Basic Unix - Part II

Jean-Yves Sgro

Updated: December 7, 2016

Contents

1 Pipes and Filters **1**
1.1 Wild cards . . . 2
1.2 Sorting . . . 3

2 Redirecting input **4**

3 Nelle's Pipeline **4**
3.1 Starting point: North Pacific Gyre . . . 4
3.2 Checking gyre data files . . . 5

4 Pattern recognition: `grep` **6**
4.1 Simple text pattern . . . 6
4.2 Regular expressions . . . 7
4.3 Which files contain a given pattern? . . . 7

5 Streaming editor: `sed` **8**

6 Editing standard output: `cut` **8**

7 Character translator: `tr` **9**
7.1 DNA into RNA . . . 10

8 Loops **11**
8.1 Variables . . . 12
8.2 For loop . . . 14

9 Nelle's Pipeline: Processing Files **17**
9.1 Nested loops . . . 17

10 Scripts **18**
10.1 ADVANCED: Additional info . . . 18

11 Summary **19**
11.1 Concepts . . . 19
11.2 Variables . . . 19
11.3 Symbols . . . 19
11.4 New commands learned: . . . 20

1 Pipes and Filters

This section reflects content from software carpentry tutorial page http://swcarpentry.github.io/shell-novice/04-pipefilter/

We will practise the commands and concepts we learned and look at the shell's most powerful feature: the ease with which it lets us *combine existing programs in new ways.*

Two main ingredients make Unix (Linux/MacOS) powerful (quoting from the tutorial):

Instead of creating enormous programs that try to do many different things, Unix programmers focus on creating lots of simple tools that each do one job well, and that work well with each other. This programming model is called "pipes and filters".

Little programs transform *a stream of input into a stream of output. Almost all of the standard Unix tools can work this way: unless told to do otherwise, they read from standard input, do something with what they've read, and write to standard output.*

The key is that any program that reads lines of text from standard input and writes lines of text to standard output can be combined with every other program that behaves this way as well. This is the power of piping.

1.1 Wild cards

We start by looking into the `molecules` directory containing six Protein Data Bank (`.pdb`) files, a plain text format that specifies the type and position of each atom in the molecule, derived by X-ray crystallography or NMR.

```
ls -C molecules
```

```
cubane.pdb  methane.pdb octane.pdb  propane.pdb
ethane.pdb  middle.sh   pentane.pdb
```

All files end with the `.pdb` filename extension. What if we wanted only files that start with the letter `p` and we did not want to specify what the rest of the file name was. In that case we would use of a new symbol: `*` called the `wild card` which is meant to match zero or more characters in a command. Therefore `p*` would represent all files that start with `p` no matter what comes after:

```
ls molecules/p*
```

```
molecules/pentane.pdb
molecules/propane.pdb
```

We could also call all the files that end with `.pdb` wtih `*.pdb`.

The wild card can replace an number of characters. On the other hand the symbol `?` represents one character and more than one `?` can be used to specify *excatly* how many characters should match.

Exercise: Try the following commands:

```
ls ?thane.pdb
ls ??thane.pdb
```

We will now use the word count commannd `wc` we learned previously to count the number of lines. But since we are only interested in the number of lines we'll use `wc -l` as we have seen before.

The command sends the results to the screen display (standard ouptut), but we know how to "capture" this information and send the results to a file instead thanks to redirection with `>` into a new file we call `lengths.txt`. We then move back into the `data-shell` directory. Since `data-shell` contains `molecules` it is the "parent" and therefore can be represented symbolically with dot-dot: `..`.

```
cd molecules
wc -l *.pdb
wc -l *.pdb > lengths.txt
cd ..
```

```
  20 cubane.pdb
  12 ethane.pdb
   9 methane.pdb
  30 octane.pdb
```

```
 21 pentane.pdb
 15 propane.pdb
107 total
```

We should now be within `data-shell`. We are going to transfer (move) file `length.txt` which is inside the `molecules` directoy and bring it into the current directory or `.`

```
mv molecules/lengths.txt .
```

We can now use it locally without affecting the content of the `molecules` directory.

1.2 Sorting

We created file `lengths.txt` and we know how to see it's content with `cat`.

```
cat lengths.txt
```

This reflects exactly what we saw on the display earlier and lines are printed in the same order. The first column contains the length (number of lines) for each file.

1.2.1 Numerical

Introducing `sort` that can "*sort lines of text files*" according to the `man` pages.

For now our interest is to sort by numerical values of the first column. To ensure that the sorting is numerical we add `-n`. The `lengths.txt` file will not be modified and the sorted results will be sent to the screen display (standard output.)

```
sort -n lengths.txt
```

```
  9 methane.pdb
 12 ethane.pdb
 15 propane.pdb
 20 cubane.pdb
 21 pentane.pdb
 30 octane.pdb
107 total
```

The order is now changed and reflects increasing numerical values from top to bottom.

We can also save these results into a new file:

```
sort -n lengths.txt > num_sorted-lengths.txt
```

1.2.2 Alphabetical (on chosen column)

The first column (also called "field") is the default column onto which `sort` will operate.

What if we wanted to sort by file name instead, for example to make the list alphabetical?

The first problem is that the name of the files are in the **second** column. Therefore we need to find a way to tell `sort` that's what we want to use.

Inspecting the `man` pages we can discover that:

```
man sort
```

```
-k, --key=POS1[,POS2]
              start a key at POS1, end it at POS2 (origin 1)
```

Therefore we can modify our sort command and make the sorting alphabetical which is the default. We only need to tell `sort` to look into the second column. We can write this as `-k 2` or alternatively with `--key=2`

```
sort -k 2 lengths.txt
```

```
  20 cubane.pdb
  12 ethane.pdb
   9 methane.pdb
  30 octane.pdb
  21 pentane.pdb
  15 propane.pdb
 107 total
```

2 Redirecting input

We already learned about redirecting standard output into a new file with `>` or appending the redirected output into an existing file with `>>`.

In the same way there is a command to redirect standard input. We already know how to use the output of one program to serve as input of the next program via piping, for example as in `ls | wc`.

But what if the output is already contained within a file?

We already used that in a command line `cat animals.txt  | wc`.

However there is another way to do that with the symbol `<`. It is the reverse symbol of redirecting *into* a file. Now we are redirecting *from* the file as demonstrated here:

```
wc < lengths.txt
```

```
       7      14     138
```

In this way we (re)directed the content of the file `lengths.txt` ***as*** standard input into command `wc`.

3 Nelle's Pipeline[1]

We can now look more closing at the data the software carprentry tutorial

3.1 Starting point: North Pacific Gyre

Nelle Nemo, a marine biologist, has just returned from a six-month survey of the North Pacific Gyre[2], where she has been sampling gelatinous marine life in the Great Pacific Garbage Patch[3]. She has 300 samples in all, and now needs to:

1. Run each sample through an assay machine that will measure the relative abundance of 300 different proteins. The machine's output for a single sample is a file with one line for each protein.
2. Calculate statistics for each of the proteins separately using a program her supervisor wrote called `goostat`.
3. Compare the statistics for each protein with corresponding statistics for each other protein using a program one of the other graduate students wrote called `goodiff`.

[1] http://swcarpentry.github.io/shell-novice/01-intro/
[2] http://en.wikipedia.org/wiki/North_Pacific_Gyre
[3] http://en.wikipedia.org/wiki/Great_Pacific_Garbage_Patch

4. Write up results. Her supervisor would really like her to do this by the end of the month so that her paper can appear in an upcoming special issue of Aquatic Goo Letters.

It takes about half an hour for the assay machine to process each sample. The good news is that it only takes two minutes to set each one up. Since her lab has eight assay machines that she can use in parallel, this step will "only" take about two weeks.

The bad news is that if she has to run `goostat` and `goodiff` by hand, she'll have to enter filenames and click "OK" 45,150 times (300 runs of `goostat`, plus 300x299/2 runs of `goodiff`). At 30 seconds each, that will take more than two weeks. Not only would she miss her paper deadline, the chances of her typing all of those commands right are practically zero.

3.2 Checking gyre data files

Nelle has run her samples through the assay machines and created 1520 files in the `north-pacific-gyre/2012-07-03` directory. As a quick sanity check, starting from her home directory, Nelle types:

```
cd north-pacific-gyre/2012-07-03
wc -l *.txt
```

```
300 NENE01729A.txt
300 NENE01729B.txt
300 NENE01736A.txt
300 NENE01751A.txt
300 NENE01751B.txt
300 NENE01812A.txt
... ...
```

Now she types this:

```
wc -l *.txt | sort -n | head -n 3
```

```
 240 NENE02018B.txt
 300 NENE01729A.txt
 300 NENE01729B.txt
```

It seems that one of the file has only `240` lines rather than `300`. So it's `60` lines shorter! When she goes back and checks it, she sees that she did that assay at

8:00 on a Monday morning — someone was probably in using the machine on the weekend, and she forgot to reset it. Before re-running that sample, she checks to see if any files have too much data:

```
wc -l *.txt | sort -n | tail -n 5
```

```
 300 NENE02040B.txt
 300 NENE02040Z.txt
 300 NENE02043A.txt
 300 NENE02043B.txt
5082 total
```

So it all looks OK, there are not files with more than `300` lines. But the second file from top has a `Z` in its name while her samples should only be marked `A` or `B`. By convention in her lab files marked `Z` indicate samples with missing information.

Are there any other files with `Z` in the list?

```
ls *Z.txt
```

```
NENE01971Z.txt    NENE02040Z.txt
```

Note: The `ls` command only looks for files that have a `Z` before `.txt` which makes the command more specific than say `ls *Z*` for example.

Sure enough, when she checks the log on her laptop, there's no depth recorded for either of those samples. Since it's too late to get the information any other way, she must exclude those two files from her analysis. She could just delete them using `rm`, but there are actually some analyses she might do later where depth doesn't matter, so instead, she'll just be careful later on to select files using the wildcard expression `*[AB].txt`. As always, the `*` matches any number of characters; the expression `[AB]` matches either an `'A'` or a `'B'`, so this matches all the valid data files she has. We will explore these expression shortly.

4 Pattern recognition: `grep`

Pattern finding can be simple or complex. We already used pattern matching and finding when listing files with wild card `*` or `?` or `[AB]`. This is a vast subject but knowing a few commands is useful in daily life.

Most of the time pattern matching is useful to find information when files are large (contains thousands of lines) or there is a large number files (files have few lines but there are thousands of files.)

For these exercises we'll use the files at hand, but the power scales up to large numbers of files and/or data size.

The general shell program for pattern recognition is `grep` defined in the `man` pages as *file pattern searcher*.

```
DESCRIPTION
     The grep utility searches any given input files, selecting lines that
     match one or more patterns.
```

4.1 Simple text pattern

Here are a few simple examples:

Select line(s) containing patern `Ser` inside file listing amino acids:

```
grep Ser data/amino-acids.txt
```

```
Serine      Ser
```

The pattern is case-sensitive so the command `grep ser data/amino-acids.txt` would yield no results. Of course, like most commands we can add a flag (`-i`) to make the search case insensitive:

In that case the command would work:

```
grep -i ser data/amino-acids.txt
```

```
Serine      Ser
```

Of course all the power of Unix is at our fingertips and we can "pipe" commands as well.

One useful thing to do sometimes is to find a pattern but **remove** the corresponding entries. In the next example we'll first look for pattern `glu` making it case insensitive.

```
grep -i glu data/amino-acids.txt
```

```
Glutamic acid   Glu
Glutamine   Gln
```

Then we'll want to remove the entries containing the word `acid`. To remove a pattern we use the flag `-v`.

```
grep -i glu data/amino-acids.txt | grep -v acid
```

```
Glutamine   Gln
```

The word need not be complete (*e.g.* just `tamic acid`.) And if there are blank spaces within the pattern then the pattern needs to be placed within quotes:

```
grep -i glu data/amino-acids.txt | grep -v "tamic acid"
```

```
Glutamine   Gln
```

4.2 Regular expressions

The command `grep` owes its name to different possible acronyms[4]:

- "**g**lobal **r**egular **e**xpression **p**rint"
- "**g**lobal **r**egular **e**xpression **p**arser"
- "**g**et **r**egular **e**xpression **p**rocessing"

All of them refer to "regular expression" sometimes also called `regexp` or `regex`.

A regular expression constitutes a language to represent patterns, from simple to very complex.

One regular expression used above was `[AB]` used in `*[AB].txt` meaning match any file that contains the letters `A` or `B`.

Let's try this method to search the amino acids file. We know that the first letter is uppercase so the following command will list lines of amino acids that contain either `I` or `L`:

```
grep [IL] data/amino-acids.txt
```

```
Isoleucine  Ile
Leucine     Leu
Lysine      Lys
```

In that case `[IL]` is a regular expression as the pattern only needs to match one of the letters, not both. Contrast this with the following command. If we remove the brackets then it is a simple two letters text pattern and there will be no match:

```
grep IL data/amino-acids.txt
grep -i IL data/amino-acids.txt
```

The pattern can be constructed with more than just letters.

> Exercise: what is the meaning of the command:

```
ls -F | grep -v '[/@=|]'
```

> What is it's output? ____

4.3 Which files contain a given pattern?

We can also ask `grep` to check all files for a specific pattern. This will generate a list for each line of each file that contains that pattern.

The directory `creatures` contains files with DNA sequence for two fantastic creatures: `basilisk` and `unicorn`. Do any one of the contain the pattern `ACGT` exactly?

[4] http://www.acronymfinder.com/GREP.html

```
grep ACGT creatures/*
```

```
creatures/basilisk.dat:ACGTGGTCTA
creatures/unicorn.dat:ACGTGGTCTA
creatures/unicorn.dat:ACGTGGTCTA
```

We can note that the output is the file name (perhaps with a partial `path`) followed by a colon (`:`) and the line that matched the pattern. There is no space between the file name and the matched line, only the colon. Below we'll see a useful method to separate the two entities.

5 Streaming editor: `sed`

As we now know and understand, the standard output stream can go directly to the screen or be "captured" into a file or "piped" as standard input to another program.

If all goes well, the output of program one becomes the perfect output of program two. However, sometimes a little "tweaking" of the data can be useful. Enters `sed` the **s**tream **ed**itor.

`sed` can use regular expressions and is very powerful. We'll see brief example of substitution to illustrate the streaming dynamic.

We noted that the the grep results were the file name, a colon, and the file matching the pattern. Let's use `sed` to "separate" the 2 items by simply replacing the colon `:` with a blank space ' '.

To do this we'll use the `s` substitute command of `sed` that declared a pattern to find on each line (the colon), and what it should be substituted with (a white space) done globally on all lines (trailing `g` in the command). All has to be encapsulated within quotes:

```
grep ACGT creatures/* | sed 's/:/ /g'
```

```
creatures/basilisk.dat ACGTGGTCTA
creatures/unicorn.dat ACGTGGTCTA
creatures/unicorn.dat ACGTGGTCTA
```

No big deal we only have 3 lines and it would have been easy to do that by hand...

We could also use `sed` to remove the name of the `creatures` folder wihin the path for a cleaner output. However, since there is a `/` within the path name and we also use `/` to separate the various parts of the command we need to "escape" the forward slash `/` of the pathname with a backslash `\`:

```
grep ACGT creatures/* | sed 's/creatures\// /g'
```

```
 basilisk.dat:ACGTGGTCTA
 unicorn.dat:ACGTGGTCTA
 unicorn.dat:ACGTGGTCTA
```

6 Editing standard output: `cut`

There are other methods to edit the streaming standard output. We just learned about `sed` but there are others. Here we'll quiclky learn about `cut` that can "slice" columns in a table.

We learned above that we need to "get rid of" the column of matched lines or we need to only grab the first column... This is the perfect job for `cut`. The `man` pages short description is: *cut out selected portions of each line of a file.*

We can use the fact that there is a `:` colon **delimiter** between the file name and the matched lines and tell `cut` to find and ***cut*** the stream at that delimiter into two columns (or fields) and we can ask to see only the first column (field.)

```
grep ACGT creatures/* |  cut -d : -f 1
```

```
creatures/basilisk.dat
creatures/unicorn.dat
creatures/unicorn.dat
```

We can also remove the name of the directory as before:

```
grep ACGT creatures/* |  cut -d : -f 1 | sed 's/creatures\// /g'
```

```
 basilisk.dat
 unicorn.dat
 unicorn.dat
```

Following with the previous example lets now change the four letters pattern `ACGT` to a shorter pattern `ACG` of three letters. We can expext to match more lines and we can count them:

```
grep ACG creatures/* | wc
```

```
      40      40    1388
```

What if there were 100s or 1000's matches?

For this we could use the sorting command `sort` that we already know and add the flag `u` to keep only *unique* entries.

```
grep ACGT creatures/* |  cut -d : -f 1 | sed 's/creatures\// /g' | sort -u
```

```
 basilisk.dat
 unicorn.dat
```

This is indeed a very sophisticated command line with 4 command and 3 pipe symbols!

Thanks to the ***streaming*** of the data as standard input and output we were able to ask each command to accomplish a small task and pass the results to the next task. There was no need for programming a special software to accomplish what we wanted.

Note: the command `uniq` can play the same role as `sort -u`. The command could be written again as:

```
grep ACGT creatures/* |  cut -d : -f 1 | sed 's/creatures\// /g' | uniq
```

```
 basilisk.dat
 unicorn.dat
```

> Exercise: If we make the pattern only `AC` how many lines do you get?
> Without unique sorting:___________________________
> With unique sorting:_______________________________

7 Character translator: `tr`

`tr` is a very useful utility that can help with tedious tasks, even with non visible characters

```
DESCRIPTION
     The tr utility copies the standard input to the standard output with substitu-
     tion or deletion of selected characters.
```

7.1 DNA into RNA

A simple example is to change DNA into RNA for example to have an RNA version of the `unicorn.dat` file.

But we have another problem: the file starts with a header of 3 lines. We may not care about that, but it would be nice to not change the words of the header.

```
head -5 creatures/unicorn.dat
```

```
COMMON NAME: unicorn
CLASSIFICATION: equus monoceros
UPDATED: 1738-11-24
AGCCGGGTCG
CTTTACCTTA
```

We can make a copy of the header into a temporary file which can be used later for example. Here we only get the first `3` lines and send them into a file:

```
head -5 creatures/unicorn.dat > u_header.txt
```

OK. We *copied* the first three lines into a separate file, BUT the header is still present in the original file. We'll need to find a way to remove them "on the fly" within the standard input/output stream, and for this we can use `sed` again. To remove just the first line the command would be `sed 1d` but to remove the first three line the command would be `sed 1,3d`.

We can test all this and pipe the result into the `head` program. We could make the command "linear" with the `cat` command so that all the commands would go from left to right:

```
cat creatures/unicorn.dat | sed 1,3d | head -5
```

However, we learned the input redirect `<` and that is more efficient to simply tell `sed` to get it from there. In this case it makes little difference, but on a large number of very large files it could make a difference of minutes, hours or even days!

The command would then be as follows, which in plain English can be read as: "Send the content of file `unicorn.dat` contained within the directory `creatures` to the program `sed` which will remove lines `1` through `3` and send the result to `head` for final display on the screen." (The "difficulty" here is that the command first goes right to left - sending to `sed` - and then goes left to right sending to `head`.)

```
sed 1,3d < creatures/unicorn.dat | head -5
```

```
AGCCGGGTCG
CTTTACCTTA
AAGCCGAGGG
GGGTGGTACG
CCGAACATAA
```

To continue, we now invoque `tr` to translate DNA into RNA by changing the `T` into `U`. Again, at the end we collect into `head` to not print the complete result:

```
sed 1,3d < creatures/unicorn.dat | tr 'T' 'U' | head -5
```

```
AGCCGGGUCG
CUUUACCUUA
AAGCCGAGGG
GGGUGGUACG
CCGAACAUAA
```

OK it works. But, as Mr Colombo (from the TV series) always asked "just one more question" we would now like to know the sequence length, not of one line but of the entire sequence (assuming it is a contiguous sequence such as an mRNA for example.)

Let's try on one line to see what we get:

```
sed 1,3d < creatures/unicorn.dat | tr 'T' 'U' | head -1 | wc
```

```
       1       1      11
```

We see that the length of "characters" (the third number) is in fact `11` and *not* `10` if we count the bases by hand!

Why is that ?

It's because each line contains a hidden "newline" character which can be written as `\n` and is counted in the process.

For one line or two it's OK, we can do some math in our head and remove `1` or `2` but what about 100s or 1000's. So, we call `tr` to the rescue and ask it to remove (delete) the newline character:

```
sed 1,3d < creatures/unicorn.dat | tr 'T' 'U' | head -1 | tr -d '\n' | wc
```

```
       0       1      10
```

We should now see the *exact* number of bases within that one line: `10`.

We also note that there are `0` lines as newline was never seen so counting never started…

We can now apply the command to the whole sequence: We remove the first 3 lines so they are not counted as sequence. Then we change `T` to `U` but that does not change numbers, then we delete new lines and finally count. So the command is the same as above, we just remove the `head` part:

```
sed 1,3d < creatures/unicorn.dat | tr 'T' 'U' |  tr -d '\n' | wc
```

```
       0       1    1600
```

> Exercise: what would be the incorrect number of bases if we did not remove newlines `\n`?

From this point we could reconstruct a complete file with the saved header in 2 steps:

1. rename the header to e.g. `unicorn_rna.dat`. Or we can recreate what we need from scratch!
2. append the streaming output of RNA at the end of that file with `>>`

Optionally we can then move the new RNA file into the `creatures` directory.

```
# Step 1: create a new file with just the first 3 lines of the DNA version:
head -3 creatures/unicorn.dat  > unicorn_rna.dat

# Step 2: we append  output stream to the file:
sed 1,3d < creatures/unicorn.dat | tr 'T' 'U' >> unicorn_rna.dat

# We move the file down one directory:
mv unicorn_rna.dat creatures
```

Note: The last command moves the file into a directory. The directory name could be written also as `./creatures` or even `./creatures/`.

8 Loops[5]

Loops are key to productivity improvements through automation as they allow us to execute commands repetitively.

[5] http://swcarpentry.github.io/shell-novice/05-loop/

8.1 Variables

Before we dive into loops we need to learn or review the notion of ***variables***.

A variable is *a symbol or name that stands for a value.* One way to think about it is that of a container: a glass can contain water, wine or juice. In that case `glass` would be the variable name and `wine`, `water`, and `juice` would be possible *values* for that variable.

In loops, we'll use a variable, a word or even just a letter, and we'll make it a variable by adding a `$` to its name.

Let's see a simple example: we want to get every file in a folder, list the file and check the first line.

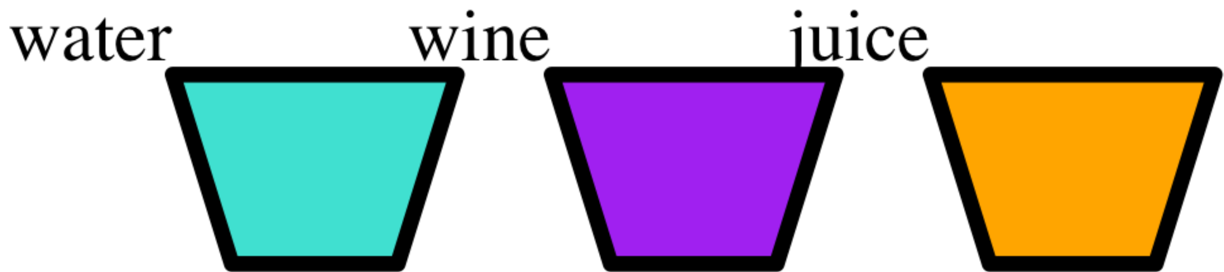

Figure 1: The glass content can vary. The glass is the variable, the content is the variable value.

Figure 1.

Continuing with this analogy:

If we write `glass` then it is the word `glass`.

If we glue a `$` to `glass` then `$glass` means the **content** or current **value** of the variable `glass` and depending on the situation it could be `wine`, `water`, or `juice` for example.

Let's try this: we will create a variable called `glass` and give it various values. We can use the command `echo` to see what the variable values are.

The command `echo` will just print back whatever it is given, ***except*** in the case of a variable precedeed by a `$`. Since we don't have a variable defined yet, we can just test the `echo` command:

```
echo glass
```

```
glass
```

First we define a variable `glass` and for this we need the command `export` to actually export that variable into the shell:

```
export glass=water
echo glass
echo $glass
```

The `echo` commands will create the following output:

```
glass
water
```

Note the difference afforded by adding the `$`.

We can illustrate this concept with the following figure, showing how the **container** (*glass*, the variable *name*) remains the same while the **content** (*water*, *wine*, *juice*, the variable *value*) changes and can be called by **`$glass`**.

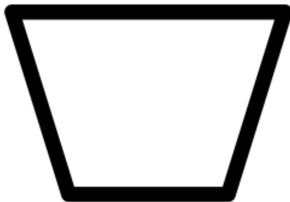

Figure 2: Separating the container and the content: ‘glass’ is the container ‘$glass’ is the content and can vary.

Figure 2.

> Exercise: give other values to `glass` with `export` and then check the result of `$glass` with `echo`. (Hint: multi-word values have to be in quotes, for example "cold water".)

8.2 For loop

A ***for loop*** simply takes a list of files and **does** something with them. We use a variable to designate the file names in succession.

We can ***do*** or apply any command to the file as it is named by the variable as we would if we took the files one by one and applied that command "by hand". To keep it simple let's just use `ls` as the *action* we want to **do** to the file.

Note: As soon as the shell sees the `for` command it will know it is a loop and the prompt will change from `$` to `>` until we are **done**.

8.2.1 Loop on files

Here we use the variable `f` simply because it is the first letter of the word *file* but we can use any letter or word we wish.

Therefore `f` will be the *glass* that will be filled successively with the *name of each file* in a list provided by `*.txt`. Thefore `$f` will represent the name of each file in the list until the end of the list.

Here the prompts are added for clarity. Note the change from`$` to `>` after the start of the `for` loop:

```
$ cd data
$ for f in *.txt
> do ls $f
> done
```

```
amino-acids.txt
animals.txt
morse.txt
planets.txt
salmon.txt
sunspot.txt
```

Note: if you use the `up` arrow to recover the previous command it will be shown as a single line for the `for` loop separated by semi-colons (`;`.)

```
for f in *.txt; do ls $f; done
```

OK, this very fancy loop does exactly the same as `ls *.txt` but the power comes from the fact that now, we can add more commands.

For example let's do this:

- call each `.txt` file
- print its name
- show the very first line
- until all files are done

We can even do that from a higher directory level if we wish!

Let's also change `f` to `file` just to show we can.

The commands are slightly indented for clarity (this time the prompt is not printed) but indentation is optional here.

```
for file in data/*.txt
do
 ls $file
 head -1 $file
done
```

```
data/amino-acids.txt
Alanine     Ala
data/animals.txt
2012-11-05,deer
data/morse.txt
0   -----
data/planets.txt
"Planet Name","Pl. Mass","Pl. Radius","Pl. Period",
data/salmon.txt
coho
data/sunspot.txt
(* Sunspot data collected by Robin McQuinn from *)
```

We could even modify the loop slightly to remove the trailing directory name remembering to "escape" the trailing slash after `data/` with a backslash. We can also add a blank line between each output for reading clarity:

```
for file in data/*.txt
do
 ls $file | sed 's/data\///g'
 head -1 $file
 echo "" # blank line
done
```

```
amino-acids.txt
Alanine     Ala

animals.txt
2012-11-05,deer

morse.txt
0   -----

planets.txt
"Planet Name","Pl. Mass","Pl. Radius","Pl. Period",

salmon.txt
coho

sunspot.txt
(* Sunspot data collected by Robin McQuinn from *)
```

> The word used after `for` to serve as *variable* can be any word we wish. So far we used `f` and `file` and we only need to add `$` when we want to extract the value of the variable.

In the same way, the *list* that follows `in` was provided by an `ls` command but an explicit list of file names, just separated by blank space would work too. For example if we only wanted to apply the loop to the 2 files `planets.txt` and `sunspot.txt` we would write

```
for file in data/planets.txt data/sunspot.txt
do
```

```
 ls $file | sed 's/data\///g'
 head -1 $file
 echo "" # blank line
done
```

```
planets.txt
"Planet Name","Pl. Mass","Pl. Radius","Pl. Period",

sunspot.txt
(* Sunspot data collected by Robin McQuinn from *)
```

8.2.2 Loop on number

The variable need not necessarily be a file name. Here is a simple example where the variable called `mynum` is simply a number

In the loop we use the command `expr` (*evaluate expression*) to add `100` to the variable number.

We add `" "` to create a blank space. All three `echo` commands are flagged with `-n` to prevent a new line and therefore all three commands within the loop are printed on a single line:

```
for mynum in 0 1 2 3
do
 echo -n $mynum " "
 echo -n   $nymum plus one hundred is equal to:
 expr $mynum + 100
done
```

```
0  plus one hundred is equal to:100
1  plus one hundred is equal to:101
2  plus one hundred is equal to:102
3  plus one hundred is equal to:103
```

8.2.3 Loop for backup

One example provided in the tutorial is to backup current versions of specific files. This example provides an illustration to the fact that the `$` marker attached to the variable name can be embedded within a new file name. For this to work properly we need to be within the `creatures` directory. To see but **not** execute the command we can ask `echo` to print to the screen what the loop would do:

```
cd creatures
for filename in *.dat
do
    echo cp $filename original-$filename
done
```

This loop would replace three `cp` commands but that depends on the number of files present ending with `.dat`.

To have the loop do the actual copying we would simply remove the word `echo` and only keep `cp`.

> Warning: next time this command is run we will have an `original-original` file and then next time an `original-original-original` file etc. Therefore this is not a very good loop!

In our caset there are only 2 or 3 files starting with either `b` or `u` so we could change the loop with a simple *regular expression* to take advantage of this fact: simply add `[bu]` in the file selector:

```
cd creatures
for filename in [bu]*.dat
do
    echo cp $filename original-$filename
done
```

9 Nelle's Pipeline: Processing Files[6]

Here is a brief summary of Nelle's pipeline. You can read the complete story on the referenced software carpentry web page.

First she goes to where some of the data files are:

```
cd north-pacific-gyre/2012-07-03
```

Depending where you are it might be best to use: `cd ~/Desktop/data-shell/north-pacific-gyre/2012-07-03/`

She is getting ready to process a large number of files andfile with a `Z` cannot be used, the "good ones" can only contain `A` or `B` and can be selected with the *regular expression* `[AB]`.

On these files she will need to apply a software written by her boss called `goostats` and rewrite the results into a new file. (Warning: a bug in the `goostats` program: if a new file is not provided the original data file will be overwritten to contain nothing!). There is no screen output provided by `goostats` and she decides to `echo` the file name being processed so she can see progress.

The final command will look like this:

```
for datafile in *[AB].txt
do echo $datafile
bash goostats $datafile stats-$datafile
done
```

When she runs her program now, it produces one line of output every five seconds or so:

```
NENE01729A.txt
NENE01729B.txt
NENE01736A.txt
...
```

9.1 Nested loops

Nested loops are a loop within a loop.

> Exercise: Nested loops (verbatim from tutorial) Suppose we want to set up up a directory structure to organize some experiments measuring the growth rate under different sugar types and different temperatures. What would be the result of the following code:

```
for sugar in fructose glucose sucrose
do
    for temperature in 25 30 37 40
    do
        mkdir $sugar-$temperature
    done
done
```

[6] http://swcarpentry.github.io/shell-novice/05-loop/

10 Scripts[7]

All commands can placed into a simple text file, and that text file can become a script!

With the power of loops, pipes and redirect powerful scripts can already be written.

The software carpentry page details the process of creating a simple script to extract the *middle* of any plain text file using the combination of `head`, `tail`, pipe `|` and the use of variables.

Please refer the that web page to complete these exercises.

10.1 ADVANCED: Additional info

The web page shows how to create useful scripts allowing users to specify *e.g.* the name of files to be used and other *arguments*.

All scripts are run on the command line with `bash script_name` (and `sh script_name` would be an alternative command.)

However it is sometimes useful to make the file an "executable" so that we don't need to add the `bash` or `sh` command to execute it.

This is accomplished with the command `chmod` to change the operating mode to add the `x` in the privilege list.

If you already followed the web page exercises and created a script called `middle.sh` you can give it execute privilege to yourself only (the *user*) with:

```
chmod u+x middle.sh
```

In that case a listing of the file will show:

```
-rwxr--r--  1 YOU  AD\Domain Users    28 Mar 29 10:34 middle.sh
```

To add the privilege to anyone, we can use the combined group `a` that contains user, group and others:

```
chmod u+a middle.sh
```

In that case a listing of the file will show:

```
-rwxr-xr-x  1 YOU  AD\Domain Users    28 Mar 29 10:34 middle.sh
```

To make use fo the software ***within the current directory*** the command would be using the `./` notation to specify its location within the current directory:

```
./middle.sh
```

To make use of the software withouth the `bash` or `sh` command as an executable without the need to specify the location (*path*) of the file it is necessary to place the software in a directory where the operating system "knows" that executables are normally present. However, most of these directories are usually "protected" by "admin" password and for a good reason!

The alternative is to create a directory *e.g.* within your own home directory, typically called `bin` into which scripts could be placed.

Then, it would be necessary to "declare" where the directory is located by modifying a special variable called `PATH` that is currently holding the name of known executables. Since `PATH` is a variable, we can find it's value with:

```
echo $PATH
```

[7] http://swcarpentry.github.io/shell-novice/06-script/

```
/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin
```

To add your own directory to this variable we can use the `export` command. Absolute *path* is necessary.

Assuming the `bin` is located in the home directory `/Users/YOU` we would add the directory first.

Warning: to not loose the existing information it is necessary to append the value of $PATH as it was before:

```
export PATH=/Users/YOU/bin:$PATH
```

We'll now have an updated `PATH`:

```
echo $PATH
```

```
/Users/YOU/bin:/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin
```

10.1.1 First line

In addition, `bash` script can have the **FIRST LINE** modified to contain the line:

```
#!/bin/sh
```

or

```
#!/bin/bash
```

This is to specify that the script is indeed a `bash` command file.

The first line can be modified to reflect the language of the script, for example if the script was written in Python 2, the first line could be `#!/usr/bin/python`.

11 Summary

11.1 Concepts

Concept	Definition
Wild card	Replace a zero to many characters with `*`
Loop	Repeat many similar commands
Path	Path to an executable as a variable (`PATH`)

11.2 Variables

Concept	Definition
`PATH`	Defines the location of executables. `echo $PATH`. change with `export`.

11.3 Symbols

Table 3: Symbols and filters

Symbol	Meaning
`*`	Wild card: replace zero to any number of characters
`?`	Replace a single character

Symbol	Meaning
`<`	Redirect content of a file *as* standard input.
`[AB]`	Match file names containing either `A` or `B` in *[AB]*

11.4 New commands learned:

Table 4: New commands in order of appearance in the text

Command	`man` page definition and/or example
`sort`	sort lines of text files.
`grep`	file pattern searcher.
`sed`	stream editor. Subsitute text. Remove lines e.g. `sed 1,3d`
`cut`	cut out selected portions of each line of a file.
`tr`	translate characters (substitute even newline).
`echo`	repeats given arguments the standard output. `-n` to suppress newline
`export`	Creates and gives value to a *variable*.
`expr`	evaluate expression.
`for` ... `in` ...	loop contruction