

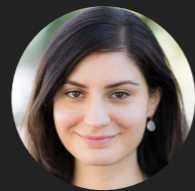
02-13-2023

# CS051A

## INTRO TO COMPUTER SCIENCE WITH TOPICS IN AI

### 8: Scoping and debugging

---



Alexandra Papoutsaki

she/her/hers

Lectures



Zilong Ye

he/him/his

Labs

Welcome to lecture 8, the lecture I hope we will catch up with all the material :)

## Lecture 8: Scoping and debugging

- ▶ Scoping
- ▶ Debugging
- ▶ Strings

Today we will talk mostly about the scope of variables and how to debug our programs, but we will also revisit strings.

## scope.py

- ▶ What will be printed out when we run scope.py?
  - ▶ When we run the file, the first two functions, `double_input` and `triple_input`, will get defined.
  - ▶ Then, the interpreter will execute the three statements at the bottom of the file, one at a time.
    - ▶ The first `print` statement will print out 20.
    - ▶ The second `print` statement will print out 30.
      - ▶ Why not 60, that is in `double_input` we assigned 20 to `val` as the first line.
      - ▶ `val = 2*val` updates the value of the parameter, **NOT** the variable outside the function.

Let's start by looking at the file `scope.py` and ask ourselves what will happen? When we run the file, the two functions, `double_input` and `triple_input` will be defined. Then the Python interpreter will move on to the bottom of the file where there are three statements it needs to execute one at a time. Once it defines the `val` variable and initializes it to 10, the first `print` statement will print out 20 and the second will print out 30. You might be wondering why not print 60? This is because the statement `val = 2*val` updates the value of the parameter locally only, **NOT** the variable outside the function.

## Scope

- ▶ The **scope** of a variable is the portion of the code we can reference that variable in and have it be valid.
- ▶ When we declare a variable *in the Python console/shell*, e.g., `x = 10`, what is its scope?
  - ▶ The scope is any shell statements/expressions typed after it.
- ▶ When we declare a variable (outside a function) *in a file*, what is its scope?
  - ▶ anywhere below it in the file. Remember, running a program is very similar to typing those commands into the shell
- ▶ When we declare a variable *inside a function*, what is its scope?
  - ▶ anywhere below it but **within the function**
- ▶ What is the scope of the parameters?
  - ▶ anywhere **within the function**
  - ▶ It doesn't really make sense outside of the function since we wouldn't have a value for it
- ▶ Additionally, the scope also defines the context for a variable reference.

In general, the scope of a variable is the portion of the code we can reference that variable in and have it be valid. If we were to work on the Python console/shell and type `x=10`, what is the scope of `x`? It's visible to any statement/expression typed within the shell after that line. In contrast, let's say we work on a `.py` file and declare a variable outside a function (this type of variable will be called a global variable). Such variables are visible anywhere in the file below that particular line. Finally, if we declare a variable within a function, the scope of that variable is anywhere below that line but within that function! Same applies to the parameters, their scope is only visible within the function.

## scope2.py

- ▶ What will be printed out when we run scope2.py?
  - ▶ The program starts at the top and declares three **global** variables x, y, and z.
  - ▶ Then, it defines the function `mystery1`.
  - ▶ Then it calls `mystery1` with the arguments `10` and `20`.
    - ▶ The parameter `a` is associated with the value `10`. The parameter `z` gets the value `20`.
      - ▶ Notice that this is **different** than the global variable `z`! In particular, when we execute `z = 100`, this reassigns the value of the local `z`, but not the global.
    - ▶ The function prints out `s`, `x`, `y`, and `z`.
      - ▶ `s` is a local variable. `x` and `y` are global. `z` is the parameter.
      - ▶ And then sets the value of the local variable `z` to `100`.
  - ▶ After `mystery1` returns, we print out the values of the global variables `x`, `y` and `z`.
    - ▶ `x` and `y` weren't changed anywhere. `z` being changed was the parameter not the global `z`.
  - ▶ What would happen if we had uncommented `mystery2` and then added a call to it at the end and ran `scope2.py`?
    - ▶ Error. The scope of `a` is only defined within `mystery1`, so it cannot be accessed anywhere outside the function.

Let's now see `scope2.py`. What do we expect that will happen when we run it? The program will define three global variables `x`, `y`, `z`, define the function `mystery1`, and then it will call `mystery1` with arguments `10` and `20`. The parameter `a` is associated with the value `10` while parameter `z` with the value `20`. Notice that this is different than the global variable `z`. In particular, `z=100`, is only applied within the function and does not affect the global variable `z`. After `mystery1` returns, `x` and `y` remain unaffected. Same applies to `z` because the change we made applied to the parameter not the global `z`. If I were to uncomment the `mystery2`, the program would produce error as the scope of `a` is only defined within `mystery1`.

## scope2.py

- ▶ What will be printed out when we run scope2.py? [cont'd]
  - ▶ We continue with creating a list, `some_list`, that contains only one item, the string “apple”.
  - ▶ We pass `some_list` to `mystery3` and we see that the contents of `some_list` were altered to “banana”.
  - ▶ Why?! Remember aliasing?
- ▶ Our `some_list` (which now contains “banana”) is now passed to `mystery4`. Why didn't `some_list` change to “pineapple”? Because `any_list` referenced a new memory location.

Now let's look at `mystery3` and `mystery4` functions that will both take a list. If we create a list called `some_list` that only contains the string “apple” and we pass it to `mystery3`, we get the following two print statements:

```
mystery3 any_list contains:['banana']
```

```
some_list contains:['banana']
```

Why?! Remember aliasing from the previous lecture?

In contrast, when we pass `any_list` to `mystery4` it stays as is. The reason is that the parameter `any_list` references a new location in memory, and does not point to the `some_list` anymore.

```
mystery4 any_list contains:['banana']
```

```
mystery4 any_list now contains:['pineapple']
```

```
some_list now contains:['banana']
```

## Lecture 8: Scoping and debugging

- ▶ Scoping
- ▶ Debugging
- ▶ Strings

Let's now switch to debugging, the process of handling things going wrong with our code. And despite our best intentions, things will go wrong :)

## When things go wrong

- ▶ The `quadruple_input` function in `debugging.py` code attempts to quadruple the input value by adding it four times.
- ▶ However, no matter the input it returns 6.

```
>>> quadruple_input(5)
6
>>> quadruple_input(4)
6
>>> quadruple_input(6)
6
```

I have provided you with a file `debugging.py` where the `quadruple_input` function attempts to quadruple the input value by adding it four times. If I run this code though, no matter what the arguments, I always get back 6?!



## Debugging

- ▶ A **bug** is a behavior in the code that is not intended.
- ▶ **Debugging** is the practice of trying to find and fix bugs.
- ▶ You might be able to look at the code and find the bug in this example. However, if you can't, you can try and add more information to your program to figure out what the problem is.
  - ▶ Adding `print` statement is one good way to figure out what your function is doing

That means my code has a bug. A bug is a behavior in our code that we did not intend for and debugging is the practice of trying to locate and fix bugs. Debugging might be as easy as looking at our code and finding the bug. However, things might be slightly more complicated and using print functions will come handy.



## debugging-with-prints.py

- ▶ If we run this version, we start to see what the problem is.
- ▶ The problem is that we've used the input parameter as the variable in the for loop and the value is getting lost!
- ▶ The fix is to use a different variable name here (e.g., i)
- ▶ When you're all done debugging and your code works, make sure to remove the print statements!
- ▶ It's worth taking ten seconds to make your print formatting nice:
  - ▶ In loop vs out of loop,
  - ▶ Iteration boundaries,
  - ▶ Labels for positions.

```
>>> quadruple_input(5)
A:  0 5
--
B:  0 0
C:  0 0
--
B:  0 1
C:  1 1
--
B:  1 2
C:  3 2
--
B:  3 3
C:  6 3
D:  6 3
6
```

For example, in the file `debugging-with-prints.py`, we will start seeing what the problem is. I have used in my for loop the input parameter as a variable! The fix is to use in the for loop a different variable name, e.g., `i`. Print statements are great but we want to make sure we clean up our code. We can also help us by formatting our print statements nicely to indicate when we are inside and outside a loop, what are the iteration boundaries, and in general label the positions.

## The debugger

- ▶ Use the little bug  icon to run a special debugging program.
  - ▶ It runs Python code in a special way that is under the control of another program
  - ▶ When the program breaks (because you asked it to with a "break point"  by clicking in the gutter near the line numbers, you can...
    - ▶ Step in/step out/step over the next line of code (if it's not stuck at an error, of course).
    - ▶ View call stack.
    - ▶ View stack frame, i.e. variables in local scope.

If print statements are not enough, we have at our disposal the debugger, a special debugging program that comes integrated in IDEs, such as PyCharm. If you locate the little bug button and add a breakpoint in your code, you can use functionalities such as step in/out/over the next line of code, see the call stack, and the stack frame. Pretty cool!

## Lecture 8: Scoping and debugging

- ▶ Scoping
- ▶ Debugging
- ▶ **Strings**

Totally different topic now, let's go back to our friends strings.

## String methods that might come handy

- ▶ Remember, strings are immutable.
- ▶ `s.lower()`: returns a copy of the string `s` converted to lower case.
- ▶ `s.replace(old, new)`: returns a copy of `s` with all occurrences of substring `old` replaced by `new`.
- ▶ `s.find(sub)`: returns the lowest index in string `s` where substring `sub` is found.
  - ▶ `s.find(sub, start)`: returns the lowest index in string `s` where substring `sub` is found within `s[start:]`.
  - ▶ `s.find(sub, start, end)`: returns the lowest index in string `s` where substring `sub` is found within `s[start:end]`.

```
>>> fruit = "Banana"
>>> fruit.lower()
'banana'
>>> fruit
'Banana'
>>> fruit.replace("a", "o")
'Bonono'
>>> fruit
'Banana'
>>> fruit = fruit.replace("a", "o")
>>> fruit
'Bonono'
>>> fruit.find("a")
-1
>>> fruit.find("o")
1
>>> fruit.find("o", 3)
3
>>> fruit.count("o")
3
>>> fruit.count("a")
0
```

A few methods that can come handy are `lower`, `replace`, `find` with the specifications above. Remember, strings are immutable and most of these methods will return you a copy of the original string. You would need to store it in a variable if you want to not lose it.

## Practice Time

- ▶ Write a function `find_letter(string, letter)` which returns the index of the first occurrence of `letter` in `string`.

```
>>> def find_letter(string, letter):
...     return string.find(letter)
...
>>> find_letter("hello", "l")
2
>>> def find_letter_alternative(string, letter):
...     for index in range(len(string)):
...         if string[index] == letter:
...             return index
...     return -1
...
>>> find_letter_alternative("hello", "l")
2
```

For the final practice round for this lecture, let's write a function that returns the index of the first occurrence of a letter in a string. There are multiple ways of going about it. We might utilize the `find` method we just learned. Or we might write a `for` loop ourselves where the first time we find a matching character, we return that specific index. If we have exhausted the entire string, we return `-1`.

## enumerate

- ▶ enumerate function returns pairs containing a count (defaults to zero) and a value
  - ▶ i.e., (0, seq[0]), (1, seq[1]), (2, seq[2]), ...

```
>>> for (key, val) in enumerate(["apple", "banana", "pineapple"]):
...     print (str(key)+ ": " + val)
...
0: apple
1: banana
2: pineapple
```

Let's look now into the enumerate function which comes handy if we want to solve problems like the one we encounter. The enumerate function takes at least one parameter, an enumerable object (so far all sequences we have seen are enumerable!), and returns pairs where the first part is the count (defaults to zero, but can be changed if passed as a second parameter), and the second part is the value. Here's how it would look like if we wanted to enumerate the list of these three fruits.

## enumerate

- ▶ You can pass a second parameter to indicate where you want the counter to start at.

```
>>> for (key, val) in enumerate(["apple", "banana", "pineapple"], 1):  
...     print (str(key)+ ": " + val)  
...  
1: apple  
2: banana  
3: pineapple
```

If we wanted the count to start a different number than 0, say 1, we can pass that as the second parameter.



## enumerate

- ▶ Works for strings, too! (Even tuples)

```
>>> for (index, char) in enumerate("hello"):
...     print(index, char)
...
0 h
1 e
2 l
3 l
4 o
```

And as we discussed, all sequences are enumerable (including strings and even tuples). Here's an example with enumerating each character in a string.

## Practice Time

- ▶ Write a function `find_letter(string, letter)` which returns the index of the first occurrence of `letter` in `string` using the `enumerate` function.

```
>>> def find_letter(string, letter):  
...     for (index, char) in enumerate(string):  
...         if char == letter:  
...             return index  
...     return -1
```

Knowing. How `enumerate` works, we can write a more elegant version of `find_letter`.

## Resources

- ▶ Textbook: [Appendix \(Debugging\)](#)
- ▶ [scope.py](#)
- ▶ [scope2.py](#)
- ▶ [debugging.py](#)
- ▶ [debugging-with-prints.py](#)

## Practice Problems

- ▶ [Practice 5 \(solution\)](#)

## Homework

- ▶ Assignment 4