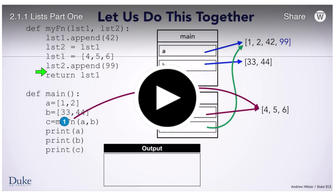
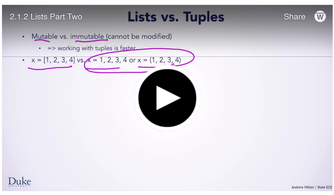
Lists: Sequences of Data



Lists Part Two

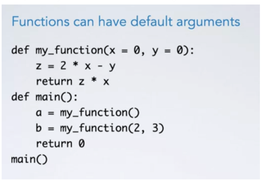


Day 6, Part 1

Now it is time to do **09_list_max** on the Mastery Learning Platform. Login to the server, go to **09_list_max** in your git repository, and read the README for directions. When you have passed that assignment, return to Sakai and continue with the content here.

Default Arguments Revisited

Recall that when we introduce you to function, we briefly mentioned the idea of default arguments:



Here, we can call my_function() to implicitly pass 0 for x and y, or my_function(2,3) to explicitly pass 2 for x and 3 for y. We could also call my_function(9) to explicitly pass 9 for x and implicitly pass 0 for y.

Now that you have seen lists, we want to take a moment to revisit default arguments to warn you about a danger of them.

Consider the following code:

```
# This function does something very bad...
def f(x=[]):
    x.append(42)
    return x


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

Here, the programmer wrote function **f** to append 42 to a list and return it. E.g., if given [3], add 42 making [3, 42] and then return [3, 42]. The programmer wrote a default argument for x of [], i.e., the empty list. While this may seem like a reasonable idea, it actually does something very surprising. We would expect this code to print

- [42]
- [42]

However, it actually prints

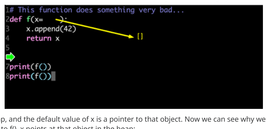
- [42]
- [42, 42]

Which seems quite surprising!

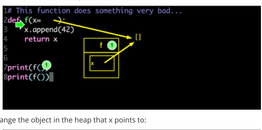
The problem here is that we have a **mutable default argument:** that is a default argument, that can be changed. When we do x.append(42), we are changing ("mutating") x. As a general rule in Python: **only use default arguments that are immutable**. Note that immutable means "cannot be changed". Strings, tuples, and numbers are all immutable and safe to use as default arguments. Lists are mutable and thus not safe to use. Several other things you will learn about later (sets, maps, and objects of your own classes) are also mutable and thus not safe to use.

At this point, you are probably wondering how Python gets this result. The reason is in when the default argument is evaluated. When Python executes the code above, it starts at line 1 and works down. When Python sees def f(x=[]): on line 2, it binds the name f to the function body. However, it also evaluates the default argument expressions at this time (once in the evaluation of the whole code). In this case, x = []. Remember that [] makes an empty list in the heap, and x points at that list.

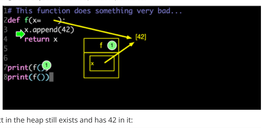
In particular, after Python evaluates the definition of the function *f*, the state of the program looks roughly like this:



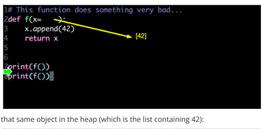
That is, we have made a list in the heap, and the default value of x is a pointer to that object. Now we can see why we get the behavior that we saw earlier. When we execute the first call to f(), x points at that object in the heap:



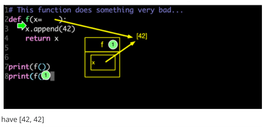
Then when we do x.append(42), we change the object in the heap that x points to:



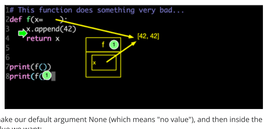
Now, when we return from *f*, the object in the heap still exists and has 42 in it:



Now when we call f() again, x points at that same object in the heap (which is the list containing 42):



So when we append 42 to that list, we have [42, 42]



So how do we fix this problem? We make our default argument None (which means "no value"), and then inside the function, we check if the value of the argument is None, and set it to the value we want:

```
# This function fixes the problem
def f(x=None):
    if (x is None):
        x = []
        pass
    x.append(42)
    return x


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

Now, when we call f() it is the same as calling f(None). The first thing f does is check if x is None. If so, it makes x be the empty list. How is this different from before? Each time Python evaluates [] we get a different empty list. So every time we do x = [] on line 4, Python will make a new empty list in the heap, and make x point at that. Line 4 will get evaluated for every call to f(), whereas the default argument only gets evaluated once.

This idiom is common in Python. Anytime you want to have a function with a default argument that is mutable, instead make the default value None, then inside the function, check for None and set the value to what you really want.

Heart Rate Example

Heart Rate Example