

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

Lecture 8: Scoping and debugging

- ▶ Scoping
- ▶ Debugging
- ▶ 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.

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.

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.

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.

Lecture 8: Scoping and debugging

- ▶ Scoping
- ▶ Debugging
- ▶ Strings

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
```


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

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
```

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.

Lecture 8: Scoping and debugging

- ▶ Scoping
- ▶ Debugging
- ▶ 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
```

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
```

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
```

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
```


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
```

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
```

Resources

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

Practice Problems

- ▶ [Practice 5 \(solution\)](#)

Homework

- ▶ Assignment 4