# Defining Functions of your Own

## 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** | **Example** |
| --- | --- | --- |
| Typewriter font | Text to be written verbatim | sep='' |
| Emphasized | A place where you can use an arbitrary expression. | integerValue |
| **Bold** | A place where you can use an arbitrary identifier. | **variableName** |
| Normal text | A description of what goes in that position,without giving explicit syntax | A digit, 0-9 |

A more complete example of using this typography with several parts would be a description of an assignment statement:

**variableName** = someExpression

with an arbitrary identifier, the specific symbol =, and an expression.

I try to make the parts that are not verbatim to be descriptive of the expected use.

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

## 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 Pythondef keyword, short for define.

Read for now:

|  |  |
| --- | --- |
| 1  2  3  4  5 | **def** happyBirthdayEmily(): *#program does nothing as written*  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:

Line 1: The heading contains def, the name of the function, parentheses, and finally a colon. A more general syntax is

def **function\_name**():

Lines 2-5: 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.

**Note**

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.

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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | *'''Function definition and invocation.'''*  **def** happyBirthdayEmily():  print("Happy Birthday to you!")  print("Happy Birthday to you!")  print("Happy Birthday, dear Emily.")  print("Happy Birthday to you!")  happyBirthdayEmily()  happyBirthdayEmily() |

The execution sequence is different from the textual sequence:

1. Lines 3-7: 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 9: this is not indented inside any definition, so the interpreter executes it directly, calling happyBirthdayEmily() while remembering where to return.
3. Lines 3-7: The code of the function is executed for the first time, printing out the song.
4. End of line 9: Back from the function call. continue on.
5. Line 10: the function is called again while this location is remembered.
6. Lines 3-7: The function is executed again, printing out the song again.
7. End of line 10: 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 ....

## 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:

*'''Function definitions and invocation.'''*

**def** happyBirthdayEmily():

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. :

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | *'''Function definitions and invocation.'''*  **def** happyBirthdayEmily():  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!")  **def** main():  happyBirthdayEmily()  happyBirthdayAndre()  main() |

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 3-17: Definitions are read and remembered
2. Line 19: The only statement outside definitions, is executed directly. This location is remembered as main is executed.
3. Line 15: Start on main
4. Line 16. This location is remembered as execution jumps to happyBirthdayEmily
5. Lines 3-7 are executed and Emily is sung to.
6. Return to the end of Line 16: Back from happyBirthdayEmily function call
7. Line 17: Now happyBirthdayAndre is called as this location is remembered.
8. Lines 9-13: Sing to Andre
9. Return to the end of line 17: Back from happyBirthdayAndre function call, done with main
10. Return to the end of line 19: 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.

### Poem Function Exercise

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.

## 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 justperson 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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | *'''Function with parameter.'''*  **def** happyBirthday(person):  print("Happy Birthday to you!")  print("Happy Birthday to you!")  print("Happy Birthday, dear " + person + ".")  print("Happy Birthday to you!")  happyBirthday('Emily')  happyBirthday('Andre') |

In the definition heading for happyBirthday, person is referred to as 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 3-7: Definition remembered
2. Line 9: Call to happyBirthday, with actual parameter 'Emily'.
3. Line 3: 'Emily' is passed to the function, so person = 'Emily'.
4. Lines 4-7: The song is printed, with 'Emily' used as the value of person in line 4: printing
5. Happy Birthday, dear Emily.
6. End of line 9 after returning from the function call
7. Line 10: Call to happyBirthday, this time with actual parameter 'Andre'
8. Line 3: 'Andre' is passed to the function, so person = 'Andre'.
9. Lines 4-7: The song is printed, with 'Andre' used as the value of person in line 4: printing
10. Happy Birthday, dear Andre.
11. End of line 10 after returning from the function call, and the program is over.

**Note**

Be sure you completely understand birthday6.py and the sequence of execution! It illustrates extremely important ideas that many people miss the first time!

It is essential to understand the difference between

1. Defining a function (lines 3-7) with the def heading including formal parameter names, where the code is merely instructions to be remembered, not acted on immediately.
2. Calling a function with actual parameter values to be substituted for the formal parameters and have the function code actually run when the instruction containing the call is run. Also note that the function can be called multiple times with different expressions as the actual parameters (line 9 and again in line 10).

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

**Note**

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:

*'''Function with parameter called in main'''*

**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()

In birthday6.py, the function calls in lines 9 and 10 were outside any function definition, so they did actually lead to immediate execution of the function. In birthday7.py the calls to happyBirthday are inside another function definition (main), so they are not actually run until the function main is run (from the last line, outside any function).

See [Birthday Function Exercise](http://anh.cs.luc.edu/python/hands-on/3.1/handsonHtml/functions.html#birthdayfunctionex).

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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | *'''User input supplies function parameter'''*  **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:
   1. Have a value passed in through a parameter (from line 10 to line 3).
   2. Prompt the user, and obtain data from the keyboard (line 11).
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 a variable name used in an actual parameter match a formal parameter name. (Here we have the value of userName in main becoming the value of person inhappyBirthday.)

Now that we have nested function calls, it is worth looking further at tracebacks from execution errors. If I add a line to main in birthday7.py:

happyBirthday(2)

as in example file birthdayBad.py, and then run it, you get something close to:

Traceback (most recent call last):

File “/hands-on/../examples/birthdayBad.py”, line 15, in <module>

main()

File “/hands-on/../examples/birthdayBad.py”, line 13, in main

happyBirthday(2)

File “/hands-on/../examples/birthdayBad.py”, line 6, in happyBirthday

print(“Happy Birthday, dear ” + person + ”.”)

TypeError: Can’t convert ‘int’ object to str implicitly

Your file folder is probably different than /hands-on/examples. The last three lines are most important, giving the line number where the error was detected, the text of the line in question, and a description of what problem was found. Often that is all you need to look at, but this example illustrates that the genesis of the problem may be far away from the line where the error was detected. Going further up the traceback, you find the sequence of function calls that led to the line where the error was detected. You can see that in main I call happyBirthday with the bad parameter, 2.

## Multiple Function Parameters

A function can have more than one parameter in a parameter list separated by commas. Here the example program addition5.py changes example program addition4a.py, using a function to make it easy to display many sum problems. Read and follow the code, and then run:

*'''Display any number of sum problems with a function.*

*Handle keyboard input separately.*

*'''*

**def** sumProblem(x, y):

sum = x + y

sentence = 'The sum of {} and {} is {}.'.format(x, y, sum)

print(sentence)

**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 a function call with actual parameters, f(actual1, actual2, actual3), calling a function f with definition heading:

**def** f(formal1, formal2, formal3):

acts approximately as if the first lines executed inside the called function f were

formal1 = actual1

formal2 = actual2

formal3 = actual3

Functions provide extremely important functionality to programs, allowing tasks 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.

## 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)=x2

then it follows that f(3) is 32, and f(3)+f(4) is 32 + 42

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, taken from example program return1.py. Read and run:

*'''A simple function returning a value, used in an expression'''*

**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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | *'''A function returning a string and using a local variable'''*  **def** lastFirst(firstName, lastName):  separator = ', '  result = lastName + separator + firstName  **return** result  print(lastFirst('Benjamin', 'Franklin'))  print(lastFirst('Andrew', 'Harrington')) |

The code above has a new feature, variables separator and result are given a value inside 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 [Local Scope](http://anh.cs.luc.edu/python/hands-on/3.1/handsonHtml/functions.html#local-scope).

Details of the execution:

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

**Warning**

If you see a ‘None’ is your printed 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!

In general functions should do a single thing. You can easily combine a sequence of functions, and you have more flexibility in the combinations if each does just one unified thing. The function sumProblem in addition5.py does two things: It creates a sentence, and prints it. If that is all you have, you are out of luck if you want to do something different with the sentence string. A better way is to have a function that just creates the sentence, and returns it for whatever further use you want. Printing is one possibility, done in addition6.py:

*'''Display a sum problems with a function returning a string,*

*not printing directly.*

*'''*

**def** sumProblemString(x, y):

sum = x + y

**return** 'The sum of {} and {} is {}.'.format(x, y, sum)

**def** main():

print(sumProblemString(2, 3))

print(sumProblemString(1234567890123, 535790269358))

a = int(input("Enter an integer: "))

b = int(input("Enter another integer: "))

print(sumProblemString(a, b))

main()

.