15.1. Optional Parameters

In the treatment of functions so far, each function definition specifies zero or more formal parameters and each function invocation provides exactly that many values. Sometimes it is convenient to have **optional parameters** that can be specified or omitted. When an optional parameter is omitted from a function invocation, the formal parameter is bound to a **default value**. When the optional parameter is included, then the formal parameter is bound to the value provided. Optional parameters are convenient when a function is almost always used in a simple way, but it’s nice to allow it to be used in a more complex way, with non-default values specified for the optional parameters.

Consider, for example, the int function, which you have used previously. Its first parameter, which is required, specifies the object that you wish to convert to an integer. For example, if you call in on a string, int("100"), the return value will be the integer 100.

That’s the most common way programmers want to convert strings to integers. Sometimes, however, they are working with numbers in some other “base” rather than base 10. For example, in base 8, the rightmost digit is ones, the next digit to the left is 8s, and the one to the left of that is the 64s place (8\*\*2).

The int function provides an optional parameter for the base. When it is not specified, the number is converted to an integer assuming the original number was in base 10. We say that 10 is the default value. So int("100") is the same as invoking int("100", 10). We can override the default of 10 by supplying a different value.

print(int("100"))

print(int("100", 10)) # same thing, 10 is the default value for the base

print(int("100", 8)) # now the base is 8, so the result is 1\*64 = 64

​

**Note**

Tom Lehrer’s New Math

Some math educators believe that elementary school students will get a much deeper understanding of the place-value system, and set a foundation for learning algebra later, if they learn to do arithmetic not only in base-10 but also in base-8 and other bases. This was part of a movement called “The New Math”, though it’s not so new now (I had it when I was in elementary school!) Tom Lehrer made a really funny song about it, and it’s set with visuals in several YouTube renditions now. Try this very nice [lip-synched version](http://www.youtube.com/watch?v=DfCJgC2zezw).

When defining a function, you can specify a default value for a parameter. That parameter then becomes an optional parameter when the function is called. The way to specify a default value is with an assignment statement inside the parameter list. Consider the following code, for example.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Python 3.3   |  |  |  | | --- | --- | --- | |  | 1 | initial = 7 | | 2 | def f(x, y =3, z=initial): | | 3 | print("x, y, z, are: " + str(x) + ", " + str(y) + ", " + str(z)) | | 4 |  | | 5 | f(2) | | 6 | f(2, 5) | | 7 | f(2, 5, 8) | | |  |  | | --- | --- | | Frames | Objects | |
|  | |

Notice the different bindings of x, y, and z on the three invocations of f. The first time, y and z have their default values, 3 and 7. The second time, y gets the value 5 that is passed in, but z still gets the default value of 7. The last time, z gets the value 8 that is passed in.

If you want to provide a non-default value for the third parameter (z), you also need to provide a value for the second item (y). We will see in the next section a mechanism called keyword parameters that lets you specify a value for z without specifying a value for y.

**Note**

This is a second, related but slightly different use of = than we have seen previously. In a stand-alone assignment statement, not part of a function definition, x=3 assigns 3 to the variable x. As part of specifying the parameters in a function definition, x=3 says that 3 is the *default* value for x, used *only when* no value is provided during the function invocation.

There are two tricky things that can confuse you with default values. The first is that the default value is determined at the time that the function is defined, not at the time that it is invoked. So in the example above, if we wanted to invoke the function f with a value of 10 for z, we cannot simply set initial = 10 right before invoking f. See what happens in the code below, where z still gets the value 7 when f is invoked without specifying a value for z.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Python 3.3   |  |  |  | | --- | --- | --- | |  | 1 | initial = 7 | | 2 | def f(x, y =3, z=initial): | | 3 | print("x, y, z, are: " + str(x) + ", " + str(y) + ", " + str(z)) | | 4 |  | | 5 | initial = 10 | | 6 | f(2) | | |  |  | | --- | --- | | Frames | Objects | |
|  | |

The second tricky thing is that if the default value is set to a mutable object, such as a list or a dictionary, that object will be shared in all invocations of the function. This can get very confusing, so I suggest that you never set a default value that is a mutable object. For example, follow the exceution of this one carefully.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Python 3.3   |  |  |  | | --- | --- | --- | |  | 1 | def f(a, L=[]): | | 2 | L.append(a) | | 3 | return L | | 4 |  | | 5 | print(f(1)) | | 6 | print(f(2)) | | 7 | print(f(3)) | | 8 | print(f(4, ["Hello"])) | | 9 | print(f(5, ["Hello"])) | | |  |  | | --- | --- | | Frames | Objects | |
|  | |

When the default value is used, the same list is shared. But on lines 8 and 9 two different copies of the list [“Hello”] are provided, so the 4 that is appended is not present in the list that is printed on line 9.

**Check your understanding**

advfuncs-1-1: What will the following code print?

**def** f(x = 0, y = 1):

**return** x \* y

**print**(f())

Top of Form

A. 0  
B. 1  
C. None  
D. Runtime error since no parameters are passed in the call to f.

Bottom of Form

advfuncs-1-2: What will the following code print?

**def** f(x = 0, y = 1):

**return** x \* y

**print**(f(1))

Top of Form

A. 0  
B. 1  
C. None  
D. Runtime error since the second parameter value is missing.

Bottom of Form

**3.** Write a function called str\_mult that takes in a required string parameter and an optional integer parameter. The default value for the integer parameter should be 3. The function should return the string multiplied by the integer parameter.

15.2. Keyword Parameters

In the previous section, on [Optional Parameters](https://fopp.umsi.education/runestone/static/fopp/AdvancedFunctions/OptionalParameters.html#optional-params-chap) you learned how to define default values for formal parameters, which made it optional to provide values for those parameters when invoking the functions.

In this chapter, you’ll see one more way to invoke functions with optional parameters, with keyword-based parameter passing. This is particularly convenient when there are several optional parameters and you want to provide a value for one of the later parameters while not providing a value for the earlier ones.

The online official python documentation includes a tutorial on optional parameters which covers the topic quite well. Please read the content there: [Keyword arguments](http://docs.python.org/3/tutorial/controlflow.html#keyword-arguments)

Don’t worry about the def cheeseshop(kind, \*arguments, \*\*keywords): example. You should be able to get by without understanding \*parameters and \*\*parameters in this course. But do make sure you understand the stuff above that.

The basic idea of passing arguments by keyword is very simple. When invoking a function, inside the parentheses there are always 0 or more values, separated by commas. With keyword arguments, some of the values can be of the form paramname = <expr> instead of just <expr>. Note that when you have paramname = <expr> in a function definition, it is defining the default value for a parameter when no value is provided in the invocation; when you have paramname = <expr> in the invocation, it is supplying a value, overriding the default for that paramname.

To make it easier to follow the details of the examples in the official python tutorial, you can step through them in CodeLens.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Python 3.3   |  |  |  | | --- | --- | --- | |  | 1 | def parrot(voltage, state='a stiff', action='voom', type='Norwegian Blue'): | | 2 | print("-- This parrot wouldn't" + action,) | | 3 | print("if you put" + str(voltage) + "volts through it.") | | 4 | print("-- Lovely plumage, the" +  type) | | 5 | print("-- It's " + state + "!") | | 6 |  | | 7 | parrot(1000)    # 1 positional argument | | 8 | parrot(voltage=1000)   # 1 keyword argument | | 9 | parrot(voltage=1000000, action='VOOOOOM') # 2 keyword arguments | | 10 | parrot(action='VOOOOOM', voltage=1000000) # 2 keyword arguments | | 11 | parrot('a million', 'bereft of life', 'jump') # 3 positional arguments | | 12 | parrot('a thousand', state='pushing up the daisies') # 1 positional, 1 keyword | |

## **15.2.1. Keyword Parameters with .format**

Earlier you learned how to use the format method for strings, which allows you to structure strings like fill-in-the-blank sentences. Now that you’ve learned about optional and keyword parameters, we can introduce a new way to use the format method.

This other option is to specifically refer to keywords for interpolation values, like below.

names\_scores = [("Jack",[67,89,91]),("Emily",[72,95,42]),("Taylor",[83,92,86])]

for name, scores in names\_scores:

print("The scores {nm} got were: {s1},{s2},{s3}.".format(nm=name,s1=scores[0],s2=scores[1],s3=scores[2]))

​

Sometimes, you may want to use the .format method to insert the same value into a string multiple times. You can do this by simply passing the same string into the format method, assuming you have included {} s in the string everywhere you want to interpolate them. But you can also use positional passing references to do this! The order in which you pass arguments into the format method matters: the first one is argument 0, the second is argument 1, and so on.

For example,

# this works

names = ["Jack","Jill","Mary"]

for n in names:

print("'{}!' she yelled. '{}! {}, {}!'".format(n,n,n,"say hello"))

​

# but this also works!

names = ["Jack","Jill","Mary"]

for n in names:

print("'{0}!' she yelled. '{0}! {0}, {1}!'".format(n,"say hello"))

​

**Check your understanding**

advfuncs-2-1: What value will be printed for z?

initial = 7

**def** f(x, y = 3, z = initial):

**print**("x, y, z are:", x, y, z)

f(2, 5)

Top of Form

A. 2  
B. 3  
C. 5  
D. 7  
E. Runtime error since not enough values are passed in the call to f

Bottom of Form

advfuncs-2-2: What value will be printed for y?

initial = 7

**def** f(x, y = 3, z = initial):

**print**("x, y, z are:", x, y, z)

f(2, z = 10)

Top of Form

A. 2  
B. 3  
C. 5  
D. 10  
E. Runtime error since no value is provided for y, which comes before z

Bottom of Form

advfuncs-2-3: What value will be printed for x?

initial = 7

**def** f(x, y = 3, z = initial):

**print**("x, y, z are:", x, y, z)

f(2, x=5)

Top of Form

A. 2  
B. 3  
C. 5  
D. 7  
E. Runtime error since two different values are provided for x

Bottom of Form

advfuncs-2-4: What value will be printed for z?

initial = 7

**def** f(x, y = 3, z = initial):

**print** "x, y, z are:", x, y, z

initial = 0

f(2)

Top of Form

A. 2  
B. 7  
C. 0  
D. Runtime error since two different values are provided for initial.

Bottom of Form

advfuncs-2-5: What value will be printed below?

names = ["Alexey", "Catalina", "Mitsuki", "Pablo"]

**print**("'{first}!' she yelled. 'Come here, {first}! {f\_one}, {f\_two}, and {f\_three} are here!'".format(first = names[1], f\_one = names[0], f\_two = names[2], f\_three = names[3]))

Top of Form

A. 'first!' she yelled. 'Come here, first! f\_one, f\_two, and f\_three are here!'  
B. 'Alexey!' she yelled. 'Come here, Alexey! Catalina, Misuki, and Pablo are here!'  
C. 'Catalina!' she yelled. 'Come here, Catalina! Alexey, Misuki, and Pablo are here!'  
D. There is an error. You cannot repeatedly use the keyword parameters.

Bottom of Form

**5.** Define a function called multiply. It should have one required parameter, a string. It should also have one optional parameter, an integer, named mult\_int, with a default value of 10. The function should return the string multiplied by the integer. (i.e.: Given inputs “Hello”, mult\_int=3, the function should return “HelloHelloHello”)

​

def multiply():

​

​

**6.** Currently the function is supposed to take 1 required parameter, and 2 optional parameters, however the code doesn’t work. Fix the code so that it passes the test. This should only require changing one line of code.

​

def waste(var = "Water", mar, marble = "type"):

final\_string = var + " " + marble + " " + mar

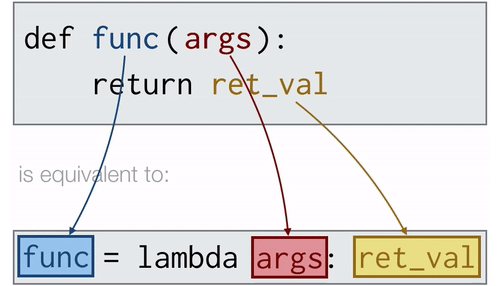
return final\_string

15.3. Anonymous functions with lambda expressions

To further drive home the idea that we are passing a function object as a parameter to the sorted object, let’s see an alternative notation for creating a function, a **lambda expression**. The syntax of a lambda expression is the word “lambda” followed by parameter names, separated by commas but not inside (parentheses), followed by a colon and then an expression. lambda arguments: expression yields a function object. This unnamed object behaves just like the function object constructed below.

**def** fname(arguments):

**return** expression



Consider the following code

def f(x):

return x - 1

​

print(f)

print(type(f))

print(f(3))

​

print(lambda x: x-2)

print(type(lambda x: x-2))

print((lambda x: x-2)(6))

​

Note the paralells between the two. At line 4, f is bound to a function object. Its printed representation is “<function f>”. At line 8, the lambda expression produces a function object. Because it is unnamed (anonymous), its printed representation doesn’t include a name for it, “<function <lambda>>”. Both are of type ‘function’.

A function, whether named or anonymous, can be called by placing parentheses () after it. In this case, because there is one parameter, there is one value in parentheses. This works the same way for the named function and the anonymous function produced by the lambda expression. The lambda expression had to go in parentheses just for the purposes of grouping all its contents together. Without the extra parentheses around it on line 10, the interpreter would group things differently and make a function of x that returns x - 2(6).

Some students find it more natural to work with lambda expressions than to refer to a function by name. Others find the syntax of lambda expressions confusing. It’s up to you which version you want to use though you will need to be able to read and understand lambda expressions that are written by others. In all the examples below, both ways of doing it will be illustrated.

Say we want to create a function that takes a string and returns the last character in that string. What might this look like with the functions you’ve used before?

def last\_char(s):

return s[-1]

​

To re-write this using lambda notation, we can do the following:

last\_char = (lambda s: s[-1])

​

(ac15\_3\_3)

Note that neither function is actually invoked. Look at the parallels between the two structures. The parameters are defined in both functions with the variable s. In the typical function, we have to use the keyword return to send back the value. In a lambda function, that is not necessary - whatever is placed after the colon is what will be returned.

**Check Your Understanding**

advfuncs-3-1: If the input to this lambda function is a number, what is returned?

(**lambda** x: -x)

Top of Form

A. A string with a - in front of the number.  
B. A number of the opposite sign (positive number becomes negative, negative becomes positive).  
C. Nothing is returned because there is no return statement.

Bottom of Form