Expansions and Regular Expressions

The folders present here are the samples used in the lectures slides.

NOTE: This tutorial was prepared in an instance of CS2043 from a few years ago. Our demos directory has one more directory in it, so some of the output you see when executing these commands may be a bit different. The bulk of what is here remains very applicable, though!

1. The Kleene_star (`*`) wildcard is presented first.
2. The question mark(question_mark) (`?`) wildcard is presented next.
3. How to combine them (combined) follows afterward.

You are **heavily encouraged** to go through the slides for this lecture demo. I will reiterate / rephrase from the slides:

> *Of the commands you will learn, in this class, I strongly believe that `find` and*
> *`grep` are the two most useful in terms of daily hacking. Getting used to them is not*
> *easy, and they only become powerful when you start to wield them with regular*
> *expressions, sets, and expansions.*

To be clear, these are exceptionally confusing tools when you start learning them. Shell vs regular expression can get particularly confusing, since the same characters do different things. The best advice I can give: practice.

Pro-Tip: in the below, I am careful to "double quote" everything. This is a good practice to get into when working with these tools. Since you are executing a command
*_in your shell_*, without quotes the pattern **may expand before the command is called**.
It depends. I'll highlight a specific example below, but feel free to execute the commands with / without "double quotes" to see for yourself.

You will get *_a lot_* of exposure to this in the assignments for the course. But since we already have some files here in the demo repo, lets do something else with them :wink:

Go to the `demos` directory for Lecture 10

Some `find` examples

1. Warm-up. Find all `.tex` files in all sub-directories.

```
$ find . -name "*.tex"
./combined/bar.tex
./combined/foo.tex
./kleene_star/Lecture1.tex
./kleene_star/Presentation.tex
```

2. On the other hand, the expansions used in `find` are the `shell` expansions. These
   are **different** than regular expressions (which is what we use with `grep` below). The `*` in the above is a "glob", and expands to "anything". This is subtle, but the difference here is important.

   In your shell, `*.tex` is _any_ file that has `anything` followed by _exactly_ `.tex`. The `.` is the literal period. As you will see below with `grep`, `*` is slightly different: it modifies the thing before it. So while they both mean "0 or more", they are different. This will make more sense by the end of the demo.

```
$ find . -name "*.tex*"
./combined/bar.tex
./combined/foo.tex
./combined/foo.text
./combined/bar.text
./kleene_star/Lecture1.tex
./kleene_star/Presentation.tex
```

   We picked up the `*.tex` from before, but also found two new files: `foo.text` and
   `bar.text`, where the second `*` in `*.tex*` matched the `t` after the `x` in these files.

3. The `*` can be dangerous, though, since it matches _anything_. Consider `*.*t*`:

```
$ find . -name "*.*t*"
./question_mark/Lec2.txt
./question_mark/Lec3.txt
./question_mark/Lec1.txt
./question_mark/Lec11.txt
./combined/bar.tex
./combined/foo.tex
./combined/bar.txt
./combined/foo.txt
./combined/hello.txt
./combined/foo.text
```

```
./combined/bar.text
./combined/h3llo.txt
./kleene_star/Lecture2.txt
./kleene_star/Lectures/placeholder.txt
./kleene_star/AlecGuinness/placeholder.txt
./kleene_star/Lecture1.tex
./kleene_star/sure.txt
./kleene_star/tex/placeholder.txt
./kleene_star/Presentation.tex
./kleene_star/Lecture1.txt
```

We picked up all of the files we did previously, but also got `.txt` files. As with before, `*.` expands to everything up to and including the `.` just before the file extension. But now we have `*t*`. Since `*` can be **0** or more, `txt` is matched because it is of the form `0 anything` followed by `t` followed by `0 or more anything`. The `*` after the `t` therefore expands to `xt` for the `txt` files, `ex` for the `.tex` files, and `ext` for the `.text` files.

4. If we **only** wanted the `text` files for some reason, this is where the `?` wildcard discussed in the slides comes in. The `?` is the "any character" wildcard: it is a length 1 match of "anything":

```
$ find . -name "*.tex?"
./combined/foo.text
./combined/bar.text
```

Notice that it does _*not*_ get the `.tex` files, because the `?` in your shell is exactly one character, not 0 or 1 like it is in regular expressions (see `grep` examples below).

5. `find` is flexible. Tell it where to look and it will.

```
$ find kleene_star/ -name "*.tex"
./kleene_star/Lecture1.tex
./kleene_star/Presentation.tex
```

6. Show me all of the files present:

```
$ find . -type f
./question_mark/Lec2.txt
./question_mark/Lec3.txt
```

```
./question_mark/cake
./question_mark/Lec1.txt
./question_mark/ca
./question_mark/can
./question_mark/Lec11.txt
./question_mark/cap
./question_mark/cat
./combined/bar.tex
./combined/foo.tex
./combined/bar.txt
./combined/foo.txt
./combined/h3
./combined/hello.txt
./combined/foo.text
./combined/bar.text
./combined/h3llo.txt
./kleene_star/Lecture2.txt
./kleene_star/Lectures/placeholder.txt
./kleene_star/AlecGuinness/placeholder.txt
./kleene_star/Lecture1.tex
./kleene_star/sure.txt
./kleene_star/tex/placeholder.txt
./kleene_star/Lec.log
./kleene_star/Presentation.tex
./kleene_star/Lecture1.txt
./README.md
```

Notice that this _*is*_ different than `` `find . -name "*.*"` ``. While file extensions are common, they are not necessary!

7. Please, no file extensions? Where is your proof? Use the `` `!` `` (or `` `-not` ``) for a boolean inversion:

```
$ find . -type f ! -name "*.*"
./question_mark/cake
./question_mark/ca
./question_mark/can
./question_mark/cap
./question_mark/cat
./combined/h3
```

Check it out for yourself: this is the best example I can give for why quoting the string you give to these tools should be done. Execute

```
$ find . -type f ! -name *.*
```

Ok clearly something fishy is going on. It's hard to see what actually went down, but there was actually only **one** file that got stripped out with `! -name *.*`. Which one? Get rid of the `!`:

```
$ find . -type f -name *.*
./README.md
```

Gah! So basically, without the double quotes, the command that actually got executed was `find . -type f ! README.md`. Confirmation

```
$ echo *.*
README.md
$ echo "*.*"
*.*
```

8. Oh dear. This is getting to be too much information. Let's examine what happens with `-maxdepth`

```
$ find . -type f -maxdepth 1
find: warning: you have specified the -maxdepth option after a non-option argument
-type, but options are not positional (-maxdepth affects tests specified before it
as well as those specified after it). Please specify options before other arguments.

./README.md
```

What? Remember how I keep telling you to pay attention to the output of the terminal?

THIS IS REALLY IMPORANT. Basically, it's telling you that the way the command-line arguments is being parsed is not what you think. It is trying to supply `-maxdepth` to modify `-type`. The implementors of `find` handle this gracefully, but there are scenarios (read the `man` page, `find` is very complex) that unexpected behavior may occur.

As an aside, many command-line tools you know and love use `getopts` to parse the
command-line arguments (but not always). [A well-worded answer](http://unix.stackexchange.com/a/96805/150726)
related to this error message.

9. The right way.

```
$ find . -maxdepth 1 -type f
./README.md
```

10. Let's check out the directories.

```
$ find . -type d
.
./question_mark
./combined
./kleene_star
./kleene_star/Lectures
./kleene_star/AlecGuinness
./kleene_star/tex
```

Notice that the current directory **is** included. You asked it to, so it did.

11. Using `-mindepth` and `-maxdepth` together is a powerful trick, and can help you quickly narrow down your results. Start with a broad search, then refine until you discover whatever it is you are looking for (e.g. if you lost a folder and are trying to `find` it again). In this example we are basically saying `depth=1`, since the `min` and `max` depths are the same. Do what you gotta do.

```
$ find . -mindepth 1 -maxdepth 1 -type d
./question_mark
./combined
./kleene_star
```

Some `grep` examples

We will cover `alias` soon in class, but to make our life a little easier
we're going to set-up an `alias`. It just makes it -- for the current session only --

so that when we type `grep`, the terminal will replace it with `grep –Hn –-color=auto`.
The breakdown:

- `H`: print the filename it was found **in**
- `n`: print the line number it was found **on**
- `--color=auto`: color the actual thing matched
  - I cannot show this to you through markdown, do it on your computer!!!

Recall that I am double quoting the expressions I send to `grep`. Sometimes you can
avoid doing this, but you will be better off to get in this habit now.

```
# The `grep` we are using right now (yours will
# likely be the same, but not necessarily)
$ which grep
/usr/bin/grep

# This just makes it so we do not have to type
#      -Hn --color=auto
# every time for the remaining examples
$ alias grep='grep -Hn --color=auto'

# Make sure it got set correctly. `which` will
# show us if an `alias` affects a given command.
$ which grep
alias grep='grep -Hn --color=auto'
        /usr/bin/grep
```

1. Please, start slow! Ok you got it. Let's see some `f` characters.

```
# the foo.tex file
$ grep "f" combined/foo.tex
combined/foo.tex:1:foo

# the foo.text file (see slides)
$ grep "f" combined/foo.text
combined/foo.text:1:foo

# The hello.txt file.  No match
# (there is no 'f' in the file).
$ grep "f" combined/hello.txt
```

```

It's worth mentioning that `grep f combined/foo.tex` won't give problems without the
double quotes. There aren't any potential conflicts: `f` is `"f"` is `'f'`...

2. Let's use those `POSIX` sets. Seriously, make sure you have the `alias` from above
setup and are doing this on your computer...in markdown it means nothing.

```
$ grep "[[:alpha:]]" combined/h3llo.txt
combined/h3llo.txt:1:h3llo

$ grep "[[:digit:]]" combined/h3llo.txt
combined/h3llo.txt:1:h3llo

$ grep "[[:alnum:]]" combined/h3llo.txt
combined/h3llo.txt:1:h3llo
```

3. Making sets is easy! Just use `[stuff you want in the set]` with `[brackets]`. The
shell won't get mad, but a set is defined as only unique elements: `[s]` and `[ss]`
and `[sss]` are all the set `[s]`:

```
# No output: no a, b, c, or d in this file
$ grep "[abcd]" combined/h3llo.txt

# No a, but there is an h
$ grep "[ah]" combined/h3llo.txt
combined/h3llo.txt:1:h3llo

# Finds all of them!
$ grep "[hlo]" combined/h3llo.txt
combined/h3llo.txt:1:h3llo

# Duplication doesn't change the behavior
$ grep "[hloolh]" combined/h3llo.txt
combined/h3llo.txt:1:h3llo

# https://en.wikipedia.org/wiki/Palindrome
$ grep "[hloolhhloolh]" combined/h3llo.txt
combined/h3llo.txt:1:h3llo
```
```

4. Inverting sets is also easy, you use the `^` character _*inside*_ the `[brackets]`.

```
# Find all *not* :alpha: characters
$ grep "[^[:alpha:]]" combined/h3llo.txt
combined/h3llo.txt:1:h3llo

# Find all *not* :digit: characters
$ grep "[^[:digit:]]" combined/h3llo.txt
combined/h3llo.txt:1:h3llo

# Since `h3llo.txt` only contains the line
# 'h3llo', and those are all alphanumeric
# characters, there is no match.
$ grep "[^[:alnum:]]" combined/h3llo.txt
```

These examples are the opposites of what we saw in `2`. You can also easily invert the sets you made in `3`:

```
# Matches everything, since there are no 'a',
# 'b', 'c', or 'd' in the file
$ grep "[^abcd]" combined/h3llo.txt
combined/h3llo.txt:1:h3llo

# Matches everything *except* the 'h'
$ grep "[^ah]" combined/h3llo.txt
combined/h3llo.txt:1:h3llo

# Only matches the '3'
$ grep "[^hlo]" combined/h3llo.txt
combined/h3llo.txt:1:h3llo
```

5. Getting a little more complicated, let's look for `hello` (and some variants) recursively using the `-r` flag to `grep`. You must now specify a folder for `grep` to look in, like `find`. We will use `combined` for this but it doesn't matter --- specify whichever folder you are interested in.

```
# Find just 'hello'
$ grep -r "hello" combined/
combined/hello.txt:1:hello

# Find just h3llo
```

```
$ grep -r "h3llo" combined/
combined/h3llo.txt:1:h3llo

# The `.` says "any single character"
$ grep -r "h.llo" combined/
combined/hello.txt:1:hello
combined/h3llo.txt:1:h3llo

# Make it a little clearer, only el and 3l matched
$ grep -r ".l" combined/
combined/hello.txt:1:hello
combined/h3llo.txt:1:h3llo
```

As you can see, this is where having the filename printed by `grep` starts to become really helpful. In daily usage of `grep`, you are likely trying to figure out not just is there *any* file that has that pattern, but rather *which* file(s) have it.
For example, say you were trying to re-factor some code. You had defined a variable
`the_variable` and used it in a bunch of files. `grep` can help you find all of the files that use `the_variable`.

Do I suggest using command-line tools to re-factor a code base? Not really. But say you used an IDE or text editor to do the re-factor. You can quickly check for your own sanity whether or not the IDE / text editor got all of them.

6. Ok let's try and work with more explicit regex tools.

```
# Picks up everything to the second l
$ grep -r ".*l" combined
combined/hello.txt:1:hello
combined/h3llo.txt:1:h3llo

# Find exactly "any character" (.),
# followed by *exactly* two `l` characters
$ grep -r ".l\{2\}" combined/
combined/hello.txt:1:hello
combined/h3llo.txt:1:h3llo

# Subtle: .* matched he and h3 in this
# case, and then the two `l` from our
# explicit matching 'l\{2\}'
$ grep -r ".*l\{2\}" combined/
combined/hello.txt:1:hello
combined/h3llo.txt:1:h3llo
```

```
# In contrast, ? is 0 or 1 only.  It will
# match against 1 over 0 when possible, so
# in this example ? is 'e' and '3' respectively.
$ grep -r ".\?l\{2\}" combined/
combined/hello.txt:1:hello
combined/h3llo.txt:1:h3llo

# The + says "at least one".
$ grep -r ".\+l\{2\}" combined/
combined/hello.txt:1:hello
combined/h3llo.txt:1:h3llo

# Warning: * may not always be what you want
# It **DOES** find _0_ or more 'x'
$ grep -r "x*h.ll" combined/
combined/hello.txt:1:hello
combined/h3llo.txt:1:h3llo

# No output because we said "find at least one x"
$ grep -r "x\+h.ll" combined/
```

7. Alright so what is this `^` and `$` nonsense? Let's work with `.ll`. This expression just means "find any character, and then two `l` characters."

```
# Great, it works as expected
$ grep -r ".ll" combined/
combined/hello.txt:1:hello
combined/h3llo.txt:1:h3llo

# `^` says "beginning of expression"
# so no output because the `.` is a single
# character only!
$ grep -r "^.ll" combined/

# Ok fine we will just cheat and `*` the `.`
$ grep -r "^.*ll" combined/
combined/hello.txt:1:hello
combined/h3llo.txt:1:h3llo

# Oh Yeah I OWN you grep.  Or do I?
# The `$` says "end of expression"
$ grep -r "^.*ll$" combined/

# Ok well that makes sense, we're missing the `o`
```

```
$ grep -r "^.*llo$" combined/
combined/hello.txt:1:hello
combined/h3llo.txt:1:h3llo
```

8. Ok but this is really annoying. Do I **always** have to escape `?`, `+`, and `{}`?
   Yes. For ["regular" regular expressions][regregex]. Or you can use
   [_extended_ regular expressions][extregex]. This is `grep -E` or `egrep`. However,
   if you use extended regular expressions, and want say the literal `?`, you now have
   to escape this (`\?` in _extended_ regular expressions means "the question mark").
   This will come back when we cover `sed`.

   Of course, `egrep` is indeed a different command, so we need to add the same `alias`
   we did earlier to get consistent results with what we have seen so far

```
# The `egrep` we are using right now
$ which egrep
/usr/bin/egrep

# Same as before, just avoiding having to
# type '-Hn --color=auto' every time.
$ alias egrep='egrep -Hn --color=auto'

# Make sure it got set correctly. `which` will
# show us if an `alias` affects a given command.
$ which egrep
alias egrep='egrep -Hn --color=auto'
        /usr/bin/egrep
```

   Working with the examples we have already seen in `6`:

```
# Regular regex
$ grep -r ".\?l\{2\}" combined/
combined/hello.txt:1:hello
combined/h3llo.txt:1:h3llo

# Extended regex
$ egrep -r ".?l{2}" combined/
combined/hello.txt:1:hello
combined/h3llo.txt:1:h3llo

# Regular regex
$ grep -r ".\+l\{2\}" combined/
combined/hello.txt:1:hello
```

```
combined/h3llo.txt:1:h3llo

# Extended regex
$ egrep -r ".+l{2}" combined/
combined/hello.txt:1:hello
combined/h3llo.txt:1:h3llo
```

Feel free to `unalias grep` and `unalias egrep` and run the above commands to
compare the difference in output. As mentioned earlier, these were _*temporary*_ aliases --- once you close your terminal and/or open a new one, they will not be there. We will learn very soon how to make it so these stick around!

[regregex]: https://www.gnu.org/software/sed/manual/html_node/Regular-Expressions.html
[extregex]: http://tldp.org/LDP/Bash-Beginners-Guide/html/sect_04_01.html#sect_04_01_03