PIC 16: Function calls in Python

Function calls in C++

Your instructor in PIC 10A should have explained function calls in a way equivalent to what follows.

How do we understand the following code?

```
int f(int i) {
    cout << i << endl;
    i = i + 1;
    cout << i << endl;

    return i;
}

int main() {
    int j1 = 0;
    int j2 = f(j1);
    cout << j1 << ' ' << j2 << endl;

    return 0;
}
```

Well, it is equivalent to the following.

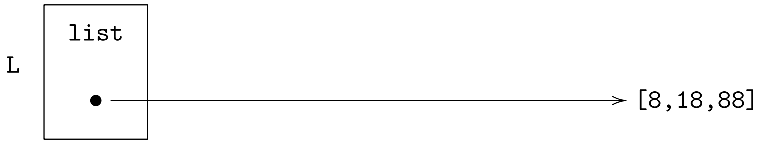
```
int main() {
    int j1 = 0;

    // We replace int j2 = f(j1)
    // by what follows...

    int j2;                 // Global variable j2 needs to be declared.

    {                       // Introduce function call scope.
        int i = j1;         // Make parameter assignments.

        cout << i << endl;  // Run function definition.
        i = i + 1;          // Run function definition.
        cout << i << endl;  // Run function definition.

        j2 = i;             // Deal with the return statement appropriately.
    }                       // End function call scope.

    cout << j1 << ' ' << j2 << endl;

    return 0;
}
```

(My C++ code snippets can be found at: http://math.ucla.edu/~mjandr/PIC10A/snippets.zip)

Function calls in Python: pass by value (pass by object reference)

Recall what the Python tutorial says about function calls: “arguments are passed using *call by value* (where the *value* is always an object *reference*, not the value of the object).”

The statement in parentheses now makes sense. After making the assignment `L = [8,18,88]`, we have the following picture.



The value of the variable is what is inside the box: `list`, and the `reference` to the object.
The value of the object is `[8,18,88]`. So the values of the variable and of the object are distinct.

Consider the following code.

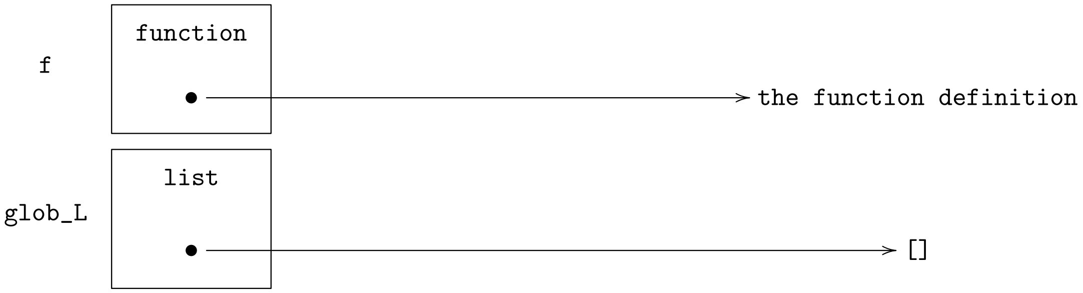
```
def f(loc_L):
    loc_L.append(0)
    loc_L = [8]

glob_L = []
f(glob_L)
print(glob_L)
```

Just like in C++, you can unpack this as follows

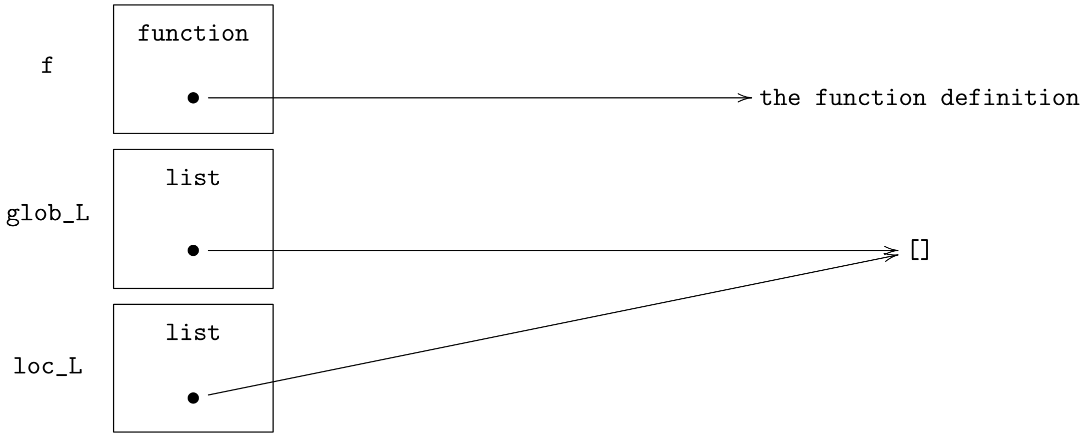
```
glob_L = []
                 # We can't introduce a scope,
                 # but you should introduce one mentally.
loc_L = glob_L   # Make parameter assignments.

loc_L.append(0)  # Run function definition.
loc_L = [8]      # Run function definition.

del loc_L        # We implement the consequences of ending our mental scope.

print(glob_L)
```

Upon running the original code we obtain the following “video”.

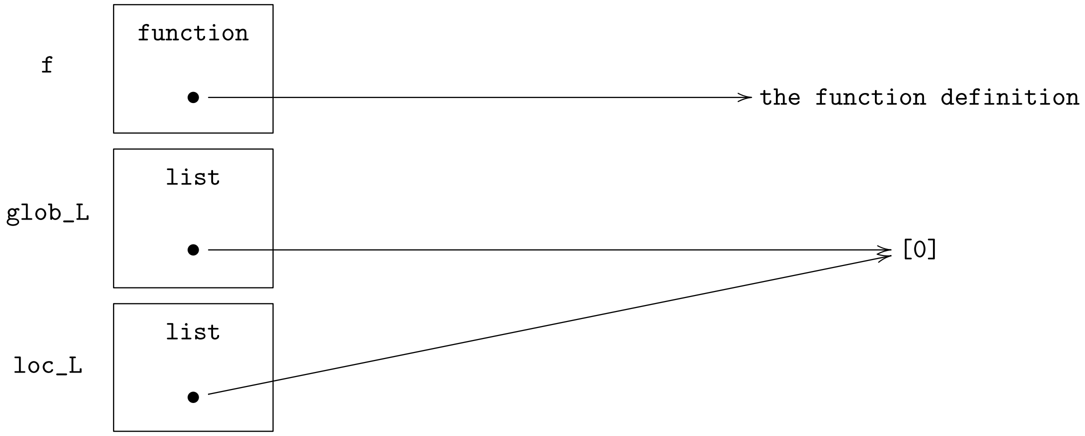
1. Before the function call.



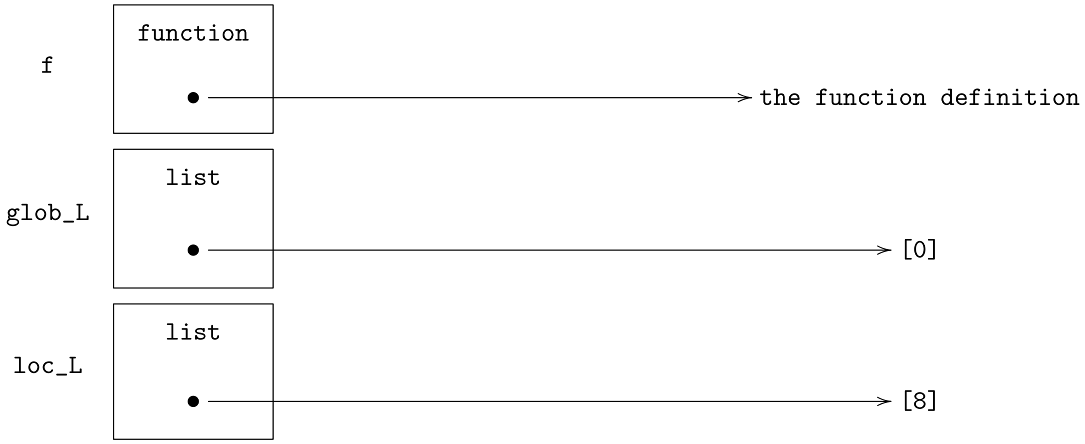
2. During the function call: `loc_L = glob_L`.



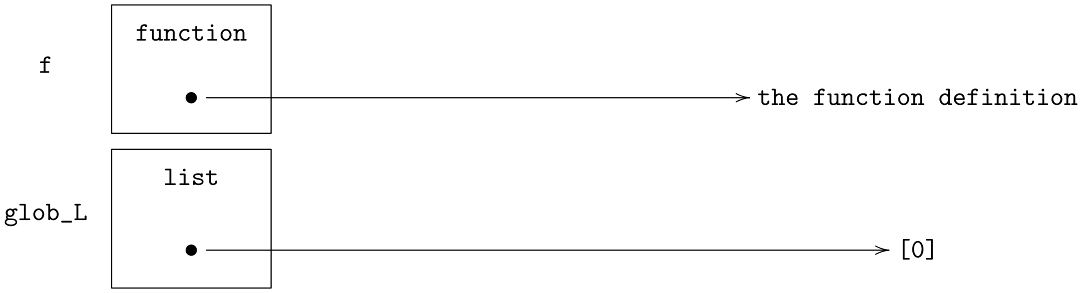
3. During the function call: `loc_L.append(0)`.



4. During the function call: `loc_L = [8]`.



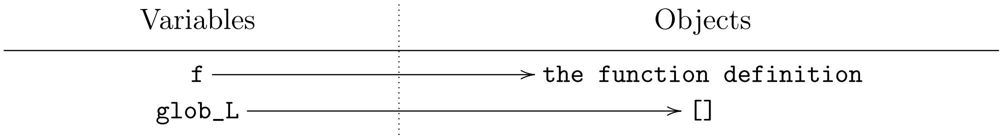
5. After the function call: `del loc_L`.



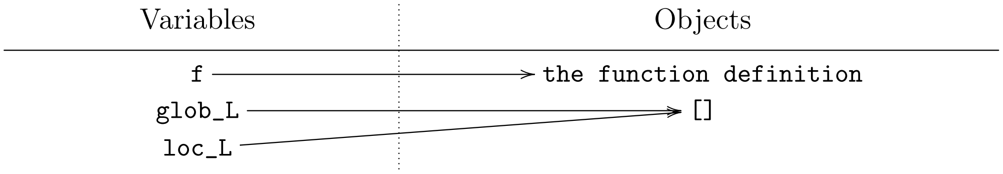
6. `[0]` is printed.

Alternatively, we can express the same information using symbol tables.
Then you just have remember that “value” means the “object reference”.

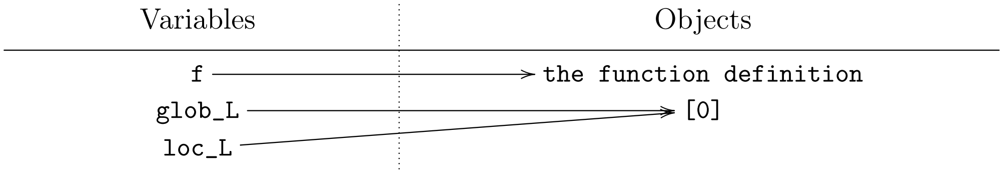
1. Before the function call.



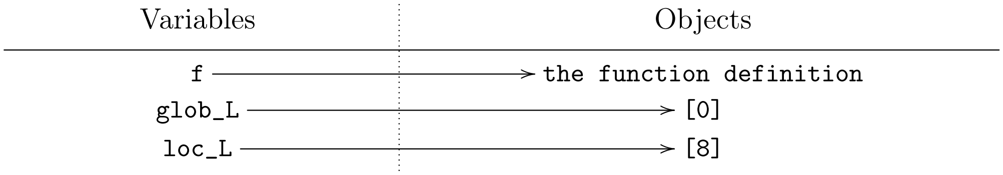
2. During the function call: `loc_L = glob_L`.



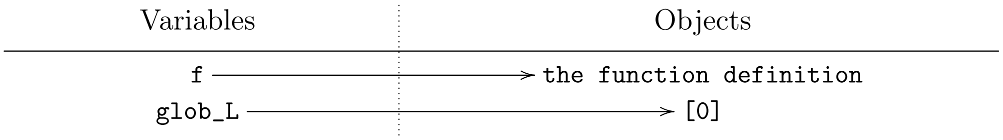
3. During the function call: `loc_L.append(0)`.



4. During the function call: `loc_L = [8]`.



5. After the function call: `del loc_L`.



6. `[0]` is printed.

Function calls in Python: scope

We've addressed the second paragraph that I told you to ignore when reading about functions, the one starting with "the actual parameters". What about the paragraph starting with "the *execution*"? I implicitly addressed some aspects of it above with the variable names `glob_L` and `loc_L`. Now I'll try to address the rest of what it says.

First, in the sentence "thus, global variables and . . ." I want you to ignore the part in parentheses. In particular, I want to avoid the keyword `global` until we absolutely have to use it (because it will probably encourage poor coding on your part).

The sentence "all variable assignments in a function store the value in the local symbol table" implies that as soon as you see `x = something` in a function definition, `x` is local to that function. As a consequence, the function parameters are local to the function because they are assigned to implicitly. You understand the rest of the paragraph provided that you understand the two examples in `funcScope.py`. They are commented extensively, but I'll explain them with the associated diagram in lecture (drawing the diagrams on here would take days).