

LING115 Lecture Note

Session#2: More fun with shell commands

1. Introduction

So we learned a little bit about Linux/Unix: its file system and basic shell commands. In this session, we will learn how to use shell commands more powerfully.

2. Wildcards

We know how to do several things with a single file: we know how to copy it to another directory, delete it, move it, rename it, and so forth. What if we wanted to do these things with multiple files? Fortunately, we do not have to run the commands separately for each file: we use wildcards.

A wildcard is a special character/expression that represents a set of strings. For example, the character `*` represents zero or more characters. That is, we can use it to refer to any string. So if we wanted to remove every file in the current working directory, we would enter the following:

```
]# rm *
```

This is because the command `rm *` means remove all files whose name matches `*`, which means any string. Any file name matches any string, so every file in the directory will be deleted. Below is a table listing of useful wildcards.

Wildcard	Meaning
<code>*</code>	Zero or more characters
<code>?</code>	Any single character
<code>[abcde]</code>	One of the characters listed
<code>[a-e]</code>	One of the characters in the specified range
<code>[!abcde]</code>	Any single character not listed
<code>[!a-e]</code>	Any single character not in the specified range
<code>{word,anotherword}</code>	One of the words listed

We can use wildcards to represent a substring of a filename. For example, `ling1*` would mean any string that starts with `ling1` followed by zero or more characters. So it would mean strings such as `ling1`, `ling11`, `ling115`, `ling115corpuslinguistics`, etc.

2. Streams

What we do with corpora will involve passing data around different programs: a program will process input data and return the processed output, which in turn may be used as input to another program. In Linux, the data that flow into and out from a program are called *streams*. It helps to think of data as water, especially since we are using terms like redirection and pipe below.

2.1. Standard I/O

We saw several shell commands for which we specify input. The command `cat` was one example. I asked you to specify the name of the input file as its argument. We also know what the output of this command is: the output is a print-out of the contents of the input file. In this case, the input stream is coming from a file. What about the output stream? Where is it headed? We never told our operating system anything about this, but it directed the output stream to some place by default: the terminal screen.

By default, the Linux operating system assumes that the input stream comes from the keyboard and the output stream goes out to the terminal screen. These default streams are called *standard streams*: standard input (stdio) and standard output (stdout), respectively¹. The thing about programs like `cat` is that they let the users override the standard input by specifying the file name. When we do not specify the name of the input file, the program waits for the standard input: it waits for us to enter something for it to concatenate and print out.

2.2. Redirection

The cool thing about Linux is that we have the power to *redirect* the streams. For example, `cat` sends the output stream to the screen by default, but we can force it to send the output stream to a file. In other words, we can directly save the output of `cat` as a file. In order for this example to make sense, we must first understand what `cat` really does: it *concatenates* everything in the input stream, be it a single file or multiple files, and prints out everything as if it were from a single source. To see this, pick two `.txt` files, and run `cat` using the names of the two files as its arguments. Assuming the files are named `foo1.txt` and `foo2.txt`, enter the following and you should see that the contents of both files are printed out as if it were from a single file.

```
]# cat foo1.txt foo2.txt
```

Now suppose we wanted a single file named `foo_both.txt` which contained all the lines from both files. What should we do? The output of the program is already the concatenation of lines from both files. It's just that we want it to save the output as a file rather than have the output printed on the screen. That is, we want to *direct* the output to a file named `foo_both.txt` rather than to the terminal screen.

Redirection is specified primarily by using two symbols: `>` and `<`. You can think of these as pointing towards where the stream is headed. For example, to direct the output stream from `cat` to `foo_both.txt`, we would enter the following:

```
]# cat foo1.txt foo2.txt > foo_both.txt
```

To illustrate redirection of input, let's bury our heads in the sand and assume that `cat` only takes in standard input: what we type in with the keyboard. If we wanted it to take in input from a file called `foo_in.txt`, we would enter the following:

¹ Actually, there is another kind of stream which consists of error messages from the program should it go wrong. But I will limit our discussion to input and output streams in this class.

```
]# cat < foo_in.txt
```

What is even better about redirection is that we can redirect both the standard input and the standard output in a single command as in the following example:

```
]# cat < foo_in.txt > foo_out.txt
```

That is, we let `cat` take in input from `foo_in.txt` and have its output saved as a file called `foo_out.txt`.

2.3. Piping

When we redirect the standard output to a file, for example, that is where the stream stops flowing. In order to use the file as input stream to another program, we will have to do it in the next prompt. Well, we can continue to channel the stream by piping. Imagine a number of wells connected with pipes. Water will continue to flow from one well to another as long as the two are connected. Piping in Linux is quite the same: programs are wells. Specifically, we pipe the standard output from one program to another program so that the stream now becomes the input stream as in the following example:

```
]# cat foo1.txt foo2.txt | grep -o -P '\w+'
```

We are actually running two programs in this example: `cat` and `grep`.

The command `grep` is short for global regular expression print: search the input stream globally for lines matching the regular expression and print them out. A regular expression is an expression that represents a set of strings. In that sense, it is very similar to wildcards we saw previously. In the above example, the input regular expression is `\w+`, which means any one word. There are several styles for specifying a regular expression that differ slightly from one another. In this class, we will learn Perl-style regular expression. The option `-P` in the example tells `grep` that we are using Perl-style regular expression when we say `\w+`. Finally, the option `-o` means print out the part of the line that matches the regular expression rather than printing out the whole line that contains the matching substring. In this example, it means print out each word, instead of printing out all lines that contain a word. We will learn `grep` and regular expression in greater detail later.

Anyway, in the above example, (1) the two input files are first concatenated, (2) the output stream from `cat` becomes the input stream of `grep` via piping, and (3) `grep` prints out all the words contained in the two files.

We can use piping multiple times as in the following example:

```
]# cat foo1.txt foo2.txt | grep -o -P '\w+' | sort | uniq
```

The command `sort` does exactly what its name suggests: it sorts the lines in the input stream. The command `uniq` is short for unique. It removes redundant lines in the input stream and prints out only the unique lines. The thing to keep in mind about `uniq` is that it will work properly only if the input stream

is already sorted. It should be obvious that the above prints out all the unique words from the two files `foo1.txt` and `foo2.txt`.

We can also use piping in conjunction with redirection as in the following example:

```
]# cat foo1.txt foo2.txt | grep -o -P '\w+' | sort | uniq > words.txt
```

What this example does should be obvious: it saves the list of unique words found in `foo1.txt` and `foo2.txt` as a file named `words.txt`.

3. Counting words

In the example we just saw, we essentially created a list of words that appear in a set of `.txt` files. While we are at it, let's learn a bit more `uniq` and `sort`, and a new shell command called `tr` so that we can create a word frequency list.

3.1. `uniq`

Again, `uniq` removes repeated lines in the input stream. More precisely, it discards all but one of *successive* lines from input. In a typical corpus, a word can appear anywhere and therefore the list of words that we initially print out with `grep` will contain multiple instances of each word scattered within the list. This is why we want to first sort the list before we use `uniq` to list each word only once.

By specifying `-c` as option, we can also use `uniq` to count how often each word appears in the input stream. Let's learn to do this by example. Look under `/home/ling115/sample_data/` and you will see `leaves_of_grass_gutenberg.txt`. This is an e-book copy of *Leaves of Grass* by Walt Whitman in plain-text format. There is `leaves_of_grass_gutenberg.case.tokens` in the same directory. This file contains the list of all word *tokens* that appear in the e-book: all instances of each word with case marking intact. Let's assume for now that case matters and print out the *token frequency* of each word: how often each word appears in the running text. This is what we do:

```
]# cat leaves_of_grass_gutenberg.case.tokens | sort | uniq -c
```

We can pipe the output to `less` or redirect it to a file if we to examine it closely. Part of our output should look something like the following, where the number before each word specifies the word's token frequency.

```
...
50 yourself
 2 Yourself
 5 yourselves
...
```

3.2. sort

Suppose we wanted to sort the word frequency list in descending order of token frequency: the most frequently appearing word appears at the top. Since the output from `uniq` comes with token frequency beginning each line, we can perhaps do this by first sorting the output and reversing it. That is,

```
]# cat ... | sort | uniq -c | sort -r
```

I omitted the file name in `cat ...` because I ran out of space. Reverse sorting the output from `uniq` is done by running `sort` with the option `-r`, short for reverse. But if you take a closer look at the output, you will realize that this didn't quite work. For example, you should see something like the following when you pipe the output to `less` and scroll down:

```
...
  9 accepted
 99 dead
 98 think
...
```

In brief, the problem here is that `sort` is not treating the token frequencies as numbers. We ask `sort` to treat a sequence of digits as numbers by adding `-n`, short for numeric sort. So the following is what we should enter to get a word frequency list sorted in descending order of token frequency:

```
]# cat ... | sort | uniq -c | sort -rn
```

Again, I omitted the file name in `cat ...` because I ran out of space.

3.3. tr

We often want to ignore case distinction when we count words. For example, we do not want to treat `different` and `Different` as two different words. So let's learn to compile a word frequency list with case distinction ignored using `tr`, short for translate.

Basically, we lower case every word in the output from `grep` before we pipe the stream to `sort`. This is where `tr` comes in. One way to use the command is to provide two sets of characters as its arguments. It then translates each character in the first set into the corresponding character in the second set. The character-to-character correspondence is implied by the position of each character in their respective set. For example, enter the following and type in `cow`.

```
]# tr cw mo
```

You should see it rewrite `cow` as `moO`. This is because the above command translates `c` into `m` and `w` into `o`. You are also asked to provide type in the input because `tr` assumes the standard input as its input. Hit `Ctrl + d` to return to command prompt.

So to lower case everything, we can simply run `tr` with its first argument listing all upper-case letters of the English alphabet and its second argument listing all lower-case letters of the English alphabet. We can

either list all letters of the English alphabet by specifying them from a to z one by one, or we can specify them as a range of characters: a-z for lower-case letters and A-Z for upper-case letters. The following should lower case everything in the standard input:

```
]# tr A-Z a-z
```

Since `tr` expects standard input, you must use `<` to redirect a file as its input stream.

3.4. Summary

So to wrap up, the following should create a word frequency list sorted in descending order of token frequency based on Leaves of Grass by Walt Whitman.

```
]# cat /home/ling115/sample_data/leaves_of_grass_gutenberg.txt | grep  
-o -P '\w+' | tr A-Z a-z | sort | uniq -c | sort -rn
```