School of
Science

Scripting Languages

Module 5

Regular Expressions,
Redirection, Piping and Grep

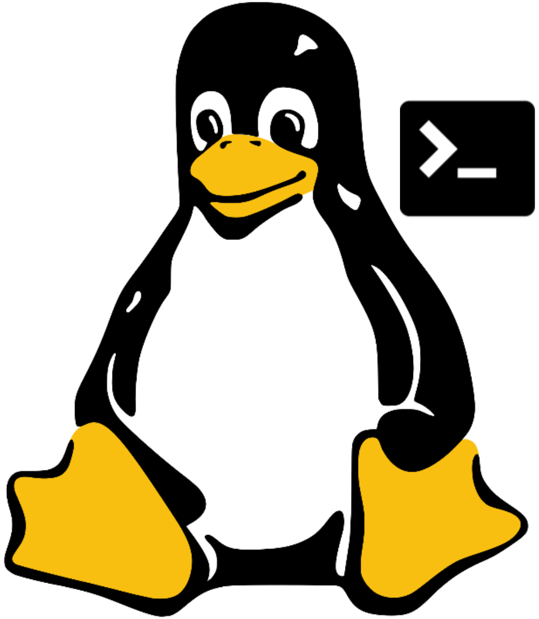
Contents

- Regular Expressions
- Regular Expression Engines
- Grep and Regex
- Anchors and Wildcards
- Extended RegEx Engine
- ERE Repetition and Optionality
- OR and Expression Grouping
- Common Grep Options
- Piping and Redirection

Learning Objectives

After completing this module, you should be able to work with:

- Regular Expressions
- Regular Expression Engines
- Grep and Regex
- Anchors and Wildcards
- Extended RegEx Engine
- ERE Repetition and Optionality
- OR and Expression Grouping
- Common Grep Options
- Piping and Redirection



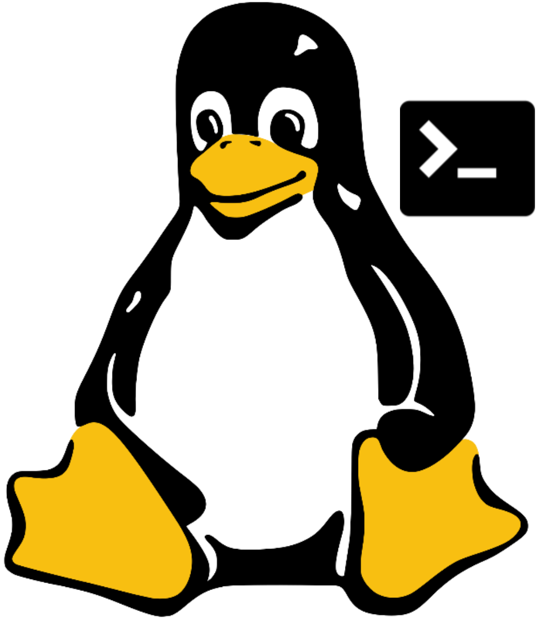
Regular Expressions

Regular Expressions

- Regular Expressions, more commonly referred to as **regex**, are used to match patterns in text
- A regex text pattern is provided to a *regex engine* to allow it to find a match
- Once a match is found, other commands and utilities can then be called upon to interact with the matched data in some way

Regex engines

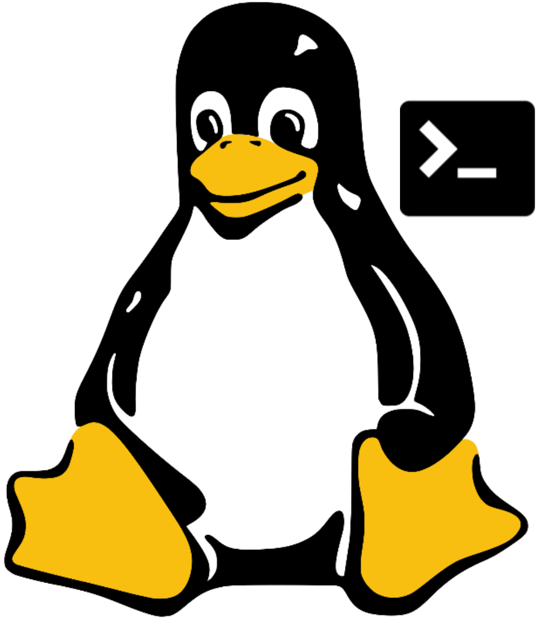
- There are two regular expression engines supported by bash commands
  - Basic Regular Expression Engine (BRE)
  - Extended Regular Expression Engine (ERE)
- These are supported by several commands and utilities, in particular **grep**, **sed** and **awk**
- **BRE** and **ERE** are also regularly used for data validation purposes as well
- The remainder of this module will examine *regex* as used with **grep**



grep

What is grep?

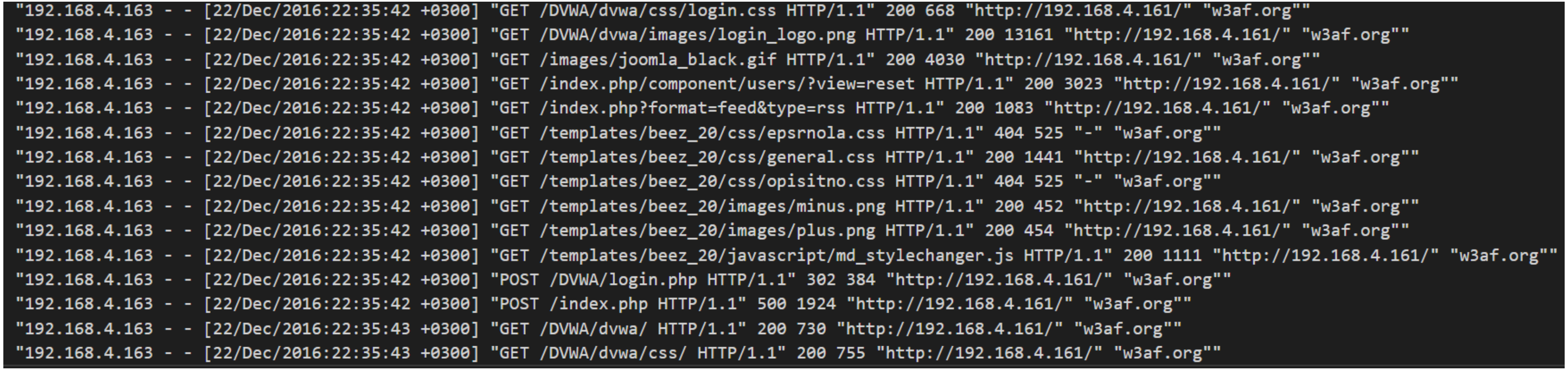
- **grep** stands for **G**lobal **R**egular **E**xpression **P**rint
- grep searches through stipulated input files for lines containing a match to a regex pattern
- When a match is found in a line, grep copies the line to standard output by default, or another output if stipulated by options or piping
- grep is highly integrated with BRE and ERE to perform the tasks it is designed to do



grep and regex

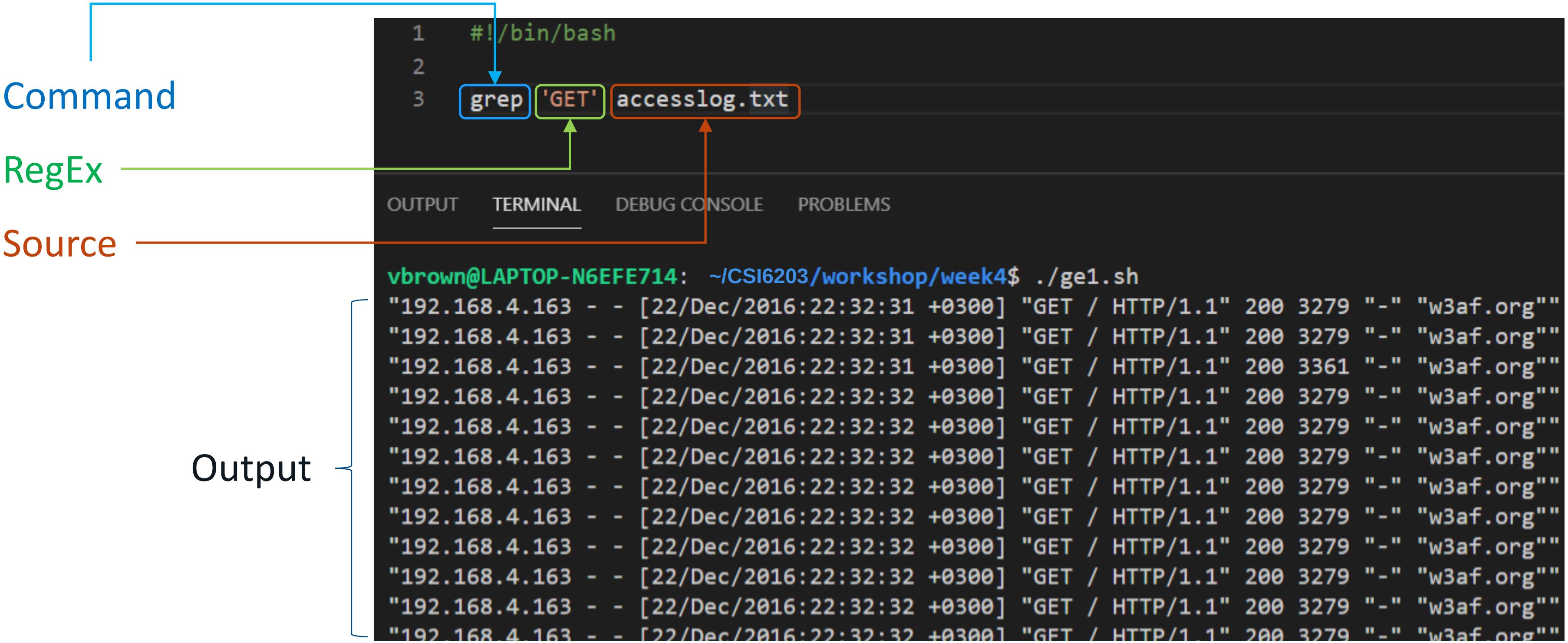
Sample Data Set

- The following grep and regex examples are based on an access log data set acquired from https://github.com/ocatak/apache-http-logs/blob/master/w3af.txt (06/07/2020)



```
"192.168.4.163 - - [22/Dec/2016:22:35:42 +0300] "GET /DVWA/dvwa/css/login.css HTTP/1.1" 200 668 "http://192.168.4.161/" "w3af.org""
"192.168.4.163 - - [22/Dec/2016:22:35:42 +0300] "GET /DVWA/dvwa/images/login_logo.png HTTP/1.1" 200 13161 "http://192.168.4.161/" "w3af.org""
"192.168.4.163 - - [22/Dec/2016:22:35:42 +0300] "GET /images/joomla_black.gif HTTP/1.1" 200 4030 "http://192.168.4.161/" "w3af.org""
"192.168.4.163 - - [22/Dec/2016:22:35:42 +0300] "GET /index.php/component/users/?view=reset HTTP/1.1" 200 3023 "http://192.168.4.161/" "w3af.org""
"192.168.4.163 - - [22/Dec/2016:22:35:42 +0300] "GET /index.php?format=feed&type=rss HTTP/1.1" 200 1083 "http://192.168.4.161/" "w3af.org""
"192.168.4.163 - - [22/Dec/2016:22:35:42 +0300] "GET /templates/beez_20/css/epsrnola.css HTTP/1.1" 404 525 "-" "w3af.org""
"192.168.4.163 - - [22/Dec/2016:22:35:42 +0300] "GET /templates/beez_20/css/general.css HTTP/1.1" 200 1441 "http://192.168.4.161/" "w3af.org""
"192.168.4.163 - - [22/Dec/2016:22:35:42 +0300] "GET /templates/beez_20/css/opisitno.css HTTP/1.1" 404 525 "-" "w3af.org""
"192.168.4.163 - - [22/Dec/2016:22:35:42 +0300] "GET /templates/beez_20/images/minus.png HTTP/1.1" 200 452 "http://192.168.4.161/" "w3af.org""
"192.168.4.163 - - [22/Dec/2016:22:35:42 +0300] "GET /templates/beez_20/images/plus.png HTTP/1.1" 200 454 "http://192.168.4.161/" "w3af.org""
"192.168.4.163 - - [22/Dec/2016:22:35:42 +0300] "GET /templates/beez_20/javascript/md_stylechanger.js HTTP/1.1" 200 1111 "http://192.168.4.161/" "w3af.org""
"192.168.4.163 - - [22/Dec/2016:22:35:42 +0300] "POST /DVWA/login.php HTTP/1.1" 302 384 "http://192.168.4.161/" "w3af.org""
"192.168.4.163 - - [22/Dec/2016:22:35:42 +0300] "POST /index.php HTTP/1.1" 500 1924 "http://192.168.4.161/" "w3af.org""
"192.168.4.163 - - [22/Dec/2016:22:35:43 +0300] "GET /DVWA/dvwa/ HTTP/1.1" 200 730 "http://192.168.4.161/" "w3af.org""
"192.168.4.163 - - [22/Dec/2016:22:35:43 +0300] "GET /DVWA/dvwa/css/ HTTP/1.1" 200 755 "http://192.168.4.161/" "w3af.org""
```

Simple BRE matching



Command

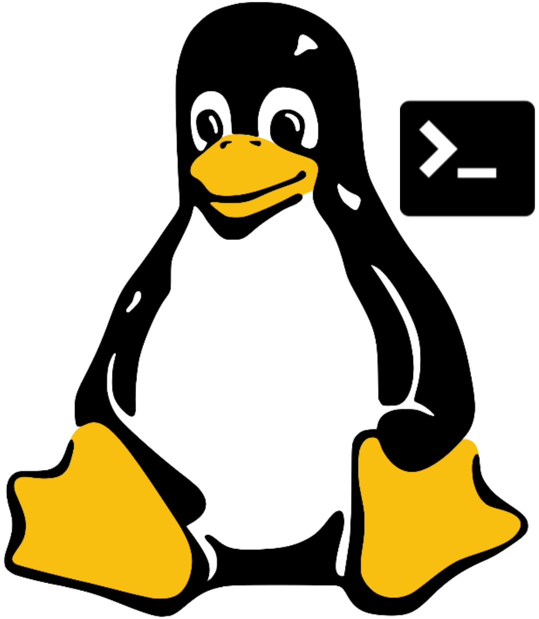
RegEx

Source

Output

```
#!/bin/bash

grep 'GET' accesslog.txt
```

```
vbrown@LAPTOP-N6EFE714: ~/CSI6203/workshop/week4$ ./ge1.sh
"192.168.4.163 - - [22/Dec/2016:22:32:31 +0300] "GET / HTTP/1.1" 200 3279 "-" "w3af.org""
"192.168.4.163 - - [22/Dec/2016:22:32:31 +0300] "GET / HTTP/1.1" 200 3279 "-" "w3af.org""
"192.168.4.163 - - [22/Dec/2016:22:32:31 +0300] "GET / HTTP/1.1" 200 3361 "-" "w3af.org""
"192.168.4.163 - - [22/Dec/2016:22:32:32 +0300] "GET / HTTP/1.1" 200 3279 "-" "w3af.org""
"192.168.4.163 - - [22/Dec/2016:22:32:32 +0300] "GET / HTTP/1.1" 200 3279 "-" "w3af.org""
"192.168.4.163 - - [22/Dec/2016:22:32:32 +0300] "GET / HTTP/1.1" 200 3279 "-" "w3af.org""
"192.168.4.163 - - [22/Dec/2016:22:32:32 +0300] "GET / HTTP/1.1" 200 3279 "-" "w3af.org""
"192.168.4.163 - - [22/Dec/2016:22:32:32 +0300] "GET / HTTP/1.1" 200 3279 "-" "w3af.org""
"192.168.4.163 - - [22/Dec/2016:22:32:32 +0300] "GET / HTTP/1.1" 200 3279 "-" "w3af.org""
"192.168.4.163 - - [22/Dec/2016:22:32:32 +0300] "GET / HTTP/1.1" 200 3279 "-" "w3af.org""
"192.168.4.163 - - [22/Dec/2016:22:32:32 +0300] "GET / HTTP/1.1" 200 3279 "-" "w3af.org""
"192.168.4.163 - - [22/Dec/2016:22:32:32 +0300] "GET / HTTP/1.1" 200 3279 "-" "w3af.org""
```



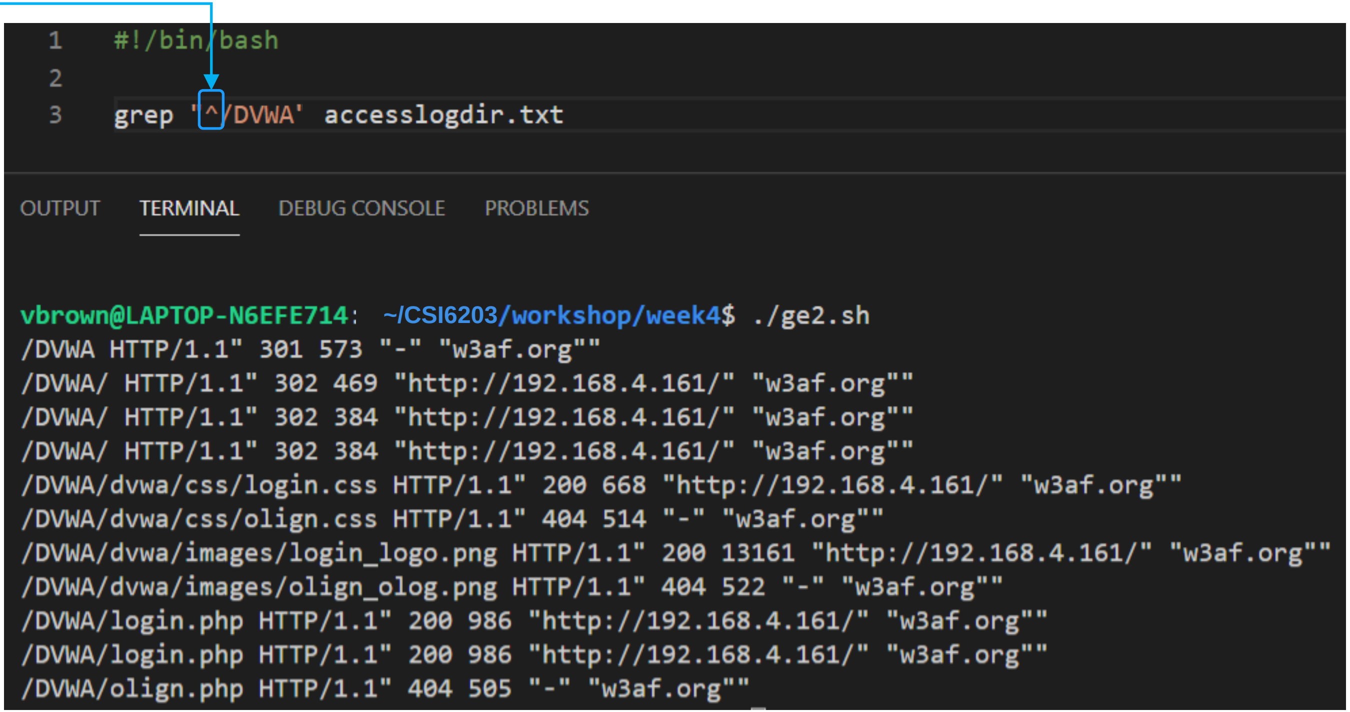
Anchors and Wildcards

Anchor characters

- Regex patterns can use special characters called anchor points to represent specific locations within the text
- The two most common anchor points are
  - The start of the line ‘**^**’
    (The ^ symbol is the circum accent, or circum for short)
  - The end of the line ‘**$**’

Start of line anchor ^

start of line (circum) anchor



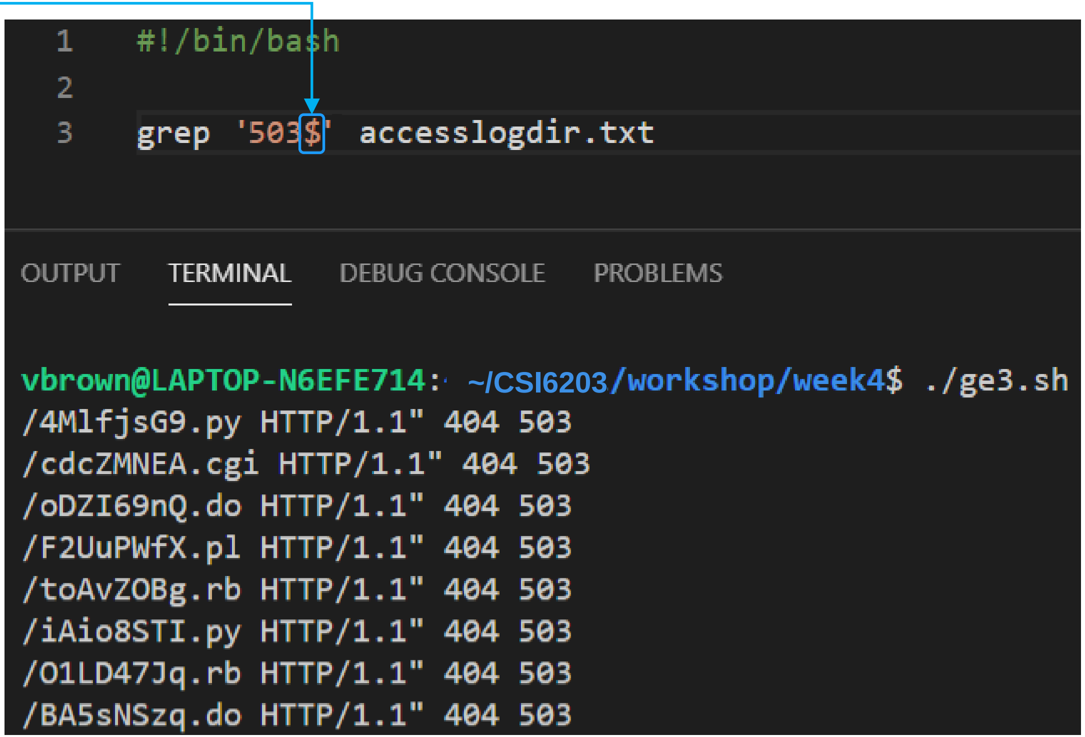
```
#!/bin/bash

grep '^/DVWA'  accesslogdir.txt
```

Output

```
vbrown@LAPTOP-N6EFE714: ~/CSI6203/workshop/week4$ ./ge2.sh
/DVWA HTTP/1.1" 301 573 "-" "w3af.org""
/DVWA/ HTTP/1.1" 302 469 "http://192.168.4.161/" "w3af.org""
/DVWA/ HTTP/1.1" 302 384 "http://192.168.4.161/" "w3af.org""
/DVWA/ HTTP/1.1" 302 384 "http://192.168.4.161/" "w3af.org""
/DVWA/dvwa/css/login.css HTTP/1.1" 200 668 "http://192.168.4.161/" "w3af.org""
/DVWA/dvwa/css/olign.css HTTP/1.1" 404 514 "-" "w3af.org""
/DVWA/dvwa/images/login_logo.png HTTP/1.1" 200 13161 "http://192.168.4.161/" "w3af.org""
/DVWA/dvwa/images/olign_olog.png HTTP/1.1" 404 522 "-" "w3af.org""
/DVWA/login.php HTTP/1.1" 200 986 "http://192.168.4.161/" "w3af.org""
/DVWA/login.php HTTP/1.1" 200 986 "http://192.168.4.161/" "w3af.org""
/DVWA/olign.php HTTP/1.1" 404 505 "-" "w3af.org""
```

End of line anchor $

end of line ($)
anchor



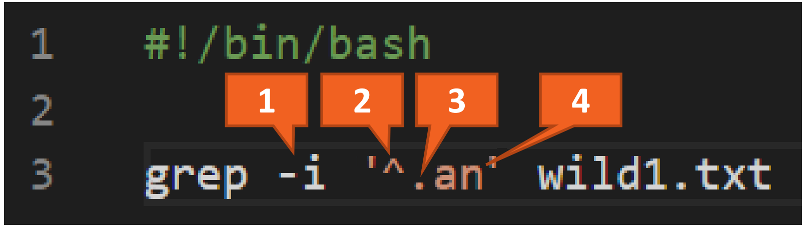
```
#!/bin/bash

grep '503$' accesslogdir.txt
```

```
vbrown@LAPTOP-N6EFE714: ~/CSI6203/workshop/week4$ ./ge3.sh
/4MlfjsG9.py HTTP/1.1" 404 503
/cdcZMNEA.cgi HTTP/1.1" 404 503
/oDZI69nQ.do HTTP/1.1" 404 503
/F2UuPWfX.pl HTTP/1.1" 404 503
/toAvZOBg.rb HTTP/1.1" 404 503
/iAio8STI.py HTTP/1.1" 404 503
/O1LD47Jq.rb HTTP/1.1" 404 503
/BA5sNSzq.do HTTP/1.1" 404 503
```

Output

Wildcard characters

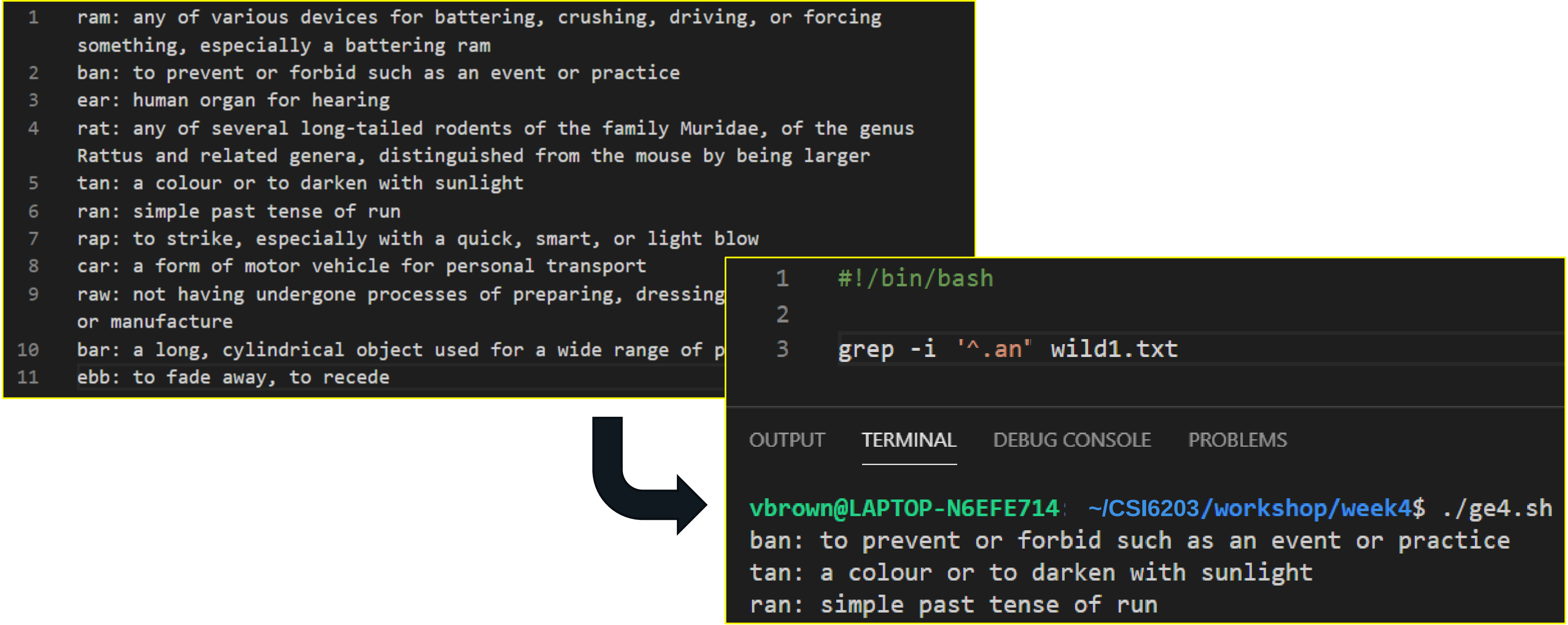
- Wildcards are characters that could match a range of characters
- In regex, the most common wildcard is dot ‘.’
- The *dot* character can be used to represent any character, e.g. find lines that start with a string ending in *an*



```
#!/bin/bash

grep -i '^.an' wild1.txt
```

1. Make match case insensitive
2. Must occur at start of line
3. Any character acceptable
4. String must end in ***an***

Wildcard characters



```
ram: any of various devices for battering, crushing, driving, or forcing
something, especially a battering ram
ban: to prevent or forbid such as an event or practice
ear: human organ for hearing
rat: any of several long-tailed rodents of the family Muridae, of the genus
Rattus and related genera, distinguished from the mouse by being larger
tan: a colour or to darken with sunlight
ran: simple past tense of run
rap: to strike, especially with a quick, smart, or light blow
car: a form of motor vehicle for personal transport
raw: not having undergone processes of preparing, dressing
or manufacture
bar: a long, cylindrical object used for a wide range of p
ebb: to fade away, to recede
```

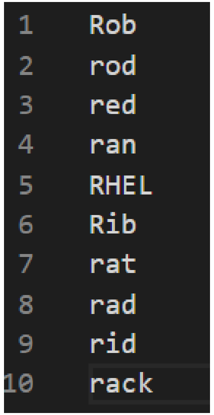
```bash
#!/bin/bash

grep -i '^.an' wild1.txt
```

```
vbrown@LAPTOP-N6EFE714: ~/CSI6203/workshop/week4$ ./ge4.sh
ban: to prevent or forbid such as an event or practice
tan: a colour or to darken with sunlight
ran: simple past tense of run
```

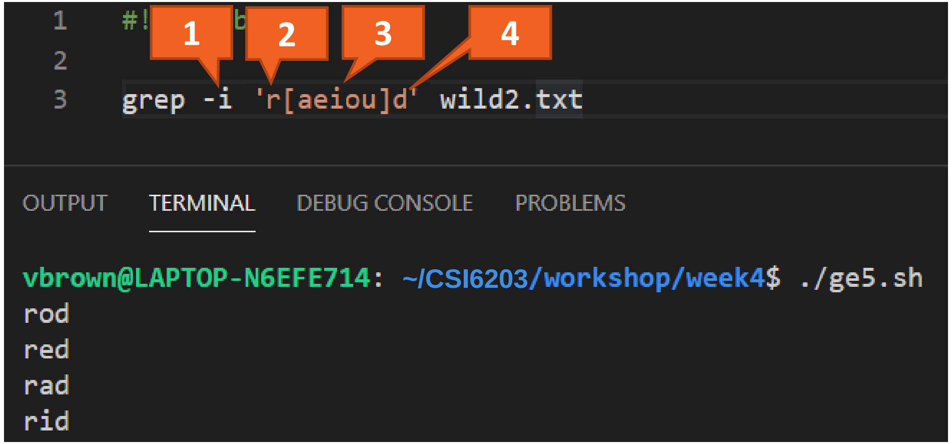
Classed wildcards

- Square brackets [ ] are used to restrict a wildcard to be only one of a set of values

- In this example, find lines contain a string starting with R/r followed by any single instance of a *vowel* and ending with **d**



```
Rob
rod
red
ran
RHEL
Rib
rat
rad
rid
rack
```



```
#!

grep -i 'r[aeiou]d'  wild2.txt
```

```
vbrown@LAPTOP-N6EFE714: ~/CSI6203/workshop/week4$ ./ge5.sh
rod
red
rad
rid
```

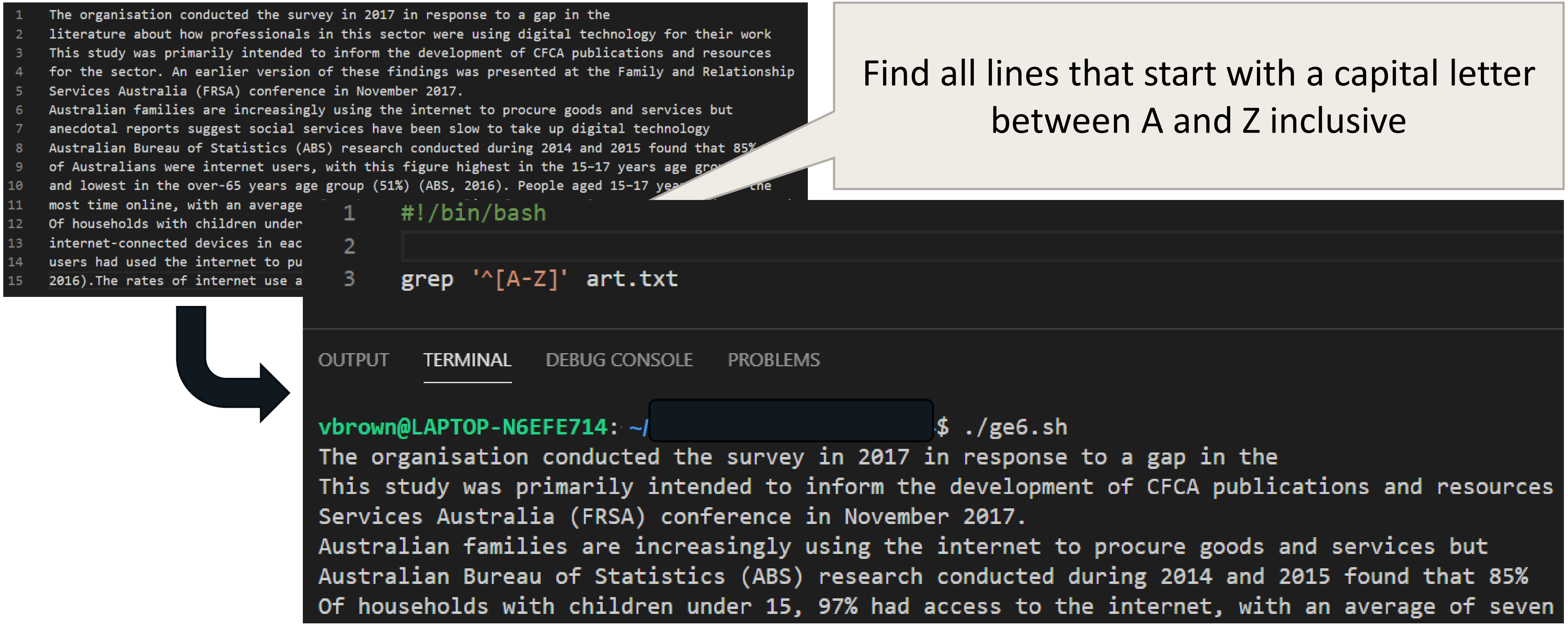
1. Make match case insensitive
2. String must start with R/r
3. Followed by any single vowel instance
4. String must end in a ***d***

Specify allowable range with [ ]

- Square brackets **[ ]** can also specify a range of allowable potential characters
- A string example would be `grep “[A-Z]” text.txt`, i.e look for lines that contain capital letters from A to Z inclusive
- A numeric example would be `grep “[0-9]” text.txt`, i.e look for lines that contain a number

School of
Science

[ ] example - string



```
The organisation conducted the survey in 2017 in response to a gap in the
literature about how professionals in this sector were using digital technology for their work
This study was primarily intended to inform the development of CFCA publications and resources
for the sector. An earlier version of these findings was presented at the Family and Relationship
Services Australia (FRSA) conference in November 2017.
Australian families are increasingly using the internet to procure goods and services but
anecdotal reports suggest social services have been slow to take up digital technology
Australian Bureau of Statistics (ABS) research conducted during 2014 and 2015 found that 85%
of Australians were internet users, with this figure highest in the 15–17 years age gro
and lowest in the over-65 years age group (51%) (ABS, 2016). People aged 15–17 yea      the
most time online, with an average
Of households with children under
internet-connected devices in eac
users had used the internet to pu
2016).The rates of internet use a
```

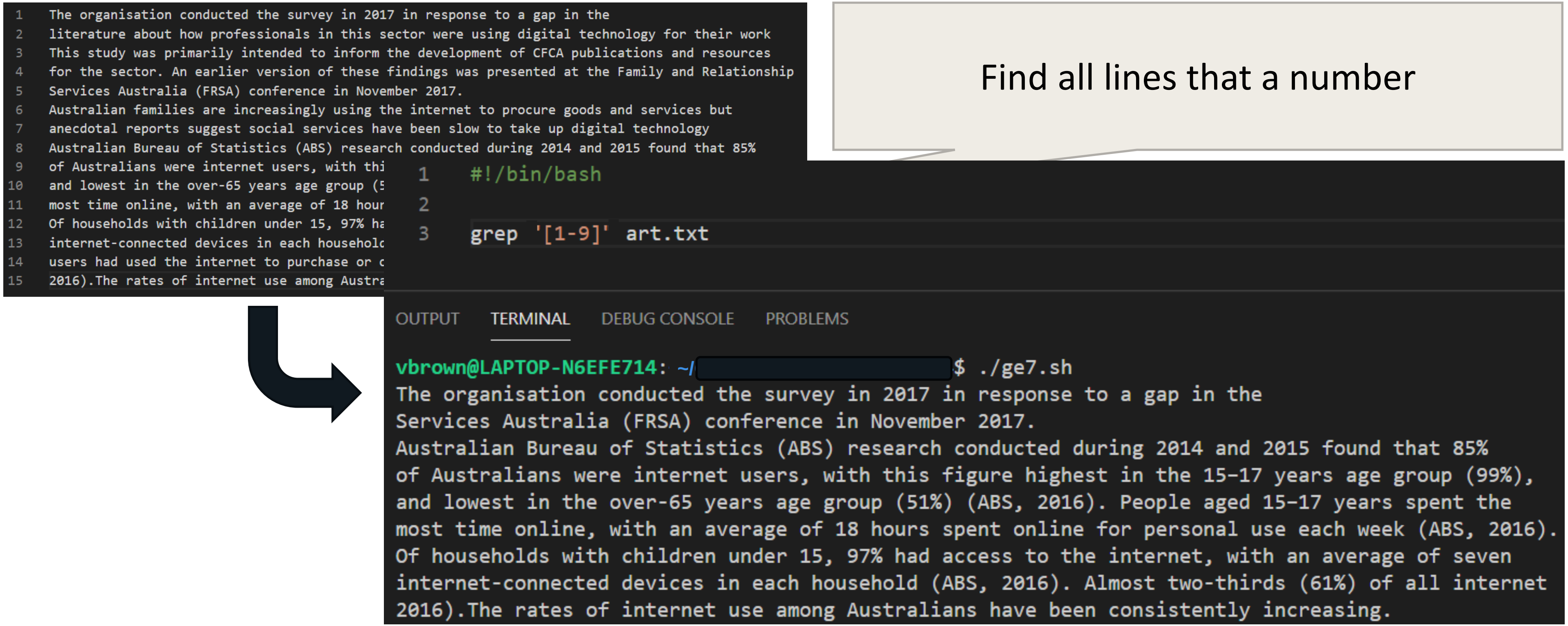
```
#!/bin/bash

grep '^[A-Z]' art.txt
```

OUTPUT   TERMINAL   DEBUG CONSOLE   PROBLEMS

```
vbrown@LAPTOP-N6EFE714:~/[redacted]$ ./ge6.sh
The organisation conducted the survey in 2017 in response to a gap in the
This study was primarily intended to inform the development of CFCA publications and resources
Services Australia (FRSA) conference in November 2017.
Australian families are increasingly using the internet to procure goods and services but
Australian Bureau of Statistics (ABS) research conducted during 2014 and 2015 found that 85%
Of households with children under 15, 97% had access to the internet, with an average of seven
```

Find all lines that start with a capital letter between A and Z inclusive

[ ] example - numeric



```
The organisation conducted the survey in 2017 in response to a gap in the
literature about how professionals in this sector were using digital technology for their work
This study was primarily intended to inform the development of CFCA publications and resources
for the sector. An earlier version of these findings was presented at the Family and Relationship
Services Australia (FRSA) conference in November 2017.
Australian families are increasingly using the internet to procure goods and services but
anecdotal reports suggest social services have been slow to take up digital technology
Australian Bureau of Statistics (ABS) research conducted during 2014 and 2015 found that 85%
of Australians were internet users, with thi
and lowest in the over-65 years age group (5
most time online, with an average of 18 hour
Of households with children under 15, 97% ha
internet-connected devices in each househol
users had used the internet to purchase or c
2016).The rates of internet use among Austra
```

Find all lines that a number

```bash
#!/bin/bash

grep '[1-9]' art.txt
```

```
vbrown@LAPTOP-N6EFE714: ~/[redacted]$ ./ge7.sh
The organisation conducted the survey in 2017 in response to a gap in the
Services Australia (FRSA) conference in November 2017.
Australian Bureau of Statistics (ABS) research conducted during 2014 and 2015 found that 85%
of Australians were internet users, with this figure highest in the 15–17 years age group (99%),
and lowest in the over-65 years age group (51%) (ABS, 2016). People aged 15–17 years spent the
most time online, with an average of 18 hours spent online for personal use each week (ABS, 2016).
Of households with children under 15, 97% had access to the internet, with an average of seven
internet-connected devices in each household (ABS, 2016). Almost two-thirds (61%) of all internet
2016).The rates of internet use among Australians have been consistently increasing.
```

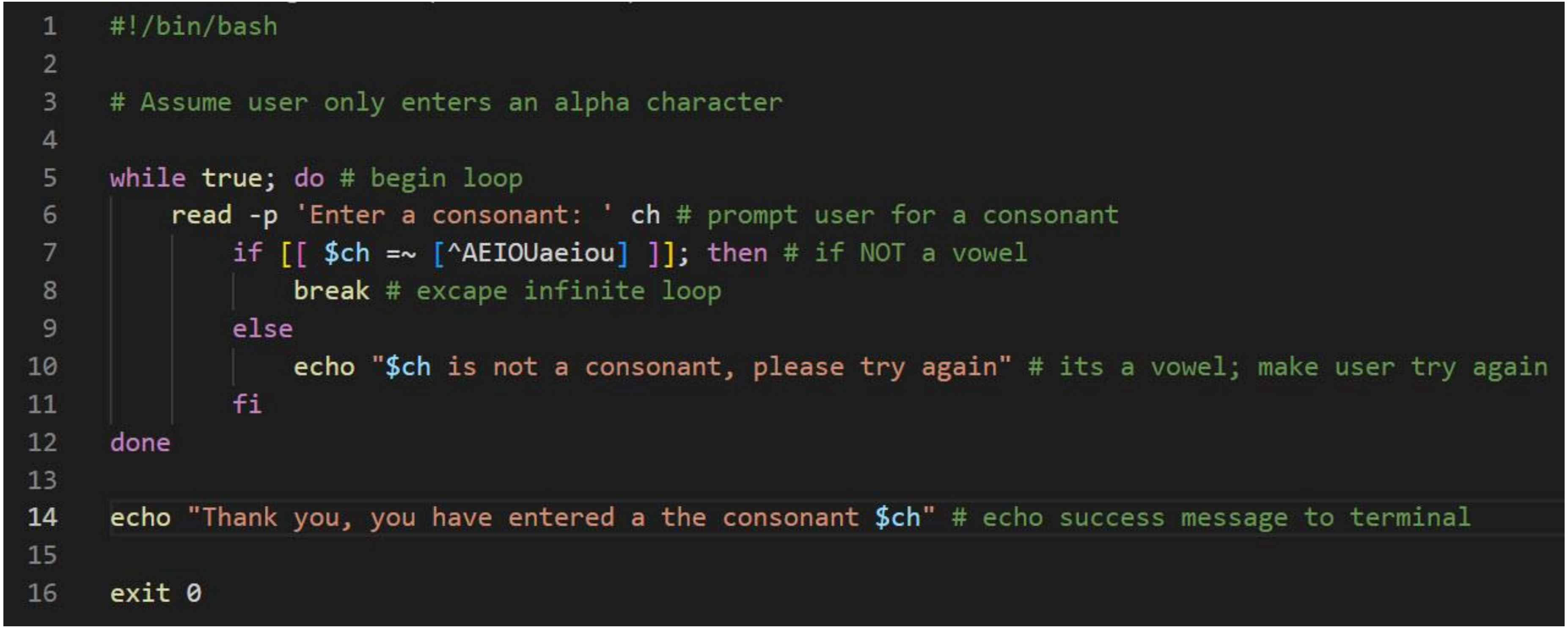
[^] excluding a range

- You can also exclude a range of characters by placing them by preceding them immediately with the carat symbol (^)
- A example to reject a single character input that is a vowel would be:

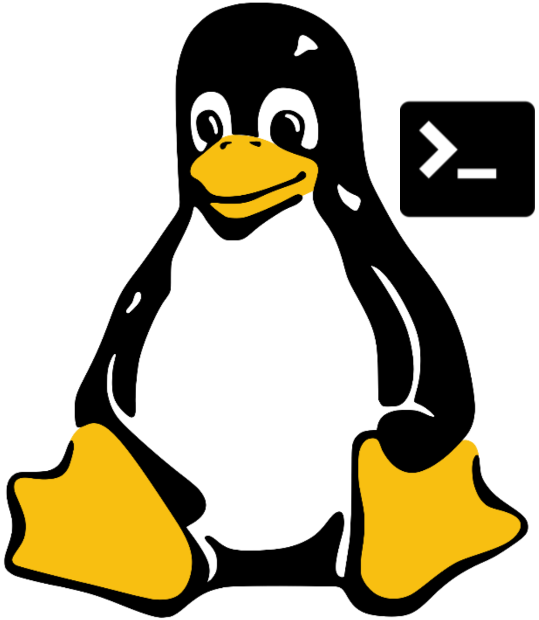
```
if [[ $ch =~ [^AEIOUaeiou] ]]; then # if any character other than a vowel is provided, then this equates to true
```

- See example on next slide for this approach in action

[^] excluding a range



```
#!/bin/bash

# Assume user only enters an alpha character

while true; do # begin loop
    read -p 'Enter a consonant: ' ch # prompt user for a consonant
        if [[ $ch =~ [^AEIOUaeiou] ]]; then # if NOT a vowel
            break # excape infinite loop
        else
            echo "$ch is not a consonant, please try again" # its a vowel; make user try again
        fi
done

echo "Thank you, you have entered a the consonant $ch" # echo success message to terminal

exit 0
```



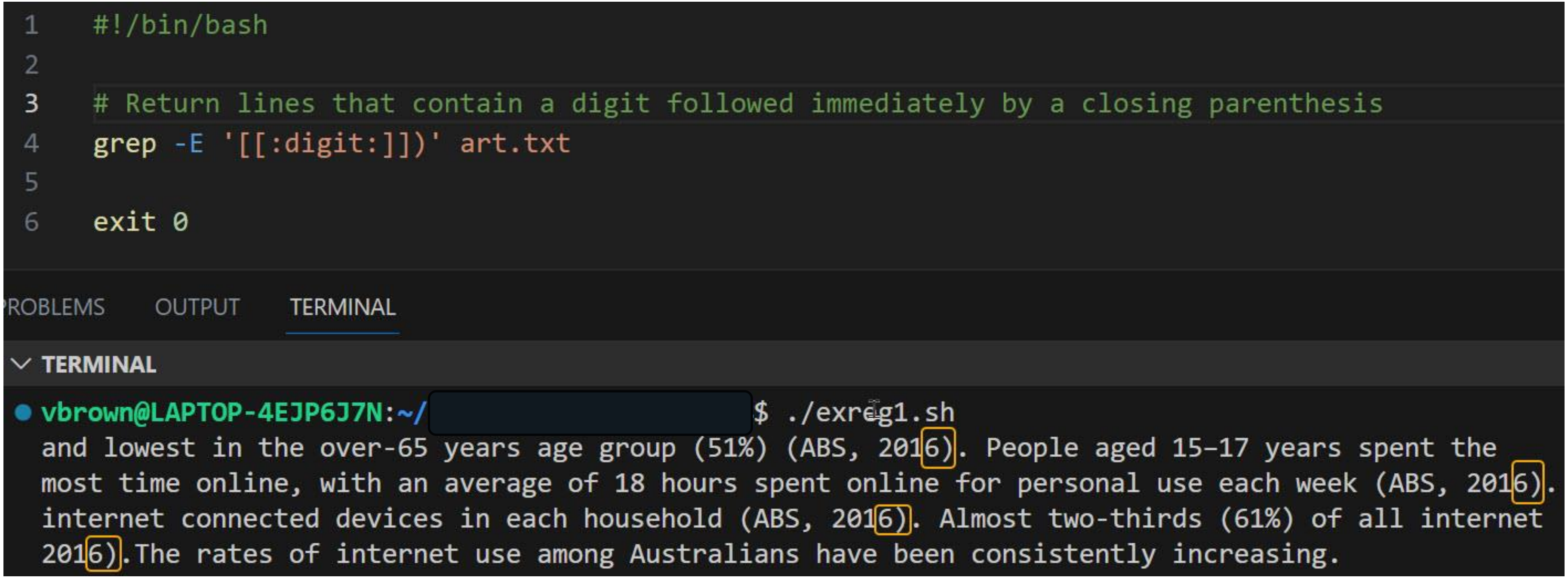
Extended RegEx Engine

Extended Regex

- ERE can also match with several other collections of classes

Pattern	Effect
[[:alpha:]]	Alphabetical character A-z, a-z
[[:alnum:]]	Alphanumeric character A-z, a-z, 0-9
[[:digit:]]	Digit 0-9
[[:upper:]]	Uppercase A-Z
[[:lower:]]	Lowercase a-z
[[:space:]]	Any whitespace character (space tab newline)
[[:blank:]]	Space or tab
[[:punct:]]	Punctuation character e.g. “!,.;”

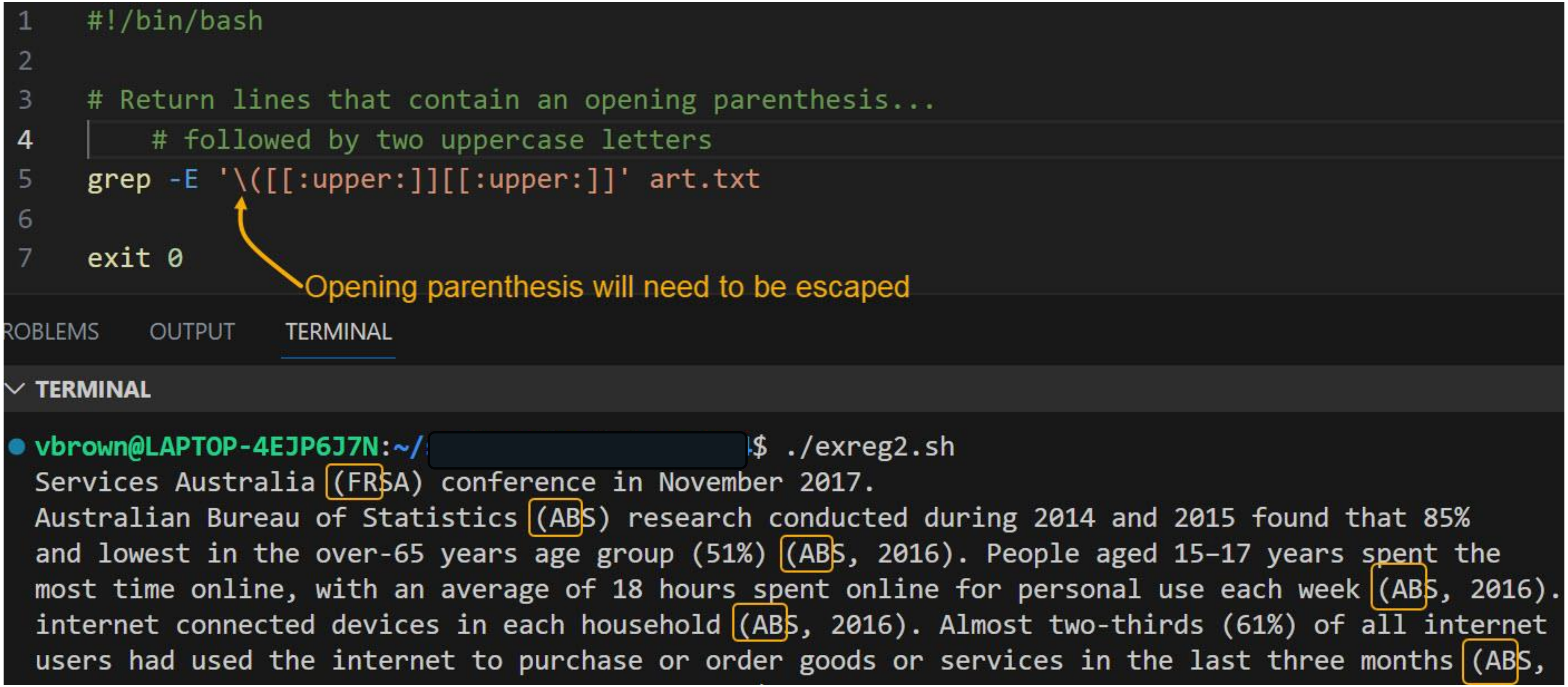
Example ERE class - [[:digit:]]



```bash
#!/bin/bash

# Return lines that contain a digit followed immediately by a closing parenthesis
grep -E '[[:digit:]])' art.txt

exit 0
```

```
vbrown@LAPTOP-4EJP6J7N:~/[redacted]$ ./exreg1.sh
and lowest in the over-65 years age group (51%) (ABS, 2016). People aged 15–17 years spent the
most time online, with an average of 18 hours spent online for personal use each week (ABS, 2016).
internet connected devices in each household (ABS, 2016). Almost two-thirds (61%) of all internet
2016).The rates of internet use among Australians have been consistently increasing.
```

Example ERE class - [[:upper:]]

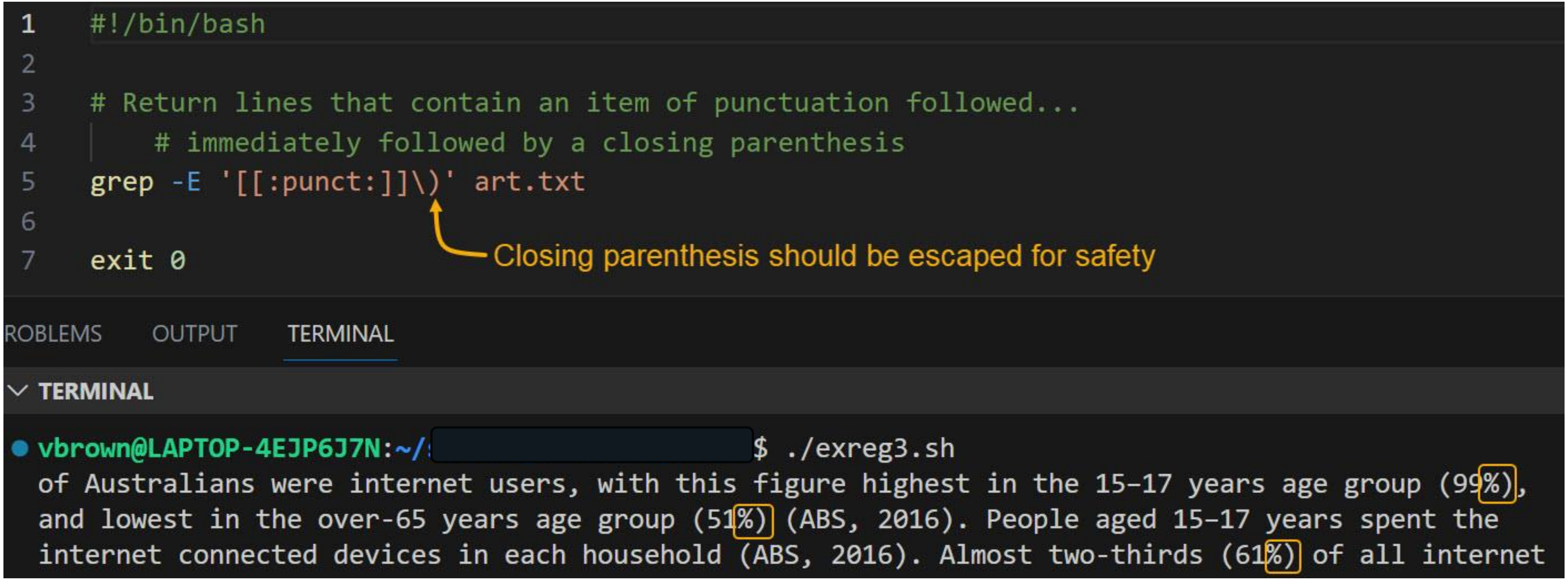


```
#!/bin/bash

# Return lines that contain an opening parenthesis...
    # followed by two uppercase letters
grep -E '\([[:upper:]][[:upper:]]' art.txt

exit 0
```

Opening parenthesis will need to be escaped

```
vbrown@LAPTOP-4EJP6J7N:~/[redacted]$ ./exreg2.sh
Services Australia (FRSA) conference in November 2017.
Australian Bureau of Statistics (ABS) research conducted during 2014 and 2015 found that 85%
and lowest in the over-65 years age group (51%) (ABS, 2016). People aged 15-17 years spent the
most time online, with an average of 18 hours spent online for personal use each week (ABS, 2016).
internet connected devices in each household (ABS, 2016). Almost two-thirds (61%) of all internet
users had used the internet to purchase or order goods or services in the last three months (ABS,
```

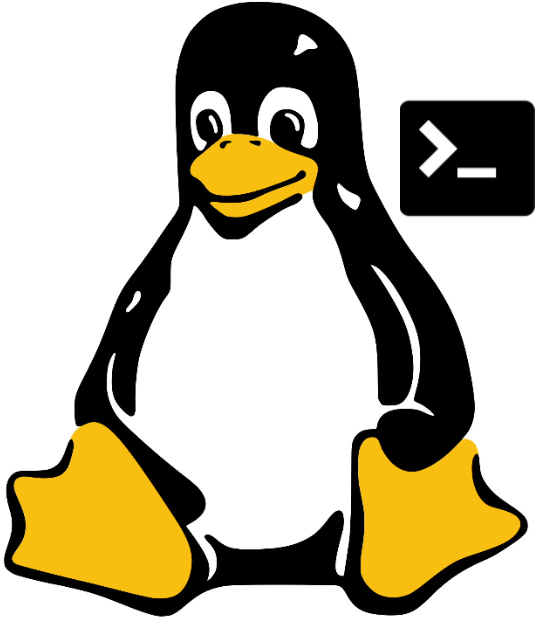
Example ERE class - [[:punct:]]



```
#!/bin/bash

# Return lines that contain an item of punctuation followed...
    # immediately followed by a closing parenthesis
grep -E '[[:punct:]]\)' art.txt

exit 0
```

Closing parenthesis should be escaped for safety

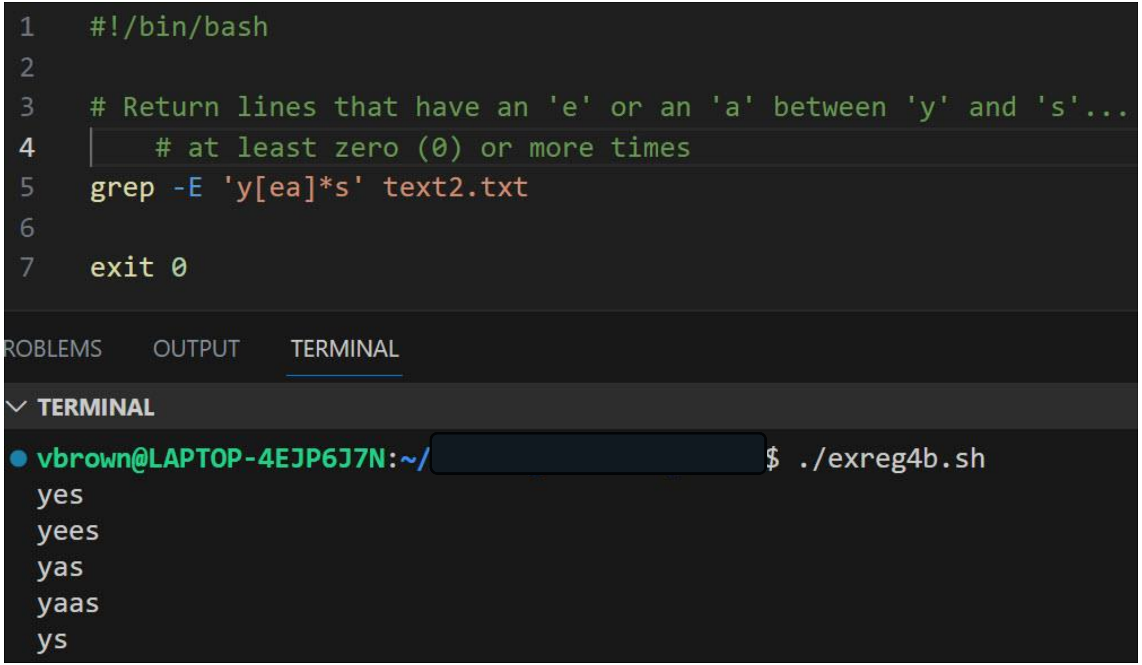
```
vbrown@LAPTOP-4EJP6J7N:~/[redacted]$ ./exreg3.sh
  of Australians were internet users, with this figure highest in the 15–17 years age group (99%),
  and lowest in the over-65 years age group (51%) (ABS, 2016). People aged 15–17 years spent the
  internet connected devices in each household (ABS, 2016). Almost two-thirds (61%) of all internet
```



ERE Repetition and Optionality

The Asterisk Wildcard

- The asterisk (*) wildcard indicates that the preceding part of the pattern is to be repeated **0 or more** times
- For example, the grep regex on the right would match the strings ys, *yes*, *yees, yas* and *yaas*, but **not** *yos*, *yus* or *yis*



```
#!/bin/bash

# Return lines that have an 'e' or an 'a' between 'y' and 's'...
    # at least zero (0) or more times
grep -E 'y[ea]*s' text2.txt

exit 0
```

```
vbrown@LAPTOP-4EJP6J7N:~/[redacted]$ ./exreg4b.sh
yes
yees
yas
yaas
ys
```

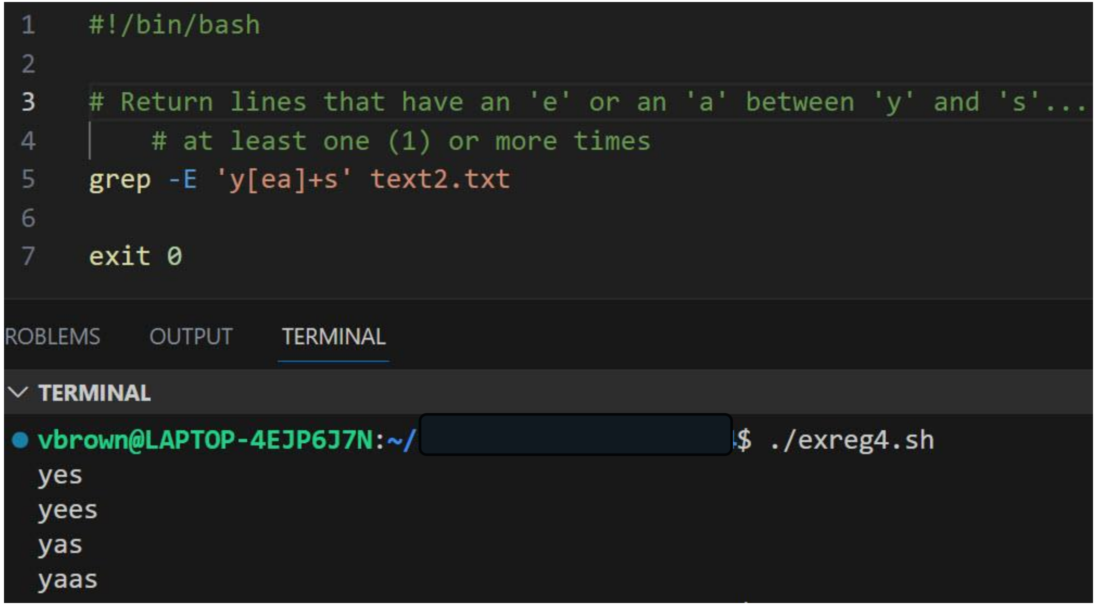
ERE

In the ERE syntax, there are even more useful and versatile pattern matching operators including:

- +
- ?
- {}
- |
- ()

ERE Plus +

- The Plus character “**+**” acts similarly to the asterisk “*****” except instead of *0 or more* repetitions, there must be at least **one or more** repetitions
- For example, the grep pattern search to the right would return the strings *yes*, *yees*, *yas* and *yaas*, but not *ys, yos, yus or yis*



```
#!/bin/bash

# Return lines that have an 'e' or an 'a' between 'y' and 's'...
    # at least one (1) or more times
grep -E 'y[ea]+s' text2.txt

exit 0
```

ROBLEMS OUTPUT TERMINAL

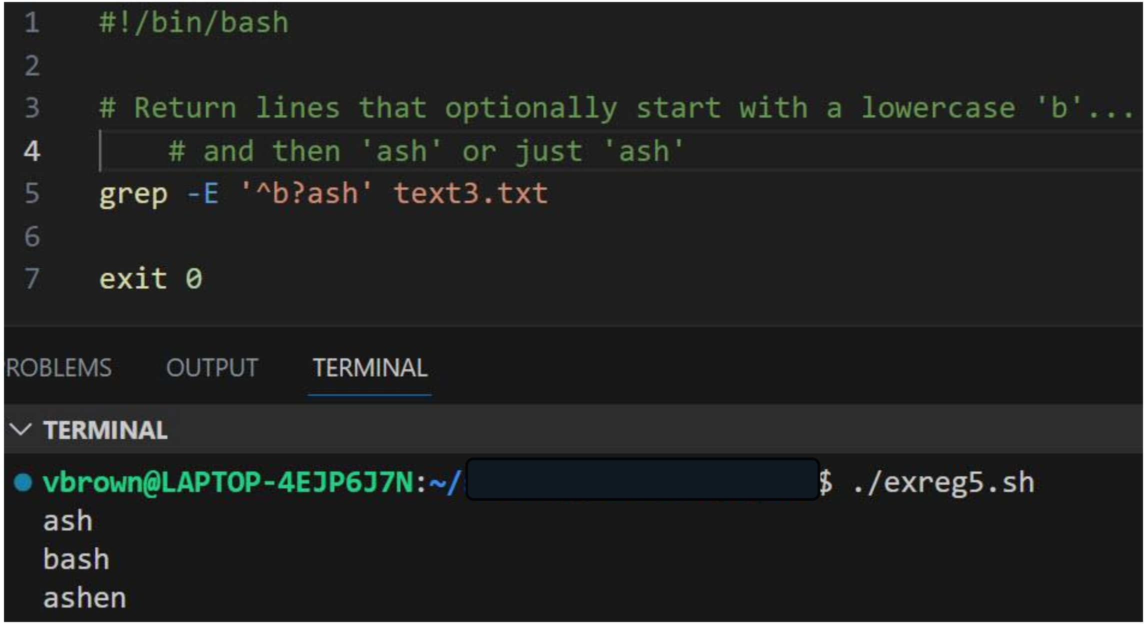
TERMINAL

```
vbrown@LAPTOP-4EJP6J7N:~/[redacted]$ ./exreg4.sh
yes
yees
yas
yaas
```

Question Mark ?

- The question mark character **?** acts as an optionality operator, meaning that the preceding character may or may not be present in the pattern being sought

- For example, the grep search to the right would find *ash*, *bash* and *ashen*, but not *tash*, *cash* or *dash*



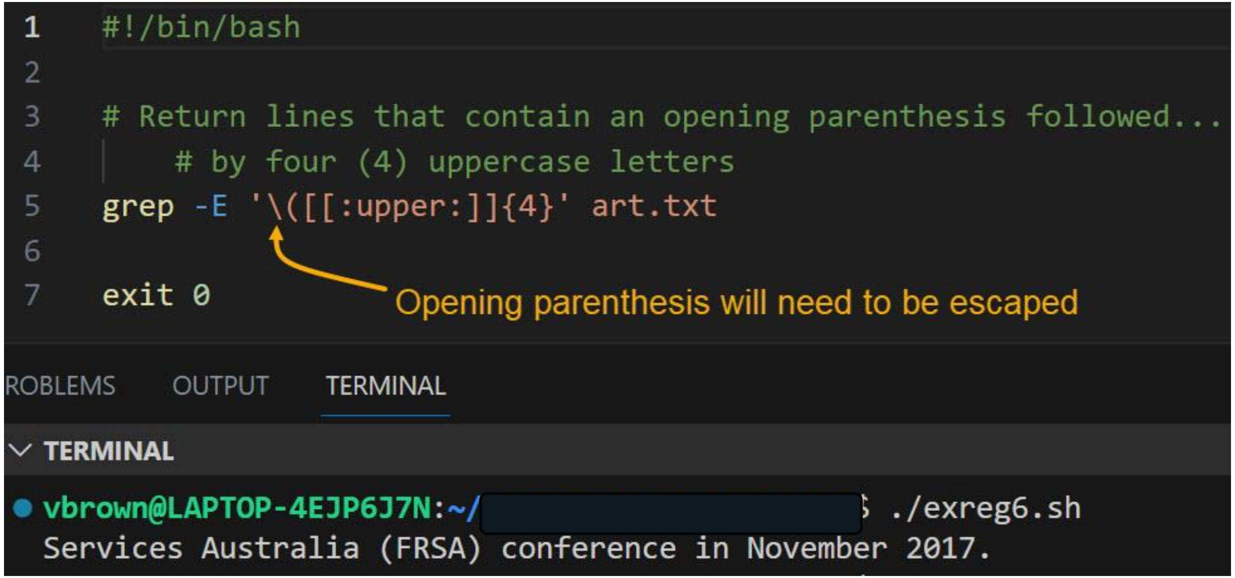
```
#!/bin/bash

# Return lines that optionally start with a lowercase 'b'...
    # and then 'ash' or just 'ash'
grep -E '^b?ash' text3.txt

exit 0
```

```
vbrown@LAPTOP-4EJP6J7N:~/[redacted]$ ./exreg5.sh
ash
bash
ashen
```

Curly Braces { }

- Curly braces **{ }** are used to specify a specific number of repetitions of a character or sequence
- For example, the grep pattern search to the right will return lines that contain an opening parenthesis followed by four (4) uppercase letters

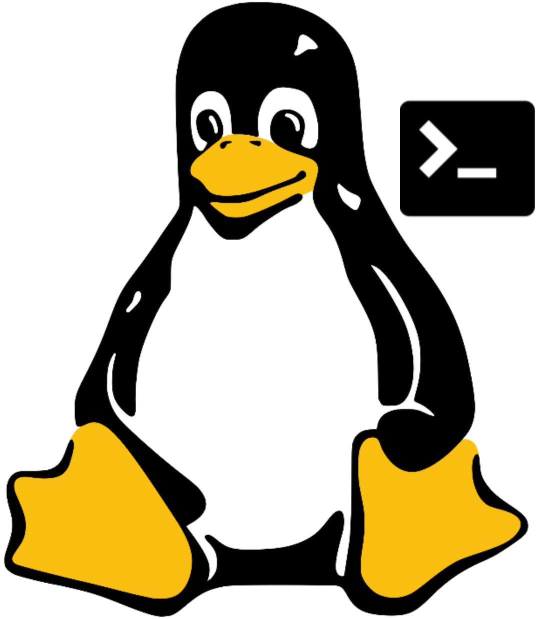
Note: The opening parenthesis in this regex example will need to be escaped or a “no closing” error will be generated



```bash
#!/bin/bash

# Return lines that contain an opening parenthesis followed...
    # by four (4) uppercase letters
grep -E '\([[:upper:]]{4}' art.txt

exit 0
```

Opening parenthesis will need to be escaped

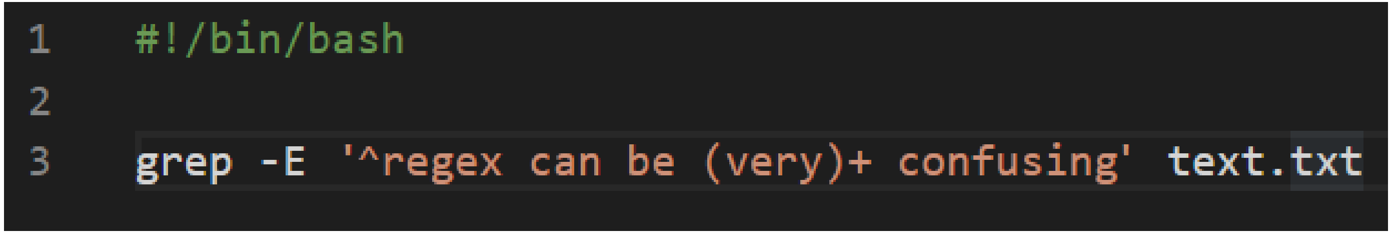
```
vbrown@LAPTOP-4EJP6J7N:~/[redacted]$ ./exreg6.sh
Services Australia (FRSA) conference in November 2017.
```



OR and Expression Grouping

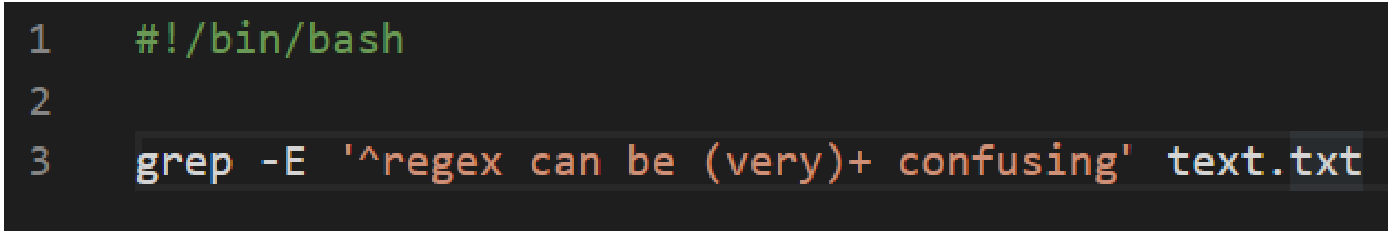
Expression Grouping

- Using parentheses **( )** regex patterns can be grouped together to allow for more complex search patterns to be constructed
- In the example below, (very) must be positionally present, at least once, as the **+** immediately after it indicates
- **Note:** `grep -E` *escapes* traditional usage of special characters. For example, the command `grep -E '{1'` searches for the two-character string `{1` instead of reporting a syntax error in the regular expression



```
#!/bin/bash

grep -E '^regex can be (very)+ confusing' text.txt
```

Expression Grouping



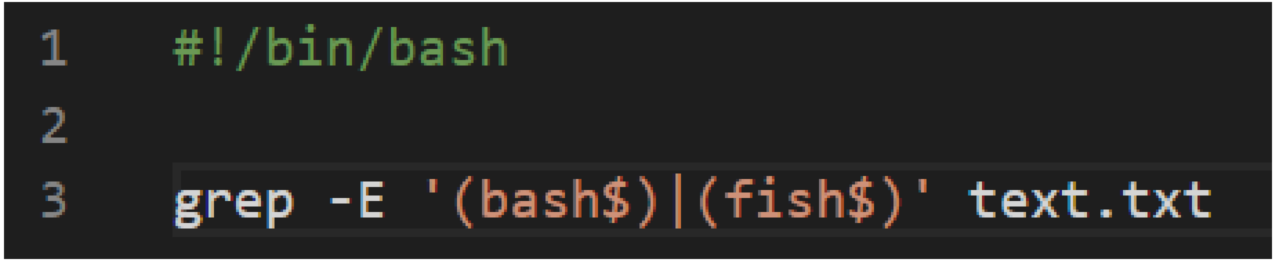
```
#!/bin/bash

grep -E '^regex can be (very)+ confusing' text.txt
```

Potential Matches:

```
regex can be very confusing
regex can be very very confusing
regex can be very very very very confusing
```

OR |

- In bash, the pipe operator “**|**” is usually used to redirecting the output of one script or command to the input of another
- Within the context of a regular expression however, it takes on the functionality of **or**
- For example, in the grep pattern search to the right, lines will be returned that <u>end</u> with either the string *bash* or the string *fish*

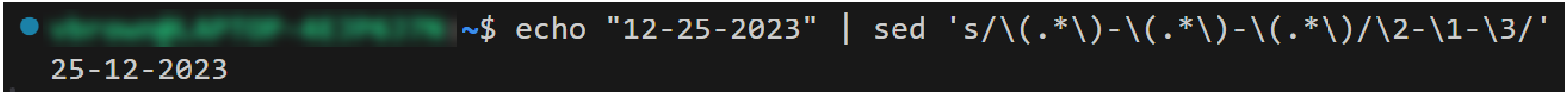


```
#!/bin/bash

grep -E '(bash$)|(fish$)' text.txt
```

Groups and Backreferences

- A group **()** can also serve as a match that can then be reused within the same regex expression
- This is useful for matching repeated sequences and capturing parts of the input for later use using backreferences
- **Backreferences** are references to previously matched groups within the same regular expression
- They are indicated by a backslash \ followed by the number of the group identified ordinally from left to right, e.g., **\1** for the first group, **\2** for the second group, and so on

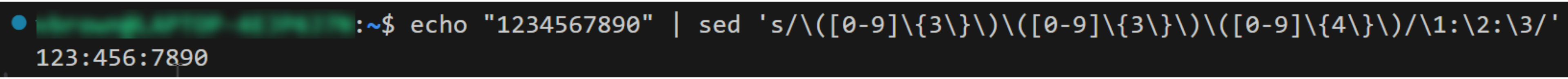
Groups/Backreferences Example 1



```
[redacted]~$ echo "12-25-2023" | sed 's/\(.*\)-\(.*\)-\(.*\)/\2-\1-\3/'
25-12-2023
```

- The goal here is to alter the input date 12-25-2023 which is in the MM-DD-YYYY (US) format to the DD-MM-YYYY (International) format
- In the first part of the sed substitution, the date is broken into its three (3) components using the dash (-) as the delimiter between them, placing each component into a group with MM being **\1**, DD being **\2** and YYYY being **\3**
- Then in the replace part of the sed statement, these groups are re-ordered to get the DD-MM-YYYY format required

Groups/Backreferences Example 2



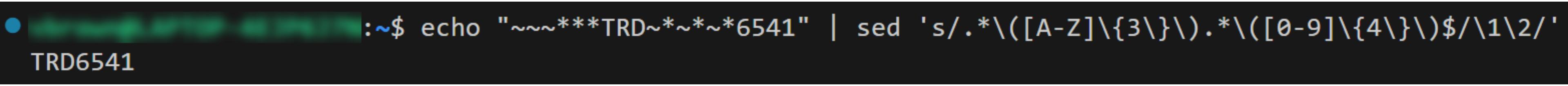
```
[redacted]:~$ echo "1234567890" | sed 's/\([0-9]\{3\}\)\([0-9]\{3\}\)\([0-9]\{4\}\)/\1:\2:\3/'
123:456:7890
```

- The goal here is to break the input number up into three parts delimited by a colon (:)
- In the first part of the sed substitution, the input number is broken up into three (3) groups, the first two to be three (3) digits in length, and the last to be four (4) digits in length
- Then in the replace part of the sed statement, these groups, these groups are referred to again with colons (:) between them to get the desired results

Groups/Backreferences Examples 3 and 4

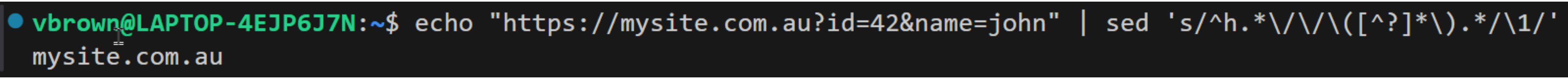
- Now carefully analyse the two examples below and figure out how they work to achieve the outputs shown

EXAMPLE 3



```
[redacted]:~$ echo "~~~***TRD~*~*~*6541" | sed 's/.*\([A-Z]\{3\}\).*\([0-9]\{4\}\)$/\1\2/'
TRD6541
```

EXAMPLE 4



```
vbrown@LAPTOP-4EJP6J7N:~$ echo "https://mysite.com.au?id=42&name=john" | sed 's/^h.*\/\/\([^?]*\).*/\1/'
mysite.com.au
```

Dealing with special characters in regex

Using escapes (\) for special characters

- In both bash and regex, there are many characters that serve a special purpose
- Two examples are the *asterisk* (*****) interpreted to mean "*zero or more occurrences of a character*" in regex, and **[** to be the *test* command in bash
- When characters are interpreted in this way, they are said to be acting as *meta-characters*
- However, there will be occasions when you need a meta-character to be treated as a normal character in a regular expression
- To achieve this, you must use an escape (****) to ensure a meta-character is treated like a normal character

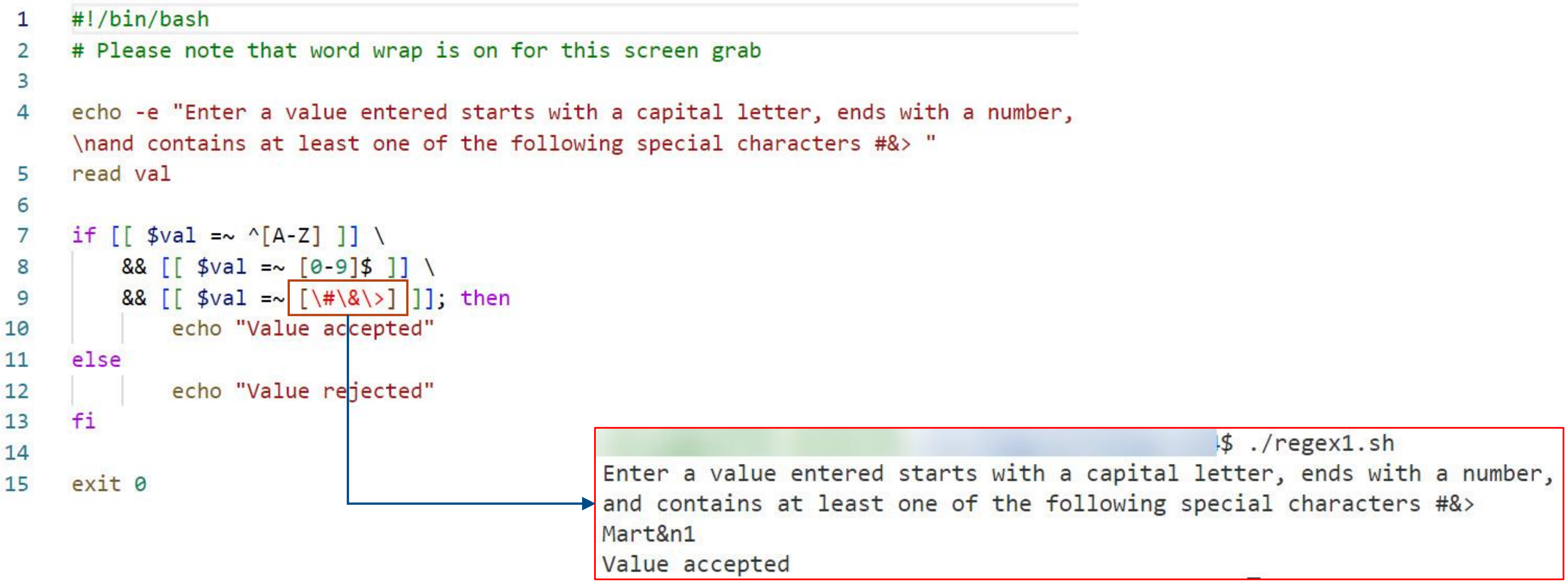
Using escapes (\) for special characters

- As a rule, the special characters in the tables to the right need to be escaped to be interpreted as normal characters in a regular expression
- Note however, that this can vary slightly from one Linux distribution to the next

#	hash
-	hyphen
;	semi-colon
&	ampersand
<	Left angular bracket
>	Right angular bracket
[ ]	Left/right square bracket
{ }	Left/right curly braces

‘	Single quote
“	Double quote
	space
\	Backslash
/	Forward slash
( )	Left/right parenthesis
`	backtick

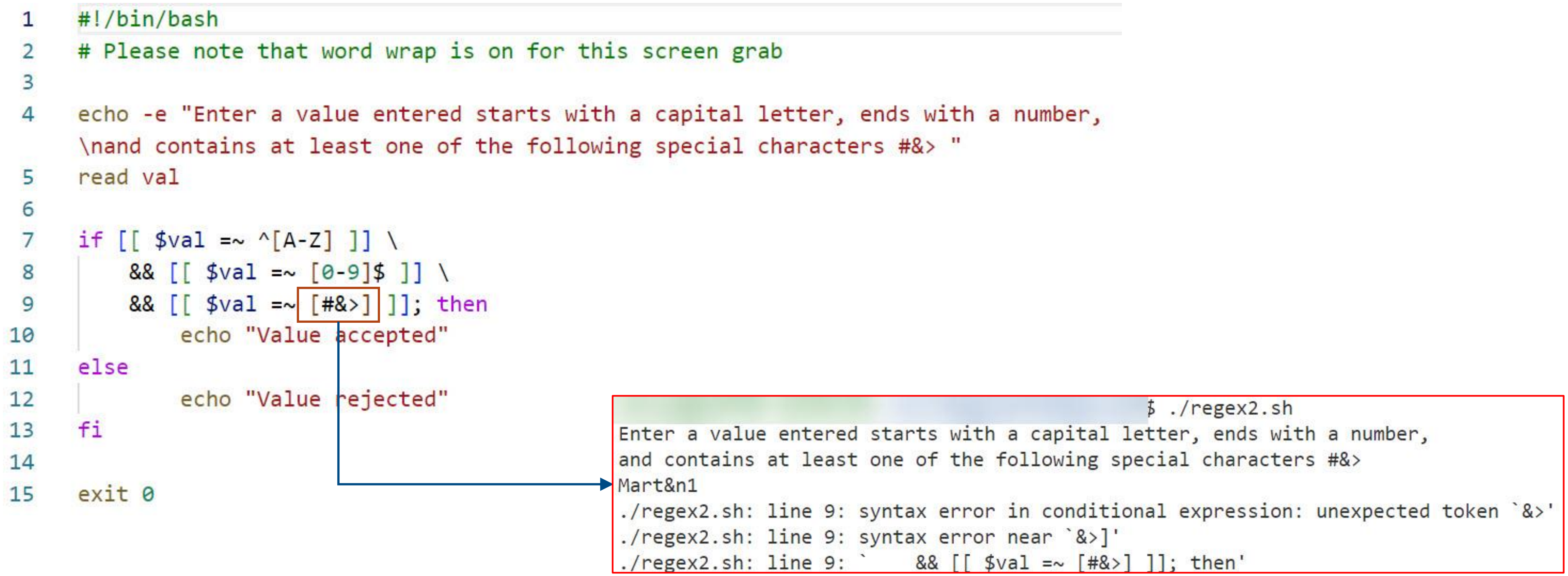
Example – metacharacters escaped



```
#!/bin/bash
# Please note that word wrap is on for this screen grab

echo -e "Enter a value entered starts with a capital letter, ends with a number, \nand contains at least one of the following special characters #&> "
read val

if [[ $val =~ ^[A-Z] ]] \
    && [[ $val =~ [0-9]$ ]] \
    && [[ $val =~ [\#\&\>] ]]; then
        echo "Value accepted"
else
        echo "Value rejected"
fi

exit 0
```

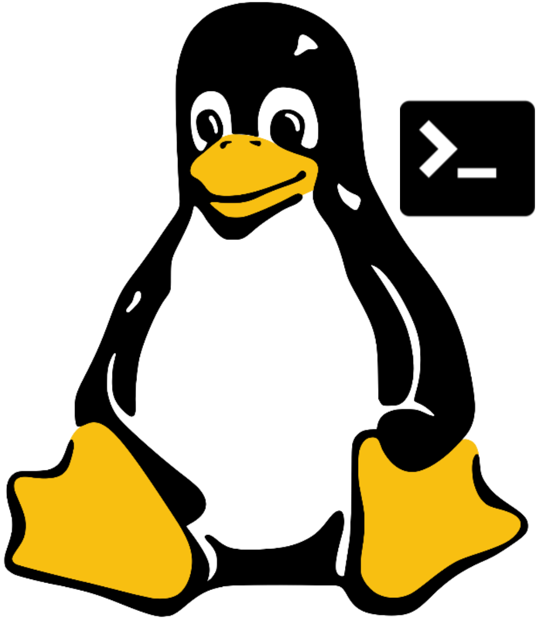
```
[redacted]$ ./regex1.sh
Enter a value entered starts with a capital letter, ends with a number,
and contains at least one of the following special characters #&>
Mart&n1
Value accepted
```

Example – metacharacters not escaped



```
1    #!/bin/bash
2    # Please note that word wrap is on for this screen grab
3
4    echo -e "Enter a value entered starts with a capital letter, ends with a number,
     \nand contains at least one of the following special characters #&> "
5    read val
6
7    if [[ $val =~ ^[A-Z] ]] \
8        && [[ $val =~ [0-9]$ ]] \
9        && [[ $val =~ [#&>] ]]; then
10           echo "Value accepted"
11   else
12           echo "Value rejected"
13   fi
14
15   exit 0
```

```
[redacted]$ ./regex2.sh
Enter a value entered starts with a capital letter, ends with a number,
and contains at least one of the following special characters #&>
Mart&n1
./regex2.sh: line 9: syntax error in conditional expression: unexpected token `&>'
./regex2.sh: line 9: syntax error near `&>]'
./regex2.sh: line 9: `    && [[ $val =~ [#&>] ]]; then'
```

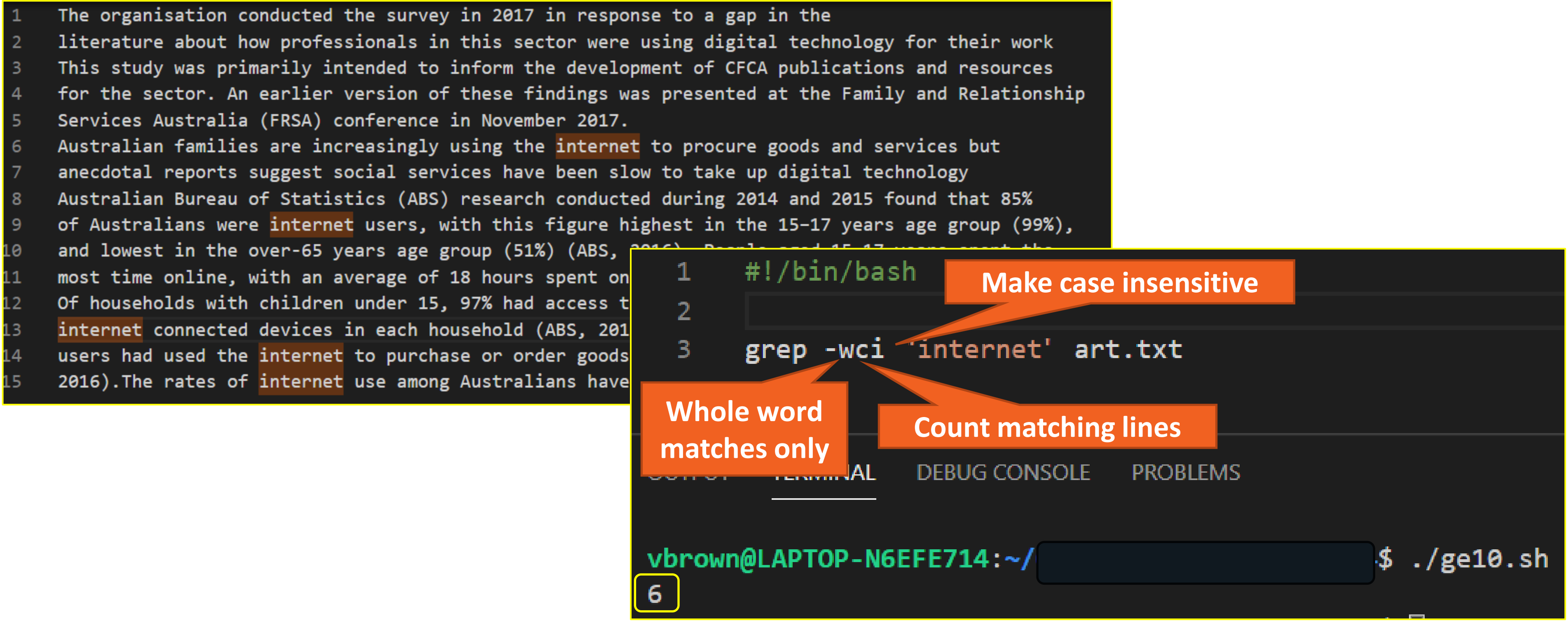


grep options

Common grep options

Option	Description
-c	Suppress normal output; instead print a count of matching lines for each input file
-E	Interpret PATTERN as an extended regular expression.
-i	Ignore case distinctions in both the PATTERN and the input files.
-m NUM	Stop reading a file after NUM matching lines.
-n	Prefix each line of output with the line number within its input file.
-o	Show only the part of a matching line that matches PATTERN.
-v	Invert the sense of matching, to select non-matching lines
-w	Select only those lines containing matches that form whole words

grep Options Example



```
The organisation conducted the survey in 2017 in response to a gap in the
literature about how professionals in this sector were using digital technology for their work
This study was primarily intended to inform the development of CFCA publications and resources
for the sector. An earlier version of these findings was presented at the Family and Relationship
Services Australia (FRSA) conference in November 2017.
Australian families are increasingly using the internet to procure goods and services but
anecdotal reports suggest social services have been slow to take up digital technology
Australian Bureau of Statistics (ABS) research conducted during 2014 and 2015 found that 85%
of Australians were internet users, with this figure highest in the 15-17 years age group (99%),
and lowest in the over-65 years age group (51%) (ABS,
most time online, with an average of 18 hours spent on
Of households with children under 15, 97% had access t
internet connected devices in each household (ABS, 201
users had used the internet to purchase or order goods
2016).The rates of internet use among Australians have
```

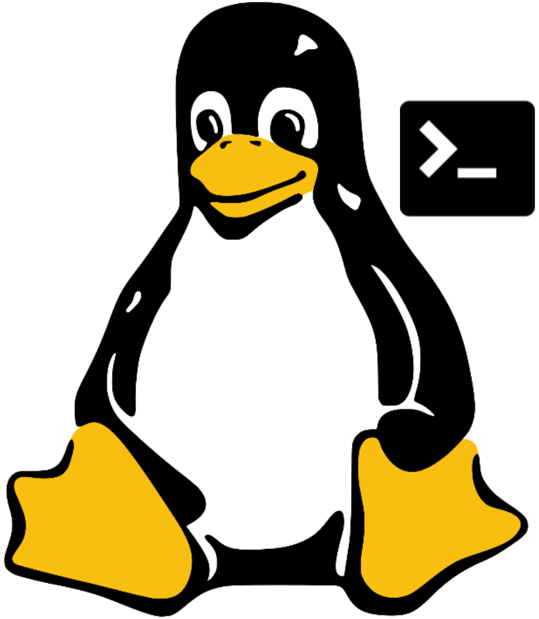
```
#!/bin/bash

grep -wci 'internet' art.txt
```

Make case insensitive

Whole word matches only

Count matching lines

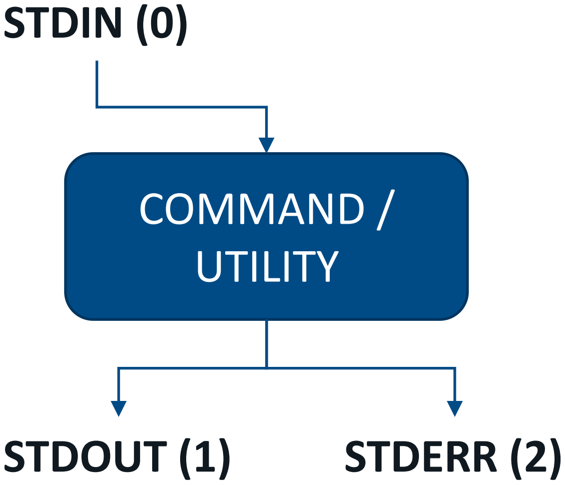
```
vbrown@LAPTOP-N6EFE714:~/[redacted]$ ./ge10.sh
6
```



Piping and Redirection

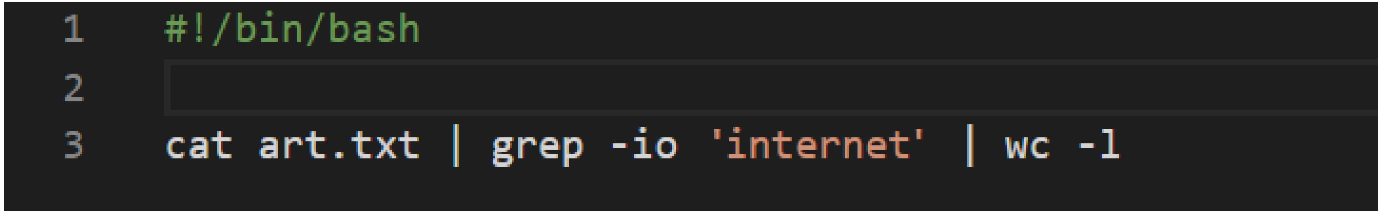
Bash Data Streams

- As discussed in Module 2, bash commands and utilities have automatic access to three (3) data streams:
  - **STDIN (0)** - Standard Input, this is the stream that feeds data into a command or utility
  - **STDOUT (1)** - Standard Output, this is the stream that outputs data from the command or utility; the terminal by default
  - **STDERR (2)** - Standard Error, this is for error messages; also defaults to the terminal)



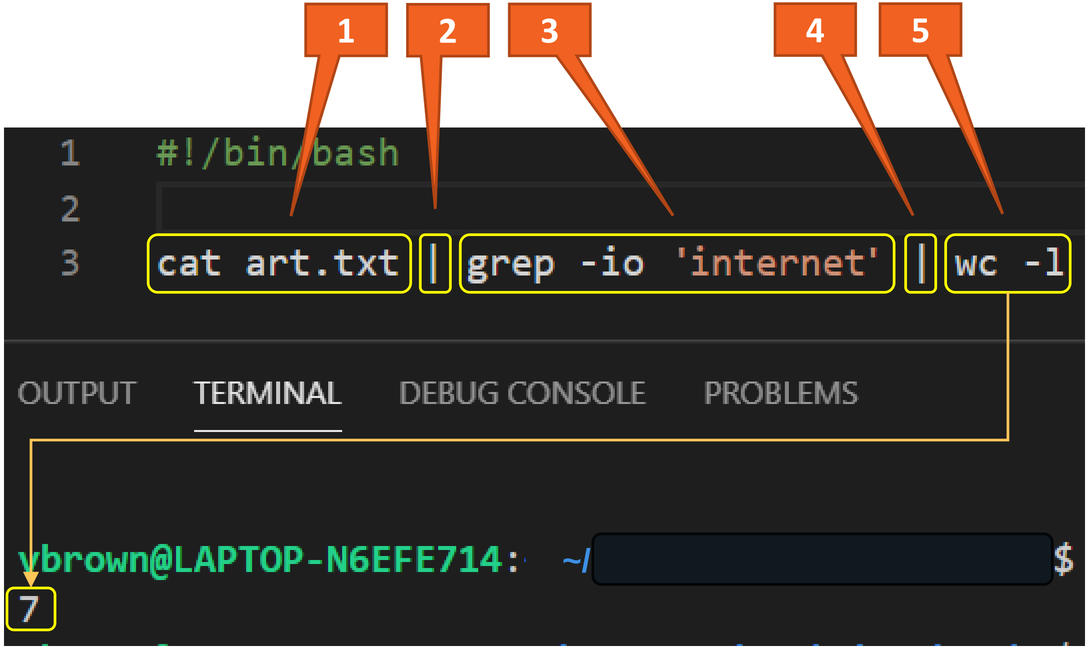
Bash Data Streams

- Bash **pipes** and **redirection** allows streams between command, utilities and files to be connected in specific sequence to manipulate data in useful and flexible ways
- In the example below, three (3) commands are used to count the number of instances of a whole string in a file



```
#!/bin/bash

cat art.txt | grep -io 'internet' | wc -l
```

Piping Example



```
#!/bin/bash

cat art.txt | grep -io 'internet' | wc -l
```

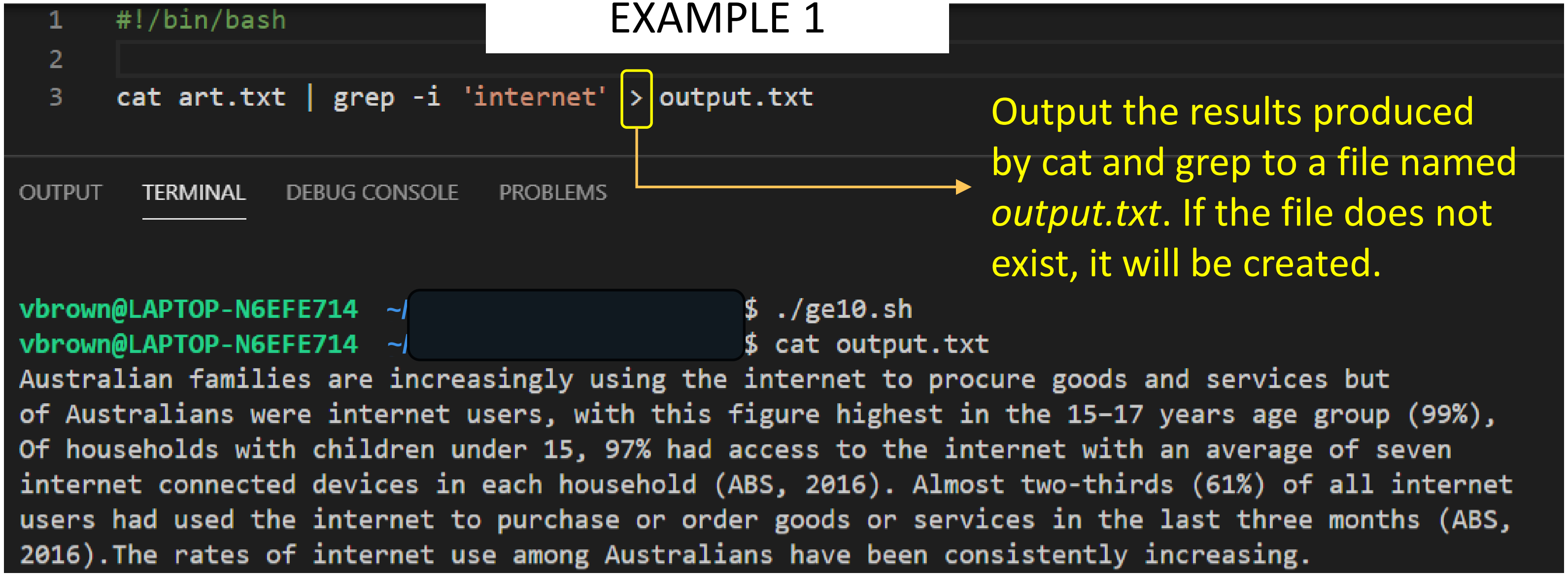
```
vbrown@LAPTOP-N6EFE714: ~/[redacted]$
7
```

1	The **cat** command makes a copy of the file named *art.txt*
2	Pipes the data acquired by **cat** to the **STDIN** of the next command (*grep*)
3	The **grep** command retrieves lines of data it receives for each instance of the whole string *internet* in a case-insensitive mode
4	Pipes the data acquired by *grep* to the **STDIN** of the next command (*Word Count*)
5	The Word Count command will count all instances of the string *internet* in the data received line by line

Redirection

- The standard input and standard output can be redirected to use files instead using the redirection operators **<** and **>**

EXAMPLE 1

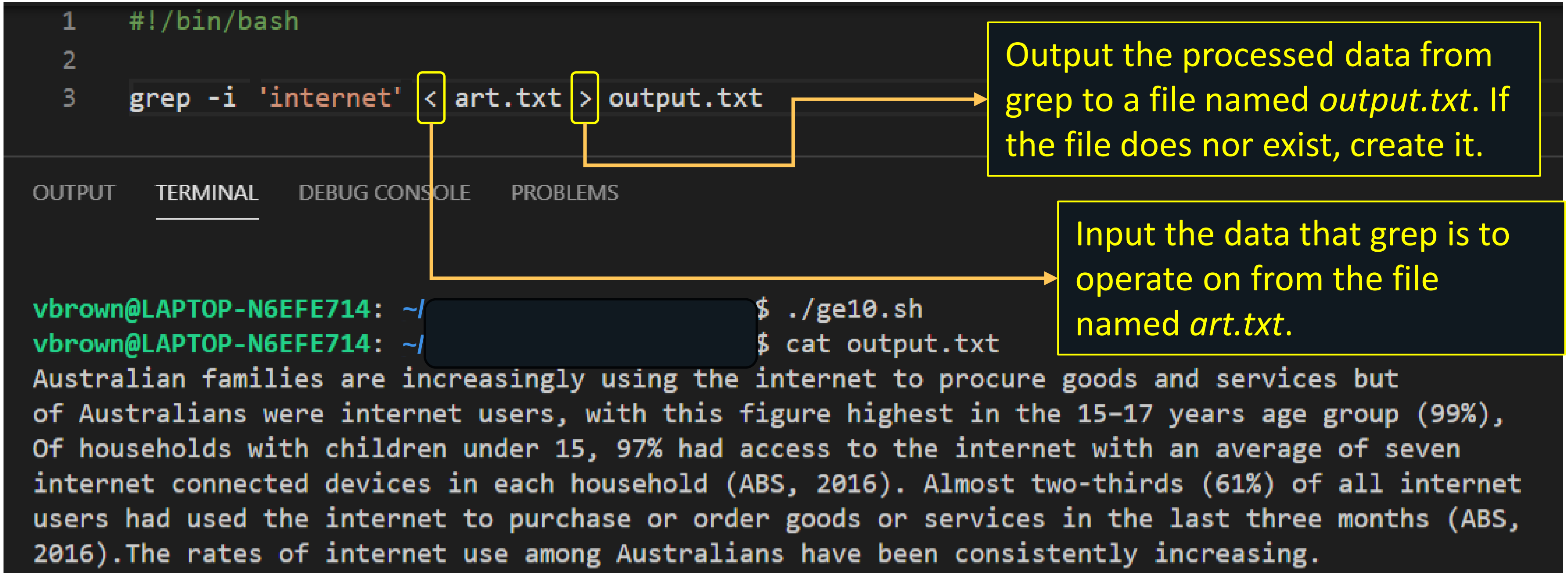


```
#!/bin/bash

cat art.txt | grep -i 'internet' > output.txt
```

```
vbrown@LAPTOP-N6EFE714 ~/[redacted]$ ./ge10.sh
vbrown@LAPTOP-N6EFE714 ~/$ cat output.txt
Australian families are increasingly using the internet to procure goods and services but
of Australians were internet users, with this figure highest in the 15-17 years age group (99%),
Of households with children under 15, 97% had access to the internet with an average of seven
internet connected devices in each household (ABS, 2016). Almost two-thirds (61%) of all internet
users had used the internet to purchase or order goods or services in the last three months (ABS,
2016).The rates of internet use among Australians have been consistently increasing.
```

Output the results produced by cat and grep to a file named *output.txt*. If the file does not exist, it will be created.

Redirection Example 2

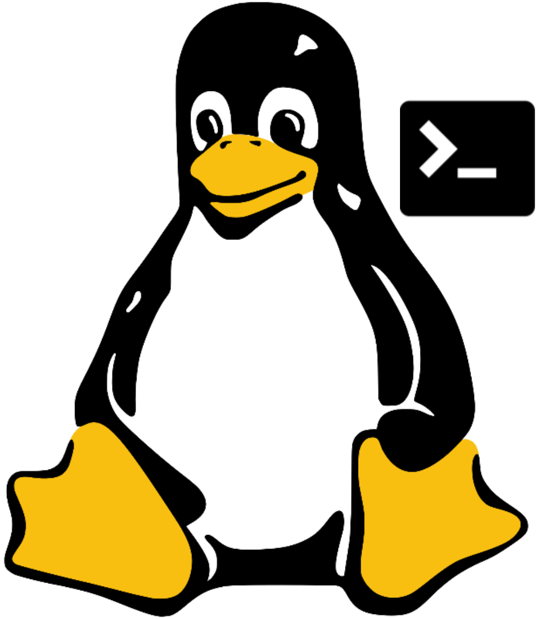


```
#!/bin/bash

grep -i 'internet' < art.txt > output.txt
```

```
vbrown@LAPTOP-N6EFE714: ~/[redacted]$ ./ge10.sh
vbrown@LAPTOP-N6EFE714: ~/[redacted]$ cat output.txt
Australian families are increasingly using the internet to procure goods and services but
of Australians were internet users, with this figure highest in the 15-17 years age group (99%),
Of households with children under 15, 97% had access to the internet with an average of seven
internet connected devices in each household (ABS, 2016). Almost two-thirds (61%) of all internet
users had used the internet to purchase or order goods or services in the last three months (ABS,
2016).The rates of internet use among Australians have been consistently increasing.
```

Output the processed data from grep to a file named *output.txt*. If the file does nor exist, create it.

Input the data that grep is to operate on from the file named *art.txt*.

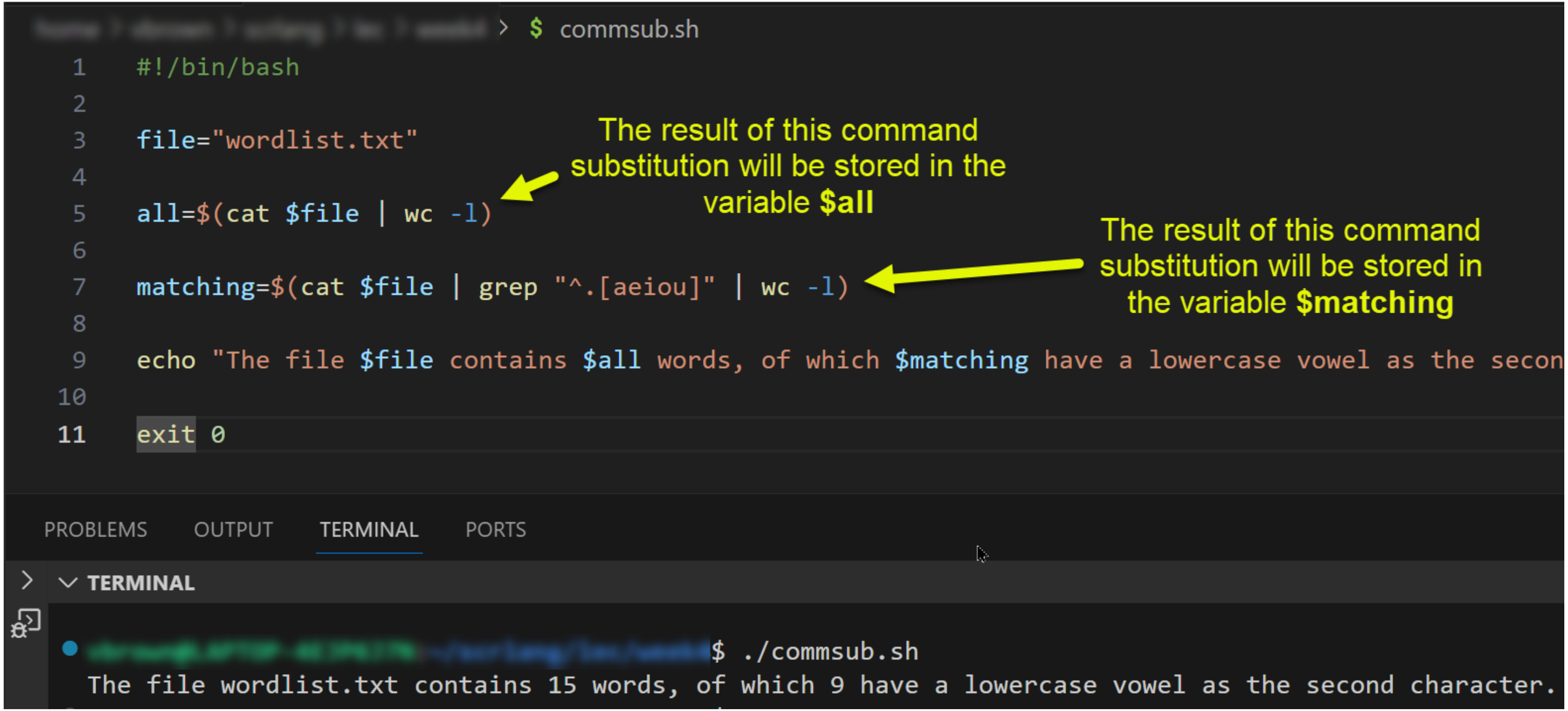


Command Substitution and Process Substitution

Command Substitution

- Command substitution allows the execution of a command for the purpose of directing its output into another command or assignment
- The syntax for command substitution is **$(**…**)**
- The older backtick delimiters can also be used for the same purpose, i.e. **`**…**`**
- The shell executes any command or series of commands inside the **$( )** or **` `** and then **replaces** the command substitution expression with the actual output it generates, with any trailing newlines removed
- This is very useful in capturing the output of a command into a variable or passing it on as an argument to yet another command

Command Substitution



[redacted] > $ commsub.sh

```bash
#!/bin/bash

file="wordlist.txt"

all=$(cat $file | wc -l)

matching=$(cat $file | grep "^.[aeiou]" | wc -l)

echo "The file $file contains $all words, of which $matching have a lowercase vowel as the secon

exit 0
```

The result of this command substitution will be stored in the variable **$all**

The result of this command substitution will be stored in the variable **$matching**

```
[redacted]$ ./commsub.sh
The file wordlist.txt contains 15 words, of which 9 have a lowercase vowel as the second character.
```

Command Substitution

When using command substitution, be mindful of the following:

- Can slow a script down as command substitutions run in their own **subshell**; this is especially in cases where the substituted command produces a large amount of output
- Excessive command substitutions can become complex and hard to read
- Variables modified or created in a command substitution subshell are not available in the parent shell script
- If a command substitution fails for any reason, you might not receive an explicit notification making it difficult to diagnose resultant script problems

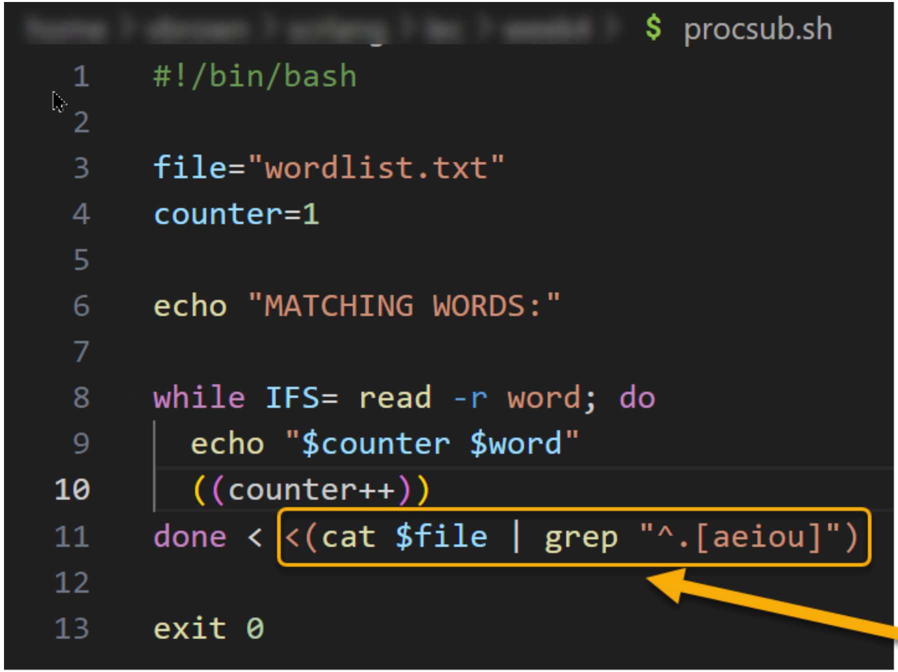
Process Substitution

- Process substitution allows the output of a command or series of commands as if this output were a file
- This is very useful when a command expects a file as input but you want to provide it with the output of another command
- The syntax for process substitution is **<(...)** for input and **>(...)** for output

Process Substitution

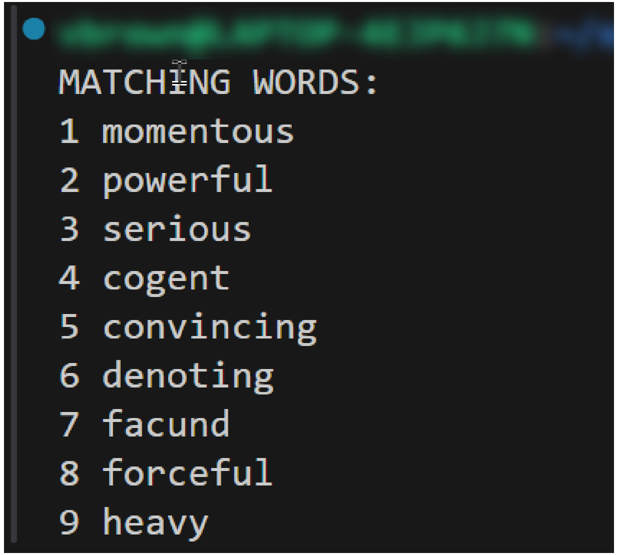
- When you use process substitution with **<(...)**, bash replaces the **<(...)** with a temporary file, e.g., */dev/fd/77*
- This file behaves like a pipe, which the command can read from as if it were reading from a regular file
- This is very useful for commands that do not accept standard input (`stdin`) or when you need to pass multiple streams of data to a command

Process Substitution



[redacted] $ procsub.sh

```bash
#!/bin/bash

file="wordlist.txt"
counter=1

echo "MATCHING WORDS:"

while IFS= read -r word; do
  echo "$counter $word"
  ((counter++))
done < <(cat $file | grep "^.[aeiou]")

exit 0
```



[redacted]

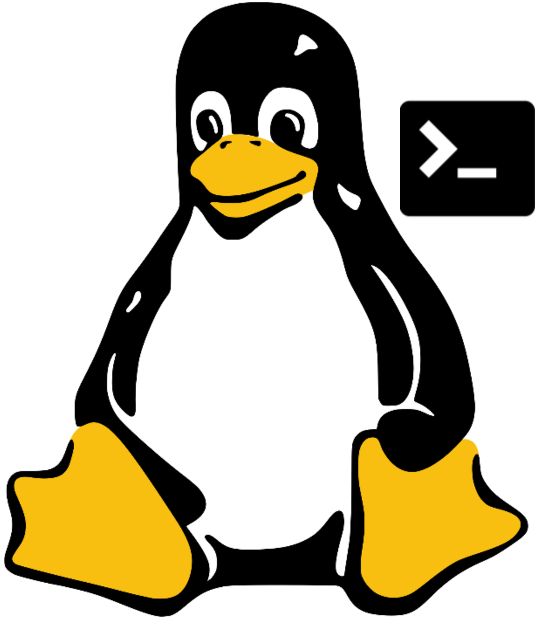
```
MATCHING WORDS:
1 momentous
2 powerful
3 serious
4 cogent
5 convincing
6 denoting
7 facund
8 forceful
9 heavy
```

The output of this will be treated like a file that the while loop can read and work with

Process Substitution

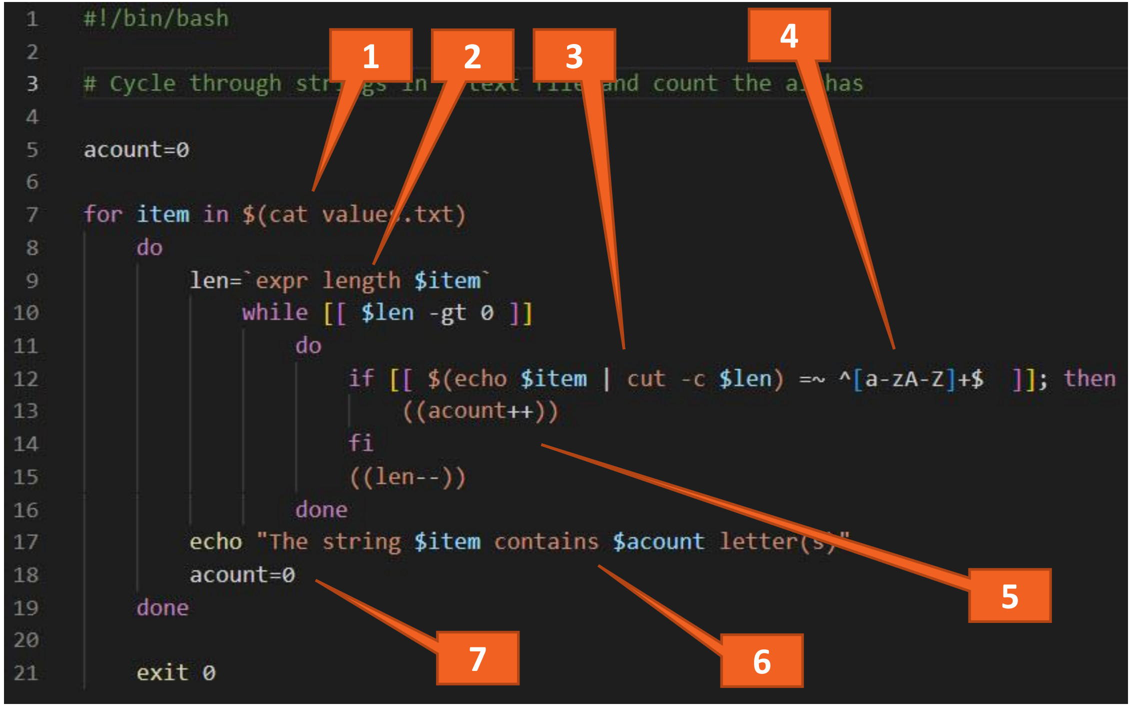
When using process substitution, be mindful of the following:

- It is not supported by all shells
- Excessive process substitutions can become complex and hard to read
- If a process substitution fails for any reason, you might not receive an explicit notification making it difficult to diagnose resultant script problems



regex, grep, piping and redirection applied

Example 1

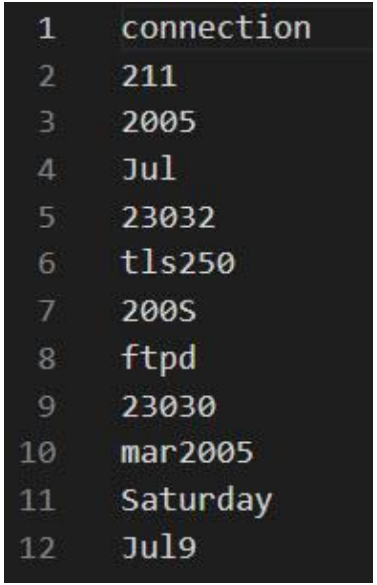


```
#!/bin/bash

# Cycle through strings in text file and count the alphas

acount=0

for item in $(cat values.txt)
    do
        len=`expr length $item`
            while [[ $len -gt 0 ]]
                do
                    if [[ $(echo $item | cut -c $len) =~ ^[a-zA-Z]+$  ]]; then
                        ((acount++))
                    fi
                    ((len--))
                done
        echo "The string $item contains $acount letter(s)"
        acount=0
    done

    exit 0
```

1	Use command substitution to get string values from file to be processed by **for** loop
2	Get the length of the current string in the loop using **expr length**
3	Isolate to the character in the string that corresponds to string length using **cut**
4	Check if an alphabetical character using *regex*, if so, increment alpha counter
5	Decrement the len value by one (1)
6	Print result to terminal
7	Reset the alpha counter to zero (0)

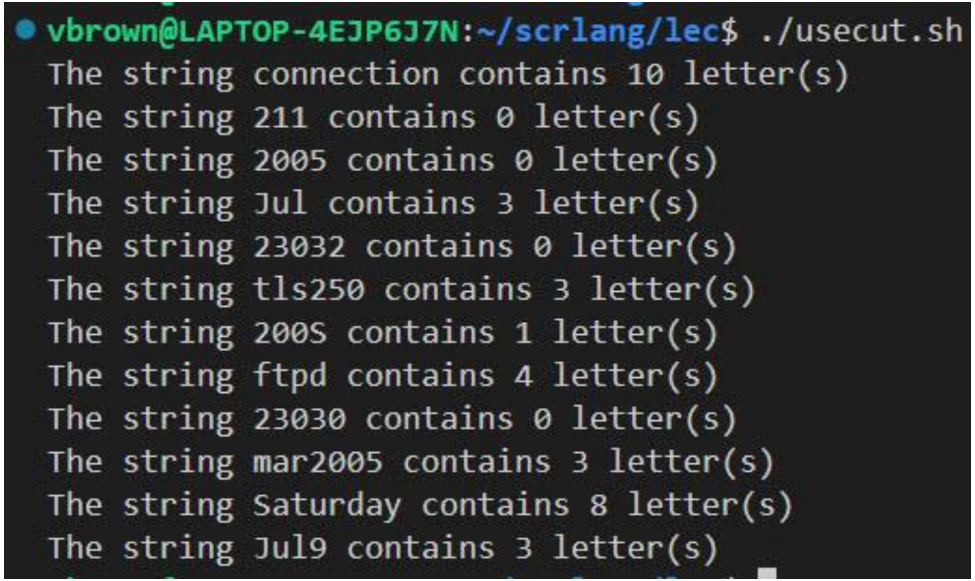
Example 1 cont…

values.txt



```
connection
211
2005
Jul
23032
tls250
200S
ftpd
23030
mar2005
Saturday
Jul9
```

output.txt



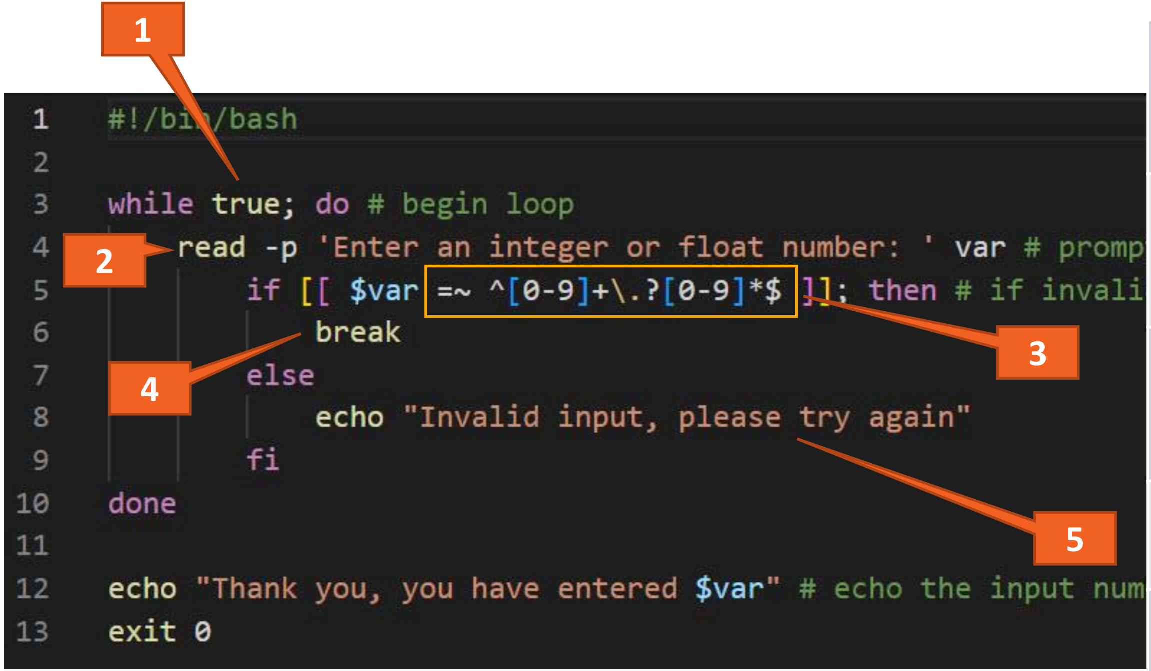
```
vbrown@LAPTOP-4EJP6J7N:~/scrlang/lec$ ./usecut.sh
 The string connection contains 10 letter(s)
 The string 211 contains 0 letter(s)
 The string 2005 contains 0 letter(s)
 The string Jul contains 3 letter(s)
 The string 23032 contains 0 letter(s)
 The string tls250 contains 3 letter(s)
 The string 200S contains 1 letter(s)
 The string ftpd contains 4 letter(s)
 The string 23030 contains 0 letter(s)
 The string mar2005 contains 3 letter(s)
 The string Saturday contains 8 letter(s)
 The string Jul9 contains 3 letter(s)
```

The cut command

- The **cut** command in bash is used to extract parts of a line from a file
- The syntax for the cut command is **`cut [options] <file>`**
- The cut command has a number of options used to specify which parts of the line to extract, most commonly:

Flag	Purpose
-c	Specifies the column number(s) to extract, e.g. **`cut -c 1-3 <file>`** extracts first three columns of file
-d	Specifies the delimiter character (default is a space), e.g. **`cut -d , -f 1,2 <file>`** extracts first two fields of a file, with fields separated by commas
-f	Specifies the field number(s) to extract, e.g. **`cut -f 1,2 <file>`** extracts the first two fields of the file
-b	Specifies the byte number(s) to extract, e.g. **`cut -b 1-10 <file>`**
-r	Specifies a regular expression to use for extracting fields, e.g. **`cut -f 1 -d " " -r "^foo" <file>`**

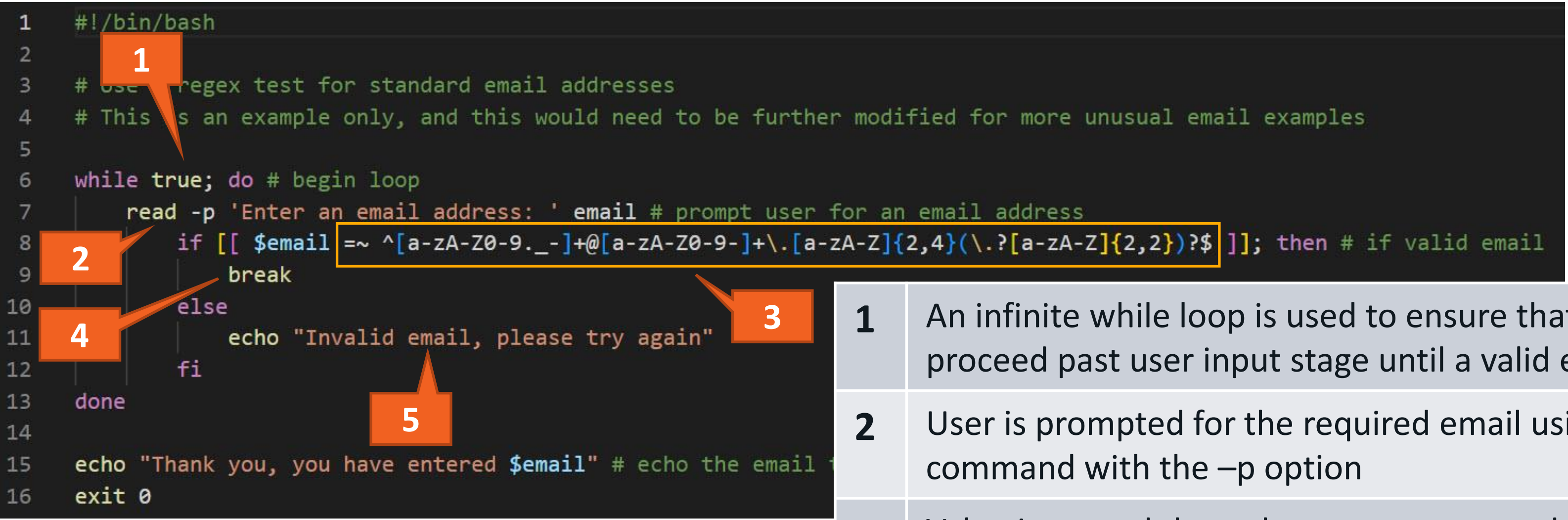
Example 2



```
#!/bin/bash

while true; do # begin loop
    read -p 'Enter an integer or float number: ' var # promp
        if [[ $var =~ ^[0-9]+\.?[0-9]*$ ]]; then # if invali
            break
        else
            echo "Invalid input, please try again"
        fi
done

echo "Thank you, you have entered $var" # echo the input num
exit 0
```

1	An infinite while loop is used to ensure that script will not proceed past user input stage until a valid value is entered
2	User is prompted for the required value using the read command with the –p option
3	Value is passed through regex pattern to determine if it is a match for the stipulated pattern, i.e. is an int or float
4	If a match, break is used to break out of the infinite while loop
5	If not a match, user is prompted to try again

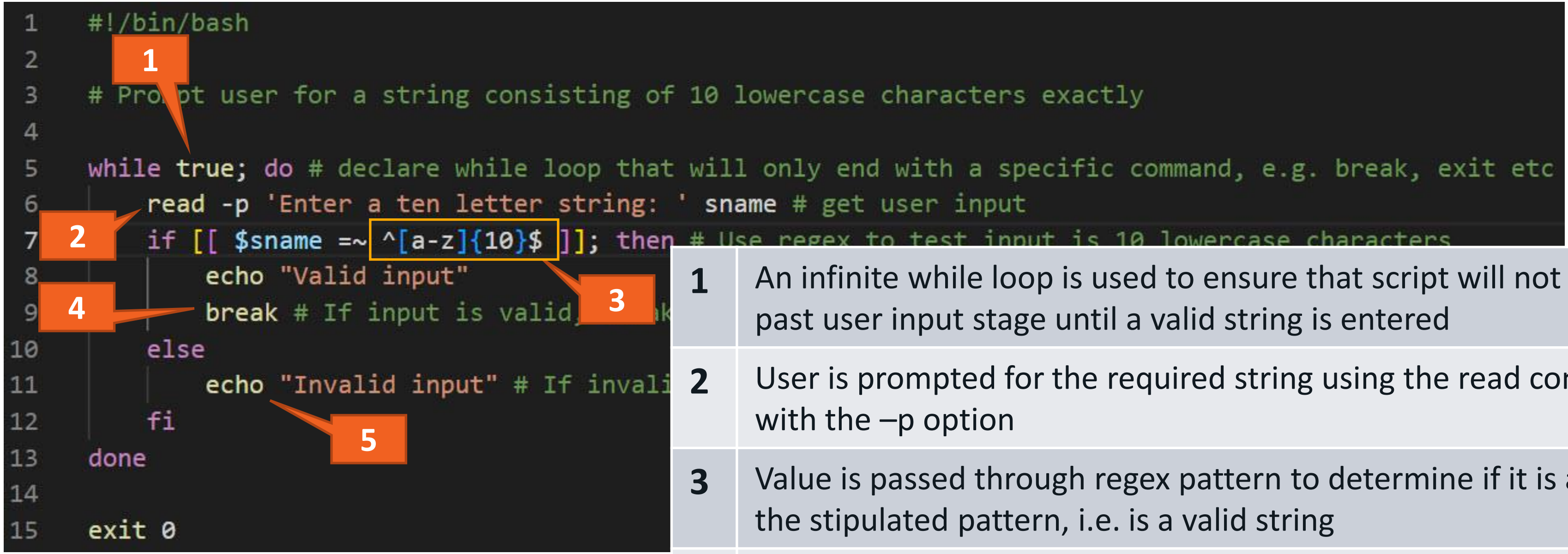
Example 3



```
#!/bin/bash

# Use a regex test for standard email addresses
# This is an example only, and this would need to be further modified for more unusual email examples

while true; do # begin loop
    read -p 'Enter an email address: ' email # prompt user for an email address
        if [[ $email =~ ^[a-zA-Z0-9._-]+@[a-zA-Z0-9-]+\.[a-zA-Z]{2,4}(\.?[a-zA-Z]{2,2})?$ ]]; then # if valid email
            break
        else
            echo "Invalid email, please try again"
        fi
done

echo "Thank you, you have entered $email" # echo the email t
exit 0
```

1	An infinite while loop is used to ensure that script will not proceed past user input stage until a valid email is entered
2	User is prompted for the required email using the read command with the –p option
3	Value is passed through regex pattern to determine if it is a match for the stipulated pattern, i.e. is a valid email
4	If a match, break is used to break out of the infinite while loop
5	If not a match, user is prompted to try again

Example 4

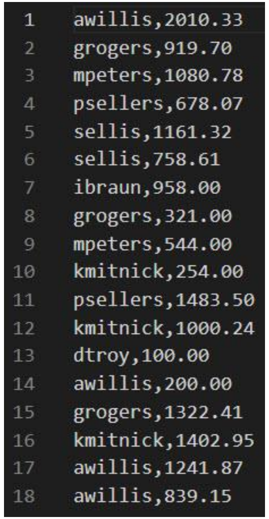


```bash
#!/bin/bash

# Prompt user for a string consisting of 10 lowercase characters exactly

while true; do # declare while loop that will only end with a specific command, e.g. break, exit etc
    read -p 'Enter a ten letter string: ' sname # get user input
    if [[ $sname =~ ^[a-z]{10}$ ]]; then # Use regex to test input is 10 lowercase characters
        echo "Valid input"
        break # If input is valid,
    else
        echo "Invalid input" # If invali
    fi
done

exit 0
```

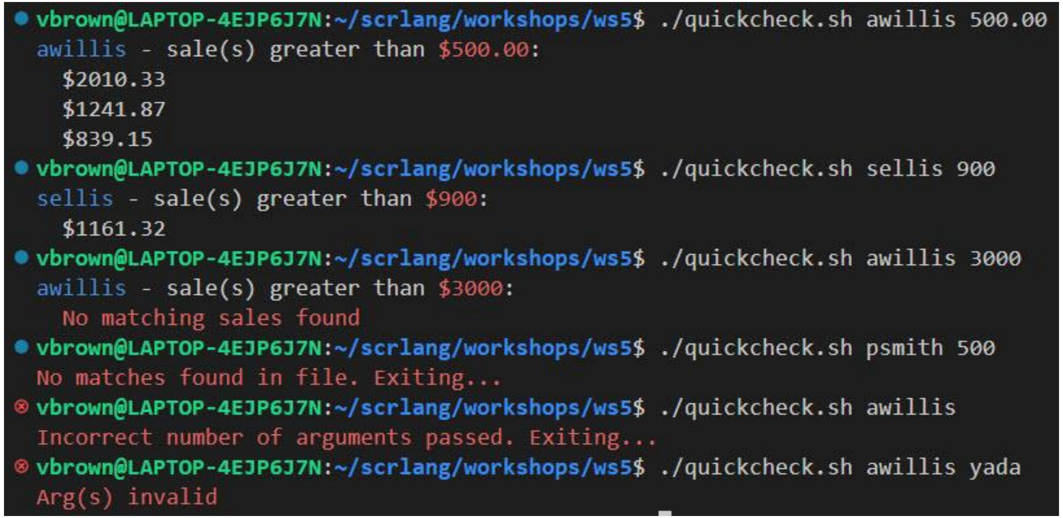
1	An infinite while loop is used to ensure that script will not proceed past user input stage until a valid string is entered
2	User is prompted for the required string using the read command with the –p option
3	Value is passed through regex pattern to determine if it is a match for the stipulated pattern, i.e. is a valid string
4	If a match, break is used to break out of the infinite while loop
5	If not a match, user is prompted to try again

Example 5

- The script example that follows retrieve the sales amounts made by a salesperson that are greater than a value provided by the user
- The user must provide a salesperson username and floor value at the command line and the scripts operates on a file named staff-sales.csv

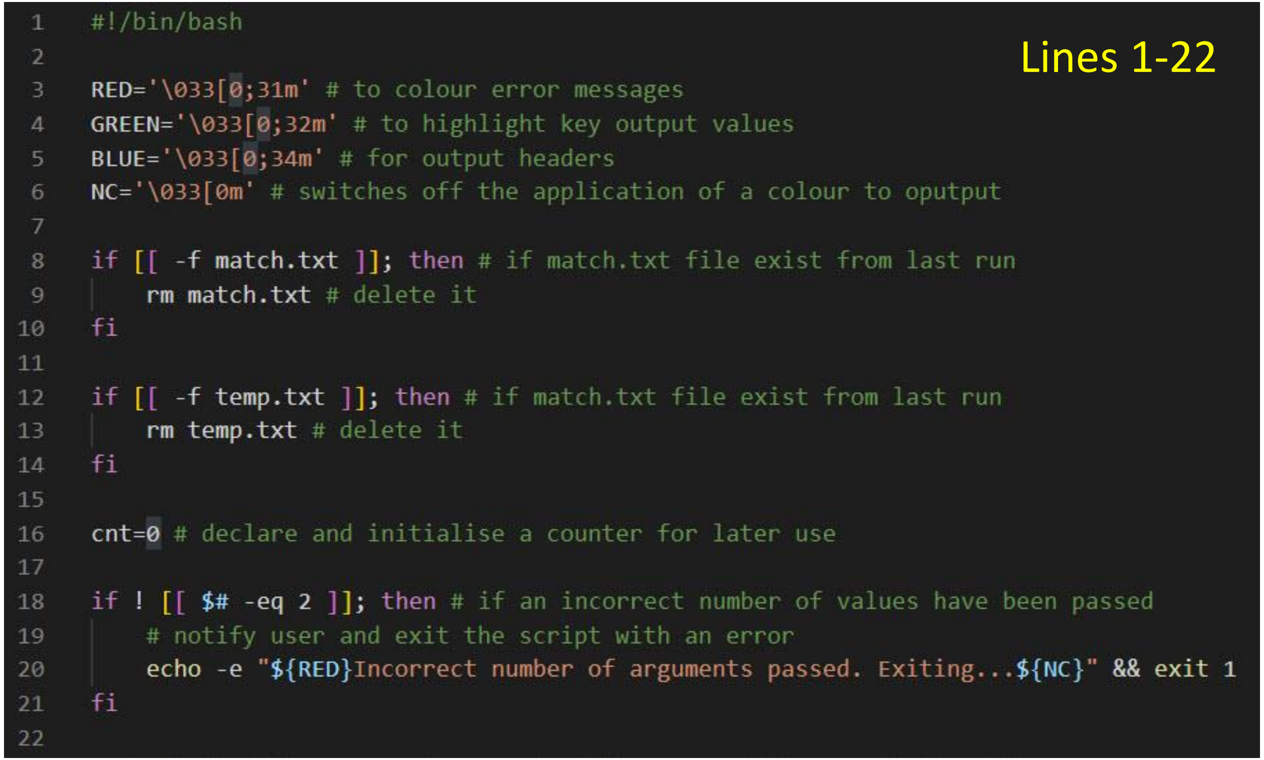


```
awillis,2010.33
grogers,919.70
mpeters,1080.78
psellers,678.07
sellis,1161.32
sellis,758.61
ibraun,958.00
grogers,321.00
mpeters,544.00
kmitnick,254.00
psellers,1483.50
kmitnick,1000.24
dtroy,100.00
awillis,200.00
grogers,1322.41
kmitnick,1402.95
awillis,1241.87
awillis,839.15
```



```
vbrown@LAPTOP-4EJP6J7N:~/scrlang/workshops/ws5$ ./quickcheck.sh awillis 500.00
awillis - sale(s) greater than $500.00:
  $2010.33
  $1241.87
  $839.15
vbrown@LAPTOP-4EJP6J7N:~/scrlang/workshops/ws5$ ./quickcheck.sh sellis 900
sellis - sale(s) greater than $900:
  $1161.32
vbrown@LAPTOP-4EJP6J7N:~/scrlang/workshops/ws5$ ./quickcheck.sh awillis 3000
awillis - sale(s) greater than $3000:
  No matching sales found
vbrown@LAPTOP-4EJP6J7N:~/scrlang/workshops/ws5$ ./quickcheck.sh psmith 500
No matches found in file. Exiting...
vbrown@LAPTOP-4EJP6J7N:~/scrlang/workshops/ws5$ ./quickcheck.sh awillis
Incorrect number of arguments passed. Exiting...
vbrown@LAPTOP-4EJP6J7N:~/scrlang/workshops/ws5$ ./quickcheck.sh awillis yada
Arg(s) invalid
```

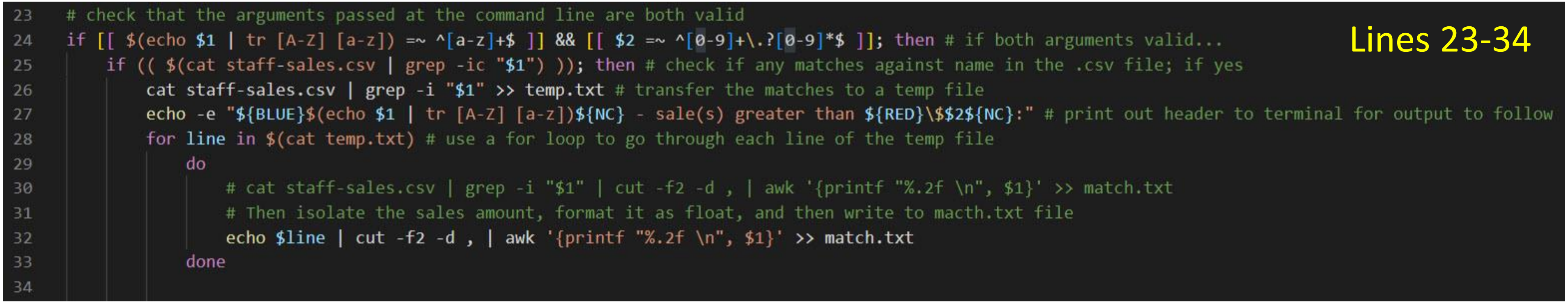
Example 5 cont…



Lines 1-22

```bash
1   #!/bin/bash
2
3   RED='\033[0;31m' # to colour error messages
4   GREEN='\033[0;32m' # to highlight key output values
5   BLUE='\033[0;34m' # for output headers
6   NC='\033[0m' # switches off the application of a colour to oputput
7
8   if [[ -f match.txt ]]; then # if match.txt file exist from last run
9       rm match.txt # delete it
10  fi
11
12  if [[ -f temp.txt ]]; then # if match.txt file exist from last run
13      rm temp.txt # delete it
14  fi
15
16  cnt=0 # declare and initialise a counter for later use
17
18  if ! [[ $# -eq 2 ]]; then # if an incorrect number of values have been passed
19      # notify user and exit the script with an error
20      echo -e "${RED}Incorrect number of arguments passed. Exiting...${NC}" && exit 1
21  fi
22
```

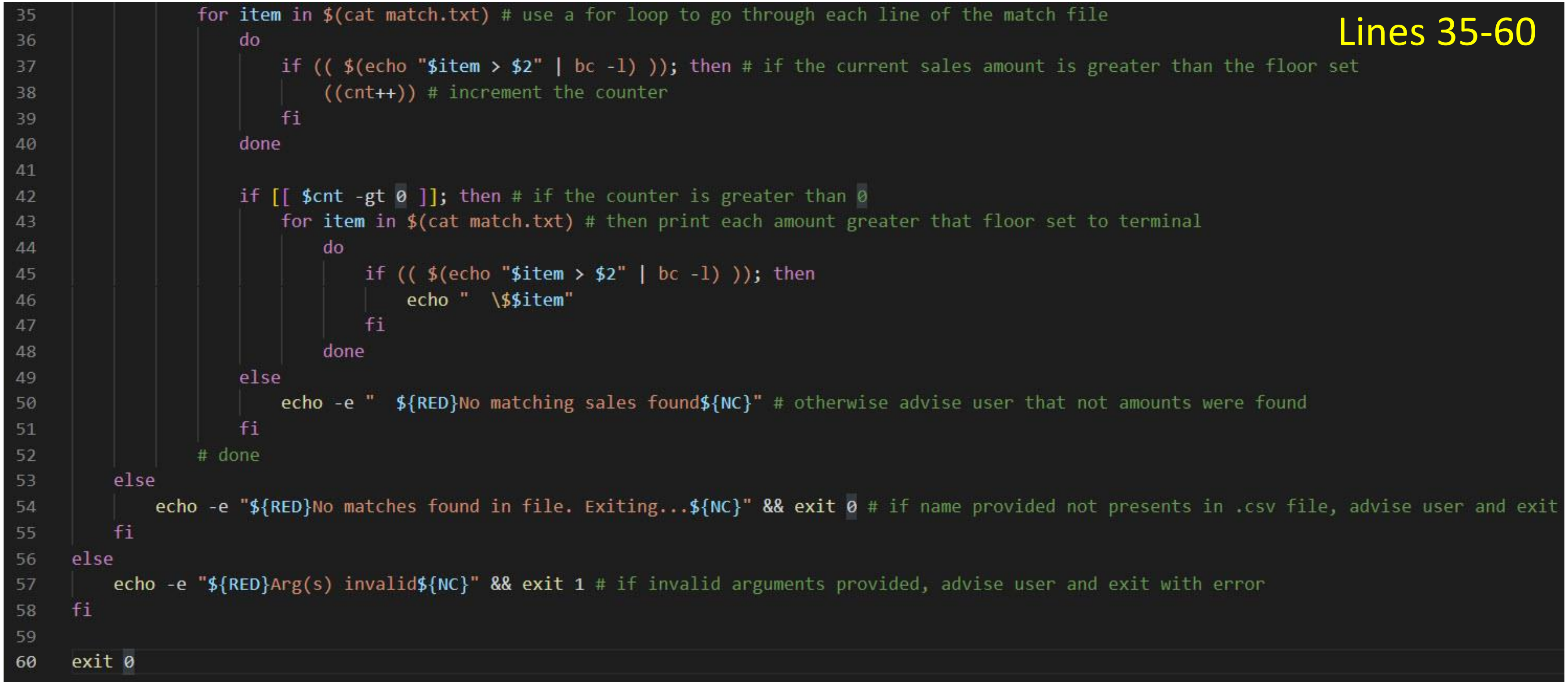
Example 5 cont…



```
23  # check that the arguments passed at the command line are both valid
24  if [[ $(echo $1 | tr [A-Z] [a-z]) =~ ^[a-z]+$ ]] && [[ $2 =~ ^[0-9]+\.?[0-9]*$ ]]; then # if both arguments valid...
25      if (( $(cat staff-sales.csv | grep -ic "$1") )); then # check if any matches against name in the .csv file; if yes
26          cat staff-sales.csv | grep -i "$1" >> temp.txt # transfer the matches to a temp file
27          echo -e "${BLUE}$(echo $1 | tr [A-Z] [a-z])${NC} - sale(s) greater than ${RED}\$$2${NC}:" # print out header to terminal for output to follow
28          for line in $(cat temp.txt) # use a for loop to go through each line of the temp file
29              do
30                  # cat staff-sales.csv | grep -i "$1" | cut -f2 -d , | awk '{printf "%.2f \n", $1}' >> match.txt
31                  # Then isolate the sales amount, format it as float, and then write to macth.txt file
32                  echo $line | cut -f2 -d , | awk '{printf "%.2f \n", $1}' >> match.txt
33              done
34
```

Lines 23-34

Example 5 cont…



Lines 35-60

```
35              for item in $(cat match.txt) # use a for loop to go through each line of the match file
36                  do
37                      if (( $(echo "$item > $2" | bc -l) )); then # if the current sales amount is greater than the floor set
38                          ((cnt++)) # increment the counter
39                      fi
40                  done
41  
42                  if [[ $cnt -gt 0 ]]; then # if the counter is greater than 0
43                      for item in $(cat match.txt) # then print each amount greater that floor set to terminal
44                          do
45                              if (( $(echo "$item > $2" | bc -l) )); then
46                                  echo "  \$$item"
47                              fi
48                          done
49                  else
50                      echo -e "  ${RED}No matching sales found${NC}" # otherwise advise user that not amounts were found
51                  fi
52              # done
53      else
54          echo -e "${RED}No matches found in file. Exiting...${NC}" && exit 0 # if name provided not presents in .csv file, advise user and exit
55      fi
56  else
57      echo -e "${RED}Arg(s) invalid${NC}" && exit 1 # if invalid arguments provided, advise user and exit with error
58  fi
59  
60  exit 0
```

Summary

Terms to Know

- Regular Expressions
- Regular Expression Engines
- Grep and Regex
- Anchors and Wildcards
- Extended RegEx Engine
- ERE Repetition and Optionality
- OR and Expression Grouping
- Common Grep Options
- Piping and Redirection