Working with Data - CLI Tools

> These activities will need to be performed in a Terminal and not using the Jupyter Notebook play button.
>
> If using `VSCode` you can right-click on the `cli_tools.ipynb` file and select `Open in Integrated Terminal`

`jq`

`jq` is a lightweight and flexible command-line JSON processor that is like 'sed' for JSON data. It allows you to slice, filter, map and transform structured data with ease. It is a powerful tool which can be used on the command line or within shell scripts, allowing you to quickly filter and transform JSON outputs from other commands or APIs

See the following documentation to learn more about jq

https://jqlang.github.io/jq/

Request the JSON users content from the previous example and pass it to `jq`

```
curl -s 'https://jsonplaceholder.typicode.com/users' | jq .
```

Read the `basic_example.json` file

```
jq . basic_example.json
```

Another way to read in the `basic_example.json` file

```
cat basic_example.json | jq
```

Return all items of the list

```
jq '.[]' basic_example.json
```

Return the first item or index 0 from the list

```
jq '.[0]' basic_example.json
```

Get the value of a key called `name` in the first list item

```
jq '.[0].name' basic_example.json
```

Get all names from the list

```
jq '.[].name' basic_example.json
```

Create an object with the values of the `id` and `name` keys

```
jq '.[] | {id, name}' basic_example.json
```

Join the IDs and Names in the format `id - name`

```
jq  '.[] | {id, name} | join (" - ")' basic_example.json
```

Remove the quotes

```
jq --help | grep raw
echo
jq -r '.[] | {id, name} | join (" - ")' basic_example.json
```

Only return items which match a criteria

```
jq '.[] | select((.name == "Ervin Howell") or ( .name | contains("Leanne")))' basic_example.
```

Only return items which match a criteria and then only print the value of the `address` key in that item

```
jq '.[] | select(.name == "Ervin Howell") | {address}' basic_example.json
```

Only return items which match a criteria and then only print all keys foudn in `address`

```
jq '.[] | select(.name == "Ervin Howell") | {address} | .address | keys '  basic_example.jso
```

Only return items which match a criteria and then print the value of a specific key in that item

```
jq '.[] | select(.name == "Ervin Howell") | {address} | .address.city ' basic_example.json
```

Show the JSON for a new user

```
cat new_user.json
```

Append it as an item to the list in `basic_example.json`

```
jq '. += [input]' basic_example.json new_user.json
```

Notice that `basic_example.json` does not contain the new user?

```
cat basic_example.json
```

Write the new appended list to a temporary file and print the contents

```
jq '. += [input]' basic_example.json new_user.json > temp_list_of_users.json
cat temp_list_of_users.json
```

Use this temporary file in a shell script

```
echo -e "Using jq in a shell script \n"

list_of_names=$(jq -r '.[].name | @sh' temp_list_of_users.json)

for i in "${list_of_names[@]}"; do
    echo "$i"
done
```

"The `@foo` syntax is used to format and escape strings, which is useful for building URLs, documents in a language like HTML or XML, and so forth."

"`@sh`: The input is escaped suitable for use in a command-line for a POSIX shell. If the input is an array, the output will be a series of space-separated strings."

https://devdocs.io/jq/

"`@` Used in filter expressions to refer to the current node being processed."

https://support.smartbear.com/alertsite/docs/monitors/api/endpoint/jsonpath.html

Remove the temporary file

```
rm temp_list_of_users.json
```

Working with Kubernetes and `kubectl`

> The `kubernetes-nodes.json` and `kubernetes-pods.json` files were created using the `kubectl get nodes -json` and `kubectl get pods --all-namespaces -json` commands
>
> You can also pipe this output straight into `jq`
>
> e.g. `kubectl get pods --all-namespaces -o json | jq '.'`

```
cat kubernetes-nodes.json | jq '.'
```

Find the Kubernetes node name

```
cat kubernetes-nodes.json | jq '.items[] | .metadata.name'
```

Create an object containing the name and status of each node

```
cat kubernetes-nodes.json | jq '.items[] |{name: .metadata.name, conditions: .status.conditi
```

Filter to just the name, status, and message

```
cat kubernetes-nodes.json | jq '.items[] |{name: .metadata.name, status: .status.conditions|
```

It can be consolidated further. Note the array [] around the conditions

```
cat kubernetes-nodes.json | jq '.items[] |{name: .metadata.name, conditions: [.status.condit
```

Have a look at the Kubernetes pods

```
cat kubernetes-pods.json | jq '.items[]'
```

Find the failing pods

```
cat kubernetes-pods.json | jq '.items[] | select(.status.phase != "Running") | {name: .metac
```

A few more CLI examples

sed

The `sed` command, short for 'stream editor', is a powerful utility in Unix and Linux systems. It is primarily used for text manipulation and transformation in a file or input stream. It can perform complex editing operations with just a few keystrokes, making it a valuable tool for text processing.

A sed command usually takes the following form:

```
s/REGEXP/REPLACEMENT/FLAGS
```

Using `sed` to substitute

```
echo 'Hello World' | sed 's/World/Universe/'
```

Using `sed` to delete lines

```
echo -e 'line1\nline2' | sed '1d'
```

Using `sed` to append a line after a match

```
echo -e 'line1\nline2' | sed '/line1/a\This is a new line.'
```

tr

The `tr` command is a powerful utility in Unix-like operating systems, standing for 'translate'. It reads characters from standard input, transforms them based on a set of specified rules, and writes the result to standard output.

Using `tr` to transform - lowercase to UPPERCASE

```
cat kubernetes-nodes.json | jq -r '.items[] | .metadata.name'
echo
cat kubernetes-nodes.json | jq -r '.items[] | .metadata.name' | tr [:lower:] [:upper:]
```

Using tr to transform - replace hyphen with underscore

```
cat kubernetes-nodes.json | jq -r '.items[] | .metadata.name' | tr '-' '_'
```

Using tr to transform - delete all the numbers

```
cat kubernetes-nodes.json | jq -r '.items[] | .metadata.name' | tr -d '[:digit:]'
```

cut

The `cut` command is a utility in Unix and Linux used for cutting out sections from each line of files and writing the result to standard output. It can be used to cut parts of a line by specifying a delimiter or a character position.

> The `kubectl-pod-output` file was created using the command, `kubectl get pods -A > kubectl-pod-output`

```
cat kubectl-pod-output
```

Using cut to select the first character

```
cat kubectl-pod-output | cut -c 1
```

Using cut to select fields using a delimeter

```
cat kubectl-pod-output | tr -s ' ' | cut -d ' ' -f 2,3
```

awk

The `awk` command in Linux is a powerful tool for processing text files, particularly those involving data arranged in columns. Named after its original authors Aho, Weinberger, and Kernighan, awk is not just a command, but a scripting language in its own right designed for data manipulation.

This is only `a fraction of a fraction` of what is possible with awk. If you want to learn more have a look at tutorials such as this one:

Getting started with AWK

Select the same fields as above but this time using awk

```
cat kubectl-pod-output | awk '{print $2,$3}'
```

Correctly output the format

```
cat kubectl-pod-output | awk '{printf "%-30s %-100s\n",$1,$2}'
```

Gawk: Effective AWK Programming

- `printf "%-30s %-100s\n",$1,$2`: pretty print whats between the quotes, `" "`, and insert the values in variables `$1` and `$2`

- `printf`: you can specify the width to use for each item, as well as various formatting choices for numbers (such as what output base to use, whether to print an exponent, whether to print a sign, and how many digits to print after the decimal point)
- `%-30s`: left align the string value in variable `$1` and make the column 30 characters wide
- `%-100s`: left align the string value in variable `$2` and make the column 30 characters wide
- `\n`: newline character
- `$1,$2`: variables holding data

  Try removing the hyphen, `-`, or changing the 30 and 100 to different values to see what happens.

`awk` can perform mathematical operations on columns

```
echo -e '1\n2\n3' | awk '{sum += $1} END {print sum}'
```

`awk` supports conditional statements like `if`

```
echo -e '10\n20\n30' | awk '{if ($1 > 15) print $1}'
```

`$1` refers to the first field (or column) in the input. `awk` automatically splits the input into fields based on a specified delimiter (spaces or tabs by default), and each field can be accessed using `$n`, where n is the number of the field.

In the previous example `awk` is reading each line of the input. `$1` represents the first (and only) field on each line. The command checks if the value in this field is greater than 15, and if so, it prints the value.