School of
**Science**

# Scripting Languages

## Module 8

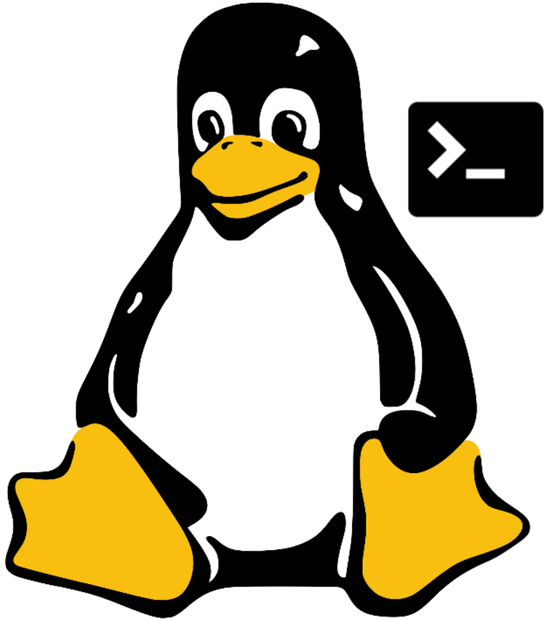
curl, wget and Manipulating Data Using Awk

## Contents

## Learning Objectives

After finishing this module, you should be able to:

- Execute scripts that use wget and curl
- Execute scripts that use AWK
- Process streams using either sed or AWK or a combination of the two
- Execute scripts that use AWK to process strings and real numbers
- Use AWK in conjunction with other commands such as grep and sed, and utilising regular expressions to produce required solutions



# wget and curl

# wget

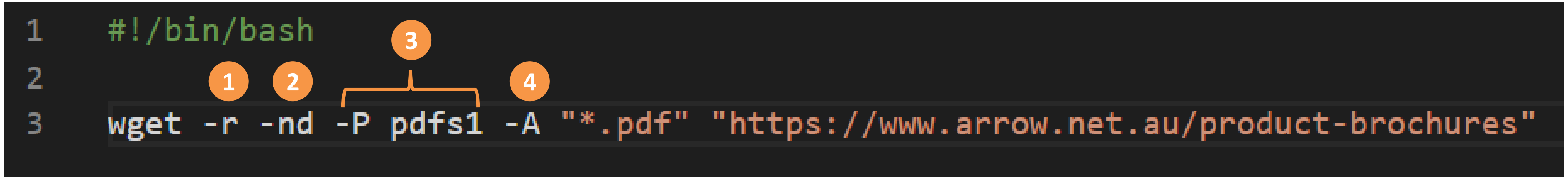
- **wget** is a utility for non-interactive download of files from the web. It supports HTTP, HTTPS and FTP
- wget can follow HTML/XHTML links and reproduce a local version of remote website, duplicating the directory structure precisely in a process known as *recursive downloading*
- The most fundamental way to use wget is to provide it with the address of a file you wish to download to your local machine

EXAMPLE:

```
$ wget https://www.asite.com/pdfs/prod_brochure.pdf
```

- This command would download the file into the same directory that the .sh script is running in

## wget



```bash
#!/bin/bash

wget -r -nd -P pdfs1 -A "*.pdf" "https://www.arrow.net.au/product-brochures"
```

|  | Command | Full Name | Function |
|---|---|---|---|
| **1** | r | --recursive | Turn on recursive retrieving. The default maximum depth is 5. |
| **2** | nd | --no-directories | Do not create a hierarchy of directories when retrieving recursively. With this option turned on, all files will get saved to the current directory unless another directory is specified |
| **3** | P *prefix* | --directory-prefix=prefix | Set directory prefix to prefix. The directory prefix is the directory where all other files and subdirectories will be saved to, i.e. the top of the retrieval tree. The default is . (the current directory). |
| **4** | A *acclist* | --accept acclist | Specify comma-separated lists of file name suffixes or patterns to accept. If any of the wildcard characters, *, ?, [ or ], appear in an element of acclist, it will be treated as a pattern, rather than a suffix. The pattern must be enclosed in quotes to prevent shell expansion. |

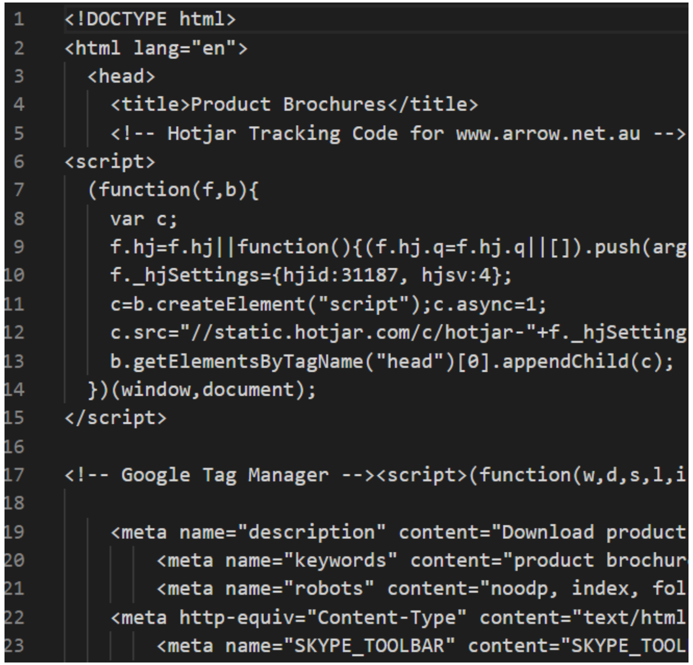
## curl

- **curl** supports piping and operates similarly to the cat command, passing data to stdout and accepting data from stdin
- curl performs single-shot transfers of data and user-specified URLs, and does not possess recursive downloading or HTML parsing capability
- curl supports more protocols that wget (HTTP, HTTPS and FTP support only) including FTPS, Gopher, SCP, SFTP, TFTP, TELNET, DICT, LDAP, LDAPS, FILE, POP3, IMAP, SMB/CIFS, SMTP, RTMP and RTSP
- curl supports more HTTP authentication methods, especially over HTTP proxies: Basic, Digest, NTLM and Negotiate as well as several SOCKS protocol versions for proxy access
- curl is bidirectional possessing upload and sending capabilities, HTTP multipart/form-data sending, gzip and deflate Content-Encoding and automatic decompression

# curl

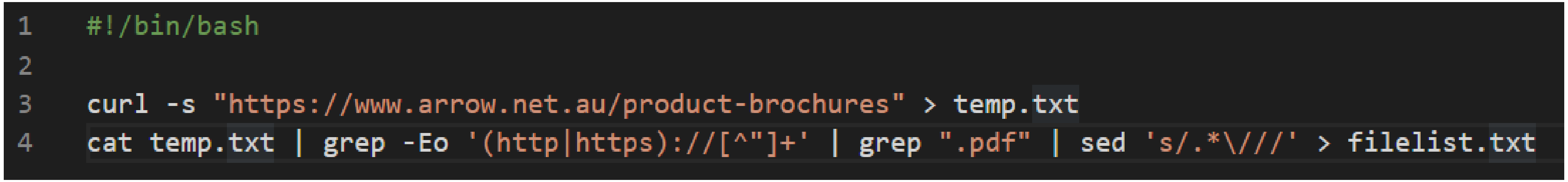
```
curl -s "https://www.arrow.net.au/product-brochures" > temp.txt
```

- This command downloads the *.htm/.html/.php* file (code) the URL points to and writes it to a file named *temp.txt* on-the-fly
- The –s means silent, which suppresses the curl download connection and progression data from being echoed to the screen.



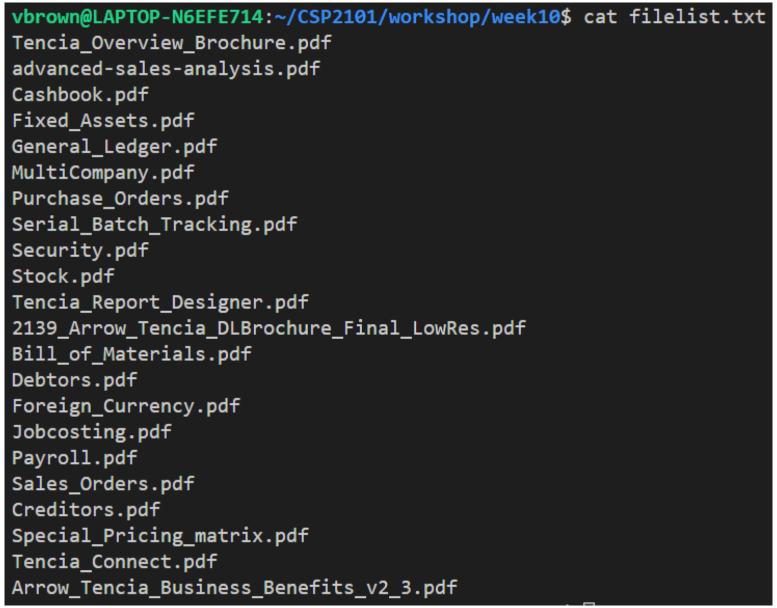
```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>Product Brochures</title>
    <!-- Hotjar Tracking Code for www.arrow.net.au -->
<script>
  (function(f,b){
    var c;
    f.hj=f.hj||function(){(f.hj.q=f.hj.q||[]).push(arg
    f._hjSettings={hjid:31187, hjsv:4};
    c=b.createElement("script");c.async=1;
    c.src="//static.hotjar.com/c/hotjar-"+f._hjSetting
    b.getElementsByTagName("head")[0].appendChild(c);
  })(window,document);
</script>

<!-- Google Tag Manager --><script>(function(w,d,s,l,i

    <meta name="description" content="Download product
      <meta name="keywords" content="product brochur
      <meta name="robots" content="noodp, index, fol
    <meta http-equiv="Content-Type" content="text/html
      <meta name="SKYPE_TOOLBAR" content="SKYPE_TOOL
```

# Curl applied

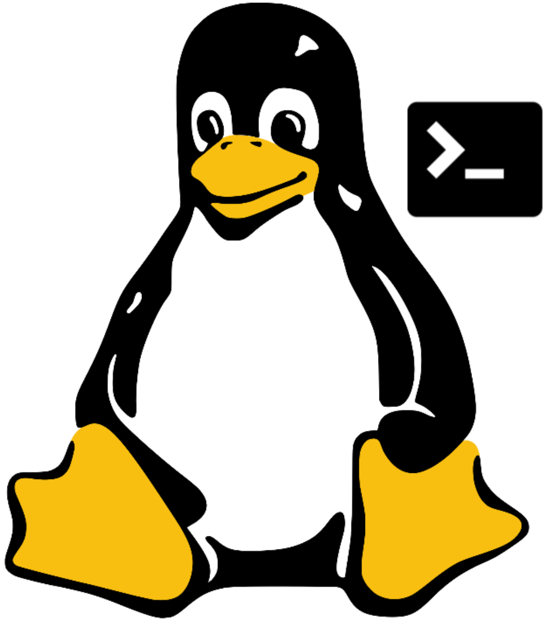


```bash
#!/bin/bash

curl -s "https://www.arrow.net.au/product-brochures" > temp.txt
cat temp.txt | grep -Eo '(http|https)://[^"]+' | grep ".pdf" | sed 's/.*\///' > filelist.txt
```

1. Acquire URL file (code)
2. Write to text file
3. Pass contents of text file to grep to:
   - **a.** **Strip** all *non-url* elements
   - **b.** **Strip** all lines that don't contain ".pdf"
   - **c.** **Strip** all *paths* leaving only the file names themselves



```
vbrown@LAPTOP-N6EFE714:~/CSP2101/workshop/week10$ cat filelist.txt
Tencia_Overview_Brochure.pdf
advanced-sales-analysis.pdf
Cashbook.pdf
Fixed_Assets.pdf
General_Ledger.pdf
MultiCompany.pdf
Purchase_Orders.pdf
Serial_Batch_Tracking.pdf
Security.pdf
Stock.pdf
Tencia_Report_Designer.pdf
2139_Arrow_Tencia_DLBrochure_Final_LowRes.pdf
Bill_of_Materials.pdf
Debtors.pdf
Foreign_Currency.pdf
Jobcosting.pdf
Payroll.pdf
Sales_Orders.pdf
Creditors.pdf
Special_Pricing_matrix.pdf
Tencia_Connect.pdf
Arrow_Tencia_Business_Benefits_v2_3.pdf
```



## Introduction to **awk**

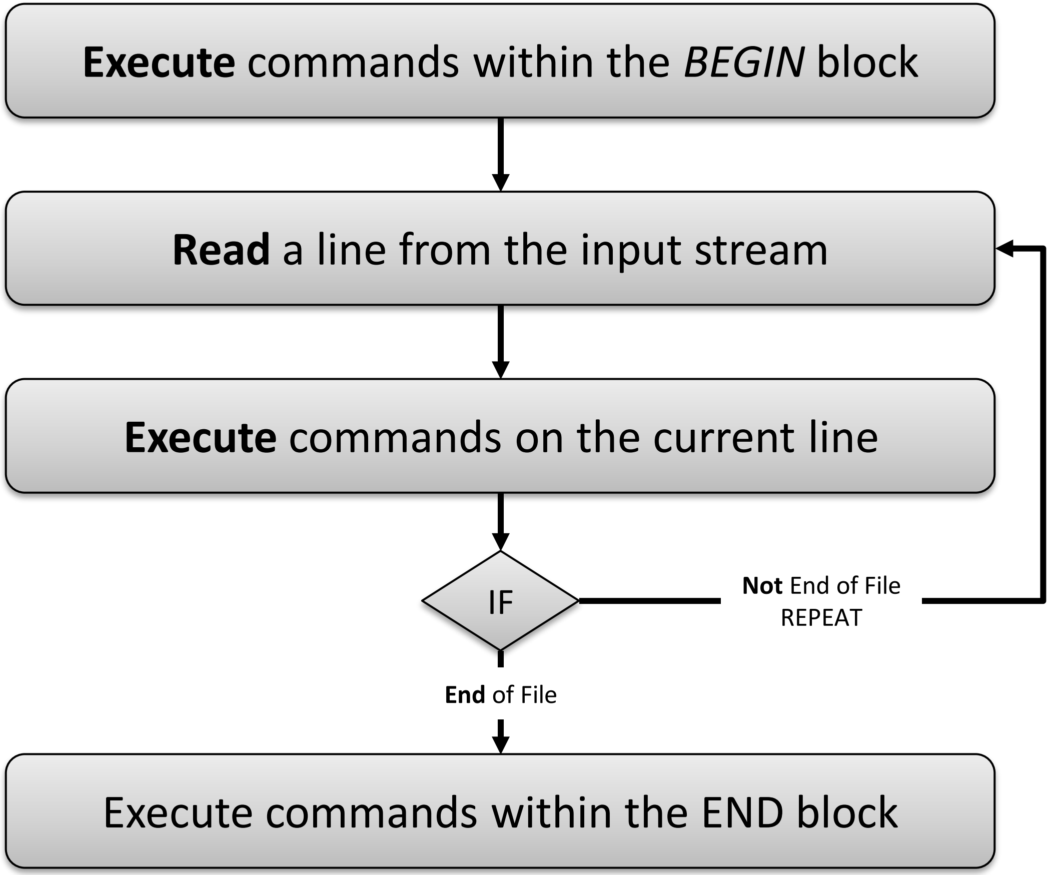
## awk

- AWK is a highly-versatile utility that can be used to perform a wide range of useful tasks within the bash environment, including:
  - ✓ Text processing,
  - ✓ Producing formatted text reports,
  - ✓ Performing arithmetic operations,
  - ✓ Performing string operations
- Unlike other key utilities such as sed and grep, AWK is actually an entire programming language in its own right, in which you can:
  - ✓ Define variables
  - ✓ Use string and arithmetic operators
  - ✓ Use control flow and loops

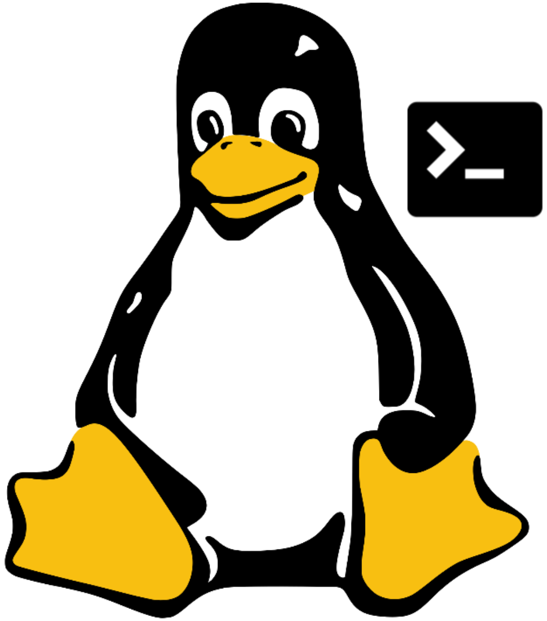
## awk History

- AWK was first developed by bell labs in the 1970s
- It was named after the three programmers who originally designed it Alfred Aho, Peter Weinberger and Brian Kernighan (AWK)
- Over the years there have been many different implementations of awk, but the two most common forms in use today are:
  - gawk (used in linux based systems)
  - BWK (used in bsd and MacOS based systems)

## The AWK workflow



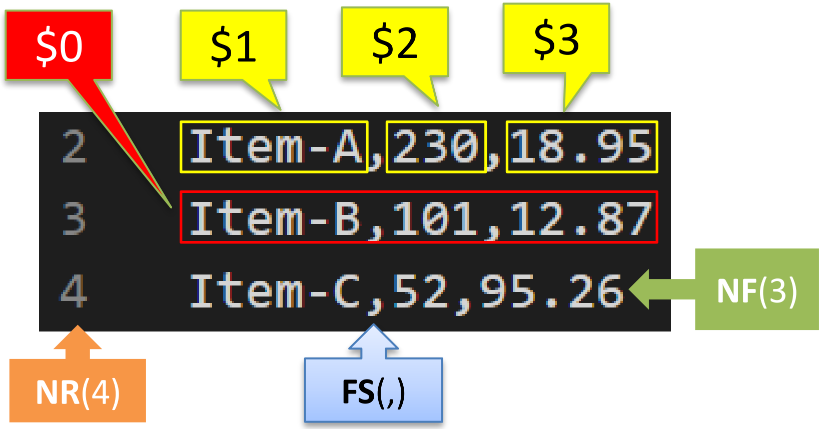
**Execute** commands within the *BEGIN* block
**Read** a line from the input stream
**Execute** commands on the current line
IF
**Not** End of File
REPEAT
**End** of File
Execute commands within the END block



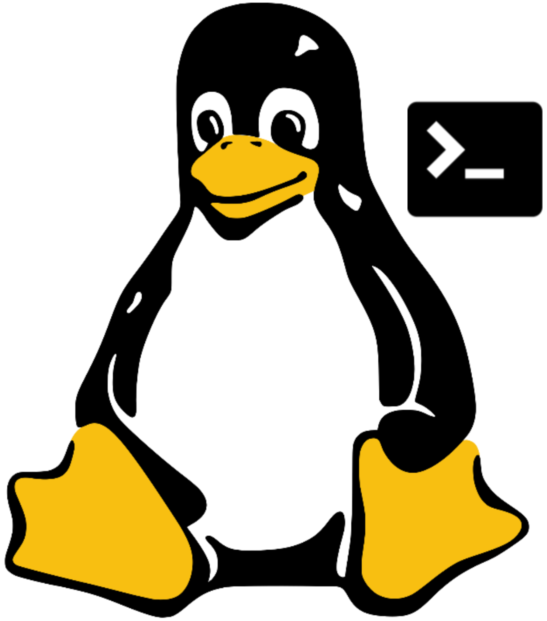
## **awk** default variables

# AWK Default Variables

| Variable | Purpose |
|---|---|
| $0 | Entire record |
| $1 | Field #1 in record |
| $2 | Field #2 in record |
| $3 | Field #3 in record |
| *…and so on* | |
| NF | Stores # fields in record |
| NR | Stores line number of each record |
| FS | File separator |



```
Item-A,230,18.95
Item-B,101,12.87
Item-C,52,95.26
```



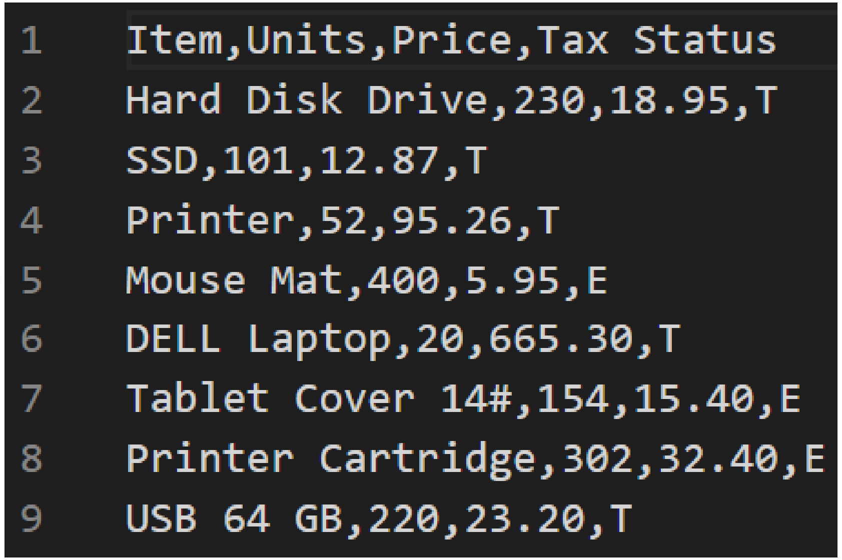
## awk Structure

## The AWK Structure

```
awk BEGIN {awk-commands}
/pattern/ {commands}
END {awk-commands}
```

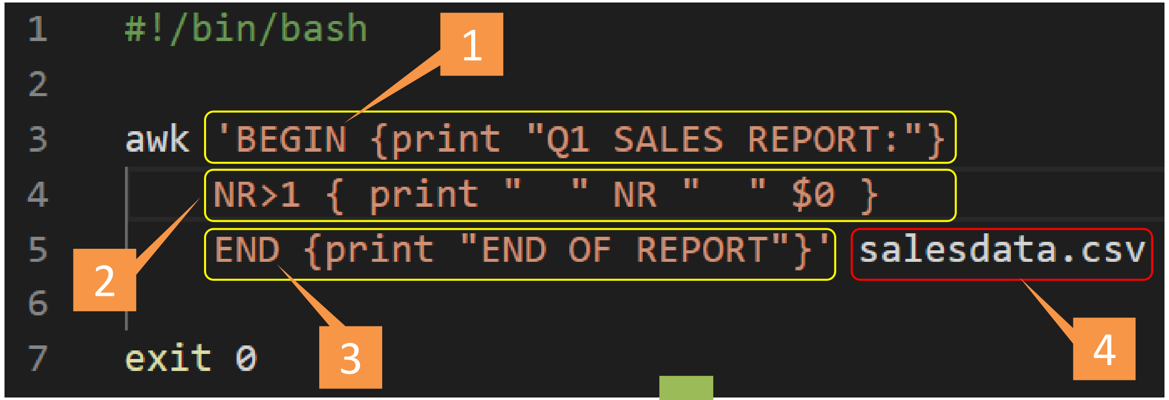
▇ *optional elements*

We will use the data in **salesdata.csv** to demonstrate awk in action

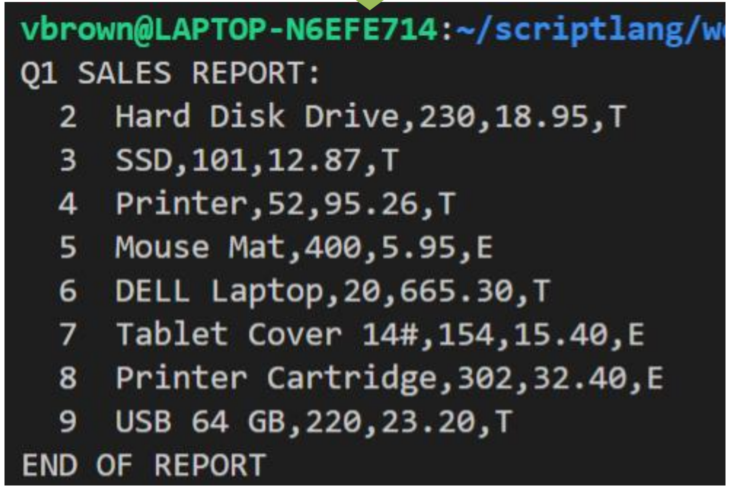


```
Item,Units,Price,Tax Status
Hard Disk Drive,230,18.95,T
SSD,101,12.87,T
Printer,52,95.26,T
Mouse Mat,400,5.95,E
DELL Laptop,20,665.30,T
Tablet Cover 14#,154,15.40,E
Printer Cartridge,302,32.40,E
USB 64 GB,220,23.20,T
```

# The awk pattern-command structure

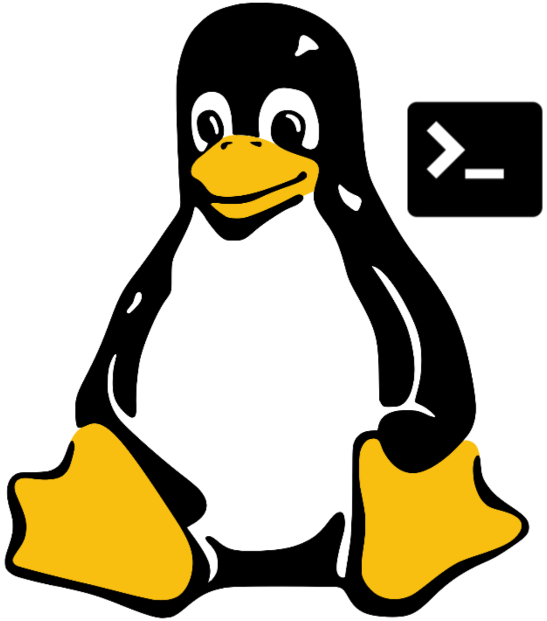


```
#!/bin/bash

awk 'BEGIN {print "Q1 SALES REPORT:"}
     NR>1 { print "   " NR "   " $0 }
     END {print "END OF REPORT"}' salesdata.csv

exit 0
```



```
vbrown@LAPTOP-N6EFE714:~/scriptlang/w
Q1 SALES REPORT:
   2   Hard Disk Drive,230,18.95,T
   3   SSD,101,12.87,T
   4   Printer,52,95.26,T
   5   Mouse Mat,400,5.95,E
   6   DELL Laptop,20,665.30,T
   7   Tablet Cover 14#,154,15.40,E
   8   Printer Cartridge,302,32.40,E
   9   USB 64 GB,220,23.20,T
END OF REPORT
```

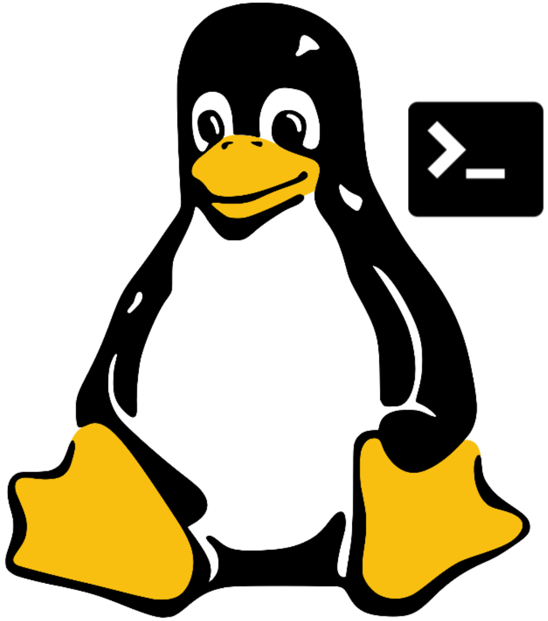
1. BEGIN pattern: Actions awk will execute **once** *before* any input lines are read. The BEGIN pattern is optional.
2. MAIN PATTERN/ACTION: Specifies what is to be done with each line, or specific fields within each line contained with the data handed to awk. One or more statements within curly braces { } are compulsory.
3. END pattern: Actions awk will execute **once** *after* any input lines are read. The END pattern is optional.
4. The DATA SOURCE: Specified **after** the *close* }' of the awk structure if data has not be piped through prior from another process



## awk Formatting

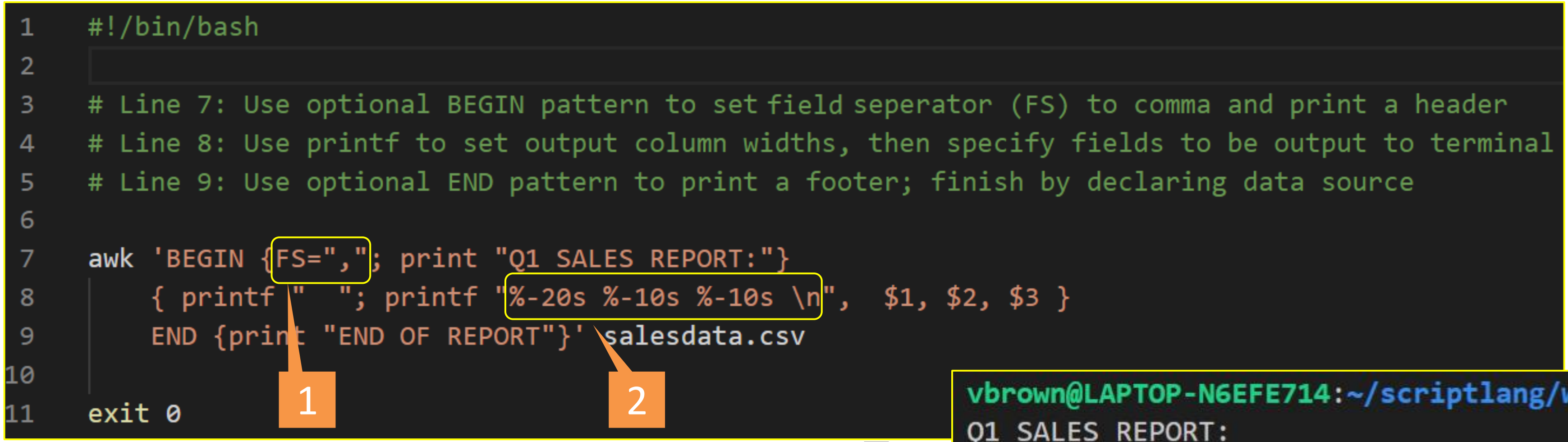
## printf format codes

| Code | Result |
|---|---|
| %s | String |
| %5s | String with a minimum of 5 characters |
| %f | Number (floating point) |
| %5f | Number with a minimum of 5 characters |
| %5.2f | Number with a minimum of 5 characters<br>and 2 decimal places |
| %d | Whole number (Decimal Integer) |
| %c | Single Character |

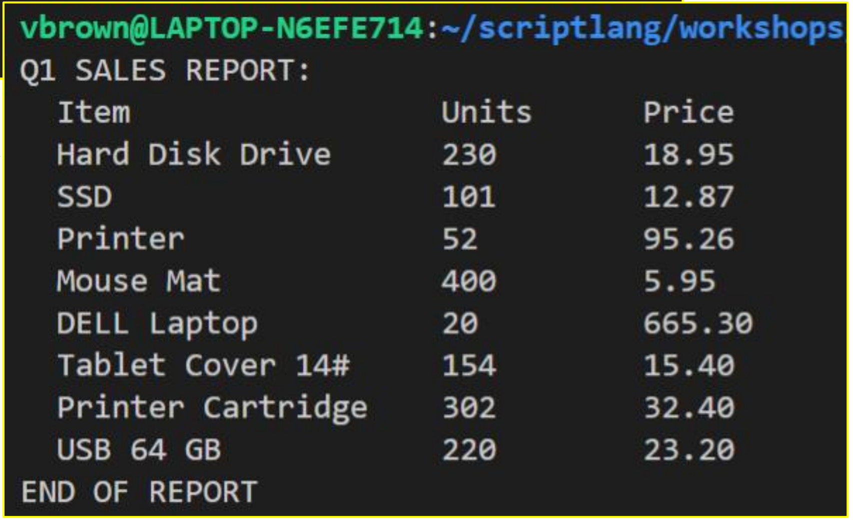


## awk in action

# Setting the field separator and output format



```
#!/bin/bash

# Line 7: Use optional BEGIN pattern to set field seperator (FS) to comma and print a header
# Line 8: Use printf to set output column widths, then specify fields to be output to terminal
# Line 9: Use optional END pattern to print a footer; finish by declaring data source

awk 'BEGIN {FS=","; print "Q1 SALES REPORT:"}
    { printf "  "; printf "%-20s %-10s %-10s \n",  $1, $2, $3 }
    END {print "END OF REPORT"}' salesdata.csv

exit 0
```

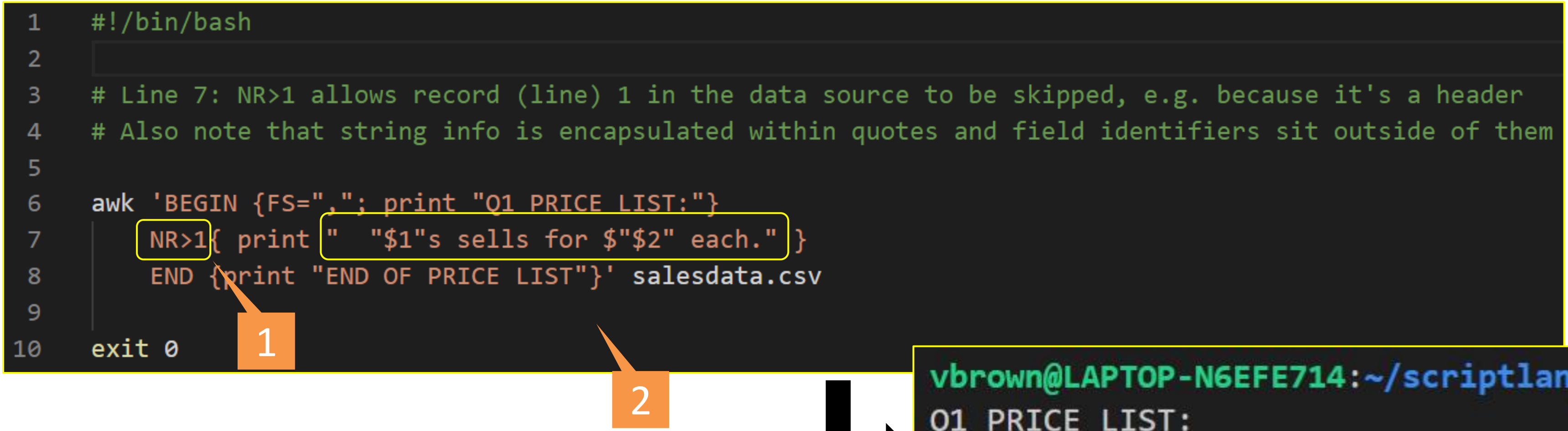


```
vbrown@LAPTOP-N6EFE714:~/scriptlang/workshops
Q1 SALES REPORT:
  Item                 Units      Price
  Hard Disk Drive      230        18.95
  SSD                  101        12.87
  Printer              52         95.26
  Mouse Mat            400        5.95
  DELL Laptop          20         665.30
  Tablet Cover 14#     154        15.40
  Printer Cartridge    302        32.40
  USB 64 GB            220        23.20
END OF REPORT
```

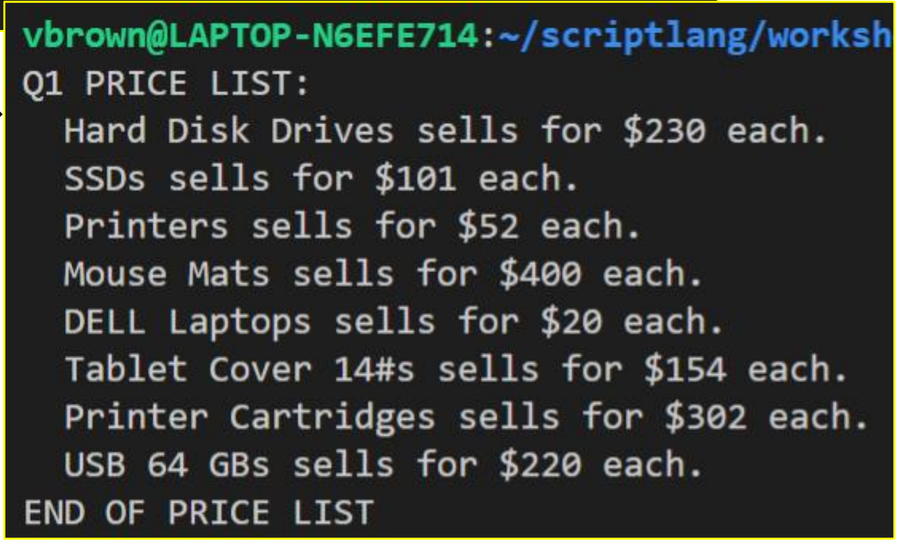
1. The FS="," pattern tells awk that the field separator in the data source is a comma

2. The `%-20s %-10s %-10s \n` pattern provides awk with the formatting pattern of the field output commands that immediately follow

# Specifying records and inserting string info



```
1   #!/bin/bash
2
3   # Line 7: NR>1 allows record (line) 1 in the data source to be skipped, e.g. because it's a header
4   # Also note that string info is encapsulated within quotes and field identifiers sit outside of them
5
6   awk 'BEGIN {FS=","; print "Q1 PRICE LIST:"}
7       NR>1{ print "  "$1"s sells for $"$2" each." }
8       END {print "END OF PRICE LIST"}' salesdata.csv
9
10  exit 0
```

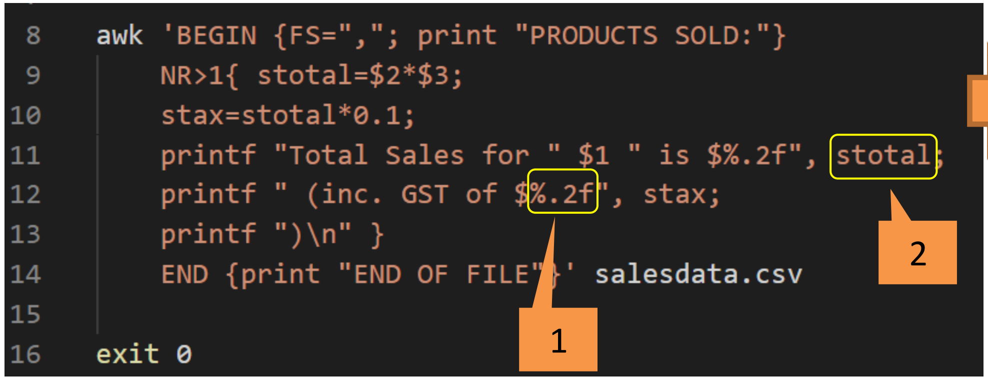


```
vbrown@LAPTOP-N6EFE714:~/scriptlang/worksh
Q1 PRICE LIST:
  Hard Disk Drives sells for $230 each.
  SSDs sells for $101 each.
  Printers sells for $52 each.
  Mouse Mats sells for $400 each.
  DELL Laptops sells for $20 each.
  Tablet Cover 14#s sells for $154 each.
  Printer Cartridges sells for $302 each.
  USB 64 GBs sells for $220 each.
END OF PRICE LIST
```

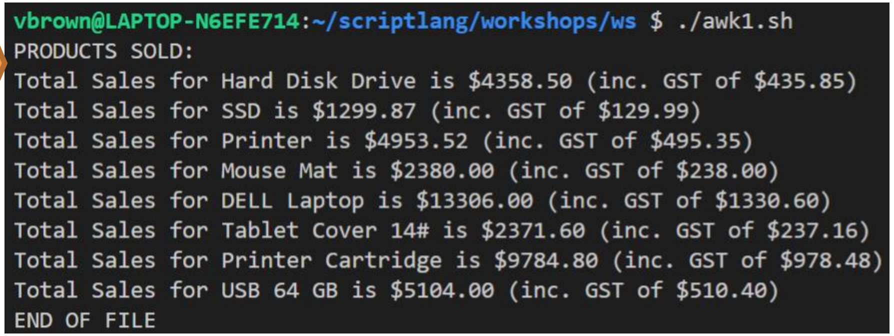
1. The NR>1 test tells awk to *skip* the first record (line) in the source file, in this case because it is an unwanted header

2. Also note that string info is *encapsulated* within quotes " " and field identifiers, e.g. $1 $2 sit outside of them

# Declared variables and output formatting



```
 8    awk 'BEGIN {FS=","; print "PRODUCTS SOLD:"}
 9        NR>1{ stotal=$2*$3;
10        stax=stotal*0.1;
11        printf "Total Sales for " $1 " is $%.2f", stotal;
12        printf " (inc. GST of $%.2f", stax;
13        printf ")\n" }
14        END {print "END OF FILE"}' salesdata.csv
15
16    exit 0
```



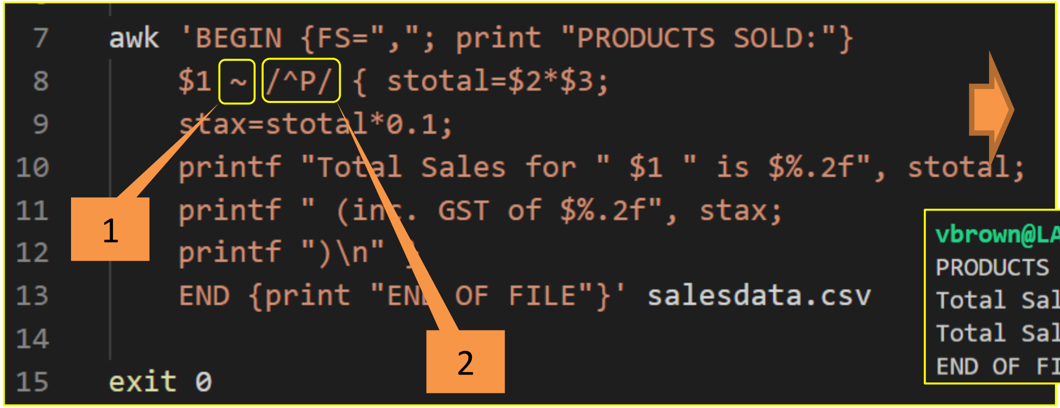
```
vbrown@LAPTOP-N6EFE714:~/scriptlang/workshops/ws $ ./awk1.sh
PRODUCTS SOLD:
Total Sales for Hard Disk Drive is $4358.50 (inc. GST of $435.85)
Total Sales for SSD is $1299.87 (inc. GST of $129.99)
Total Sales for Printer is $4953.52 (inc. GST of $495.35)
Total Sales for Mouse Mat is $2380.00 (inc. GST of $238.00)
Total Sales for DELL Laptop is $13306.00 (inc. GST of $1330.60)
Total Sales for Tablet Cover 14# is $2371.60 (inc. GST of $237.16)
Total Sales for Printer Cartridge is $9784.80 (inc. GST of $978.48)
Total Sales for USB 64 GB is $5104.00 (inc. GST of $510.40)
END OF FILE
```

**Line 9**: Declare a variable named **stotal** to which *product* of field $2 and field $3 is assigned
**Line 10**: Declare a variable named **stax** to which *10% of stotal* is assigned
**Line 11**: Print product name and its sales total to terminal formatted to two decimal places
**Line 12**: Print GST component to terminal formatted to two decimal places

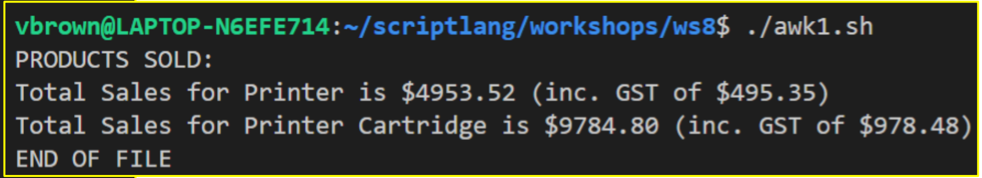
**Note:**

1. The `%.2f` code formats the value that follows, i.e. `stotal` to two (2) decimal places
2. ‘Programmer declared’ awk variables do **not** use the *$* prepend, either when declared or when used in awk command sequences

# Identifying specific records based on a regex



```
 7  awk 'BEGIN {FS=","; print "PRODUCTS SOLD:"}
 8      $1 ~ /^P/ { stotal=$2*$3;
 9      stax=stotal*0.1;
10      printf "Total Sales for " $1 " is $%.2f", stotal;
11      printf " (inc. GST of $%.2f", stax;
12      printf ")\n" }
13      END {print "END OF FILE"}' salesdata.csv
14
15  exit 0
```

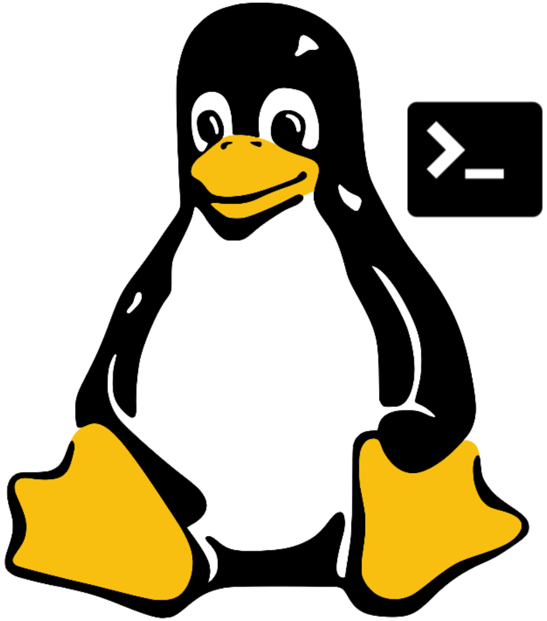


```
vbrown@LAPTOP-N6EFE714:~/scriptlang/workshops/ws8$ ./awk1.sh
PRODUCTS SOLD:
Total Sales for Printer is $4953.52 (inc. GST of $495.35)
Total Sales for Printer Cartridge is $9784.80 (inc. GST of $978.48)
END OF FILE
```

**Line 8:** Declares a pattern that determines which records (lines) will be acted upon by awk commands. The pattern in this case is those records (lines) in which field **$1** *starts* with uppercase *P*. The pattern is placed immediately **before** awk's *main command block*, i.e. just before the opening {

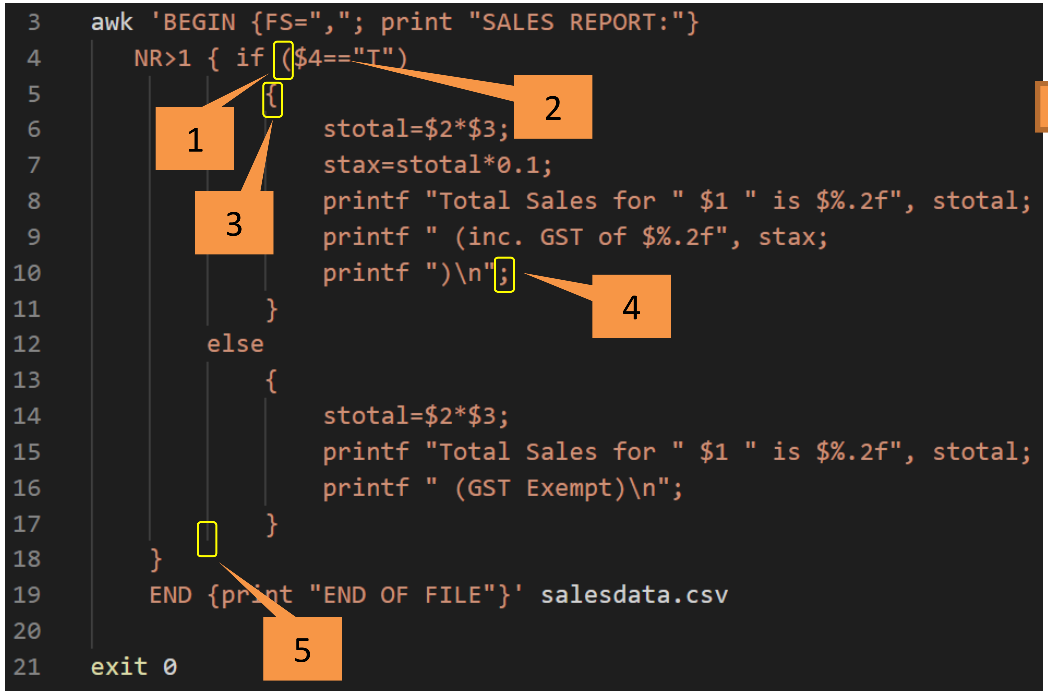
**Note:**

1. The ~ symbol in the context means contains. !~ inverts this, i.e. does **not** contain
2. In awk, the regular expression pattern is delimited by forward slashes /^P/

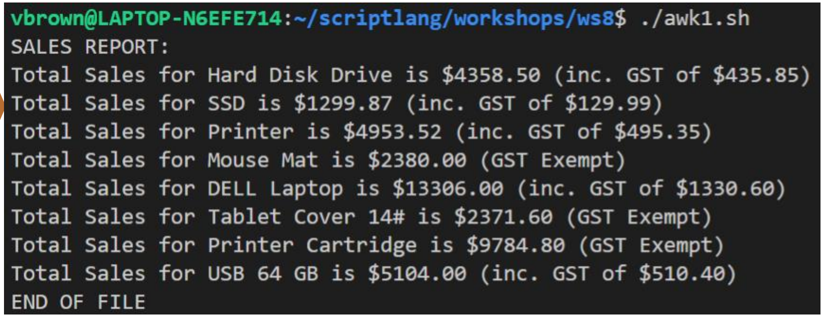


## **awk** Conditional Statements

# Using an IF control structure in awk



```
awk 'BEGIN {FS=","; print "SALES REPORT:"}
    NR>1 { if ($4=="T")
            {
                stotal=$2*$3;
                stax=stotal*0.1;
                printf "Total Sales for " $1 " is $%.2f", stotal;
                printf " (inc. GST of $%.2f", stax;
                printf ")\n";
            }
        else
            {
                stotal=$2*$3;
                printf "Total Sales for " $1 " is $%.2f", stotal;
                printf " (GST Exempt)\n";
            }
    }
    END {print "END OF FILE"}' salesdata.csv

exit 0
```

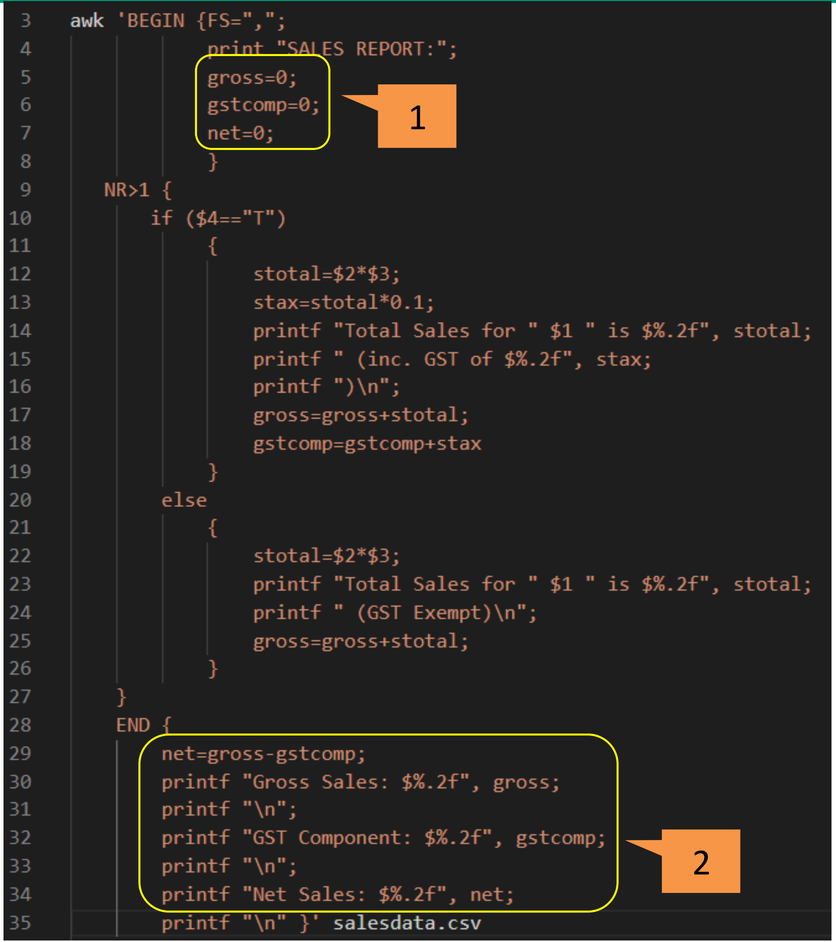


```
vbrown@LAPTOP-N6EFE714:~/scriptlang/workshops/ws8$ ./awk1.sh
SALES REPORT:
Total Sales for Hard Disk Drive is $4358.50 (inc. GST of $435.85)
Total Sales for SSD is $1299.87 (inc. GST of $129.99)
Total Sales for Printer is $4953.52 (inc. GST of $495.35)
Total Sales for Mouse Mat is $2380.00 (GST Exempt)
Total Sales for DELL Laptop is $13306.00 (inc. GST of $1330.60)
Total Sales for Tablet Cover 14# is $2371.60 (GST Exempt)
Total Sales for Printer Cartridge is $9784.80 (GST Exempt)
Total Sales for USB 64 GB is $5104.00 (inc. GST of $510.40)
END OF FILE
```

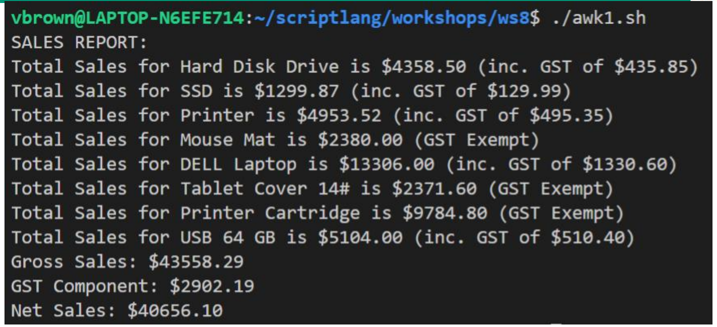
1. Logical test is encapsulated within started parenthesis
2. Assignations differ from shell script, more like PHP and others
3. Command must be enclosed in { } (unless only one command)
4. Commands must be terminated with semi-colon ;
5. If structure does not require close statement, e.g. endif

**Note:**
If more than two (2) options apply, then `else if (condition-x)` is used

# A more complex awk example



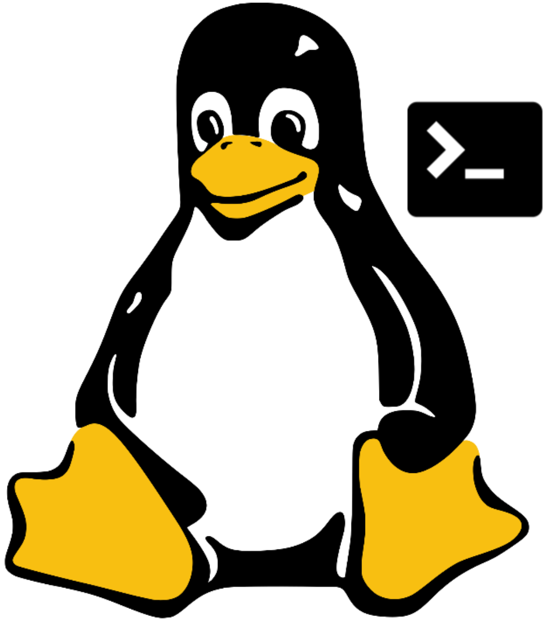
```
awk 'BEGIN {FS=",";
            print "SALES REPORT:";
            gross=0;
            gstcomp=0;
            net=0;
            }
  NR>1 {
      if ($4=="T")
            {
                stotal=$2*$3;
                stax=stotal*0.1;
                printf "Total Sales for " $1 " is $%.2f", stotal;
                printf " (inc. GST of $%.2f", stax;
                printf ")\n";
                gross=gross+stotal;
                gstcomp=gstcomp+stax
            }
        else
            {
                stotal=$2*$3;
                printf "Total Sales for " $1 " is $%.2f", stotal;
                printf " (GST Exempt)\n";
                gross=gross+stotal;
            }
   }
   END {
        net=gross-gstcomp;
        printf "Gross Sales: $%.2f", gross;
        printf "\n";
        printf "GST Component: $%.2f", gstcomp;
        printf "\n";
        printf "Net Sales: $%.2f", net;
        printf "\n" }' salesdata.csv
```



```
vbrown@LAPTOP-N6EFE714:~/scriptlang/workshops/ws8$ ./awk1.sh
SALES REPORT:
Total Sales for Hard Disk Drive is $4358.50 (inc. GST of $435.85)
Total Sales for SSD is $1299.87 (inc. GST of $129.99)
Total Sales for Printer is $4953.52 (inc. GST of $495.35)
Total Sales for Mouse Mat is $2380.00 (GST Exempt)
Total Sales for DELL Laptop is $13306.00 (inc. GST of $1330.60)
Total Sales for Tablet Cover 14# is $2371.60 (GST Exempt)
Total Sales for Printer Cartridge is $9784.80 (GST Exempt)
Total Sales for USB 64 GB is $5104.00 (inc. GST of $510.40)
Gross Sales: $43558.29
GST Component: $2902.19
Net Sales: $40656.10
```

1. The optional BEGIN statement can be used to *declare* and *initialise* variables that are to be used **across** *all records* collectively, rather than record by record, e.g. `gross`, `gstcomp` and `net`

2. The optional END statement can be used to *calculate totals* **after** all records have been processed for *summary* purposes



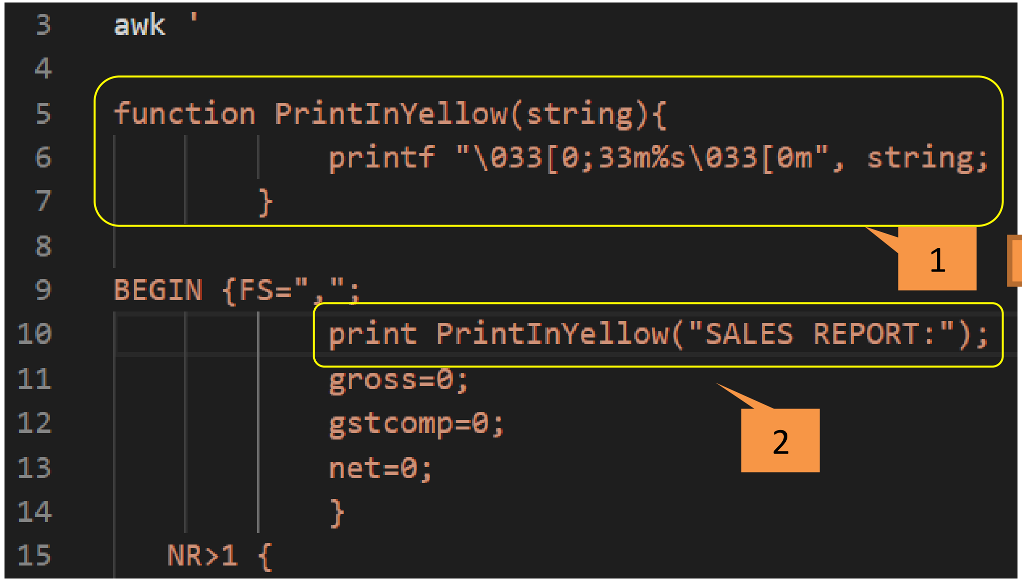
## awk Functions

## Functions

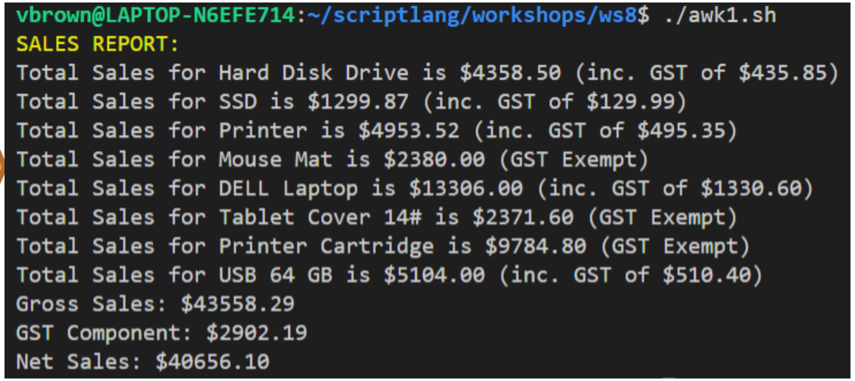
- Functions in awk behave in a similar way to functions in bash
- Just like shell script, awk functions are useful for breaking scripts up into logical modules and reducing the need for repeated code

| Function | Purpose |
|---|---|
| sin() | Sine |
| cos() | Cosine |
| tan() | Tangent |
| sqrt() | Square root |
| exp() | Exponential |
| log() | Logarithm |
| rand() | Random Number Generator |
| length() | String length |
| split() | String splitter |
| toupper() | Convert string to uppercase |
| tolower() | Convert string to lowercase |

# Custom awk functions

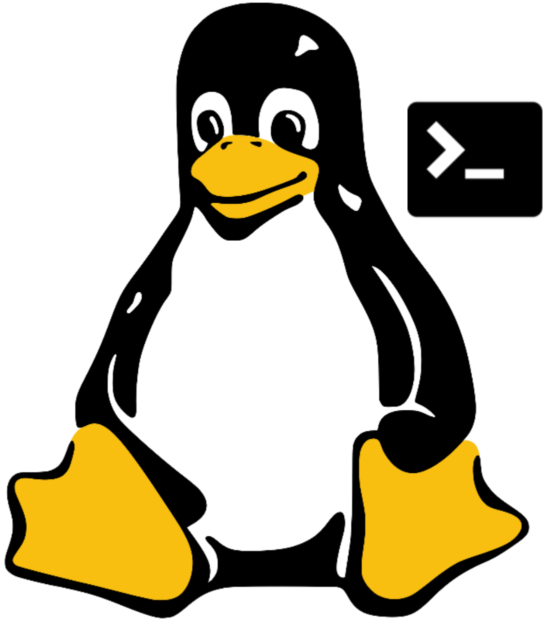


```
awk '

function PrintInYellow(string){
            printf "\033[0;33m%s\033[0m", string;
        }

BEGIN {FS=",";
            print PrintInYellow("SALES REPORT:");
            gross=0;
            gstcomp=0;
            net=0;
            }
    NR>1 {
```



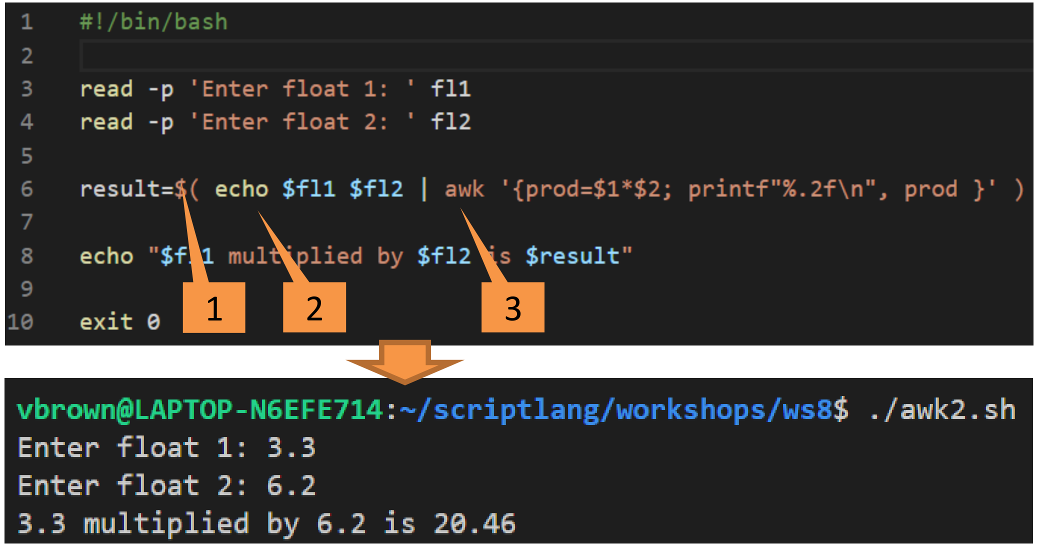
```
vbrown@LAPTOP-N6EFE714:~/scriptlang/workshops/ws8$ ./awk1.sh
SALES REPORT:
Total Sales for Hard Disk Drive is $4358.50 (inc. GST of $435.85)
Total Sales for SSD is $1299.87 (inc. GST of $129.99)
Total Sales for Printer is $4953.52 (inc. GST of $495.35)
Total Sales for Mouse Mat is $2380.00 (GST Exempt)
Total Sales for DELL Laptop is $13306.00 (inc. GST of $1330.60)
Total Sales for Tablet Cover 14# is $2371.60 (GST Exempt)
Total Sales for Printer Cartridge is $9784.80 (GST Exempt)
Total Sales for USB 64 GB is $5104.00 (inc. GST of $510.40)
Gross Sales: $43558.29
GST Component: $2902.19
Net Sales: $40656.10
```

1. Declare the custom function immediately after awks opening single parenthesis ‘
2. Apply the functions in the awk command body as required



## Integrating **awk** in shell scripts

# Using awk to work with floats 1

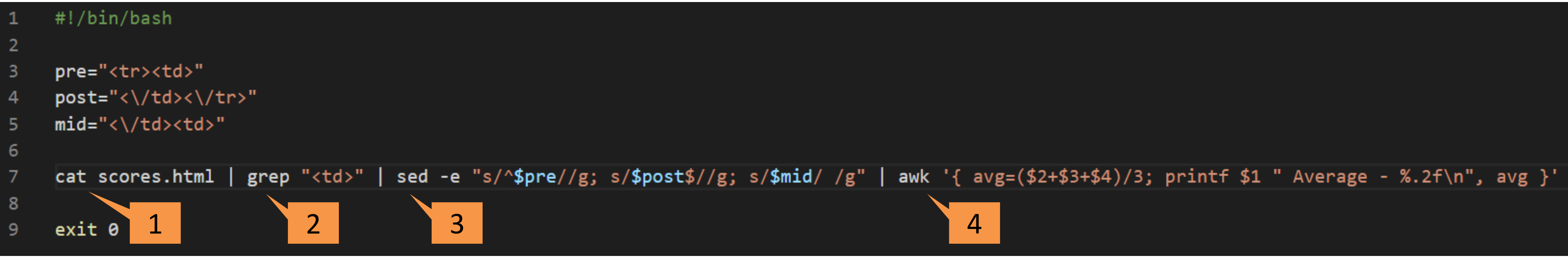


```
#!/bin/bash

read -p 'Enter float 1: ' fl1
read -p 'Enter float 2: ' fl2

result=$( echo $fl1 $fl2 | awk '{prod=$1*$2; printf"%.2f\n", prod }' )

echo "$fl1 multiplied by $fl2 is $result"

exit 0
```

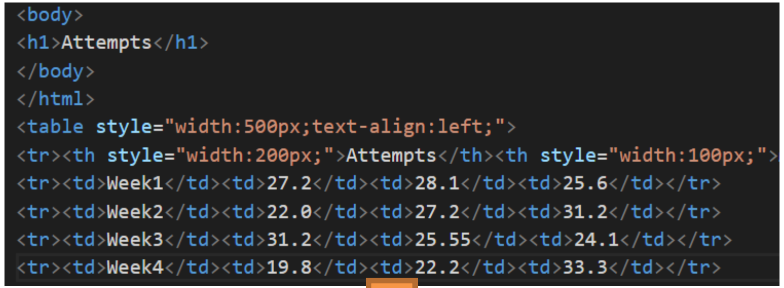
```
vbrown@LAPTOP-N6EFE714:~/scriptlang/workshops/ws8$ ./awk2.sh
Enter float 1: 3.3
Enter float 2: 6.2
3.3 multiplied by 6.2 is 20.46
```

1. Uses *command substitution* to immediately assign the results of command sequence to a variable named `$result`
2. `Echo` used to provide stored inputs (`$fl1`, `$fl2`) to `awk` via *piping*
3. Then `awk` makes the required calculations, the product of which is stored in `$result`

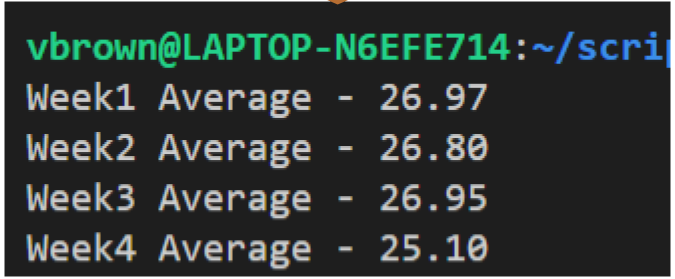
# Using awk to work with floats 2



```bash
#!/bin/bash

pre="<tr><td>"
post="<\/td><\/tr>"
mid="<\/td><td>"

cat scores.html | grep "<td>" | sed -e "s/^$pre//g; s/$post$//g; s/$mid/ /g" | awk '{ avg=($2+$3+$4)/3; printf $1 " Average - %.2f\n", avg }'

exit 0
```



```html
<body>
<h1>Attempts</h1>
</body>
</html>
<table style="width:500px;text-align:left;">
<tr><th style="width:200px;">Attempts</th><th style="width:100px;">
<tr><td>Week1</td><td>27.2</td><td>28.1</td><td>25.6</td></tr>
<tr><td>Week2</td><td>22.0</td><td>27.2</td><td>31.2</td></tr>
<tr><td>Week3</td><td>31.2</td><td>25.55</td><td>24.1</td></tr>
<tr><td>Week4</td><td>19.8</td><td>22.2</td><td>33.3</td></tr>
```



```
vbrown@LAPTOP-N6EFE714:~/scri
Week1 Average - 26.97
Week2 Average - 26.80
Week3 Average - 26.95
Week4 Average - 25.10
```

1. Grab the content of `scores.html` and *pipe* it through to `grep`
2. Use `grep` to eliminate all lines passed to it except those that contain `<td>` and then *pipe* through to `sed`
3. Use `sed` to eliminate all remaining HTML tags, ensuring that the `</td></td>` sequence is replace by a space, then *pipe* the results to `awk`
4. Use `awk` to calculate the averages of the float values on each of the four records that remain and print to the terminal formatted to two (2) decimal places

## References and Further Reading

- **Ebrahim, M. and Mallet, A. (2018) *Mastering Linux Based Scripting* (2nd Ed.), Chapters 10, 12, 13**
- https://likegeeks.com/awk-command
- https://www.tutorialspoint.com/awk/awk_built_in_functions

## Terms to Review and Know

- wget and curl
- regex in awk
- If statements in awk
- Functions in awk
- awk scripts
- awk formatting
- parsing
- fields
- records
- printf