

Lab 04

Basic Scripting With the Python Language

Goals

In lab 4 we will endeavor to learn some of the basics of using a high level scripting language in the Unix software development environment. You will build several scripts to perform small tasks using the Python scripting language.

Thus the goals of this lab are:

- 1) Learn the basics of the Python high level scripting language
- 2) Learn about using scripts to perform basic filtering and I/O tasks
- 3) Become more familiar with standard Unix filtering processes and commands.

Instructions

For this and all future labs, all of your work will be done and submitted through your own student repository on the nisl class server. If you have not done already, you need to check out a working copy of your personal repository before you can begin this lab.

Use the in-class lab time not only to find the answers to the questions and complete the given tasks, but to practice your skills in using the Python programming language, executing scripts and commands from the command line, and building small filters to accomplish tasks in the Unix programming environment.. The first portion of the lab is to be completed in class so that you may ask the instructor for help and suggestions for any problems you may be experiencing.

Exercise 4.0: Preliminaries

Check out a working copy of your students repository if you haven't already done so. Your student repository will have a url designation of the form:

<http://nisl.tamu-commerce.edu/repo/csci553/username>

Where you need to substitute your own nisl account username for the last portion above.

In your working copy, create and add to your repository a directory called *lab04* (make sure you name it exactly as shown, make sure you not only create the directory, but you run the appropriate command to add the directory to be under version control). All work for this lab is to be added and checked into the *lab04* directory of your repository.

Exercise 4.1: Hello World a la Python

Create a basic hello world python script. You should do the following:

- Create a file called `hello.py`, add it to your repository.
- Use the `#!` method we described in class to specify that the file should be interpreted using the python interpreter.
- Make the script executable.
 - Also make sure you do an appropriate action so that your python scripting commands can be found in your path. I would suggest for the duration of this lab that you add the current working directory `'.'` to the end of your path.
- Have the script print out something, like "Hello World!" or whatever you wish.
- When your script can be successfully executed from the command line, commit your new file to the repository.

Exercise 4.2: Building a Simple Filter

There are many methods for building a simple filter in Python, to read lines from standard input and write results to standard output. We will use the fileinput python library to create and perform simple filtering tasks.

Create and add a file called simplefilter.py to your lab04 working directory. For this step and for all future steps, you should make the script interpretable by the python interpreter and executable from the command line by performing the tasks of part 4.1 on your script file.

Add the following lines to your simplefilter.py script:

```
import fileinput

for line in fileinput.input():
    print "Got line: <%s>" % line
```

The fileinput library is a set of routines for doing exactly this very common task of reading lines of text from standard input so that they can be processed and filtered. The import keyword is how you tell a python script that you want to use commands or functions from a given library (look at the library reference manual on the python web site: <http://docs.python.org/lib/lib.html> for documentation on the fileinput and all other python libraries).

You can run your simplefilter.py script interactively to see how it works:

```
[harry@nisl lab04]$ simplefilter.py
Hello world
This is my second line
yet another line
Got line: <Hello world
>
Got line: <This is my second line
>
Got line: <yet another line
>
[harry@nisl lab04]$
```

In the above example I typed in the first 3 lines (in italics). I then typed in a Ctrl-d. After entering Ctrl-d, the text will be sent to the python script, and you will get the results shown where the text of the 3 lines that were typed in are displayed to standard output. You may need to enter a Ctrl-d on a line by itself in order to exit from running the script interactively.

Here is another way that you can use a pipe to test your simple filter. Create a file and type in some lines of text. Then use the cat command and a pipe to send the input into your filter:

```
[harry@nisl lab04]$ vi junk
... I edited the file called junk here and typed in a few lines of text ...
[harry@nisl lab04]$ cat junk
First line
another line
This is my last line
[harry@nisl lab04]$ cat junk | simplefilter.py
Got line: <First line
>
Got line: <another line
>
Got line: <This is my last line
>
```

Exercise 4.2.1 Slices and Fixing the Output

Notice that in the previous example, when the script echo's the line back to standard output, that a newline is being printed (and thus the > character ends up on the next new line by itself). This is because the line returned by calling `fileinput.input()` in the script captures and returns the trailing newline character entered by the user after each line. We would like to clean up the output by stripping off that newline character. Use an array slice to remove the trailing newline character before it is printed out. One hint, you do not need to use something like `strlen` (as was used in the lab03 `filter.c` program) in order to determine the last character of an array. Recall that you can use an index like `-1` to specify a location of the array relative from the end.

Fix the output of your script to remove the newline character before the line is displayed. If you make this change correctly, then you should see something similar to the following if you rerun your script with some input:

```
[harry@nisl lab04]$ cat junk | simplefilter.py
Got line: <First line>
Got line: <another line>
Got line: <This is my last line>
```

E.g. notice that the newline has been removed when echoing the lines to standard output. When you are satisfied that you have striped the newline correctly, commit your changes so far of the `simplefilter.py` script to the repository.

Exercise 4.2.2 Conditional Statements

Your next task is to modify the `simplefilter.py` filter to actually do some filtering. Change the logic in your filter loop to perform the following task. If the line is equal to the string "Hello" then you should print out the response "Hi, nice to meet you". Any other line should elicit the response "Say the magic word". You will need to use a conditional statement to perform this bit of logic. An example of a correctly functioning program would look like this:

```
[harry@nisl lab04]$ simplefilter.py
Hi
How are you
Hello
Say the magic word
Say the magic word
Hi, nice to meet you
```

Once again this is an example of running the filter interactively. The first 3 lines were input by the user and saved in a buffer. When a `Ctrl-d` is hit, then the script is executed and you get the resulting responses to the 3 lines of entered text. Once you are satisfied with your work, commit your `simplefilter.py` script to the repository.

Exercise 4.2.3: Some Simple String Processing

In this exercise we will look at a few string processing commands that Python has available for use as built in functions. Strings types are actually encapsulated objects, as in the Java or other object oriented languages. Thus a variable that holds a string in Python has many simple functions available to it to perform manipulations of the string.

As an example, run an interactive python session to explore string data types a bit more:

```
[harry@nisl lab04]$ python
Python 2.4.4 (#1, Oct 23 2006, 13:58:00)
[GCC 4.1.1 20061011 (Red Hat 4.1.1-30)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
```

```
>>> s = "This is a String. This is only my String. Testing 1 2 3 4."
>>> print s
This is a String. This is only my String. Testing 1 2 3 4.

>>> dir(s)
['_add_', '__class__', '__contains__', '__delattr__', '__doc__', '__eq__', '__ge__',
'__getattr__', '__getitem__', '__getnewargs__', '__getslice__', '__gt__',
'__hash__', '__init__', '__le__', '__len__', '__lt__', '__mod__', '__mul__', '__ne__',
'__new__', '__reduce__', '__reduce_ex__', '__repr__', '__rmod__', '__rmul__',
'__setattr__', '__str__', 'capitalize', 'center', 'count', 'decode', 'encode',
'endswith', 'expandtabs', 'find', 'index', 'isalnum', 'isalpha', 'isdigit', 'islower',
'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'replace',
'rfind', 'rindex', 'rjust', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith',
'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']
```

The `dir()` command can be used in an interactive session to discover all of the functions and attributes that an object contains. Since a string variable is actually an object, there are methods that are available that can be used to manipulate the string. For example, continuing from the previous interactive session:

```
>>> s.lower()
'this is a string. this is only my string. testing 1 2 3 4.'
>>> s.upper()
'THIS IS A STRING. THIS IS ONLY MY STRING. TESTING 1 2 3 4.'
>>> s.swapcase()
'tHIS IS A sTRING. tHIS IS ONLY MY sTRING. tESTING 1 2 3 4.'
```

This is an example of invoking the `lower()` `upper()` and `swapcase()` methods on the string object `s`.

Modify your `simplefilter.py` script from exercise 4.2. Note that the line variable is actually a string, like the string `s` object in the previous example. Modify your `simplefilter.py` script so that it does a case insensitive search for the magic word. For example, once modified you should be able to run an interactive session that looks like the following:

```
[harry@nisl lab04]$ simplefilter.py
This is a line
Another line
Hello
hello
hElLo
Say the magic word
Say the magic word
Hi, nice to meet you
Hi, nice to meet you
Hi, nice to meet you
```

Once you are satisfied with your work, commit your `simplefilter.py` script to the repository.

Exercise 4.3: Functions and More Strings

In this part of the lab we will look at defining functions in a Python script. As you saw in the previous section of the lab, there are many built in methods that can be called in order to manipulate string objects. Unfortunately, reversing a string is not one of them. We could add a method to the basic Python string object to perform this, by extending the Python language. But this would be beyond the scope of this lab. So instead we will define a simple python function that will take a single string as a parameter, and will return a new string that is the result of reversing the character order of the characters in the input string.

Before we start, a few hints on how you might solve this task. Your function will need to access and manipulate the characters of a string. Strings in Python are sequences, just like lists, and characters may be accessed using list access syntax. For example, to access the individual characters of a string you could (the following was done from an interactive python session):

```
>>> orig = "Hi!"
>>> for char in orig:
...     print char
...
H
i
!
```

Notice that in the for loop, each char is accessed one by one, and printed out to standard output. The same task can be accomplished by using an explicit index variable in a loop:

```
>>> for i in range(0, len(orig)):
...     print orig[i]
...
H
i
!
```

This could especially useful to, for example, access the characters starting from the end and progressing to the beginning of the string:

```
>>> i = len(orig)-1
>>> while (i >= 0):
...     print orig[i]
...     i -= 1
...
!
i
H
```

One other example/hint that might be useful in implementing your reverse() function. Recall that strings are overloaded in Python so that you can use the + operator to concatenate strings together. So for example, you could append or prepend a character to a string doing things like:

```
>>> orig+'M'
'Hi!M'
>>> 'M'+orig
'Mhi!'
```

With those example in mind, you are to write a filter that will read lines from standard input, and write the reversed version of the line to standard output. Create an interpretable and executable python script file called reverse.py for this portion of the lab. Your file should look something like this:

```
#!/usr/bin/env python
import fileinput

#
# your reverse function defined here
#

for line in fileinput.input():
    print reverse(line)
```

Of course you need to provide your implementation of the reverse() function at the location indicated. As stated above, your reverse() function takes a string as input. It should returned the reversed string as the result of calling the function. Also, to make your output more readable, you should strip the newline from your line before calling the reverse function, as you did in part 4.2.1

Once your reverse function is working properly, you should be able to run it with a file or interactively, and get a result similar to this:

```
[harry@nisl lab04]$ cat junk
First line
another line
This is my last line
[harry@nisl lab04]$ cat junk | reverse.py
enil tsriF
enil rehtona
enil tsal ym si sihT
```

Once you are satisfied with your `reverse.py` script, commit it to your repository.

Exercise 4.4: Palindrome Filter in Python

In this part of the lab we will reimplement the palindrome filter from lab 3 as a Python script. I have created the code to do the command line option parsing for you. Your task will be to implement the logic in order to perform palindrome filtering, as well as the logic to support the various command line arguments.

Copy the Python script from `/home/csci553/classfiles/palindrome.py` to your `lab04` directory. Also go ahead and add and commit the file to your repository. You should perform and test each of the following tasks in order on the file, and after each one you should commit your changes to the repository.

4.4.1 Implement Basic Palindrome Test

The script provided does nothing yet except for parsing command line arguments. It simply echos all lines from standard input back to standard output. Your first task is to implement palindrome filtering. You need to test each line to see if it is equal to the reverse of the line. You should copy and use your reverse function from part 4.3 to accomplish this. Also remember that, as in lab03, your `reverse()` function will probably not work well unless you strip off the trailing newline from the lines read in from standard input. You will need to do this, as you learned in part 4.1, before calling your `reverse()` function and comparing the reversed string to the original line.

Remember to commit you code once you have your basic palindrome logic working.

4.4.2 Implement the `minPalindromeLength` / `singleton` Options

This is a simple option to implement. Add code to skip lines that are less than the minimum length that is being filtered for. Hint: loops in Python support `break` and `continue` keywords as in the C and Java programming languages.

Once again, commit your changes when you have the length test working.

4.4.3 Implement the `ignoreCase` Option

Also a relatively simple option to implement. Using a string method, such as `upper` or `lower`, have your palindrome filter ignore differences in case by converting the line and its reversed string to one or the other before testing for equality. Make sure that you don't break the functionality to not ignore case when this option is not provided.

Once again, commit your changes when you have the `ignoreCase` option working.

4.4.4 Implement the `invertMatch` Option

Implement the logic to invert the normal sense of matching, so that when the `-v` flag is given your filter finds non-palindromes. Commit your changes when you succeed implementing this option.

Lab 04 Extra Credit: Reimplement the wc Command as a Python Script

The following task can be undertaken for extra credit. The extra credit will be worth 15 extra points for undergraduate students, and 5 extra points for graduate students.

The task is to reimplement the `wc` command as a python script. Call your script `wc.py`. You should be able to use the `palindrome.py` as a base for your work. You need to support the `-l`, `-w` and `-c` options. In fact, asking for help from your `wc.py` filter should produce the following usage message:

```
[harry@nisl lab04]$ wc.py -h
Usage: wc.py [options]

options:
  -h, --help            display this help and exit
  -c, --bytes           print the byte counts
  -l, --lines           print the newline counts
  -w, --words           print the word counts
```

You will need to, of course, correctly specify and parse the options using the `optparse` Python library, as well as implement the logic to display the counts. Your implementation should give exactly the same results when filtering standard input as the `wc` command:

```
[harry@nisl lab04]$ cat hello.py | wc
   3   6  51
[harry@nisl lab04]$ cat hello.py | wc -w
6
[harry@nisl lab04]$ cat hello.py | wc.py
   3   6  51
[harry@nisl lab04]$ cat hello.py | wc.py -w
6
```

To get full credit you need to implement the logic and the indicated options, and add and commit your `wc.py` script to your `lab04` repository directory.