COMP(2041|9044) 26T1 — Unix Filters

https://www.cse.unsw.edu.au/~cs2041/26T1/

What is a filter?

- A **filter** is a program that transforms a byte stream.

On Unix-like systems filters are commands that:

- read bytes from their standard input or specified files
- perform useful transformations on the stream
- write the transformed bytes to their standard output
- most filters work on text, UTF-8 or perhaps just ASCII
- most filters are line-based, a few are byte based or character-based

Using Filters

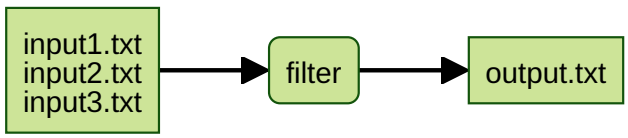
Shell I/O redirection can be used to specify filter source and destination files:

```
$ filter < input.txt > output.txt
$ < input.txt filter > output.txt
$ < input.txt > output.txt filter
```



Alternatively, most filters allow input files to be specified as arguments:

```
$ filter input1.txt input2.txt input3.txt > output.txt
```



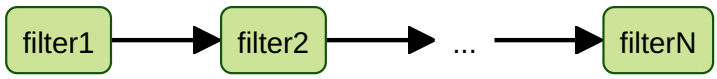
Using Filters

In isolation, filters are reasonably useful

In combination, they provide a very powerful problem-solving toolkit.

Filters are normally used in combination via a pipeline:

```
filter1 | filter2 | ... | filterN
```



Note: similar style of problem-solving to function composition.

Using Filters

- Unix filters use common conventions for command line arguments:
  - input can be specified by a list of file names
  - if no files are mentioned, the filter reads from standard input
    - which may have been re-directed from a file
  - the filename **-** corresponds to standard input
- for example:

```
# read from the file data1
$ filter data1
# or
$ filter < data1
# read from the files data1 data2 data3
$ filter data1 data2 data3
# read from data1, then stdin, then data2
$ filter data1 - data2
```

If a filter doesn't cope with named sources, you use **`cat`** at the start of the pipeline

Filters: Options

Filters normally perform multiple variations on a task.

Selection of the variation is accomplished via command-line options:

- options are introduced by a **-** ("minus" or "dash")
- options have a "short" form, **-** followed by a single letter (e.g. `-v`)
- options have a "long" form, **--** followed by a word (e.g. `--verbose`)
- short form options can usually be combined (e.g. `-av` vs `-a -v`)
- `--help` (`-h` or sometimes `-?`) often gives a list of all command-line options

Filters: Options

Most filters have *many* options for controlling their behaviour.

Unix **man**ual entries describe how each option works.

To find what filters are available: `apropos` *keyword*

The solution to all your problems: ***RTFM***

`cat` - the simplest filter

- `cat` command copies its input to output unchanged (identity filter).
- `cat` - given filenames, con**cat**enates them onto stdout.
- `cat` - given no filenames, copies stdin to stdout unchanged.

```
$ cat hello.c
#include <stdio.h>

int main(void) {
    printf("hello\n");
}
$ cat < hello.c
#include <stdio.h>

int main(void) {
    printf("hello\n");
}
```


`cat`: some options

Some useful **`cat`** options:

-n	**n**umber output lines (starting from 1)
-A	display non-printing characters - handy for debugging (not ava ilable on mac)
-s	**s**queeze consecutive blank lines into single blank line

See also: the `tac` command - which reverses the order of lines.

See also: the `rev` command - which reverses the order of characters in lines.

cat: implemented in C - passing bytes from input to stdout

```c
// write bytes of stream to stdout
void process_stream(FILE *stream) {
    int byte;
    while ((byte = fgetc(stream)) != EOF) {
        if (fputc(byte, stdout) == EOF) {
            perror("cat:");
            exit(EXIT_SUCCESS);
        }
    }
}
```

source code for cat.c

cat: implemented in C - where does input come from?

```c
// process files given as arguments
// if no arguments process stdin
int main(int argc, char *argv[]) {
    if (argc == 1) {
        process_stream(stdin);
    } else {
        for (int i = 1; i < argc; i++) {
            if (!strcmp(argv[i], "-")) {
                process_stream(stdin);
            } else {
                FILE *in = fopen(argv[i], "r");
                if (in == NULL) {
                    fprintf(stderr, "%s: %s: ", argv[0], argv[i]);
                    perror("");
                    return EXIT_FAILURE;
                }
                process_stream(in);
                fclose(in);
            }
        }
    }
    return EXIT_SUCCESS;
}
```

source code for cat.c

cat: implemented in Python

```python
def process_stream(stream):
    """
    copy bytes of f to stdout
    """
    for line in stream:
        print(line, end="")
def main():
    """
    process files given as arguments, if no arguments process stdin
    """
    if not sys.argv[1:]:
        process_stream(sys.stdin)
    else:
        for pathname in sys.argv[1:]:
            if pathname == "-":
                process_stream(sys.stdin)
            else:
                with open(pathname, "r") as f:
```

common task - filter particular lines

- very commonly we need a filter which passes only particular lines
- often we want line number added to line
- useful if filename add (for when input is coming from multiple files)
- one common need: select lines containing string(s)

selecting lines containing a string - C

```c
// print lines containing the specified substring
void process_stream(FILE *stream, char *name, char *substring) {
    char *line = NULL;
    size_t line_size = 0;
    int line_number = 1;
    while (getline(&line, &line_size, stream) > 0) {
        if (strstr(line, substring) != NULL) {
            printf("%s:%d:%s", name, line_number, line);
        }
        line_number++;
    }
    free(line);
}
```

source code for fgrep.c

selecting lines containing a string - C - main function

```c
// if no arguments process stdin
int main(int argc, char *argv[]) {
    if (argc == 2) {
        process_stream(stdin, "<stdin>", argv[1]);
    } else {
        for (int i = 2; i < argc; i++) {
            FILE *in = fopen(argv[i], "r");
            if (in == NULL) {
                fprintf(stderr, "%s: %s: ", argv[0], argv[i]);
                perror("");
                return 1;
            }
            process_stream(in, argv[i], argv[1]);
            fclose(in);
        }
    }
```

source code for fgrep.c

selecting lines containing a string - Python

```
def process_stream(f, name, substring):
    """
    print lines containing substring
    equivalent to `grep -FHn`
    """
    for (line_number, line) in enumerate(f, start=1):
        if substring in line:
            print(f'{name}:{line_number}:{line}', end='')
def main():
    """
    process files given as arguments, if no arguments process stdin
    """
    if   len(sys.argv) == 1:
        sys.exit(f"Usage: {sys.argv[0]} <NEEDLE> [FILE]...")
    elif len(sys.argv) == 2:
        process_stream(sys.stdin, "<stdin>", sys.argv[1])
    elif len(sys.argv) > 2:
        for pathname in sys.argv[2:]:
            with open(pathname, 'r') as f:
                process_stream(f, pathname, sys.argv[1])
```

source code for fgrep.py

Matching Any of A Set of String

- previous programs too limited for many uses
- often need to select lines containing any of a set of strings
- set may be huge or infinite
- **regular expressions**: concise powerful notation for sets of strings
- concept - famous theoretician Stephen Kleene 1950s
- syntax & implementation - Turing award winner - Ken Thompson (1968)
- originally editors (qed, ed, vi, emacs, ...) & compilers

Regular Expressions

- a *regular expression* (regex) often thought of as a pattern
- but think of it as defining a set of strings.
- set may be small, huge or infinite
- regular expressions libraries available for most languages.
- many tools make use of regular expressions for searching
- effective use of regular expressions makes you much more productive
- incredibly useful for manipulation of textual data
- POSIX standard(s) for regular expressions

Regular Expressions Basics

- Unless a character has a special meaning it matches itself
  - e.g. **a** has no special meaning so it matches **a**.
- p***** denotes zero or more repetitions of p.
  - e.g. **b*** matches the empty string and: **b**, **bb**, **bbb**, **bbbb** ...
  - note this is an infinite set of strings
- $pattern_1$**|**$pattern_2$ denotes the union of $pattern_1$ and $pattern_2$.
  - e.g **perl|python|ruby** matches any of: **perl**, **python** or **ruby**
    - **|** is sometimes called alternation
- Parentheses are used for grouping
  - e.g. **c(,c)*** matches: **c c,c c,c,c c,c,c,c** ...
    - and **(d|e)*(f|g)** matches **f**, **g**, **df**, **dg**, **ef**, **eg**, **ddf**, **deg**, **edf**, **edg**, **eef**, ...
- backslash **** removes any special meaning of the following character
  - e.g. ***** matches an ***** instead of indicating repetition
- Any regular expression can be written using only **()*|**
  - but many syntax features are present for convenience & clarity.

Convenient Regular Expressions for matching Single Characters

- **.** (dot) matches any single character.
- Square brackets provide convenient matching of any **one** of a set of characters.
- **[***listOfCharacters***]** matches any **single** character from ***listOfCharacters***.
  - e.g. **[aeiouAEIOU]** matches any (English) vowel.
- A shorthand is available for ranges of characters **[**$first - last$**]**
  - e.g **[a-e] [a-z] [0-9] [a-zA-Z] [A-Za-z] [a-zA-Z0-9]**
- Square brackets matching can be inverted with an **^**
- **[^***listOfCharacters***]** matches any **single** character except those in ***listOfCharacters***.
  - e.g **[^a-e]** matches any character *except* one of the first five lowercase letters
- Other characters lose their special meaning inside bracket expressions.
- e.g. `[^X]*X` matches any characters up to and including the first X

Anchoring Matches

- regular expressions may be used to match against a whole string
  - e.g re.fullmatch in Python
- regular expressions are often used to match a substring (part of a string)
  - e.g `grep` prints lines containing a substring matching the regular expression
  - e.g `re.search` in Python (`re.match` matches only at start of string)
- when matching part of a string you can limit matches to the start or end of a string (or both)
- start of the string is denoted by **^** (uparrow)
  - `^hello` matches a string starting with **hello**
  - note **^** has two meanings in regular expressions
  - e.g. `^[abc]` matches **a** or **b** or **c** at the start of a string.
  - e.g. `[^abc]` matches any character except **a** or **b** or **c** (anywhere in the string)
- the end of the string is denoted by **$** (dollar)
  - `cat$` matches `cat` at the end of a string.
  - **^cat.*dog$** matches any string starting `cat` and finishing dog.

Convenient Regular Expressions for Repetition

For convenience and readability, more special characters denoting repetition:

- p***** denotes zero or more repetitions of p
- p**+** denotes one or more repetitions of p
  - e.g. **`[0-9]+`** matches any sequence of digits (i.e. matches integers)
  - e.g. **`[-'a-zA-Z]+`** matches any sequence of letters/hyphens/apostrophes
    - this pattern could be used to match words in a piece of English text,
    - e.g. **`it's`**, **`John`**, ...
- p**?** denotes zero or one occurrence of p
- p{n**}** denotes n repetitions of p
  - e.g **`z[0-9]{7}`** matches a UNSW zid
- p{n,m**}** denotes n to m repetitions of p
- p{n,**}** denotes n or more repetitions of p
- p{**,**m**}** denotes m or less repetitions of p

Regular Expression Examples

Regex	Possible Matches
`abc`	abc
`a.c`	abc, aac, acc, aXc, a2c, ...
`ab*c`	abc, ac, abbc, abbbc, ...
`a|the`	a, the
`[a-z]`	a, b, c, ... z

websites https://regex101.com and https://regexr.com highly recommend to help let you understand regex matching

`grep`: select lines matching a pattern

- **`grep`** copies to stdout lines that match a specified regular expression.
  - some regular expression characters also special meaning to Shell
  - when run from Shell regular expression often needs single quotes
  - `grep` is an acronym for **G**lobally search with **R**egular **E**xpressions and **P**rint

Some useful **`grep`** options:

`-E`	use extended regular expression syntax
`-i`	ignore upper/lower-case difference in matching
`-v`	only display lines that *do not* match the pattern
`-c`	print a count of matching lines
`-w`	only match pattern if it makes a complete word
`-x`	only match pattern if it makes a complete line

grep and friends

- `grep -F` match strings only (no regular expressions)
  - use if you don't need regex
  - faster
  - avoids bugs from regex syntax accidentally occurring in your match string
- `grep -G` or `grep` matches a subset of regular expressions, e.g. no **+ ? | () {}**
  - faster than `grep -E` but this is rarely important these days
  - generally just use `grep -E`
- `grep -E` (extended grep) matches full POSIX regular expressions
  - `grep -E` is what you want most of the time
- `grep -P` POSIX regular expressions + Perl extensions
  - standard Python regex include some but not all Perl extensions
  - use if you need Perl/Python regex extensions
  - PCRE library widely used (e.g. Apache)

wc: word counter

- **wc** summarizes its input as a single line.
- often useful as last command in pipeline
- also useful in shell scripts
- other filters may have counting options, e.g. **`grep -c`**
- some useful **wc** options:

-c	print the number of bytes (-m for characters)
-w	print the number of **w**ords (non-white space) only
-l	print the number of **l**ines only

- by default, **wc** prints the number of line, words, and bytes in its input, e.g

```
$ wc /etc/passwd
  49   79 2793 /etc/passwd
```


wc in C

```
int n_lines = 0;
int n_words = 0;
int n_chars = 0;
int in_word = 0;
int c;
while ((c = fgetc(in)) != EOF) {
    n_chars++;
    if (c == '\n') {
        n_lines++;
    }
    if (isspace(c)) {
        in_word = 0;
    } else if (!in_word) {
        in_word = 1;
        n_words++;
    }
}
printf("%d %d %d %s\n", n_lines, n_words, n_chars, name);
```

source code for wc.c

wc in Python

```
def process_stream(pathname, stream):
    """
    count lines, words, chars in stream
    """
    lines = 0
    words = 0
    characters = 0
    for line in stream:
        lines += line.endswith(os.linesep)
        words += len(line.split())
        characters += len(line)
    print(f"{lines:>6} {words:>6} {characters:>6} {pathname}")
```

source code for wc.py

tr: transliterate characters

- **tr** reads chars and writes characters, mapping (replacing) some chars with others.
- the mapping is specified as 2 arguments: **tr** sourceChars destChars**
- each char in **sourceChars** is mapped to the corresponding char in **destChars**. For example:

```
tr 'abc' '123' < someText
```

$sourceChars$ = **'abc'**, $destChars$ = **'123'**: **a** $\rightarrow$ **1** **b** $\rightarrow$ **2** **c** $\rightarrow$ **3**

- **tr** doesn't accept file names on the command line - it uses stdin only
- **tr** is not line-based it works with individual chars
- most **tr** implementations do not support multi-char characters (UTF8)
  - they really work with bytes not characters!

tr: transliterate characters

Chars that are not in $sourceChars$ are copied unchanged to output.

If there is no corresponding char (i.e. $destChars$ is shorter than $sourceChars$), then the last char in $destChars$ is used.

Shorthands are available for specifying char lists:

E.g. `'a-z'` is equivalent to `'abcdefghijklmnopqrstuvwxyz'`

Note: newlines can be modified if the mapping specification requires it.

tr: transliterate characters

Some useful **tr** options:

-c	map all bytes *not* occurring in $sourceChars$ (**c**omplement)
-s	**s**queeze adjacent repeated characters out (only copy the first)
-d	**d**elete all characters in $sourceChars$ (no $destChars$)

```
# map all upper-case letters to lower-case equivalents
tr 'A-Z' 'a-z' < text
# naive encryption (a->b, b->c, ... z->a)
tr 'a-zA-Z' 'b-zaB-ZA' < text
# remove all digits from input
tr -d '0-9' < text
# break text file into individual words, one per line
tr -cs 'a-zA-Z0-9' '\n' < text
```


head/tail: select first/last lines

- **head** prints the first n (default 10) lines of input.
- The **tail** prints the last n lines of input.
- **-n** option changes number of lines head/tail prints.
  - e.g. **tail -n 30 file** prints last 30 lines of `file`.
- Combine `head` and `tail` to select a range of lines.
  - **head -n 100 | tail -n 20** copies lines 81..100 to output.
- **head** & **tail** mostly used with stdin or single file
  - if multiple files specified output prefixed with name

Delimited Input

Many filters are able to work with text data formatted as *fields* (columns in spreadsheet terms).

Such filters typically have an option for specifying the delimiter or field separator.
(Unfortunately, they often make different assumptions about the default column separator)

Example (tab-separated columns):

```
John    50
Wen     75
Andrew  33
Wenjie  95
Yang    93
Sowmya  96
```

Delimited Input

Or vertical bar-separated columns, CSE enrollment file:

```
COMP1511|2252424|Abbot, Andrew John    |3727|1|M
COMP2511|2211222|Abdurjh, Saeed        |3640|2|M
COMP1511|2250631|Accent, Aac-Ek-Murhg  |3640|1|M
COMP1521|2250127|Addison, Blair        |3971|1|F
COMP4141|2190705|Allen, David Peter    |3645|4|M
COMP4960|2190705|Allen, David Pater    |3645|4|M
```

Or colon-separated columns, e.g: Unix password file

```
root:*:0:0:root:/root:/bin/bash
jas:*/44Ko:100:100:John Shepherd:/home/jas:/bin/bash
cs1521:*:101:101:COMP1521:/home/cs1521:/bin/bash
cs2041:*:102:102:COMP2041:/home/cs2041:/bin/bash
cs3311:*:103:103:COMP3311:/home/cs3311:/bin/bash
```


cut: vertical slice

The `cut` command prints selected parts of input lines.

- `cut` can select fields, column separator defaults to tab
- `cut` can select a range of character positions
- some useful **`cut`** options:

-f$listOfCols$	print only the specified fields (tab-separated) on output
-c$listOfPos$	print only chars in the specified positions
-dc	use character c as the field separator

- lists are specified as ranges (e.g. `1-5`) or comma-separated (e.g. `2,4,5`).
- `cut` has no way to refer to "last column" without counting the columns.
  - awk has this and usually `sed` or `perl` will solve problem too
- `cut` has no way to reorder columns.
  - awk has this and usually `sed` or `perl` will solve problem too

cut: vertical slice - Examples

```
# print the first column
cut -f1 data
# print the first three columns
cut -f1-3 data
# print the first and fourth columns
cut -f1,4 data
# print all columns after the third
cut -f4- data
# print the first three columns, if '|'-separated
cut -d'|' -f-3 data
# print the first five chars on each line
cut -c1-5 data
```

sort: sort lines

The `sort` command copies input to output but ensures that the output is arranged in some particular order of lines.

By default, sorting is based on the first characters in the line.

Other features of `sort`:

- understands that text data sometimes occurs in delimited fields.
  (so, can also sort fields (columns) other than the first (which is the default))
- can distinguish numbers and sort appropriately
- can ignore punctuation or case differences
- can sort files "in place" as well as behaving like a filter
- capable of sorting *very large* files

sort: sort lines

Some useful **`sort`** options:

-r	sort in descending order (**r**everse sort)
-n	sort numerically rather than lexicographically
-d	dictionary order: ignore non-letters and non-digits
-tc	use character c to separate columns (default: non-blank to blank transition)
-kn	sort on column n

Note: often need to put quotes (`' '`) around the separator character **c** to prevent the shell from mis-interpreting it.

Hint: to specify Tab as the field delimiter with an interactive shell like bash, type CTRL-v before pressing the Tab key.

sort: sort lines - examples

```
# sort numbers in 3rd column in descending order
sort -nr -k3 data
# sort the password file based on user name
sort -t: -k5 /etc/passwd
```

sort in Python

```
import sys
def process_stream(f):
    """
    print lines of stream in sorted order
    """
    print("".join(sorted(f)), end="")
def main():
    """
    process files given as arguments, if no arguments process stdin
    """
    if len(sys.argv) == 1:
        process_stream(sys.stdin)
    else:
        with open(sys.argv[1], 'r') as f:
            process_stream(f)
```

source code for sort.py

uniq: remove or count duplicates

uniq remove all but one copy of **adjacent** identical lines.

Some useful **uniq** options:

-c	also print number of times each line is duplicated
-d	only print (one copy of) duplicated lines
-u	only print lines that occur uniquely (once only)

- extremely useful data analysis/summary tool
- often preceded by **cut**
- almost always preceded by **sort** (to ensure identical lines are adjacent)
- for example:

```
# extract first field, sort, and tally
cut -f1 data | sort | uniq -c
```


uniq in C

```
// cope stream to stdout except for repeated lines
void process_stream(FILE *stream) {
    char *line = NULL;
    size_t line_size = 0;
    char *last_line = NULL;
    size_t last_line_size = 0;
    while (getline(&line, &line_size, stream) > 0) {
        if (last_line == NULL || strcmp(line, last_line) != 0) {
            fputs(line, stdout);
        }
        // grow last_line if line has grown
        if (last_line_size != line_size) {
            last_line = realloc(last_line, line_size);
            assert(last_line != NULL);
            last_line_size = line_size;
        }
        strncpy(last_line, line, line_size);
    }
    free(line);
    free(last_line);
}
```

source code for uniq.c

uniq in Python

```
import sys
def process_stream(stream):
    """
    copy stream to stdout except for repeated lines
    """
    last_line = None
    for line in stream:
        if last_line is None or line != last_line:
            print(line, end='')
        last_line = line
def main():
    """
    process files given as arguments, if no arguments process stdin
    """
    if not sys.argv[1:]:
        process_stream(sys.stdin)
    else:
        for pathname in sys.argv[1:]:
            with open(pathname, 'r') as f:
                process_stream(f)
```

source code for uniq.py

sed: stream editor

- interactive editors are used to change files, e.g.: vim, emac, sublime, atom, notepad)
- **sed** is an editor for streams (pipelines)
  - **sed** can also be used to change files
- **sed** is not interactive - editing commands specified on command-line or in a file

How sed works:

- read each line of input
- check if it matches any patterns or line-ranges
- apply related editing commands to the line
- write the transformed line to output

sed: stream editor

The editing commands are very powerful and subsume the actions of many of the filters looked at so far.

In addition, sed can:

- partition lines based on patterns rather than columns
- extract ranges of lines based on patterns or line numbers

Two important sed options

-n	do not print lines by default - applies all editing commands as normal but displays no output, unless p appended to edit command
-E	extended regular expressions like grep you often want this

sed: stream editor

Editing commands:

p	print the current line
d	delete (don't print) the current line
s/*regex***/***replace***/**	substitute first occurrence of string matching ***regex*** by ***replace*** string
s/*regex***/***replace***/g**	substitute all occurrences of string matching ***regex*** by ***replace*** string
q	terminate execution of sed

sed: stream editor

All editing commands can be qualified by line addresses or line selector patterns to limit lines where command is applied:

line_number	selects the specified line
*start_line_number***,***end_line_number*	selects all lines between specified line numbers
/*regex***/**	selects all lines that match ***regex***
/*regex1***/,/***regex2***/**	selects all lines between lines matching regex1 and regex2

sed: stream editor - examples

```
# print all lines
sed -n 'p' < file

# print the first 10 lines
sed '10q' < file
sed -n '1,10p' < file

#print lines 81 to 100
sed -n '81,100p' < file

#print the last 10 lines of the file?
sed -n '$-10,$p' < file  # does NOT work
```

sed: stream editor - more examples

```
# print only lines containing 'xyz'
sed -n '/xyz/p' < file

# print only lines NOT containing 'xyz'
sed '/xyz/d' < file

# show the passwd file, displaying only the
# lines from "root" up to "nobody" (i.e. system accounts)
sed -n '/^root/,/^nobody/p' /etc/passwd

# remove first column from ':'-separated file
sed 's/[^:]*://' datafile

# reverse the order of the first two columns
sed -E 's/([^:]*):([^:]*):(.*)$/\2:\1:\3/'
```


change a file with sed

Read & writing a file simultaneously is dangerous.

This command will leave `story.txt` zero-length.

```
sed 's/[aeiou]//g' story.txt > story.txt  # DANGER story.txt will be destroyed
```

The shell truncates `story.txt` to zero length before running `sed`.

`sed` finds nothing in the file to read.

A simple work-around is a temporary file (there are issues which we will discuss later)

```
sed 's/[aeiou]//g' story.txt > story.txt.new
mv story.txt.new story.txt
```

Some `sed` implementations have a command-line option **`-i`** which does this:

```
sed -i 's/[aeiou]//g' story.txt
```


find: search for files

The `find` commands allows you to search for files based on specified properties

- entire directory trees, testing each file for the required property.
- takes actions for all matching files - default action is print the filename
- very useful as first stage of pipeline, but can specify operation as well

Invocation: **`find`** ***directories tests actions***

where

- the ***tests*** examine file properties like name, type, modification date
- the ***actions*** can be simply to print the name or execute an arbitrary command on the matched file

find: search for files - examples

```
# find all the HTML files below /home/z5234567/public_html
find  /home/z5234567/public_html -name '*.html'

# find all your files/dirs changed in the last 2 days
find ~ -mtime -2

# show info on files changed in the last 2 days
find ~ -mtime -2 -type f -exec ls -l {} \;

# show info on directories changed in the last week
find ~ -mtime -7 -type d -exec ls -ld {} \;

#find directories either new or  '07' in their name
find ~ -type d \( -name '*07*' -o -mtime -1 \)
```

Note: ~ above is shell syntax for your home directory

find: search for files - more examples

```
# find all new HTML files below ~/public_html
find ~/public_html -name '*.html' -mtime -1

# find background colours in my HTML files
find ~/public_html -name '*.html' -exec grep -H 'bgcolor' {} \;

# above could also be accomplished via ...
grep -r 'bgcolor' ~/public_html

# make sure that all HTML files are accessible
find ~/public_html -name '*.html' -exec chmod 644 {} \;

#remove any really old files ... Danger!
find /home/andrewt -type f -mtime +364 -exec rm {} \;
find /home/andrewt -type f -mtime +364 -ok rm {} \;
```


join: database operator (advanced)

`join` merges two files using the values in a field in each file as a common key.

The key field can be in a different position in each file, but the files must be ordered on that field. The default key field is 1.

Some useful **`join`** options:

`-1` *k*	key field in first file is *k*
`-2` *k*	key field in second file is *k*
`-a` *N*	print a line for each unpairable line in file *N* (1 or 2)
`-i`	ignore case
`-t` *c*	tab character is *c*

join: database operator

```
$ cat data1
Bugs Bunny      1953
Daffy Duck      1948
Donald Duck     1939
Goofy   1952
Mickey Mouse    1937
Nemo    2003
Road Runner     1949
```

```
$ cat data2
Warners Bugs Bunny
Warners Daffy Duck
Disney  Goofy
Disney  Mickey Mouse
Pixar   Nemo
```

```
$ join -t'    ' -2 2 -a 1 data1 data2
Bugs Bunny      1953    Warners
Daffy Duck      1948    Warners
Donald Duck     1939
Goofy   1952    Disney
Mickey Mouse    1937    Disney
Nemo    2003    Pixar
Road Runner     1949
```


paste: combine files

The `paste` command displays several text files "in parallel" on output.

If the inputs are files a, b, c

- the first line of output is composed of the first lines of a, b, c
- the second line of output is composed of the second lines of a, b, c

Lines from each file are separated by a tab character or specified delimiter(s).

If files are different lengths, output has all lines from longest file, with empty strings for missing lines.

Interleaves lines instead with `-s` (serial) option.

paste: combine files

Example: using **`paste`** to rebuild a file broken up by **`cut`**.

```
# assume "data" is a file with 3 tab-separated columns
cut -f1 data > data1
cut -f2 data > data2
cut -f3 data > data3
paste data1 data2 data3 > newdata

#"newdata" should look the same as "data"
```

tee: send copy of pipeline to file

Simple program but useful interactively and sometimes in scripts

```
$ echo Hello Andrew | tee copy.txt
Hello Andrew
$ cat copy.txt
Hello Andrew
$
```

- a useful debugging trick is **`tee /dev/tty`** to divert a copy of a pipeline to the terminal

xargs: run commands with arguments from standard input - advanced

Some useful **`xargs`** options:

-n*max-args*	use at most *max-args* arguments per command line
-P*max-procs*	Run up to *max-procs* processes at a time
-i*replace-str*	Replace occurrences of *replace-str* with words read from stdin

- For example:

```
# remove home directories of users named Andrew:
grep Andrew /etc/passwd | cut -d: -f6 | xargs rm -r

# run make in every sub-directory below /usr/src/
# with a Makefile, run up to 8 make's in parallel
find /usr/src -name Makefile | sed 's/Makefile//' | xargs -P8 -i@ make -C @
```

- see also **`parallel`**

xargs in Python

```
import subprocess
import sys
# the real xargs runs the command multiple times if input is large
# the real xargs treats quotes specially
def main():
    input_words = [w for line in sys.stdin for w in line.split()]
    command = sys.argv[1:]
    subprocess.run(command + input_words)
```

source code for xargs.py

process substitution (AKA named pipes) - advanced

- bash provides process substitution - interesting & useful, but bash-only
  - does not work with other shells
  - uses temporary named pipes
- syntax is **<(*command*)**
  - eg. **`diff <(sort file1) <(sort file2)`**
    - runs **`sort file1`** and **`sort file2`** then passes fake filenames to **`diff`** as arguments
- or **>(*command*)**
  - eg **`tar cf - somedir | tee >(shasum > dir.tgz.shasum) >(md5sum > dir.tgz.md5sum) > dir.tar`**
    - runs **`shasum`**, **`md5sum`**, and creates **`dir.tar`** all with the output of **`tar`**
  - most of the time pipes **|** are enough
- useful for commands which don't provide any way of reading from stdin or writing to stdout
  - but check if **-** as filename works
  - and also **`/dev/stdin`** **`/dev/stdout`** may be available
- useful to combine two pipelines

Filter summary by type

- *Horizontal slicing* - select subset of lines: **`cat`**, **`head`**, **`tail`**, **`grep`**, **`sed`**, **`uniq`**
- *Vertical slicing* - select subset of columns: **`cut`** , **`sed`**
- *Substitution*: **`tr`**, **`sed`**
- *Aggregation, simple statistics*: **`wc`**, **`uniq`**
- *Assembly* - combining data sources: **`paste`**, **`join`**
- *Reordering*: **`sort`**
- *Viewing* (always end of pipeline): **`more`**, **`less`**
- *File system search*: **`find`**
- *Programmable filters*: **`sed`** (also awk, `python`, `perl`, ...)