STAT 605
Data Science Computing

`grep` and regular expressions

Text data is ubiquitous

Examples:

Biostatistics (DNA/RNA/protein sequences)

Databases (e.g., census data, product inventory)

Log files (program names, IP addresses, user IDs, etc)

Medical records (case histories, doctors' notes, medication lists)

Social media (Facebook, twitter, etc)

How is text data stored?

Underlyingly, every file on your computer is just a string of bits…

0	1	1	0	0	0	1	1	0	1	1	0	0	0	0	1	0	1	1	1	0	1	0	0

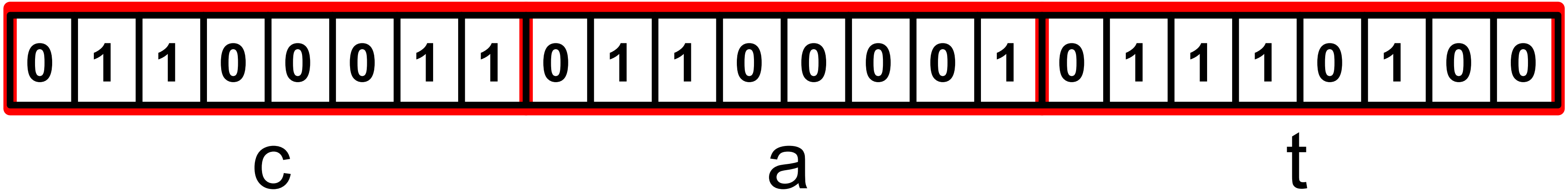
...which are broken up into (for example) bytes…

0 1 1 0 0 0 1 1	0 1 1 0 0 0 0 1	0 1 1 1 0 1 0 0

...which correspond to (in the case of text) characters.

0 1 1 0 0 0 1 1	0 1 1 0 0 0 0 1	0 1 1 1 0 1 0 0
c	a	t

How is text data stored?



Some encodings (e.g., UTF-8 and UTF-16) use “variable-length” encoding, in which different characters may use different numbers of bytes.

We’ll concentrate (today, at least) on ASCII, which uses fixed-length encodings.

ASCII (American Standard Code for Information Interchange)

8-bit* fixed-length encoding, file stored as stream of bytes

Each byte encodes a character

Letter, number, symbol or “special” characters (e.g., tabs, newlines, NULL)

Delimiter: one or more characters used to specify boundaries

Ex: space (`' '`, ASCII 32), tab (`'\t'`, ASCII 9), newline (`'\n'`, ASCII 10)

https://en.wikipedia.org/wiki/ASCII

*technically, each ASCII character is 7 bits, with the 8th bit reserved for error checking.

See https://en.wikipedia.org/wiki/Parity_bit

ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	*[NULL]*	32	20	*[SPACE]*	64	40	**@**	96	60	`
1	1	*[START OF HEADING]*	33	21	**!**	65	41	**A**	97	61	**a**
2	2	*[START OF TEXT]*	34	22	**"**	66	42	**B**	98	62	**b**
3	3	*[END OF TEXT]*	35	23	**#**	67	43	**C**	99	63	**c**
4	4	*[END OF TRANSMISSION]*	36	24	**$**	68	44	**D**	100	64	**d**
5	5	*[ENQUIRY]*	37	25	**%**	69	45	**E**	101	65	**e**
6	6	*[ACKNOWLEDGE]*	38	26	**&**	70	46	**F**	102	66	**f**
7	7	*[BELL]*	39	27	**'**	71	47	**G**	103	67	**g**
8	8	*[BACKSPACE]*	40	28	**(**	72	48	**H**	104	68	**h**
9	9	*[HORIZONTAL TAB]*	41	29	**)**	73	49	**I**	105	69	**i**
10	A	*[LINE FEED]*	42	2A	*	74	4A	**J**	106	6A	**j**
11	B	*[VERTICAL TAB]*	43	2B	**+**	75	4B	**K**	107	6B	**k**
12	C	*[FORM FEED]*	44	2C	**,**	76	4C	**L**	108	6C	**l**
13	D	*[CARRIAGE RETURN]*	45	2D	**-**	77	4D	**M**	109	6D	**m**
14	E	*[SHIFT OUT]*	46	2E	**.**	78	4E	**N**	110	6E	**n**
15	F	*[SHIFT IN]*	47	2F	**/**	79	4F	**O**	111	6F	**o**
16	10	*[DATA LINK ESCAPE]*	48	30	**0**	80	50	**P**	112	70	**p**
17	11	*[DEVICE CONTROL 1]*	49	31	**1**	81	51	**Q**	113	71	**q**
18	12	*[DEVICE CONTROL 2]*	50	32	**2**	82	52	**R**	114	72	**r**
19	13	*[DEVICE CONTROL 3]*	51	33	**3**	83	53	**S**	115	73	**s**
20	14	*[DEVICE CONTROL 4]*	52	34	**4**	84	54	**T**	116	74	**t**
21	15	*[NEGATIVE ACKNOWLEDGE]*	53	35	**5**	85	55	**U**	117	75	**u**
22	16	*[SYNCHRONOUS IDLE]*	54	36	**6**	86	56	**V**	118	76	**v**
23	17	*[ENG OF TRANS. BLOCK]*	55	37	**7**	87	57	**W**	119	77	**w**
24	18	*[CANCEL]*	56	38	**8**	88	58	**X**	120	78	**x**
25	19	*[END OF MEDIUM]*	57	39	**9**	89	59	**Y**	121	79	**y**
26	1A	*[SUBSTITUTE]*	58	3A	**:**	90	5A	**Z**	122	7A	**z**
27	1B	*[ESCAPE]*	59	3B	**;**	91	5B	**[**	123	7B	**{**
28	1C	*[FILE SEPARATOR]*	60	3C	**<**	92	5C	****	124	7C	**|**
29	1D	*[GROUP SEPARATOR]*	61	3D	**=**	93	5D	**]**	125	7D	**}**
30	1E	*[RECORD SEPARATOR]*	62	3E	**>**	94	5E	**^**	126	7E	**~**
31	1F	*[UNIT SEPARATOR]*	63	3F	**?**	95	5F	_	127	7F	*[DEL]*

Caution!

Different OSs follow slightly different conventions when saving text files!

Most common issue:

- UNIX/Linux/MacOS: newlines stored as `'\n'`
- DOS/Windows: stored as `'\r\n'` (carriage return, then newline)

When in doubt, use a tool like UNIX/Linux `xxd` (hexdump) to inspect raw bytes

`xxd` is also in MacOS; available in cygwin on Windows

Unicode

Universal encoding of (almost) all of the world’s writing systems

Each symbol is assigned a unique **code point**, a four- or five-digit hex number

- Unique number assigned to a given character U+XXXX
- ‘U+’ for unicode, XXXX is the code point (in hexadecimal)
- Example: 😎=U+1F60E, ∰=U+2230; http://www.unicode.org/ for more

Variable-length encoding

- UTF-8: 1 byte for first 128 code points, 2+ bytes for higher code points
- Result: ASCII is a subset of UTF-8

Most R files are ASCII; newer versions of Rstudio support unicode; newer versions of Python (i.e., 3+) encode scripts in unicode by default.

Matching text: regular expressions ("regexes")

Suppose I want to find all addresses in a big text document. How to do this?

Regexes describe sets of strings.

They allow concise specification for matching patterns in text

Specifics vary from one program to another (grep, vim, emacs, sed), but the basics that you learn in this course will generalize with minimal changes.

`grep`: pattern matching on the command line

`grep` takes two basic arguments:

1. A pattern to search for
2. A collection of text to search through

`grep` will look for the pattern and find everywhere it matches in the text

`grep <pattern> [filename]` searches for `pattern` in the file

Example: `grep goat example1.txt`

finds all instances of the string `goat` in the file `example1.txt`

Command line regexes: `grep`

```
keith@Steinhaus:~$ cat myfile.txt
hello world.
keith@Steinhaus:~$ grep 'hello' myfile.txt
hello world.
keith@Steinhaus:~$ grep 'goat' myfile.txt
keith@Steinhaus:~$
keith@Steinhaus:~$ cat myfile.txt | grep 'hello'
hello world.
keith@Steinhaus:~$ echo “Hello” | grep ‘hello’
keith@Steinhaus:~$
```

Searches for the string `hello` in the file `myfile.txt`, prints all matching lines to `stdout`.

String `goat` does not occur in `myfile.txt`, so no lines to print.

`grep` can also be made to search for a pattern in its `stdin`.

`grep` is case-sensitive by default. You can turn this off with the `-i` flag.

What about more complicated matches?

`grep` would not be very useful if all we could do is search for strings like ‘`dog`’

Power of regexes lies in specifying complicated patterns. Examples:

- Whitespace characters: `'\t'`, `'\n'`, `'\r'`
- Matching classes of characters (e.g., digits, whitespace, alphanumerics)
- Special characters: `. ^ $ * + ? { } [ ] \ | ( )`
  - We’ll discuss meaning of special characters shortly

Special characters must be **escaped** with backslash `'\'`

Ex: match a string containing the letter `x` followed by a period

```
keith@Steinhaus:~$ echo 'x.' | grep 'x\.'
x.
keith@Steinhaus:~$
```

Special characters: basics

Some characters have special meaning

These are: `. ^ $ * + ? { } [ ] \ | ( )`

We'll talk about some of these today; for others, see `man re_format`

Important: special characters must be escaped to match literally!

```
keith:~/regex_demo$ echo '$2' | grep '$2'
$2
keith:~/regex_demo$ echo '$2' | egrep '$2'
keith:~/regex_demo$ echo '$2' | egrep '\$2'
$2
keith:~/regex_demo$
```

We use `grep -E` or `egrep` ("extended grep") for these characters to have their special meanings

Without escaping, `$` is a special character that matches the end of a line. The escaped `\$` matches a literal `$`.

Special characters: sets and ranges

Can match “sets” of characters using square brackets:

- `‘[aeiou]’` matches any *one* of the characters `’a’,’e’,’i’,’o’,’u’`
- `‘[^aeiou]’` matches any *one* character **NOT** in the set.

```
keith:~/regex_demo$ echo 'cat' | grep 'c[aeiuo]t'
cat
keith:~/regex_demo$ echo 'cot' | grep 'c[aeiuo]t'
cot
keith:~/regex_demo$ echo 'cut' | grep 'c[aeiuo]t'
cut
keith:~/regex_demo$ echo 'cdt' | egrep 'c[aeiou]t'
keith:~/regex_demo$ echo 'cdt' | egrep 'c[^aeiou]t'
cdt
keith:~/regex_demo$
```

Special characters: sets and ranges

Can also match “ranges”:

- Ex: `'[a-z]'` matches lower case letters
  - Ranges calculated according to ASCII numbering
- Ex: `'[0-9A-Fa-f]'` will match any hexadecimal digit
- To match literal `'-'`, put it first or last (e.g. `'[-az]'`, `'[1-5-]'`)

```
keith:~/regex_demo$ echo 'a b c d' | grep '[a-d]'
a b c d
keith:~/regex_demo$ echo 'a b c d' | grep '[e-z]'
keith:~/regex_demo$ echo 'A1' | grep '[A-Z][0-9]'
A1
keith:~/regex_demo$ echo 'A1' | grep '[a-z][0-9]'
keith:~/regex_demo$ echo 'upper-case' | grep '[-xyz]case'
upper-case
keith:~/regex_demo$
```

Special characters: sets and ranges

Special characters lose special meaning inside square brackets:

- Ex: `'[(+\*)]'` will match any of `'('`, `'+'`, `'\'`, `'*'`, or `')'`
- To match `'^'` literal, make sure it **isn't** first: `'[(+*)^]'`

```
keith:~/regex_demo$ echo '2+2=4' | grep '[(+-)]'
2+2=4
keith:~/regex_demo$ echo '1=2' | grep '[(+-)]'
keith:~/regex_demo$ echo '\ is the escape character.' | grep '[\.,]'
\ is the escape character.
keith:~/regex_demo$ echo '2pi' | grep '[^a-z0-9]'
keith:~/regex_demo$ echo '2^7' | grep '[0-9][a-z^][0-9]'
2^7
keith:~/regex_demo$ echo 'e^pi' | grep '[0-9][a-z^][0-9]'
keith:~/regex_demo$
```

Special characters and sets

`'^'` : matches beginning of a line (i.e., matches "empty string" `''` at start of line)

`'$'` : matches end of a line (i.e., matches empty string before a newline)

`'.'` : wildcard, matches any character other than a newline

`'[[:space:]]'` : matches whitespace (spaces, tabs, newlines)

`'[[:digit:]]'` : matches a digit (0,1,2,3,4,5,6,7,8,9), equivalent to `[0-9]`

`'\w'` : matches a "word" character (number, letter or underscore '_')

`'\b'` : matches boundary between word (`'\w'`) and non-word characters

Example: beginning and end of lines, wildcards

```
keith:~$ echo 'bad' | egrep '^b.d$'
bad
keith:~$
```

'.' matches 'a', and start- and end-lines match correctly.

```
keith:~$ echo 'bid' | egrep '^b.d$'
bid
keith:~$
```

'.' matches 'i', and start- and end-lines match correctly.

```
keith:~$ echo 'bids' | egrep '^b.d$'
keith:~$
```

Matching fails because of 's' at end of string, which means that 'd' is not followed by end-of-line.

```
keith:~$ echo 'abad' | egrep '^b.d$'
keith:~$
```

Matching fails because of 'a' at start of string, which means that 'b' is not the start of the string.

Matching multiple substrings

Regexes may match multiple times on a single lines

`grep -o` prints each match on a separate lines.

```
keith:~$ echo 'goat goat bird goat' | grep 'goat'
goat goat bird goat
keith:~$ echo 'goat goat bird goat' | grep -o 'goat'
goat
goat
goat
keith:~$ echo '12345' | egrep -o '[[:digit:]][[:digit:]]'
12
34
keith:~$
```

Example: whitespace and boundaries

```
keith:~$ string1="c\ta t\ns\t";
keith:~$ echo -e "$string1" | egrep -o '[[:space:]]'



keith:~$ echo -e "$string1" | egrep -o '\s\b'


keith:~$
```

`'[[:space:]]'` matches any whitespace. That includes spaces, tabs and newlines.

...but `grep` searches each line of input, so the newline isn't matched-- it separates two lines.

The trailing tab in `string1` isn't matched, because it isn't followed by a whitespace-word boundary.

Reminder: `-e` flag tells `echo` to treat backslashed characters as special. So this prints the \t as a tab and the \n as a newline.

Character classes and complements

‘[[:space:]]’, equivalent to ‘\s’; complemented as ‘\S’ or ‘[^[:space]]’

‘[[:digit:]]’; complemented as ‘[^[:digit:]]’

‘\w’ complemented as ‘\W’ to match anything that **isn’t** alphanumeric or ‘_’

‘\b’ : complemented as ‘\B’ to match **NOT** at a word boundary

Character classes: complements

`'\S'` : complements `'\s'` (equivalent to `[[:space:]]`;
matches anything that **isn't** whitespace

```
keith:~$ echo -e "c\ta t\ns\t" | egrep -o "\S"
c
a
t
s
keith:~$ echo -e "c\ta t\ns\t" | egrep -o "[^[:space:]]"
c
a
t
s
keith:~$
```

`[[:space:]]` matches all whitespace characters (space, tab, newline, etc), so its complement matches everything else.

Character classes: complements

`‘[[:digit:]’` complemented as `‘[^[:digit:]]’`

```
keith:~$ echo -e 'a1 $25 2pi' | egrep -o "[[:digit:]]"
1
2
5
2
keith:~$ echo -e ‘a1 $25 2pi’ | egrep -o "[^[:digit:]]"
a


p
i
keith:~$
```

Important: we need single-quotes around the string, here. If we use double-quotes, bash interprets `$25` as “the value of the variable named 25”.

See https://www.gnu.org/software/bash/manual/html_node/Quoting.html#Quoting for more on string quoting.

Character classes: complements

'\w' : complemented as '\W'

```
keith:~$ echo 'a-b_2 $5.' | egrep -o '\w'
a
b
_
2
5
keith:~$ echo 'a-b_2 $5.' | egrep -o '\W'
-
 
$
.
keith:~$
```

'\w' matches alphanumerics and the underscore, so '\W' matches everything other than those characters.

Character classes: complements

'\b' : complemented as '\B'; matches **NOT** at a word boundary

```
keith:~$ echo 'Here is a surge of words' | egrep -o '\b[aeiou]\b'
a
keith:~$ echo 'Here is a surge of words' | egrep -o '\B[aeiou]\B'
e
u
o
keith:~$ echo 'Here is a surge of words' | egrep '\B[aeiou]\B'
Here is a surge of words
keith:~$
```

'\b' and '\B' are a bit tricky-- they match the empty string, '', **between** two other characters.

Matching and repetition

- `'*'` : zero or more of the previous item
- `'+'` : one or more of the previous item
- `'?'` : zero or one of the previous item

`'a*'` matches 0 or more instances of `'a'`, so it match the empty string between `c` and `t` in `'cat'`.

```
keith:~$ echo 'ct cat caat' | egrep -o 'ca*t'
ct
cat
caat
keith:~$ echo 'ct cat caat' | egrep -o 'ca+t'
cat
caat
keith:~$ echo 'ct cat caat' | egrep -o 'ca?t'
ct
cat
keith:~$
```

Matching and repetition

`'{4}'` : exactly four of the previous item

`'{3,}'` : three or more of previous item

`'{2,5}'` : between two and five (inclusive) of previous item

```
keith:~$ echo 'ct cat caat caaat' | egrep -o 'ca{2}t'
caat
keith:~$ echo 'ct cat caat caaat' | egrep -o 'ca{2,}t'
caat
caaat
keith:~$ echo 'ct cat caat caaat' | egrep -o 'ca{1,2}t'
cat
Caat
keith:~$
```

Test your understanding

Which of the following will match `’^[[:digit:]]{2,4}\s’`?

`‘7 a1’`

`‘747 Boeing’`

`‘C7777 C7778’`

`‘12345 ’`

`‘1234\tqq’`

`‘Boeing 747’`

Test your understanding

Which of the following will match ‘^[:digit:]]{2,4}\s’?

‘7 a1’

‘747 Boeing’

‘C7777 C7778’

‘12345 ’

‘1234\tqq’

‘Boeing 747’

Test your understanding

Which of the following will match ’^[[:digit:]]{2,4}\s’?

‘7 a1’

‘747 Boeing’

‘C7777 C7778’

‘12345 ’

‘1234\tqq’

‘Boeing 747’

Test your understanding

Which of the following will match `'^[[:digit:]]{2,4}\s'`?

`'7 a1'`

`'747 Boeing'`

`'C7777 C7778'`

`'12345 '`

`'1234\tqq'`

`'Boeing 747'`

Test your understanding

Which of the following will match '^[[:digit:]]{2,4}\s'?

'7 a1'

'747 Boeing'

'C7777 C7778'

'12345 '

'1234\tqq'

'Boeing 747'

Test your understanding

Which of the following will match `'^[[:digit:]]{2,4}\s'`?

`'7 a1'`

`'747 Boeing'`

`'C7777 C7778'`

`'12345 '`

`'1234\tqq'`

`'Boeing 747'`

Test your understanding

Which of the following will match `'^[[:digit:]]{2,4}\s'`?

`'7 a1'`

`'747 Boeing'`

`'C7777 C7778'`

`'12345 '`

`'1234\tqq'`

`'Boeing 747'`

Test your understanding

Which of the following will match `r’^[[:digit:]]{2,4}\s’`?

`‘7 a1’`

`‘747 Boeing’`

`‘C7777 C7778’`

`‘12345 ’`

`‘1234\tqq’`

`‘Boeing 747’`

Or clauses: |

`|` ("pipe") is a special character that allows one to specify "or" clauses

Example: I want to match the word "cat" ***or*** the word "dog"

Solution: `(cat|dog)`

Note: parentheses are not strictly necessary here, but parentheses tend to make for easier reading and avoid possible ambiguity. It's a good habit to just use them.

```
keith:~$ echo "cat" | egrep '(cat|dog)'
cat
keith:~$ echo "dog" | egrep '(cat|dog)'
dog
keith:~$ echo "goat" | egrep '(cat|dog)'
keith:~$
```

Or clauses: `|` is greedy!

What happens when an expression using pipe can match many different ways?

What's going on here?!

```
keith:~$ echo "aaaa" | egrep -o "a|aa|aaa"
aaa
a
keith:~$
```

Matching with `'|'` is ***greedy***

Tries to match as much of the string as possible with the regex.

When it cannot make a longer match, it returns the match…

...and starts trying to make another.

Note: this behavior can be changed using flags. Refer to the documentation.

Matching, greediness and laziness

The opposite of greedy matching is lazy matching

Perl regexes (a slight variant of the egrep regexes), can be made lazy with `?`

```
keith:~$ echo "aaaa" | egrep -o "a+"
aaaa
keith:~$ echo "aaaa" | grep -P -o "a+?"
a
a
a
a
keith:~$
```

`'a+'` gobbles up the whole string, because the regex is greedy by default.

In Perl regexes, `'?'` modifies operators like `'+'` and `'*'` to **not** be greedy, and we get lazy matching.

`-P` flag tells grep to use Perl regexes (very similar to extended `grep`).

Backreferences

Can refer to an earlier match ***within the same regex!***

- A **group** is a portion of the regular expression inside parentheses
- `'\N'`, where N is a number, references the N-th group

Example: find strings of the form `'X X'`, where X is any non-whitespace string.

```
keith:~$ echo "cat cat" | egrep "([^[:space:]]+) \1"
cat cat
keith:~$ echo "cat dog" | egrep "([^[:space:]]+) \1"
keith:~$
```

`[^[:space:]]+` matches `'cat'`. Then `\1` attempts to match a second copy of the same string that `^[:space:]]+` matched.

Backreferences

Backrefs allows very complicated pattern matching!

Test your understanding:

Describe what strings `‘([[:digit:]]+)([A-Z]+):\1+\2’` matches

What about `‘([a-zA-Z]+).*\1’`?

Backreferences

Backrefs allows very complicated pattern matching!

Test your understanding:

Describe what strings `‘([[:digit:]]+)([A-Z]+):\1+\2’` matches

What about `‘([a-zA-Z]+).*\1’`?

`‘([[:digit:]]+)([A-Z]+):\1+\2’`

Matches strings of the form `XY:X+Y`, where `X` is a string of one or more digits, and `Y` is a string of more than one capital letters.

Backreferences

Backrefs allows very complicated pattern matching!

Test your understanding:

Describe what strings `‘([[:digit:]]+)([A-Z]+):\1+\2’` matches

What about `‘([a-zA-Z]+).*\1’`?

`‘([[:digit:]]+)([A-Z]+):\1+\2’`

Matches strings of the form `XY:X+Y`, where `X` is a string of one or more digits, and `Y` is a string of more than one capital letters.

`‘([a-zA-Z]+).*\1’`

Matches strings of the form `XYX`, where `X` is a string of one or more letters, And `Y` is a string of zero or more characters (other than newlines)

Backreferences

Backrefs allows very complicated pattern matching!

Test your understanding:

Describe what strings `‘([[:digit:]]+)([A-Z]+):\1+\2’` matches?

What about `‘([a-zA-Z]+).*\1’`?

Tougher question:

Is it possible to write a regular expression that matches palindromes?

Answer: Strictly speaking, no. https://en.wikipedia.org/wiki/Regular_language

Better answer: ...but if your matcher provides enough bells and whistles...

Backreferences

All that being said…

> **From `man regex`:** Back references are a dreadful botch, posing major problems for efficient implementations. They are also somewhat vaguely defined (does "a\(\(b\)*\2\)*d" match "abbbd"?). Avoid using them.

I wouldn’t go so far as to say “never use backrefs”...

... but they can cause confusion, so write them carefully!

Debugging

When in doubt, test your regexes!

- A bit of googling will find you lots of tools for doing this…
- ...or you can just do your testing directly on the command line

Try to come up with string examples that should and shouldn’t match your regex

Don’t forget the edge cases

- Should the empty string match your pattern?
- What about a string of length one?

Generating strings: brace expansions

We can use similar ideas to those in regexes to generate sequences of strings

String list: generates a sequence of strings

`{first_string,second_string,third_string}`

expands into `first_string second_string third_string`

```
keith:~$ echo {'string1','string2','string3'}
string1 string2 string3
keith:~$ echo "hello "{'Alice','Bob','Carol'}", nice to meet you."
hello Alice, nice to meet you. hello Bob, nice to meet you. hello Carol,
nice to meet you.
keith:~$ echo {A,B,C}{1,2}
A1 A2 B1 B2 C1 C2
keith:~$ echo {A,B,C}{1,2,{x,y,z}}
A1 A2 Ax Ay Az B1 B2 Bx By Bz C1 C2 Cx Cy Cz
keith:~$
```

We can nest brace expansions. The rules for expansion remain the same, but apply recursively.

Generating strings: brace expansions

Range list: generates a sequence of ordered strings

`{a..b}` expands into a sequence from `a` to `b` separated by spaces

```
keith:~$ echo {1..12}
1 2 3 4 5 6 7 8 9 10 11 12
keith:~$ echo {a..h}
a b c d e f g h
keith:~$ echo {10..1}
10 9 8 7 6 5 4 3 2 1
keith:~$ echo {A..F}{1,2,3}
A1 A2 A3 B1 B2 B3 C1 C2 C3 D1 D2 D3 E1 E2 E3 F1 F2 F3
keith:~$
```

We can create sequences of numbers, letters, any characters, really-- `bash` does its best to figure out what we meant.

Using brace expansions to describe files

Example: `cat u{wisc,mich,mass}.txt`

will `cat` the files `uwisc.txt`, `umich.txt` and `umass.txt`

```
keith:~$ ls
keith:~$ for f in `echo "1 2 3 4 5"`; do echo "$f" > ${f}.txt; done
keith:~$ ls
1.txt  2.txt  3.txt  4.txt  5.txt
keith:~$ ls [2-4].txt
2.txt  3.txt  4.txt
keith:~$ rm {1,3,5}.txt
keith:~$ ls
2.txt  4.txt
keith:~$
```

Curly braces around the variable name avoid ambiguity in string concatenation.

Notation very similar to the regex set notation is available on the command line.

We can also use regex-like patterns

Example: `ls my_dir/*.txt` will list all `.txt` files in the directory `my_dir`

Exercises

1) Write a regular expression that matches all strings consisting only of letters (either upper or lower-case) and that start and end with a vowel (though not necessarily the same vowel), with one or more consonants in between. For the purposes of this question, a vowel is one of a, e, i, o and u (whether upper or lower case) and a consonant is any other letter except y (so y is neither a consonant nor a vowel).
2) Write a shell script that counts how many numbers from 1000 to 5000 inclusive consist entirely of the digits {1,2,3}. **Hint:** use a brace expansion to enumerate 1000 to 5000, pipe it to `grep`, then pipe it to `wc`.