Bash String Processing

Introduction

Over the years, the `bash` shell has acquired lots of new bells and whistles. Some of these are very useful in shell scripts; but they don't seem well known. This page mostly discusses the newer ones, especially those that modify strings. In some cases, they provide useful alternatives to such old standbys as `tr` and `sed`, with gains in speed. The general theme is avoiding pipelines.

In the examples below, I'll assume that `string` is a shell variable that contains some character string. It might be as short as a single character, or it might be the contents of a whole document.

Case Conversions

One of the obscure enhancements that can be discovered by reading the `man` page for `bash` is the case-conversion pair:

```
newstring=${string^^}   # the string, converted to UPPER CASE
newstring=${string,,}   # the string, converted to lower case
```

(You can also convert just the *first* letter by using a *single* `^` or `,` .) Notice that the original variable, `string`, is not changed.

Normally, we think of doing this by using the `tr` command:

```
newstring=`echo "$string" | tr '[a-z]' '[A-Z]'`
newstring=`echo "$string" | tr '[A-Z]' '[a-z]'`
```

Of course, that involves spawning a new process. Actually, as the `man` page for `tr` tells you, this isn't optimal; depending on your locale setting, you might get unexpected results. It's safer to say

```
newstring=`echo "$string" | tr '[:lower:]' '[:upper:]'`
newstring=`echo "$string" | tr '[:upper:]' '[:lower:]'`
```

Using `tr` is certainly more readable; but it also takes a lot longer to type. How about execution time?

Timing tests

Here's a code snippet that does nothing, a hundred thousand times:

```
str1=X
```

```
i=0
time (
while [ $i -lt 100000 ]
do
        let i++
done
)
```

On my machine — currently, a 3 GHz (6000 bogomips) dual-core Pentium box — that takes about 1.57 seconds. That's the `bash` overhead for running the useless loop. Nearly all of that is “user” time; the “sys” time is only a few dozen milliseconds.

Now let's add the line

```
str2=${str1^^}
```

to the loop, just after the `let` statement. The execution time jumps to about 2.3 seconds; so executing the added line 100,000 times took about 0.7 second. That's about 7 microseconds per execution.

Now, let's try putting the line

```
str2=`echo "$str1" | tr '[:lower:]' '[:upper:]'`
```

inside the loop instead. The execution time is now a whopping $1^{m}\,33^{s}$ of real time — but only 3 seconds of user and 7 sec of system time! Apparently, the system gives both `bash` and `tr` a thousand one-millisecond time-slices a second, and then takes a vacation until the next round millisecond comes up.

If we try to even things up a bit by making the initial string longer, we find practically the same times for the version using `tr`, but about 0.2 second longer than before for the all-shell version, if the string to convert is `"Hello, world!"`. Clearly, we need a really big string to measure `bash`'s speed accurately.

So let's initialize the original string with the line

```
str1=`cat /usr/share/dict/american-english`
```

which is a text file of 931708 characters. For this big file, a single cycle through the loop is enough: it takes `bash` about 45.7 seconds, all but a few milliseconds of which is “user” time. On the other hand, the `tr` version takes only 0.24 seconds to process the big text file.

Clearly, there's a trade-off here that depends on the size of the string to be converted. Evidently, the context switch required to invoke `tr` is the bottleneck when the string is short; but `tr` is so much more efficient than `bash` in converting big strings that it's faster when the string exceeds a few thousand characters. I find my machine takes about 1.55 milliseconds to process a string about 4100 characters long, regardless of which method is used. (About a quarter of a millisecond is used by the system when `tr` is invoked; presumably, that's the time required to set up the pipeline and make the context switch.)

`sed`-like Substitutions

Likewise, you can often make `bash` act enough like `sed` to avoid using a pipeline. The syntax is

```
newstring=${oldstring/pattern/replacement}
```

Notice that there is no trailing slash, as in `sed` or `vi`: the closing brace terminates the substitution string.

The catch is that only shell-type patterns (like those used in pathname expansion) can be used, not the elaborate regular expressions recognized by `sed`. Also, only a single replacement normally occurs; but you can simulate a "global" replacement by using *two* slashes before the pattern:

```
newstring=${oldstring//pattern/replacement}
```

A handy use for this trick is in sanitizing user input. For example, you might want to convert a filename into a form that's safe to use as (part of) a shell-variable name: filenames can contain hyphens and other special characters that are not allowed in variable names, which can only be alphanumeric. So, to clean up a dirty string:

```
clean=${dirty//[-+=.,]/_}
```

If we had set `dirty='a,b.c=d-e+f'`, the line above converts the dangerous characters to underscores, forming the clean string: `a_b_c_d_e_f`, which can be used safely in a shell script.

And you can omit the replacement string, thereby deleting the offensive parts entirely. So, for example,

```
cleaned=${dirty//[-+=.,]}
```

is equivalent to

```
cleaned=`echo $dirty | sed -e 's/[-+=.,]//g'`
```

or

```
cleaned=`echo $dirty | tr -d '+=.,-'`
```

where we have to put the hyphen last so `tr` won't think it's an option.

Be careful: `sed` and `tr` allow the use of ranges like `'A-Z'` and `'0-9'` ; but `bash` requires you to either enumerate these, or to use character classes like `[:upper:]` or `[:digit:]` *within* the brackets that define the pattern list.

You can even force the pattern to appear at the beginning or the end of the string being edited, by prefixing `pattern` with `#` (for the start) or `%` (for the end).

Faking `basename` and `dirname`

This use of `#` to mark the beginning of an edited string, and `%` for the end, can also be used to simulate the `basename` and `dirname` commands in shell scripts:

```
dirpath=${path%/*}
```

extracts the part of the `path` variable *before* the last slash; and

```
base=${path##*/}
```

yields the part *after* the last slash. **CAUTION**: Notice that the asterisk goes *between* the slash and the `##`, but *after* the `%`.

That's because

```
${varname#pattern}
```

trims the **shortest prefix** from the contents of the shell variable `varname` that matches the shell-pattern `pattern` ; and

```
${varname##pattern}
```

trims the **longest prefix** that matches the pattern from the contents of the shell variable. Likewise,

```
${varname%pattern}
```

trims the **shortest suffix** from the contents of the shell variable `varname` that matches the shell-pattern `pattern` ; and

```
${varname%%pattern}
```

trims the **longest suffix** that matches the pattern from the contents of the shell variable. You can see that the general rule here is: a *single* `#` or `%` to match the *shortest* part; or a *double* `##` or `%%` to match the *longest* part.

But be careful. If you just feed a bare filename instead of a pathname to `dirname`, you get just a dot `[.]`; but if there are no slashes in the variable you process with the hack above, you get the filename back, unaltered: because there were no slashes in it, nothing got removed. So this trick isn't a complete replacement for `dirname`.

Another use of `basename` is to remove a suffix from a filename. We often need to do this in shell scripts when we want to generate an output file with the same basename but a different extension from an input file. For example, to convert `file.old` to `file.new`, you could use

```
newname=`basename $oldname .old`.new
```

so that, if you had set `oldname` to `file.old` , `newname` would be set to `file.new` . But it's faster to say

```
newname=${oldname%.old}.new
```

(Notice that we have to use the `%` operation here, even though the generic replacement for `basename` given above uses the `##` operation. That's because we're trimming off a suffix rather than a prefix, in this case.) If you didn't know the old file extension, you could still replace it by saying

```
newname=${oldname%.*}.new
```

This way of trimming off a prefix or a suffix is also useful for separating numbers that contain a decimal point into the integer and fractional parts. For example, if we set `DECIMAL=123.4567`, we can get the part before the decimal as

```
INTEGER=${DECIMAL%.*}
```

and the digits of the fraction as

```
FRACT=${DECIMAL#*.}
```

Numerical operations

Speaking of digits, you can also perform simple integer arithmetic in `bash` without having to invoke another process, such as `expr`. Remember that the `let` operation automatically invokes arithmetic evaluation on its operands. So

```
let sum=5+2
```

will store 7 in `sum`. Of course, the operands on the right side can just as well be shell variables; so, if `x` and `y` are numerical, you could

```
let sum=x+y
```

which is both more compact and faster than

```
sum=`expr $x + $y`
```

If you want to space the expression out for better readability, you can say

```
let "sum = x + y"
```

and `bash` will do the right thing. (You have to use quotes so that `let` has just a single argument. If you don't like the quotes, you can say

```
sum=$(( x + y ))
```

but then you can't have spaces around the `=` sign.)

This way of doing arithmetic is a lot more readable than using `expr` — especially when you're doing multiplications, because `expr` has to have its arguments separated by whitespace, so the asterisk `[*]` has to be quoted:

```
product=`expr $x \* $y`
```

Yuck. Pretty ugly, compared to

```
let "product = x * y"
```

Finally, when you need to increment a counter, you can say

```
let i++
```

or

```
let j+=2
```

which is cleaner, faster, and more readable than invoking `expr`.

Sub-strings

In addition to truncating prefixes and suffixes, `bash` can extract sub-strings. To get the 2 characters that follow the first 5 in a string, you can say

```
${string:5:2}
```

for example.

This can save a lot of work when parsing replies to shell-script questions. If the shell script asks a yes/no question, you only need to check the first letter of the reply. Then

```
init=${string:0:1}
```

is what you want to test. (This gives you 1 character, starting at position 0 — in other words, the first character of the string.)

If the "offset" parameter is −1, the substring begins at the *last* character of the string; so

```
last=${string: -1:1}
```

gives you just the last character. (Note the space that's needed to separate the colon from the minus sign; this is required to avoid confusion with the colon-minus sequence used in specifying a default value.)

To get the last 2 characters, you should specify

```
last2=${string: -2:2} ;
```

note that

```
penult=${string: -2:1}
```

gives you the *next*-to-last character.

Replacing `wc`

Many invocations of `wc` can be avoided, especially when the object to be measured is small. Of course, you should avoid operating on a file directly with `wc` in constructions like

```
size=`wc -c somefile`
```

because this captures the user-friendly repetition of the filename in the output. Instead, you want to re-direct the input to `wc`:

```
size=`wc -c < somefile`
```

But if the operand is already in a shell variable, you certainly don't want to do this:

```
size=`echo -n "$string" | wc -c`
```

— particularly if the string is short — because `bash` can do the job itself:

```
size=${#string}
```

It's even possible to make `bash` fake `wc -w`, if you don't mind sacrificing the positional parameters:

```
set $string
nwords=$#
```

Back to the . . .
main LaTeX page

or the **alphabetic index page**

or the **GF home page**

or the **website overview page**