Lab 04: Modules and Scope

CMPT 145

Laboratory 04 Overview

Part 1 : Pre-Lab Reading (Slide 4)
Part 2 : Laboratory Activities (Slide 29)
Hand In : A transcript of your work (Slide 39)

Part I

Pre-Lab Reading

Scope

The concept of scope

- 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 define which names are visible from any part of your program.
- Scope is often described in terms of variables, but 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 visibility is limited to that function. This rule applies also to all names.

- Variables defined inside a function are called local variables.
- These are usable by the function while it is running.

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

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

Frames and Scope

- When a function is called, Python creates a frame.
- The frame stores all parameters and variables created in the function.
  - These are called local variables.
  - These are usable by the function while it is running.
  - The frame also stores references to the value for each variable.
  - The references refer to values stored in the heap.
- When the function returns, Python removes the frame, and the local variables literally disappear.

Assignment statements

- 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.

```
a_variable = 10
def a_function():
  print(a_variable)
```

Python has special rules that limit use of global variables.

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:

```python
a_variable = 10
def a_function():
  a_variable = 11
  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.

```python
a_list = [10]

def a_function():
  a_list.append(11)

a_function()
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 variable created on line 1.

Global variables: the cost

- A bug caused by misuse of global variables can be very difficult to find, and even more difficult to fix.
- Using the Python command `global` will slow down your function noticeably.
- Misuse of global variables will reduce robustness and adaptability and reusability.
- The bigger your program, the more you should resist using the Python command `global`.

Scripts vs. Modules

Scripts (recap)

Definition

A script is just a file containing some Python code.

- It can use functions defined in its own file
- It can import Python modules.
- Running a script (in PyCharm or on the command-line) accomplishes some work we want done.

Global Scope

Definition

The Python global scope is any code in a script outside any function.

- A script must have some code in the global scope.
- If it doesn't, the script does not do anything!

Script example

The following script has a function (lines 3-7), and then some code (lines 9-10) in the global scope.

```
 1  # count.py
 2
 3  def sum_to(x):
 4      total = 0
 5      for i in range(x+1):
 6          total += i
 7      return total
 8
 9  example = 100
10  print("Global code in count.py", sum_to(example))
```

Without lines 9-10, the script only defines a function and would do nothing else.

Example: Importing a script with global code

The following script imports the script `count.py`.

```python
import count as count

example = 50
print("Global code in count3.py", count.sum_to(example))
```

When this script runs, the global code in `count.py` runs first!

```
Global code in count.py 5050
Global code in count3.py 1275
```

Modules (recap)

- A module is also a script.
- It defines functions and other Python things.
- It may import other Python modules.
- We import a module to have access to its definitions.

We probably don't want the module to run global code.

Module example

The following module has a function (lines 3-7), but no code that runs in the global scope.

```
1 # count1.py
2
3 def sum_to(x):
4     total = 0
5     for i in range(x+1):
6         total += i
7     return total
8
9 #end of file
```

Preventing global code from executing

The following script has a function (lines 3-7), and then some code (lines 9-11) in an if statement.

```
 1  # count2.py
 2
 3  def sum_to(x):
 4      total = 0
 5      for i in range(x+1):
 6          total += i
 7      return total
 8
 9  if __name__ == ’__main__’:
10      example = 100
11      print("Global code in count2.py", sum_to(example))
```

Notes on the example

- The variable `__name__`:
  - Created by Python when a script is run.
  - A global variable!
  - Otherwise, it's just a normal Python variable.
- We can check its value, but we better not change it!
- It's value depends on how the script is used:
  - If the file is being run as a script, `__name__` has the value `'__main__'`
  - If the file is being imported as a module, `__name__` refers to the module's name as a string.

Example: Global code is not executed

The following script imports the script `count2.py`.

```python
import count2 as count

example = 50
print("Global code in count3.py", count.sum_to(example))
```

When this script runs, the global code in `count2.py` does not get executed.

```
Global code in count3.py 1275
```

Part II

Laboratory Activities

Scope

Scope

Activity

- Download the files `scope.py` and `test_scope.py` from Lab04 on Moodle.
- Study the code in both files.
- Run the test script. Observe the errors!
- Maybe add a few more tests to collect more evidence.
- Re-order your tests. You'll get different reports!
- Copy/paste the output of your test script, showing errors to the `lab04-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 `lab04-transcript.txt` file.

Advice

- The misuse of the global variable is an error that can be very hard to find.
- Faults caused by misuse of a global variable seem to change randomly.
- The larger the program, the harder to find (this example was too small to be hard)
- Keep all your global code together as much as possible.
- Scattering your global code between and around functions will cause you grief.

Scripts vs. Modules

Modules vs. Scripts

ACTIVITY:

1. Download the files: `runcount.py` and `count.py` from Lab04 on Moodle.
2. Make sure `runcount.py` runs!
3. Notice that `count.py` has no code that executes at the global level.

Running scripts

ACTIVITY:

1. Add one print statement

   ```
   print(’Global code in count’)
   ```

   to `count.py` after all the operations.
2. Run `count.py` as a script. You should see the print statement’s output.
3. Run `runcount.py` as a script. You should see `count.py`’s output.
4. Hand in the console output showing the console output described above.

Modules vs. Scripts

ACTIVITY:

1. Add the conditional to `count.py` after all the definitions:

```
if __name__ == ’__main__’:
    print(’Global code in count’)
```

2. Run `count.py` as a script. You should still see the print statement’s output.
3. Run `runcount.py` as a script. You should no longer see `count.py`’s output.
4. Hand in the console output showing the console output described above.

Part III

Hand In

What To Hand In

Hand in your `lab04-transcript.txt` file showing:

- Copy/paste from Scoping activities on Slides 30 and 32.
- The console output from the activity on Slides 36-37.