Lab 08: Modules, and Internal Functions

CMPT 145

Laboratory 08 Overview

Section 2

Pre-Lab Reading

Recap: Scope, Frames, and References

The scope of local variables

Local Scope in Python

If a variable is created within a function, its visibility is limited to that function. This rule applies to any kind of name.

- Variables created 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.
- Function parameters are always local variables too!

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 connects each name to a value on the heap, using a reference.
- When the function returns, Python removes the frame, and the names literally disappear.
    - If the frame stored the only reference to some value on the heap, when the frame disappears, so does the 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)
```

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, but it's usually a bad idea.

Global variables and mutable data types

- LAR applies to assignment statements only.

```
a_list = [10]

def a_function():
  a_list.append(11)

a_function()
print(a_list)
```

- The function modifies the mutable value by accessing it from a global variable.
- Still a terrible idea!

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`.

```
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
```

There's nothing special about this module. It contains a single function definition.

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!
- Its 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`.

```
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
```

Defining functions within functions

Defining functions within functions

- A Python function can define one or more functions internally.

```
def collatz(a):

    def coll_step(b):
        if b % 2 == 0:
            return b // 2
        else:
            return 3 * b + 1

    biggest = a
    while a > 1:
        a = coll_step(a)
        if a > biggest:
            biggest = a
    return biggest
```

Notes on the example

- The function `coll_step` is defined internally to `collatz`.
  - The internal function definition ends at Line 9.
  - Line 9 is the first line of code whose indentation aligns with the internal function's `def` on line 3.
  - `coll_step` is only visible inside `collatz`.
- We say that `collatz` encloses `coll_step`.
- This is another way to hide information.
- It is particularly useful when a function needs a helper function that no other function needs to use.

Internal definitions and scoping in Python

- Calling an internal function creates a frame like any other function.
- An internal function is local to the enclosing function.
    - No function can look from the outside into another function's scope.
- Internal functions can look outwards to access variables defined by enclosing functions.
    - Every frame has a static link that points back to the place where the function was created.
    - For an internal function, the static link points to the enclosing function.

Example

```
1  def split(alist, pivot):
2      ls, es, gs = [], [], []
3
4      def place(t, pivot):
5          if t == pivot:
6              es.append(t)
7          elif t < pivot:
8              ls.append(t)
9          else:
10             gs.append(t)
11
12     for x in alist:
13         place(x, pivot)
14     return ls, es, gs
```

The internal function `place()` has access to the variables `ls, es, gs` on line 2.

Notes on the example

- The internal function `place()` has access to the variables `ls, es, gs` on line 2.
- These three variables are external to `place()`, but internal to `split`.
- This looks like “global” variables, but is limited to a single function.
- We’ll be using this technique from time to time in the coming material.

Simplifying internal definitions

- Note that both functions have a parameter named `pivot`.
- The parameter `pivot` in `place()` shadows the parameter `pivot` in `split`
- The value of the `pivot` never changes; it's just information.
- Because internal definitions can see names in the enclosing frame, `place()` can also see the external parameter `pivot`.
- We can use Python's scoping rules to simplify the internal definition.

Example

```
1  def split(alist, pivot):
2      ls, es, gs = [], [], []
3  
4      def place(t):
5          if t == pivot:
6              es.append(t)
7          elif t < pivot:
8              ls.append(t)
9          else:
10             gs.append(t)
11 
12     for x in alist:
13         place(x)
14     return ls, es, gs
```

The internal function `place()` has access to the parameter `pivot` on line 1.

Example: factorial

- Here's another example:

```
def fact_helper(i, n, prod):
    if i == n:
        return i*prod
    else:
        return fact_helper(i+1, n, i*prod)

def factorial(n):
    if n == 0:
        return 1
    else:
        return fact_helper(1, n, 1)
```

- The helper function is so specialized that it should only be called by `factorial`.

Example: factorial

- We can move the helper function into the main function

```
def factorial(n):
    def fact_helper(i, n, prod):
        if i == n:
            return i*prod
        else:
            return fact_helper(i+1, n, i*prod)

    if n == 0:
        return 1
    else:
        return fact_helper(1, n, 1)
```

- Notice that the helper's parameter `n` shadows the parameter of the enclosing function.

Example: factorial

We can simplify the parameter list for the helper.

```
def factorial(n):
    def fact_helper(i, prod):
        if i == n:
            return i*prod
        else:
            return fact_helper(i+1,i*prod)

    if n == 0:
        return 1
    else:
        return fact_helper(1,1)
```

To `fact_helper`, `n` is like a global constant, but it's local to `factorial`.

Notes on the example

- The parameter `n` appears in the frame created when `factorial` is called.
- The frame for `fact_helper` has a static link to this frame, because `fact_helper` was created there.
- When `fact_helper` is called, it cannot find `n` in its own frame, but it can look into `factorial`'s frame to find it.
- To `fact_helper`, `n` is like a global constant, but it's local to `factorial`.

Revising the idea of scoping

- Not every programming language allows internal functions. Modern languages usually do.
- An internal function creates a frame like any other function.
- An internal function is local to the enclosing function.
- No function can look from the outside into another function's scope. Internal functions are no exception.
- Internal functions can look outwards to access variables defined by enclosing functions (or global variables).

Section 3

Laboratory Activities

Scripts vs. Modules

Modules vs. Scripts

ACTIVITY:

1. Download the files: `runcount.py` and `count.py` from Lab08 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. Copy/paste the console output showing the console output described above to your `lab08-responses.txt` file.

```
Activity: Modules and Scripts: Before
(your console output here)
```

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. Copy/paste the console output showing the console output described above to your `lab08-responses.txt` file.

```
Activity: Modules and Scripts: After
(your console output here)
```

Defining functions within functions

ACTIVITY: Factorial

1. Open the script `fact.py` provided to you on Moodle.
2. Run the script, make sure it works.
3. Using the example from 30, move the helper function into `factorial`.
4. Simplify the parameters, as in the given example.
5. Make sure that the script still works!

This is for practice, you won't hand this in.

Review: Quick Sort

- In the file `qsort.py` you'll find Python code for a simple implementation of Quick Sort.
- Quick Sort is a divide and conquer algorithm for sorting.
- The key step is splitting a list into 3 sublists.
  1. All elements smaller than a given pivot
  2. All elements equal to the given pivot
  3. All elements larger than a given pivot.
- Thus divided, a recursive call can sort the smaller and larger elements.

ACTIVITY: Quick Sort

The key step is given as a helper function named `split`.

1. Run the script, to be sure it works!
2. Move the helper function into the quick sort function.
3. Simplify the parameter list for `split`, making use of your understanding of scope for internal functions.
    - Hint: what parameters don't change in `split`?
4. Make sure the script still works!

(You'll hand in the code for this activity. See Slide 43)

Section 4

Hand In

What To Hand In

- Your `lab08-responses.txt` file showing the console output from the activity on Slides 36-37.
- Your implementation of `qsort.py` from the ACTIVITY on Slide 41