Lab 02: References, and Scope

CMPT 145

Laboratory 02 Overview

Section 1

Pre-Lab Reading

References in Python

- In this lab, we'll have another opportunity to think about references.
- Review Chapter 2 of the readings! Then do the Lab Activities starting Slide 18.

The concept of scope (review)

- When you write a program, you create named variables, functions, and parameters.
- In Python, all names are stored in frames, along with references to values.
- The rules concerning scope determine which parts of a program have access to any given variable.
- In Python, scope applies to anything you can name, including variables, parameters, and functions.

The scope of local variables

Local Scope

If a variable is created within a function, its accessibility is limited to that function. This rule applies also to all names.

- Variables defined inside a function are called local variables.

```
1 def a_function():
2   a_variable = 11
3   print(a_variable)
```

- Line 2 creates a variable accessible only inside the function.

Frames (review of Chapter 2)

- Every time a function is called, Python creates a frame for that function.
- The frame stores all variables created in that function, and all the parameters of the function as well.
  - These are the local variables.
  - Each variable in the frame has a reference to a value.
  - Values are stored as objects in the heap.
- When the function returns, Python removes the frame, and the local variables literally disappear.
  - There are exceptions, but not covered in CMPT 145.

Assignment statements (review)

- An assignment statement can create a new variable, or change an existing variable.
- This decision is based on context.

```
1 def a_function():
2   a_variable = 10
3   a_variable = 11
4   print(a_variable)
```

- Line 2: the variable is created.
- Line 3: the variable gets a new value.

The scope of global variables

Global scope

If a variable is created outside any function, it is visible to every function.

- These variables are stored in a global frame.
- The global frame is created when a script is started.
- The global frame is destroyed when a script is finished.
- A global variable is visible everywhere in the script.

```
1 a_variable = 10
2 def a_function():
3   print(a_variable)
```

Python prefers creating local variables

- Consider:

```
1 a_variable = 10
2 def a_function():
3   a_variable = 11
4   print(a_variable)
```

- Using Python's rules about names:
  - Line 1 creates a global variable
  - Line 3 creates a new local variable with the same name as the global variable.
  - Line 4 uses the local variable.
  - The global variable's value is unchanged by line 3.

Shadowing global variables

- From the previous example:

```
1 a_variable = 10
2 def a_function():
3   a_variable = 11
4   print(a_variable)
```

- We say that the new local variable shadows the global variable.
- The global variable cannot be seen because the local variable gets in the way.
- This behaviour means that by default, you cannot re-assign a global variable within a function.

Local Assignment Rule

Local Assignment Rule (LAR)

By default, Python creates a new local variable the first time its name is used on the left-side of an assignment statement within a function.

- This rule expresses Python's preference to create local variables.
- The default behaviour applies to assignment statements.
- The default behaviour can be defeated.

Global variables and mutable data types

- LAR applies to assignment statements only.
- Functions can affect mutable values of global variables.

```
1 a_list = [10]
2
3 def a_function():
4   a_list.append(11)
5
6 a_function()
7 print(a_list)
```

- This is not assignment, so LAR does not apply.
- The function modifies a mutable value through a global variable.

Global variables: Use and Misuse

- Acceptable: Global code modifying global variables.
  - A normal script is fine.
- Misuse: Modifying a global variable within a function.
  - Reduces robustness and adaptability and reusability.
  - A bug caused by misuse can be very difficult to find, and even more difficult to fix.

Global variables: Advice

Global variables

Do not modify global variables within functions.

- Python's Local Assignment Rule supports this advice.
- This advice is consistent with the best practices of Software Engineering for 40 years.

Global variables: handle with care

- Rarely, a limited use of global variables is warranted.
- You can defeat the Local Assignment Rule for a variable using the Python command `global`.

```
1 a_variable = 10
2
3 def a_function():
4   global a_variable
5   a_variable = a_variable + 1
```

- Because of line 4, line 5 changes the global variable.
- Using `global` will slow down your function noticeably.
- The bigger your program, the more you should resist using `global`.

Section 2

Laboratory Activities

References in Python

- On the following slides, you'll find a code snippet.
- For each snippet:
  1. Predict the console output.
  2. Draw the frame and heap diagram, showing all the variables and values in the snippet.
  3. Write a one sentence answer to the question posed with each example.
- You can test your understanding by running the code, provided as `references.py`

Activity: References 1

```
1 print('Example: Copying References')
2 x = [1, 2, 3]
3 y = x
4 print('before', x, y)
5 y[0] = 100
6 print('after ', x, y)
```

1. Predict the console output.
2. Draw the frame and heap diagram, showing all the variables and values in the snippet.
3. Question: Does the change to `y` on line 5 affect `x`? Explain why or why not.

Activity: References 2

```
1 print('Example: Copying Lists')
2 x = [1, 2, 3]
3 y = x[:]  # copies the list
4 print('before', x, y)
5 y[0] = 100
6 print('after ', x, y)
```

1. Predict the console output.
2. Draw the frame and heap diagram, showing all the variables and values in the snippet.
3. Question: Does the change to `y` on line 5 affect `x`? Explain why or why not.

Activity: References 3

```
1 print("Example: List expressions")
2 x = [1, 2, 3]
3 y = [4, 5, 6]
4 z = x + y
5 print('before', x, y, z)
6 x[0] = 100
7 y[0] = 999
8 print('after ', x, y, z)
```

1. Predict the console output.
2. Draw the frame and heap diagram, showing all the variables and values in the snippet.
3. Question: Do the changes to `x` and `y` on lines 6-7 affect `z`? Explain why or why not.

ACTIVITY: References 4

```
1  print('Example: multi-dimensional lists')
2
3  x = [[0, 1, 2], [5, 6, 7]]
4  y = [[10, 11, 12], [15, 16, 17]]
5  z = x + y
6  print('before', x, y, z)
7  x[0][0] = 999     # change first element of first sublist
8  y[0][0] = 100     # change first element of first sublist
9  print('after ', x, y, z)
```

1. Predict the console output.
2. Draw the frame and heap diagram, showing all the variables and values in the snippet.
3. Question: Do the changes to `x` and `y` on lines 7-8 affect `z`? Explain why or why not.

Activity: References 5

```
1 print('Example: list products')
2
3 x = [1, 2, 3] * 3
4 print('before', x)
5 x[0] = 10
6 print('after ', x)
```

1. Predict the console output.
2. Draw the frame and heap diagram, showing all the variables and values in the snippet.
3. Question: What does the change to `x` on line 5 do to `x`'s value? Explain.

Activity: References 6

```
1 print('Example: list products copy references')
2
3 x = [[0, 1, 2]] * 3
4 print('before', x)
5 x[0][0] = 10
6 print('after ', x)
```

1. Predict the console output.
2. Draw the frame and heap diagram, showing all the variables and values in the snippet.
3. Question: What does the change to `x` on line 5 do to `x`'s value? Explain.

Activity: References 7

```
1  print('Example: functions and parameters')
2
3  def change_params(a):
4      a = a - 1
5      return a
6
7  a = 5
8  b = 10
9  print('before', a, b)
10 b = change_params(a)
11 print('after ', a, b)
```

1. Predict the console output.
2. Draw the frame and heap diagram, showing all the variables and values in the snippet.
3. Question: How does line 10 affect the global variables `a` and `b`. Explain.

Activity: References 8

```
1  print('Example: functions and and immutable arguments')
2
3  def swap_ints(x, y):
4      print('inside swap_ints(), before', x, y)
5      tmp = x
6      x = y
7      y = tmp
8      print('inside swap_ints(), after ', x, y)
9
10 a = 3
11 b = 4
12 print('global scope, before', a, b)
13 swap_ints(a, b)
14 print('global scope, after', a, b)
```

1. Predict the console output.
2. Draw the frame and heap diagram, showing all the variables and values in the snippet.
3. Question: Did variables `a` and `b` get swapped? Explain.

Activity: References 9

```
1  print('Example: functions and and mutable arguments')
2
3  def swap_in_list(a_list, i, j):
4      tmp = a_list[1]
5      a_list[1] = a_list[2]
6      a_list[2] = tmp
7
8  some_list = ['a', 'list', 'of', 'words']
9  print('before', some_list)
10 swap_in_list(some_list, 1, 2)
11 print('after ', some_list)
```

1. Predict the console output.
2. Draw the frame and heap diagram, showing all the variables and values in the snippet.
3. Question: Did the function call on line 10 affect the list `some_list`?

Handin for References Activities

- Collect your one sentence answers, and place them in a file named `lab02-responses.txt`
- Example:

```
1  Activity References 1:
2  There is only one list, with two references to it
3  (variables x and y), and the assignment on line 5
4  changed this list.
5
6  Activity References 2:
7  Line 3 created a copy of the list, and the assignment
8  statement on line 5 changes the copy, but not the
9  original.
10
11 (etc)
```

- Hand-in the file `lab02-responses.txt`

Scope

Activity

- Download the files `scope.py` and `test_scope.py` from Lab02 on Moodle.
- Study the code in both files.
- Run the test script. Observe the errors! Don't fix them yet.
- Add at least 4 new test cases to the test script.
- Re-order your tests. You'll get different reports!
- Copy/paste the output of your test script, showing errors to the `lab02-transcript.txt` file.

Global variables in the module

- In the file `scope.py`, observe the global variable `duplicates` defined on line 25.
- The function `find_duplicates()` modifies this global variable (line 19).
- On any single test, `find_duplicates()` will get the right answer.
- Used multiple times, `find_duplicates()` will be incorrect.

Shadowing a global variable

Activity

- Define a local variable named `duplicates` inside the function `find_duplicates()`.
- Do not delete the global variable yet.
- Re-run the tests. The errors should be gone!
- The local variable `duplicates` shadows the global variable of the same name.
- Copy/paste the output of your test script, showing no errors to the `lab02-transcript.txt` file.
- Hand in your test script too.

Section 3

Hand In

What To Hand In

Hand in the following files:

- A file `lab02-transcript.txt` showing console output before and after testing and fixing `scope.py` (Slide 31)
- A file `test_scope.py` showing at least 4 new tests.
- A file `lab02-responses.txt` giving your one-sentence responses to the references activities.