e to remember that function parameter defaults are evaluated once, when the function is defined (i.e. when the module is loaded, or in this Jupyter notebook, when we "execute" the function definition), and not every time the function is called.

Consider the following scenario.

We are creating a grocery list, and we want our list to contain consistently formatted data with name, quantity and measurement unit:

`bananas (2 units) grapes (1 bunch) milk (1 liter) python (1 medium-rare)`

To make sure the data is consistent, we want to use a function that we can call to add the item to our list.

So we'll need to provide it our current grocery list as well as the item information to be added:

```
def add_item(name, quantity, unit, grocery_list):
    item_fmt = "{0} ({1} {2})".format(name, quantity, unit)
    grocery_list.append(item_fmt)
    return grocery_list
```

We have two stores we want to visit, so we set up two grocery lists:

```
store_1 = []
store_2 = []

add_item('bananas', 2, 'units', store_1)
add_item('grapes', 1, 'bunch', store_1)
add_item('python', 1, 'medium-rare', store_2)
['python (1 medium-rare)']

store_1
['bananas (2 units)', 'grapes (1 bunch)']

store_2
['python (1 medium-rare)']
```

Ok, working great. But let's make the function a little easier to use - if the user does not supply an existing grocery list to append the item to, let's just go ahead and default our `grocery_list` to an empty list hence starting a new shopping list:

```
def add_item(name, quantity, unit, grocery_list=[]):
    item_fmt = "{0} ({1} {2})".format(name, quantity, unit)
    grocery_list.append(item_fmt)
    return grocery_list

store_1 = add_item('bananas', 2, 'units')
add_item('grapes', 1, 'bunch', store_1)
['bananas (2 units)', 'grapes (1 bunch)']

store_1
['bananas (2 units)', 'grapes (1 bunch)']
```

OK, so that seems to be working as expected.

Let's start our second list:

```
store_2 = add_item('milk', 1, 'gallon')

print(store_2)
['bananas (2 units)', 'grapes (1 bunch)', 'milk (1 gallon)']
```

??? What's going on? Our second list somehow contains the items that are in the first list.

What happened is that the returned value in the first call we made was the default grocery list - but remember that the list was created once and for all when the function was **created** not called. So everytime we call the function, that is the **same** list being used as the default.

When we started out first list, we were adding item to that default list.

When we started our second list, we were adding items to the **same** default list (since it is the same object).

We can avoid this problem using the same pattern as in the previous example we had with the default date time value. We use None as a default value instead, and generate a new empty list (hence starting a new list) if none was provided.

```
def add_item(name, quantity, unit, grocery_list=None):
    if not grocery_list:
        grocery_list = []
    item_fmt = "{0} ({1} {2})".format(name, quantity, unit)
    grocery_list.append(item_fmt)
    return grocery_list

store_1 = add_item('bananas', 2, 'units')
add_item('grapes', 1, 'bunch', store_1)
['bananas (2 units)', 'grapes (1 bunch)']

store_2 = add_item('milk', 1, 'gallon')
store_2
['milk (1 gallon)']
```

Issue resolved!

However, there are legitimate use cases (well, almost legitimate, often we're better off using a different approach that we'll see when we look at closures), but here's a simple one.

We want our function to cache results, so that we don't recalculate something more than once.

Let's say we have a factorial function, that can be defined recursively as:

```
n! = n * (n-1)!
```

```
def factorial(n):
    if n < 1:
        return 1
    else:
        print('calculating {0}!'.format(n))
        return n * factorial(n-1)

factorial(3)
calculating 3!
calculating 2!
```

```
calculating 1!

6

factorial(3)
calculating 3!
calculating 2!
calculating 1!

6
```

As you can see we had to recalculate all those factorials the second time around.

Let's cache the results leveraging what we saw in the previous example:

```
def factorial(n, cache={}):
    if n < 1:
        return 1
    elif n in cache:
        return cache[n]
    else:
        print('calculating {0}!'.format(n))
        result = n * factorial(n-1)
        cache[n] = result
        return result

factorial(3)
calculating 3!
calculating 2!
calculating 1!

6

factorial(3)
6
```

Now as you can see, the second time around we did not have to recalculate all the factorials. In fact, to calculate higher factorials, you'll notice that we don't need to re-run *all* the recursive calls:

```
factorial(5)
calculating 5!
calculating 4!

120
```

`5!` and `4!` was calculated since they weren't cached, but since `3!` was already cached we didn't have to recalculate it - it was a quick lookup instead.

This technique is something called memoization, and we'll come back to it in much more detail when we discuss closures and decorators.