Functions and Parameter List in Python

The general syntax for creating a function is:

```
def functionName(parameters):
    … stuff …
```

The parameters are optional, and if there is more than one they are written as a sequence of comma-separated identifiers, or as a sequence of `identifier=value` pairs.

Example of a function that calculates the area of a triangle using Heron's formula:

```
def heron(a, b, c):
    s = (a + b + c) / 2
    return math.sqrt(s * (s – a)*(s – b)*(s – c))
```

Each parameter, a, b, and c, is initialized with the corresponding value that was passed as an argument. When the function is called we must supply all of the arguments, for example, `heron(3, 4, 5)`. If we give too few or too many arguments, a `TypeError` exception will be raised. When we do a call like this we are said to be using ***positional arguments***, because each argument passed is set as the value of the parameter in the corresponding position.

Some functions have parameters for which there can be a sensible **default**. We can specify a default value for a particular parameter by using the `parameter=default` syntax. If we have a method that has two parameters, for example `methodX(title, length=25)`, and a default value is specified for the second parameter, then we can call this method with just one argument, for example `methodX("Monty Python")`.

We are not forced, however, to use positional arguments. That is, we are not forced to pass our arguments in the order they appear in the function's definition—instead, we can use ***keyword arguments***, passing each argument in the form `name=value`.

As an example, here is a small function that returns the string it is given, or if it is longer than the specified length, it returns a shortened version with an indicator added:

```
def shorten(text, length=25, indicator="..."):
    if len(text) > length:
        text = text[:length – len(indicator)] + indicator
    return text
```

Here are four example calls to the method **shorten**:

```
shorten("The Oceans")                          # returns: 'The Oceans'
shorten(length=7, text="The Oceans")           # returns: 'The ...'
shorten("The Oceans", indicator="&", length=7) # returns: 'The Oc&'
shorten("The Oceans", 7, "&")                  # returns: 'The Oc&'
```

Since both length and indicator have default values, either or both can be omitted entirely, in which case the default is used—this is what happens in the *first* call.
In the *second* call we use **keyword arguments** for both of the specified parameters, so we can order them as we like.
The *third* call mixes both **positional** and **keyword arguments**. We used a positional first argument, and then two keyword arguments.

Important note: positional arguments must always **precede** keyword arguments!

The *fourth* call simply uses **positional arguments**.

The difference between a mandatory parameter and an optional parameter is that a parameter with a default is optional (because Python can use the default value), and a parameter with no default is mandatory (because Python cannot guess.)

When default values are given they are created at the time the `def` statement is executed (that is, when the function is created), not when the function is called. For immutable arguments like numbers and strings this doesn't make any difference, but for mutable arguments you need to be careful and handle that parameter in a particular way. Take a look at the following example:

Let's assume we have a small function called `append_to` that appends an element to a list:

```
def append_to(element, aList=[]):  # parameters: element and a list
    aList.append(element)
    return aList
```

Then, if we call the method two times:

```
my_list = append_to(12)          # first call to the method
print (my_list)
my_other_list = append_to(42)    # second call to the method
print (my_other_list)
```

We might expect that a new list is created each time the function is called if a second argument isn't provided. So we would assume the output after calling the above lines of code is:

```
[12]
[42]
```

In reality, however, when the `append_to` function is created the `aList` parameter is set to refer to a new list. Whenever this function is called with just the first parameter, the default list will be the one that was created at the same time as the function itself—so no new list is created. So the actual output after running the above code is:

```
[12]
[12, 42]
```

Python's default arguments are evaluated *once* when the function is defined, not each time the function is called. This means that if you use a mutable default argument and mutate it, you will have mutated that object for all future calls to the function as well.

The solution is to create a new object each time the function is called, by calling a default argument to signal that no argument was provided (i.e. use `None`). See the modified code:

```
def append_to(element, aList=None):    # use of None
      if aList is None:      # if no second parameter…
            aList = []       # create a list
      aList.append(element)
      return aList
```

Now if we call the method two times (like we did earlier), the output will be correct:

```
[12]
[42]
```

Using the modified code, we create a new list every time the function is called without a list argument. If a list argument is given, we use it.

This strategy of having a default of `None` and creating a fresh object should be used for dictionaries, lists, sets, and any other mutable data types that we want to use as default arguments.