evaluated, the function call is effectively replaced temporarily by its returned value. Inside the Python function, the value to be returned is given by the expression in the `return` statement. After the function `f` finishes executing from inside

```
print(f(3))
```

it is as if the statement temporarily became

```
print(9)
```

and similarly when executing

```
print(f(3) + f(4))
```

the interpreter first evaluates f(3) and effectively replaces the call by the returned result, 9, as if the statement temporarily became

```
print(9 + f(4))
```

and then the interpreter evaluates f(4) and effectively replaces the call by the returned result, 16, as if the statement temporarily became

```
print(9 + 16)
```

resulting finally in 25 being calculated and printed.

Python functions can return any type of data, not just numbers, and there can be any number of statements executed before the return statement. Read, follow, and run the example program `return2.py`:

```
def lastFirst(firstName, lastName):            #1
    separator = ', '                           #2
    result = lastName + separator + firstName  #3
    return result                              #4

print(lastFirst('Benjamin', 'Franklin'))       #5
print(lastFirst('Andrew', 'Harrington'))       #6
```

The code above has a new feature, variables `separator` and `result` are given a value in the function, but `separator` and `result` are not among the formal parameters. The assignments work as you would expect here. More on this shortly, in Section 1.11.8 on local scope.

Details of the execution:

(1) Lines 1-4: Remember the definition
(2) Line 5: call the function, remembering where to return
(3) Line 1: pass the parameters: `firstName = 'Benjamin'`; `lastName = 'Franklin'`
(4) Line 2: Assign the variable `separator` the value `', '`
(5) Line 3: Assign the variable `result` the value of
    `lastName + separator + firstName` which is
    `'Franklin' + ', ' + 'Benjamin'`, which evaluates to
    `'Franklin, Benjamin'`
(6) Line 4: Return `'Franklin, Benjamin'`
(7) Line 5 Use the value returned from the function call so the line effectively becomes
    `print('Franklin, Benjamin')`
    so print it.
(8) Line 6: call the function with the new actual parameters, remembering where to return
(9) Line 1: pass the parameters: `firstName = 'Andrew'`; `lastName = 'Harrington'`
(10) Lines 2-4: ... calculate and return `'Harrington, Andrew'`
(11) Line 6: Use the value returned by the function and print `'Harrington, Andrew'`

Compare `return2.py` and `addition5.py`, from the previous section. Both use functions. Both print, but where the printing is done differs. The function `sumProblem` prints directly inside the function and returns nothing explicitly. On the other hand `lastFirst` does not print anything but returns a string. The caller gets to decide what to do with the string, and above it is printed in the main program.

Open `addition5.py` again, and introduce a *common mistake*. Change the last line of the function `main` inserting `print`, so it says

```
print(sumProblem(a, b))
```

Then try running the program. The desired printing is actually done inside the function sumProblem. You introduced a statement to print what `sumProblem` *returns*. Although `sumProblem` returns nothing *explicitly*, Python does make every function return something. If there is nothing explicitly returned, the special value `None` is returned. You should see that in the Shell output. This is a fairly common error. If you see a 'None' is your output where you do not expect it, it is likely that you have printed the return value of a function that did not return anything explicitly!

EXERCISE 1.11.6.1. Create quotientReturn.py by modifying quotientProb.py from Exercise 1.11.5.1 so that the program accomplishes the same thing, but everywhere change the quotientProblem function into one called `quotientString` that merely *returns* the string rather than printing the string directly. Have the `main` function print the result of each call to the quotientString function.

1.11.7. Two Roles: Writer and Consumer of Functions. The remainder of Section1.11 covers finer points about functions that you might skip on a first reading.

We are only doing tiny examples so far to get the basic idea of functions. In much larger programs, functions are useful to manage complexity, splitting things up into logically related, modest sized pieces. Programmers are both writers of functions and consumers of the other functions called inside their functions. It is useful to keep those two roles separate:

The user of an already written function needs to know:

(1) the name of the function
(2) the order and meaning of parameters
(3) what is returned or produced by the function

How this is accomplished is not relevant at this point. For instance, you use the work of the Python development team, calling functions that are built into the language. You need know the three facts about the functions you call. You do not need to know exactly *how* the function accomplishes its purpose.

On the other hand when you *write* a function you need to figure out exactly how to accomplish your goal, name relevant variables, and write your code, which brings us to the next section.

1.11.8. Local Scope. For the logic of writing functions, it is important that the writer of a function knows the names of variables inside the function. On the other hand, if you are only using a function, maybe written by someone unknown to you, you should not care what names are given to values used internally in the implementation of the function you are calling. Python enforces this idea with *local scope* rules: Variable names initialized and used inside one function are *invisible* to other functions. Such variables are called *local* variables. For example, an elaboration of the earlier program `return2.py` might have its `lastFirst` function with its local variable `separator`, but it might also have another function that defines a `separator` variable, maybe with a different value like `'\n'`. They do not conflict. They are independent. This avoids lots of errors!

For example, the following code in the example program `badScope.py` causes an execution error. Read it and run it, and see:

```
def main():
    x = 3
    f()

def f():
    print(x)  #f does not know about the x defined in main

main()
```