Data Modelling Tools

AUTM08016

Topic 5
Files, Spreadsheets and Serialisation

Dr Diarmuid Ó Briain

Version 1.0 [02 January 2024]

TUS
Ollscoil Teicneolaíochta na Sionainne:
Lár Tíre, An tIarthar Láir
Technological University of the Shannon:
Midlands Midwest

Dr Diarmuid Ó Briain

CEng, FIEI

CISSP®

Table of Contents

Table of Figures

1. Testing if files or directories exist

It is quite important functionality for the program to be in a position to determine if files or directories exist already on the host operating system. To demonstrate this, create a directory and a file.

```
~$ mkdir my_directory
~$ echo "FILE" > my_directory/file.txt
```

In python the `os` module path function is required.

```
>>> from os import path
```

Consider the use of the `path.exists()`, `path.isdir()` and `path.isfile()` methods in the example presented in Figure 1.

```
~$ cat exists.py
#!/usr/bin/env python3

# Do this in the shell first
#   ~$ mkdir my_directory
#   ~$ echo "FILE" > my_directory/file.txt

from os import path

import sys

my_dir = "my_directory"
my_file = "my_directory/file.txt"
other_file = "my_directory/file2.txt"

print(f"'{my_dir}' exists: {path.exists(my_dir)}")
print(f"'{my_file}' exists: {path.exists(my_file)}")
print(f"'{other_file}' exists: {path.exists(other_file)}")
print(f"'{my_dir}' is a directory: {path.isdir(my_dir)}")
print(f"'{my_file}' is a directory: {path.isdir(my_file)}")
print(f"'{my_dir}' is a file: {path.isfile(my_dir)}")
print(f"'{my_file}' is a file: {path.isfile(my_file)}")

# End

```

Figure 1: exists.py

```
~$ ./exists.py
'my_directory' exists: True
'my_directory/file.txt' exists: True
'my_directory/file2.txt' exists: False
'my_directory' is a directory: True
'my_directory/file.txt' is a directory: False
'my_directory' is a file: False
'my_directory/file.txt' is a file: True
```

2. Writing to files

To write to a file on the host operating system it is necessary to '**open**' the file and assign it to a file-handle, in this example '**fh**'. The file-handle is an identifier assigned to an open file that is currently being utilised by Python.

From this point on the file can be written to by the **write()** method on the file-handle.

To finish the operation and close the file, use the **close()** method on the file-handle.

Note that the **write()** method, unlike the **print()** does not complete each instruction with a carriage return (**\n**). Therefore carriage returns will need to be added as required.

```
~$ cat write2file.py
#! /usr/bin/env python3

import sys

# // File for writing //
some_text = "Some more text"

# // Writing to a file //
fh = open("file.txt", mode="w", encoding="utf-8")
fh.write("This is a test file\nthat I want to write to")
fh.write("\n")
fh.write(some_text)
fh.write("\nfinished.")
fh.write("\n")
fh.close()

# End

```

Figure 2: Writing to files

```
~$ ./write2file.py

~$ cat file.txt
This is a test file
that I want to write to
Some more text
finished.
```

2.1 Modes

Character	Meaning
r	open for reading (default)
w	open for writing, truncating the file first
x	open for exclusive creation, failing if the file already exists
a	open for writing, appending to the end of the file if it exists
r+	open for reading & writing (beginning of file)
w+	open for reading & writing (beginning of file)
a+	open for reading & writing (end of file)

3. Deleting files

It is a simple matter to delete a file, check it exists first and then delete with the `os` module `remove()` method.

```
~$ ls my_directory/file.txt
my_directory/file.txt


>>> import os

>>> if (os.path.isfile("my_directory/file.txt")):
...     print("File file.txt exists")
...     os.remove("my_directory/file.txt")
...     print("Deleted file.txt")
...

File file.txt exists
Deleted file.txt

>>> quit()


~$ ls my_directory/file.txt
ls: cannot access 'my_directory/file.txt': No such file or directory
```

4. Reading files

4.1 Read the entire file as a string

Like writing to files, to read, the file must be opened, but with **r** for read mode. The **read()** method can be used to read the entire file or it can be read line by line using a **for** loop as demonstrated.

```
~$ cat readafile_as_string.py
#! /usr/bin/env python3

import sys

# // File for reading //
file = "unstructured_data.txt"

print()

# // Read entire file //
print("** Reading entire file, this is the output **\n")
fh = open(file, mode="r")
print((type(fh), fh))
entire_file = fh.read()
print(entire_file)
print()

# // Closing filehandle //
fh.close()

# // Read entire file (alternative format) //
print()
print("** Reading entire file, this is the output **\n")
with open(file) as fh2:
    print((type(fh2), fh2))
    entire_file2 = fh2.read()
print(entire_file2)

try:
    entire_file3 = fh2.read()
except:
    print("\nfilehandle 'fh2' was closed with the indent closure\n")

print("** splitlines() for some structure **\n")
structured_list = entire_file.splitlines()
print(structured_list)

# End

```

Figure 3: Read a file

Note the two methods demonstrated for opening and reading the file. In the first, the file is opened and assigned to the file-handle **fh**. Processing occurs until the **close()** method is called to close the file-handle.

An alternative to opening and closing the file-handle is the **with/as** structure. Lines indented after the **with/as** line are processed with the file-handle open. As soon as the indentation is remove the file-handle automatically closes.

If structure is required, the **splitlines()** method can be applied to the string.

```
~$ ./readafile_as_string.py

** Reading entire file, this is the output **

(<class '_io.TextIOWrapper'>, <_io.TextIOWrapper
name='unstructured_data.txt' mode='r' encoding='UTF-8'>)
Port      Status    Vlan  Duplex      Speed  Type
Gi0/1/0   connected   1   auto        auto   10/100/1000BaseTX
Gi0/1/1   connected   1   half-duplex 100    10/100/1000BaseTX
Gi0/1/2   notconnect  1   full-duplex 1000   10/100/1000BaseTX
Gi0/1/3   notconnect  1   auto        auto   10/100/1000BaseTX

** Reading entire file, this is the output **

(<class '_io.TextIOWrapper'>, <_io.TextIOWrapper
name='unstructured_data.txt' mode='r' encoding='UTF-8'>)
Port      Status    Vlan  Duplex      Speed  Type
Gi0/1/0   connected   1   auto        auto   10/100/1000BaseTX
Gi0/1/1   connected   1   half-duplex 100    10/100/1000BaseTX
Gi0/1/2   notconnect  1   full-duplex 1000   10/100/1000BaseTX
Gi0/1/3   notconnect  1   auto        auto   10/100/1000BaseTX

file-handle 'fh2' was closed with the indent closure

** splitlines() for some structure **

['Port      Status    Vlan  Duplex      Speed  Type ', 'Gi0/1/0
connected   1   auto        auto   10/100/1000BaseTX', 'Gi0/1/1
connected   1   half-duplex 100    10/100/1000BaseTX', 'Gi0/1/2
notconnect  1   full-duplex 1000   10/100/1000BaseTX', 'Gi0/1/3
notconnect  1   auto        auto   10/100/1000BaseTX']
```

4.2 Read file line by line

Reading in a file in bulk can present some difficulties with the manipulation of data within a program, consider this example that uses the **readlines()** method to read the file line by line into a list. This program then processes each line and splits it into a structured list of lists format.

```
~$ cat readafile_line_by_line.py
#! /usr/bin/env python3

import sys
from pprint import pprint

# // File for reading //
file = "unstructured_data.txt"
structured_list = list()

print()

# // Read unstructured data file line by line to list //
with open(file) as fh2:
    entire_file_as_list = fh2.readlines()

# // Output less-structured data //
print(type(entire_file_as_list), entire_file_as_list)

# // Process data in list //
for line in entire_file_as_list:
    structured_list.append(line.strip().split())

# // Output structured data //
print()
pprint(structured_list)

# End

```

Figure 4: Read a file line by line

```
~$ ./readafile_line_by_line.py

<class 'list'> ['Port      Status     Vlan  Duplex       Speed  Type \
n', 'Gi0/1/0   connected   1    auto        auto   10/100/1000BaseTX\
n', 'Gi0/1/1   connected   1    half-duplex 100    10/100/1000BaseTX\
n', 'Gi0/1/2   notconnect  1    full-duplex 1000   10/100/1000BaseTX\
n', 'Gi0/1/3   notconnect  1    auto        auto   10/100/1000BaseTX']

[['Port', 'Status', 'Vlan', 'Duplex', 'Speed', 'Type'],
 ['Gi0/1/0', 'connected', '1', 'auto', 'auto', '10/100/1000BaseTX'],
 ['Gi0/1/1', 'connected', '1', 'half-duplex', '100', '10/100/1000BaseTX'],
 ['Gi0/1/2', 'notconnect', '1', 'full-duplex', '1000', '10/100/1000BaseTX'],
 ['Gi0/1/3', 'notconnect', '1', 'auto', 'auto', '10/100/1000BaseTX']]
```

4.3 Importing structured data

A little trick, if it is necessary to keep structured data in a file. Instead of reading it from a file unstructured and parsing it, store the data structured in a simple python module and import it.

```
~$ cat structured_data_mod.py
str_intf = """Port      Name  Status       Vlan  Duplex Speed Type
Gi0/1/0         connect   1     auto   auto  10/100/1000BaseTX
Gi0/1/1         connect   1     half-duplex   100  10/100/1000BaseTX
Gi0/1/2         notconnect   1     full-duplex   1000  10/100/1000BaseTX
Gi0/1/3         notconnect   1     auto   auto  10/100/1000BaseTX"""

list_intf = ["Gi0/1/0", "Gi0/1/1", "Gi0/1/2", "Gi0/1/3"]

dict_intf = {"Gi0/1/0": "connect", "Gi0/1/1": "connect", "Gi0/1/2": "notconnect",
"Gi0/1/3": "notconnect"}

list_of_lists = [["PORT_NAME", "STATUS", "VLAN", "DUPLEX", "SPEED", "TYPE"],
["Gi0/1/0","connect","1","auto","auto","10/100/1000BaseTX"],
["Gi0/1/1","connect","1","half-duplex","100","10/100/1000BaseTX"],
["Gi0/1/2","notconnect","1","full-duplex","1000","10/100/1000BaseTX"],
["Gi0/1/3","notconnect","1","auto","auto","10/100/1000BaseTX"]]

dict_of_lists = {"Gi0/1/0": ["connect","1","auto","auto","10/100/1000BaseTX"],
"Gi0/1/1": ["connect","1","half-duplex","100","10/100/1000BaseTX"],
"Gi0/1/2": ["notconnect","1","full-duplex","1000","10/100/1000BaseTX"],
"Gi0/1/3": ["notconnect","1","auto","auto","10/100/1000BaseTX"]}

dict_of_dicts = {"Gi0/1/0": {"STATUS":
"connect","VLAN":"1","DUPLEX":"auto","SPEED":"auto","TYPE":"10/100/1000BaseTX"},
"Gi0/1/1": {"STATUS": "connect","VLAN":"1","DUPLEX":"half-
duplex","SPEED":"100","TYPE":"10/100/1000BaseTX"},
"Gi0/1/2": {"STATUS": "notconnect","VLAN":"1","DUPLEX":"full-
duplex","SPEED":"1000","TYPE":"10/100/1000BaseTX"},
"Gi0/1/3": {"STATUS":
"notconnect","VLAN":"1","DUPLEX":"auto","SPEED":"auto","TYPE":"10/100/1000BaseTX"}
}
```

Import the data from the ‘module’.

```
~$ python3
>>> import structured_data_mod as sdm
```

Import the string.

```
>>> sdm.str_intf
"Port      Name  Status       Vlan  Duplex Speed Type \nGi0/1/0
connect   1     auto   auto  10/100/1000BaseTX\nGi0/1/1
connect   1     half-duplex   100  10/100/1000BaseTX\nGi0/1/2
notconnect   1     full-duplex   1000  10/100/1000BaseTX\nGi0/1/3
notconnect   1     auto   auto  10/100/1000BaseTX"
```

Import a list.

```
>>> sdm.list_intf
["Gi0/1/0", "Gi0/1/1", "Gi0/1/2", "Gi0/1/3"]
```

Import a dictionary.

```
>>> sdm.dict_intf
{"Gi0/1/0": "connect", "Gi0/1/1": "connect", "Gi0/1/2": "notconnect",
"Gi0/1/3": "notconnect"}
```

Import the list of lists.

```
>>> sdm.list_of_lists
[["PORT_NAME", "STATUS", "VLAN", "DUPLEX", "SPEED", "TYPE"],
["Gi0/1/0", "connect", "1", "auto", "auto", "10/100/1000BaseTX"],
["Gi0/1/1", "connect", "1", "half-duplex", "100",
"10/100/1000BaseTX"], ["Gi0/1/2", "notconnect", "1", "full-
duplex", "1000", "10/100/1000BaseTX"], ["Gi0/1/3", "notconnect",
"1", "auto", "auto", "10/100/1000BaseTX"]]
```

Import the dictionary of lists.

```
>>> sdm.dict_of_lists
{"Gi0/1/0": ["connect", "1", "auto", "auto", "10/100/1000BaseTX"],
"Gi0/1/1": ["connect", "1", "half-duplex", "100",
"10/100/1000BaseTX"], "Gi0/1/2": ["notconnect", "1", "full-
duplex", "1000", "10/100/1000BaseTX"], "Gi0/1/3": ["notconnect",
"1", "auto", "auto", "10/100/1000BaseTX"]}
```

Import the dictionary of dictionaries.

```
>>> sdm.dict_of_dicts
{"Gi0/1/0": {"STATUS": "connect", "VLAN": "1", "DUPLEX": "auto",
"SPEED": "auto", "TYPE": "10/100/1000BaseTX"}, "Gi0/1/1":
{"STATUS": "connect", "VLAN": "1", "DUPLEX": "half-duplex",
"SPEED": "100", "TYPE": "10/100/1000BaseTX"}, "Gi0/1/2":
{"STATUS": "notconnect", "VLAN": "1", "DUPLEX": "full-duplex",
"SPEED": "1000", "TYPE": "10/100/1000BaseTX"}, "Gi0/1/3":
{"STATUS": "notconnect", "VLAN": "1", "DUPLEX": "auto", "SPEED":
"auto", "TYPE": "10/100/1000BaseTX"}}
```

5. Working with Spreadsheets

Comma-Separated Values (CSV) files allow data to be saved in a tabular format and are easily handled by spreadsheet programs such as LibreOffice Calc, Google Sheets or Microsoft Excel. CSV files have a **.csv** extension and use a comma symbol (ASCII code 44) to separate fields. CSV files are in common use because:

- They are plain-text files, making them easier for the website developers to create.
- They are easy to import into a spreadsheet or another storage database, regardless of the specific software (software independence).
- They are a useful format for organising large datasets.

5.1 Write to a CSV file

Consider the program in Figure 5.

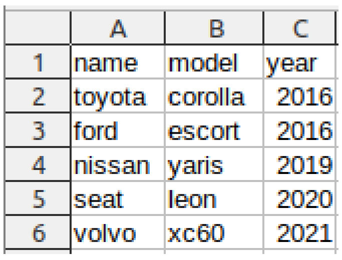
Firstly the CSV file is opened for writing, due to the **`w`** mode, using the **open()** function. A CSV writer object is created by calling the **`writer()`** function of the **`csv`** module. Data is then written to the file via writer object by calling the **`writerow()`** method to write a single line or **`writerows()`** method to write multiple lines to the CSV writer object.

The object must be closed, in this case that happens when the indent finishes, otherwise it is necessary to use the **close()** method on the file-handle.

```
~$ cat write_csv.py
#! /usr/bin/env python3
""" CSV Write """

import csv

header = ["name", "model", "year"]
data = [
    ["toyota", "corolla", "2016"],
    ["ford", "escort", "2016"],
    ["nissan", "yaris", "2019"],
    ["seat", "leon", "2020"],
    ["volvo", "xc60", "2021"],
]
cars_csv = "cars.csv"

# // Open CSV file as a file-handle //
with open(cars_csv, mode="w", encoding="utf8") as fh:
    writer = csv.writer(fh)
    writer.writerow(header)
    writer.writerows(data)

# End

```

Figure 5: Write to a CSV file

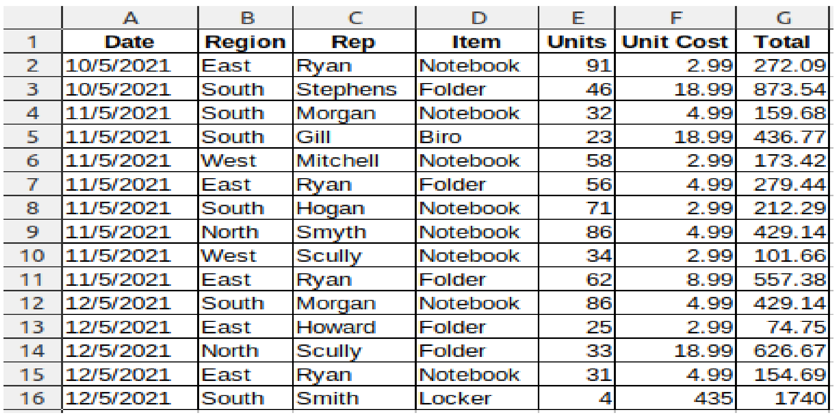
```
~$ ./write_csv.py

~$ cat cars.csv
name,model,year
toyota,corolla,2016
ford,escort,2016
nissan,yaris,2019
seat,leon,2020
volvo,xc60,2021
```



	A	B	C
1	name	model	year
2	toyota	corolla	2016
3	ford	escort	2016
4	nissan	yaris	2019
5	seat	leon	2020
6	volvo	xc60	2021

5.2 Reading from a .CSV file

```
~$ cat orders.csv
Date,Region,Rep,Item,Units,Unit Cost,Total
10/5/2021,East,Ryan,Notebook,91,2.99,272.09
10/5/2021,South,Stephens,Folder,46,18.99,873.54
11/5/2021,South,Morgan,Notebook,32,4.99,159.68
11/5/2021,South,Gill,Biro,23,18.99,436.77
11/5/2021,West,Mitchell,Notebook,58,2.99,173.42
11/5/2021,East,Ryan,Folder,56,4.99,279.44
11/5/2021,South,Hogan,Notebook,71,2.99,212.29
11/5/2021,North,Smyth,Notebook,86,4.99,429.14
11/5/2021,West,Scully,Notebook,34,2.99,101.66
11/5/2021,East,Ryan,Folder,62,8.99,557.38
12/5/2021,South,Morgan,Notebook,86,4.99,429.14
12/5/2021,East,Howard,Folder,25,2.99,74.75
12/5/2021,North,Scully,Folder,33,18.99,626.67
12/5/2021,East,Ryan,Notebook,31,4.99,154.69
12/5/2021,South,Smith,Locker,4,435,1740
```



	A	B	C	D	E	F	G
1	**Date**	**Region**	**Rep**	**Item**	**Units**	**Unit Cost**	**Total**
2	10/5/2021	East	Ryan	Notebook	91	2.99	272.09
3	10/5/2021	South	Stephens	Folder	46	18.99	873.54
4	11/5/2021	South	Morgan	Notebook	32	4.99	159.68
5	11/5/2021	South	Gill	Biro	23	18.99	436.77
6	11/5/2021	West	Mitchell	Notebook	58	2.99	173.42
7	11/5/2021	East	Ryan	Folder	56	4.99	279.44
8	11/5/2021	South	Hogan	Notebook	71	2.99	212.29
9	11/5/2021	North	Smyth	Notebook	86	4.99	429.14
10	11/5/2021	West	Scully	Notebook	34	2.99	101.66
11	11/5/2021	East	Ryan	Folder	62	8.99	557.38
12	12/5/2021	South	Morgan	Notebook	86	4.99	429.14
13	12/5/2021	East	Howard	Folder	25	2.99	74.75
14	12/5/2021	North	Scully	Folder	33	18.99	626.67
15	12/5/2021	East	Ryan	Notebook	31	4.99	154.69
16	12/5/2021	South	Smith	Locker	4	435	1740

Figure 6: A CSV file for reading

Consider the program in Figure 7 which can read the CSV file in Figure 6. The CSV module reads the **orders.csv** file. The file is opened by the open() method and due to the `r` mode, the file is in read-only status. The **reader()** method extracts the file contents and returns a list to the program.

The program returns the contents of the CSV file as a list of lists.

```
~$ cat read_csv.py
#! /usr/bin/env python3
""" CSV Read """

import csv

header = list()
data = list()
orders_csv = "orders.csv"

# // Open CSV file as a file-handle //
with open(orders_csv, mode="r", encoding="utf8") as fh:
    csv_read = list(csv.reader(fh))
    header = csv_read[0]
    data = csv_read[1:]

print(header)
print()
[print(x) for x in data]


# End

```

Figure 7: Read from a CSV file

```
~$ ./read_csv.py
['Date', 'Region', 'Rep', 'Item', 'Units', 'Unit Cost', 'Total']

['10/5/2021', 'East', 'Ryan', 'Notebook', '91', '2.99', '272.09']
['10/5/2021', 'South', 'Stephens', 'Folder', '46', '18.99', '873.54']
['11/5/2021', 'South', 'Morgan', 'Notebook', '32', '4.99', '159.68']
['11/5/2021', 'South', 'Gill', 'Biro', '23', '18.99', '436.77']
['11/5/2021', 'West', 'Mitchell', 'Notebook', '58', '2.99', '173.42']
['11/5/2021', 'East', 'Ryan', 'Folder', '56', '4.99', '279.44']
['11/5/2021', 'South', 'Hogan', 'Notebook', '71', '2.99', '212.29']
['11/5/2021', 'North', 'Smyth', 'Notebook', '86', '4.99', '429.14']
['11/5/2021', 'West', 'Scully', 'Notebook', '34', '2.99', '101.66']
['11/5/2021', 'East', 'Ryan', 'Folder', '62', '8.99', '557.38']
['12/5/2021', 'South', 'Morgan', 'Notebook', '86', '4.99', '429.14']
['12/5/2021', 'East', 'Howard', 'Folder', '25', '2.99', '74.75']
['12/5/2021', 'North', 'Scully', 'Folder', '33', '18.99', '626.67']
['12/5/2021', 'East', 'Ryan', 'Notebook', '31', '4.99', '154.69']
['12/5/2021', 'South', 'Smith', 'Locker', '4', '435', '1740']
```

Figure 8: Read from a CSV file #2

6. Exercise #5.1

Write a program called “**exercise_5.1.py**”.

Read the CSV file, **orders.csv**, adding the following data to the retrieved data and writing a new **.csv** file, **orders2.csv**, with the complete dataset.

```
["14/5/2021", "West", "Duggan", "Stapler", "5", "15", "75"]
```

The new file should look like this:

```
~$ ./exercise_8.1.py

~$ cat orders2.csv
Date,Region,Rep,Item,Units,Unit Cost,Total
10/5/2021,East,Ryan,Notebook,91,2.99,272.09
10/5/2021,South,Stephens,Folder,46,18.99,873.54
11/5/2021,South,Morgan,Notebook,32,4.99,159.68
11/5/2021,South,Gill,Biro,23,18.99,436.77
11/5/2021,West,Mitchell,Notebook,58,2.99,173.42
11/5/2021,East,Ryan,Folder,56,4.99,279.44
11/5/2021,South,Hogan,Notebook,71,2.99,212.29
11/5/2021,North,Smyth,Notebook,86,4.99,429.14
11/5/2021,West,Scully,Notebook,34,2.99,101.66
11/5/2021,East,Ryan,Folder,62,8.99,557.38
12/5/2021,South,Morgan,Notebook,86,4.99,429.14
12/5/2021,East,Howard,Folder,25,2.99,74.75
12/5/2021,North,Scully,Folder,33,18.99,626.67
12/5/2021,East,Ryan,Notebook,31,4.99,154.69
12/5/2021,South,Smith,Locker,4,435,1740
14/5/2021,West,Duggan,Stapler,5,15,75
```

7. Serialisation Protocols

Serialisation protocols convert data objects that exist within complex data structures into a byte stream for storage or transfer.

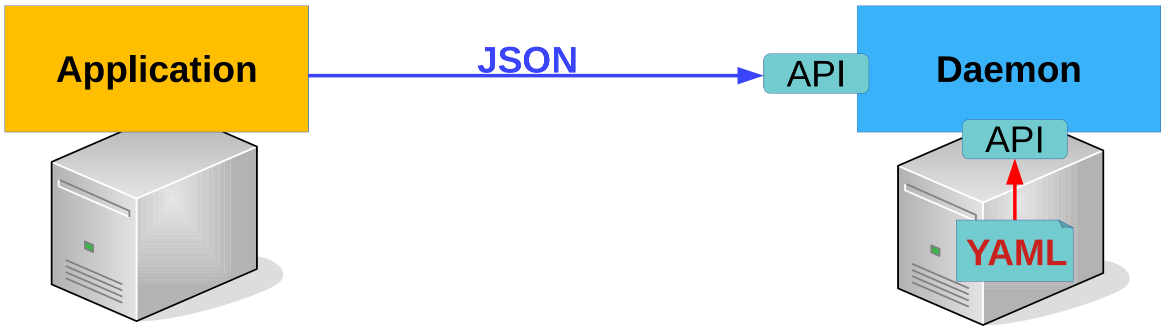


Figure 9: Serialisation Protocols

Computer systems vary in their hardware and software architecture, operating system and addressing mechanisms. Storing and exchanging data between such varying environments requires a platform-and-language-neutral data format that is easily understood.

There are a number of data serialisation protocols and their use varies depending upon factors such as data complexity, need for human readability, speed and storage space constraints. Extensible Markup Language (XML), JavaScript Object Notation (JSON) and YAML Ain't Markup Language (YAML) are some commonly used data serialisation formats.

For example XML is a nested textual human-readable and editable format. It is popular for schema based validation, metadata applications, web services data transfer, web publishing. JSON is also human readable and is a popular format for web API parameter passing in machine to machine communications, YAML is a lightweight text format that is human-readable, supports comments and easily editable. Technically YAML is a superset of JSON.

8. YAML Ain't Markup Language

8.1 Processing YAML files

YAML Ain't Markup Language (YAML) is a human-readable data-serialisation language. It is commonly used for configuration files, but it is also used in data storage (for example: debugging output) or transmission (for example: document headers).

It became the standard method of handling IP address schema in Debian flavours of GNU/Linux as **netplan**. Netplan interprets a YAML description of the required network interfaces and what each should be configured to do and it then generates all the necessary configuration for a specified renderer tool.

YAML natively supports three basic data types: scalars (such as strings, integers, and floats), lists, and associative arrays.

The official recommended filename extension for YAML files is **.yaml**; however, **.yml** is also popular, particularly in relation to its use with **Ansible**.

8.2 Installing the Python YAML module

Install Python YAML module

```
~$ python3 -m pip install pyyaml
```

8.3 YAML structures

Starting with the file in Illustration 1 and two data files, `data.yaml` and `data2.yaml`. Run the program and both files are read in, in the first case the `yaml.load(<input>, Loader=<loader>)` parses the supplied YAML document as a stream and produce the corresponding Python object. `yaml.load()` accepts a byte string, a Unicode string, an open binary file object, or an open text file object as input. A byte string or a file must be encoded with utf-8, utf-16-be or utf-16-le encoding.

The `yaml.load()` is also supplied with a loader, if the YAML data is coming from trusted sources, the `Loader=FullLoader` argument can be supplied, which loads the full YAML language and avoid arbitrary code execution. An alternative is to use the `Loader=SafeLoader`. This protects against an arbitrary Python object being generated from an unsafe YAML file, perhaps from an untrusted source, such as the Internet. It limits output to simple Python objects like integers or lists.

`yaml.safe_load(<input>)` is the same as
`yaml.load(<input>, Loader=SafeLoader)`.

```
~$ cat yaml_start.py
#! /usr/bin/env python3
""" Yaml start """

import sys
import yaml

# Variables
file = "data.yaml"
file2 = "data2.yaml"


print()
# // Open a yaml file for reading //
with open(file, mode="r") as fh:
    head_ = fh.readline()
    yaml_in = yaml.load(fh, Loader=yaml.FullLoader)
    print(f"{type(yaml_in)}\n{yaml_in}")
    print()

print("\n---------------\n")

while True:
    input("Press 'Enter' to continue")
    print()

    # // Open a multi-doc yaml file for reading //
    with open(file2, mode="r") as fh:
        head_ = fh.readline()
        yaml_in2 = yaml.load_all(fh, Loader=yaml.SafeLoader)
        print(type(yaml_in2), yaml_in2)
        for doc in yaml_in2:
            print(doc)
            print("\n---------------\n")

```

Figure 10: yaml_start.py

```
~$ cat data.yaml
---

# List
- list 1
- list 2
- list 3
- list 4

...
```

```
~$ cat data2.yaml
---

# List
- list 1
- list 2
- list 3
- list 4

...
```

Note that it is not necessary to quote the strings in the YAML file.

Run the program with the interactive switch. The program reads in the YAML file **`data.yaml`** and returns the YAML list as a Python list. The **`yaml.load()`** method can read in a single YAML **doc**. This is a file starting with **---** and ending with **...**, in this case the structure is a list.

```
~$ ./yaml_start.py

<class 'list'>
['list 1', 'list 2', 'list 3', 'list 4']

----------------

Press 'Enter' to continue
```

Pressing the '**Enter**' key moves the program into the while loop. This also returns the list in the single **doc** in the YAML file **data2.yaml**.

```
<class 'generator'> <generator object load_all at 0x7f2d9befc120>
['list 1', 'list 2', 'list 3', 'list 4']

----------------

Press 'Enter' to continue
```

Edit the **`data2.yaml`** file by:

```
~$ cat <<EOM>> data2.yaml
---

# A basic dictionary

dict1: one
dict2: two
dict3: three
dict4: four
...

EOM
```

Now press the '**Enter**' key to reread the **`data2.yaml`** file. Now as well as the list a dictionary is returned in Python.

```
{'dict1': 'one', 'dict2': 'two', 'dict3': 'three', 'dict4': 'four'}

----------------

Press 'Enter' to continue
```

Edit the **data2.yaml** file again with a complex dictionary:

```
~$ cat <<EOM>> data2.yaml
---

# More complex dictionary

dict_:
  dict1: one
  dict2: two
  dict3: three
  dict4: four

EOM
```

Now press the '**Enter**' key, now the more complex dictionary appears in Python.

```
{'dict_': {'dict1': 'one', 'dict2': 'two', 'dict3': 'three', 'dict4':
'four'}}

---------------

Press 'Enter' to continue
```

Edit the **data2.yaml** file again with a complex dictionary within a list:

```
~$ cat <<EOM>> data2.yaml
---

# Dictionary in a list

- dict_:
    dict1: one
    dict2: two
    dict3: three
    dict4: four
    dict5: null

EOM
```

Now press the '**Enter**' key, and the list of dictionary appears in Python. Note also that the YAML **null** becomes **None** in Python.

```
[{'dict_': {'dict1': 'one', 'dict2': 'two', 'dict3': 'three',
'dict4': 'four', 'dict5': None}}]

---------------

Press 'Enter' to continue
```

Once again edit the **data2.yaml** file with a complex dictionary:

```
~$ cat <<EOM>> data2.yaml
---
# Lists in a dictionary

dict_:
  dict1:
    - sometimes: true
    - othertimes: false
    - eventimes: yes
    - goodtimes: no
  dict2: two
  dict3: three
  dict4: four
  dict5: null

EOM
```

Now press the '**Enter**' key, and the list of dictionary appears in Python. Note also that the YAML **null** becomes **None** in Python. In this case note that the YAML bool statements **true** and **yes** produce **True** in Python and the **false** and **no** produce a **False** in python.

```
{'dict_': {'dict1': [{'sometimes': True}, {'othertimes': False},
{'eventimes': True}, {'goodtimes': False}], 'dict2': 'two', 'dict3':
'three', 'dict4': 'four', 'dict5': None}}

---------------

Press 'Enter' to continue
```

Finally edit the **data2.yaml** file with a other items:

```
~$ cat <<EOM>> data2.yaml
---
# Other stuff

- dict1: |
    multi line can
    look exactly
    like this.

- dict2: >
    This looks easier
    to read here within
    the YAML file.

EOM
```

Now press the '**Enter**' key, and two dictionaries within a list are returned. Consider the difference carefully. Note that the second dictionary only has `\n` at the end of a single line whereas the first dictionary has multiple `\n` to keep the multiline nature of the string.

```
[{'dict1': 'multi line can \nlook exactly \nlike this.\n'}, {'dict2':
'This looks easier to read here within the YAML file.\n'}]

---------------

Press 'Enter' to continue
```

8.4 Simple YAML reader

Illustration 1 is a simplified reader. Note the use of the **`yaml.safe_load(<input>)`** format. The program expects an input file and if not supplied with on, asks for one to be supplied.

```
~$ cat yaml_reader.py
#!/usr/bin/env python3
import sys
import yaml
from pprint import pprint


def yaml_reader(file):
    with open(file) as fh:
        return yaml.safe_load(fh)

if __name__ == "__main__":
    try:
        file = sys.argv[1]
    except IndexError:
        file = input("Enter YAML file name: ")
    print(type(yaml_reader(file)), yaml_reader(file))

```

Figure 11: Simple YAML reader

8.5 Compressed YAML

As well as the formats already considered, YAML has a compressed format. Consider the original `data.yaml` file and now consider `data_compressed.yaml` with the reader as illustrated in Figure 12. A similar example for dictionary formats is also displayed. The compressed formats are quite similar to the python structure itself.

```
~$ cat data.yaml
---

# List
- list 1
- list 2
- list 3
- list 4

...

~$ cat data_compressed.yaml
---

# List
[list 1, list 2, list 3, list 4]

...

~$ ./yaml_reader.py data.yaml
<class 'list'> ['list 1', 'list 2', 'list 3', 'list 4']

~$ ./yaml_reader.py data_compressed.yaml
<class 'list'> ['list 1', 'list 2', 'list 3', 'list 4']

~$ cat data_dict.yaml
---

# A basic dictionary

dict1: one
dict2: two
dict3: three
dict4: four
...

~$ cat data_dict_compressed.yaml
---

# A basic dictionary

{dict1: one, dict2: two, dict3: three, dict4: four}
...

~$ ./yaml_reader.py data_dict.yaml
<class 'dict'> {'dict1': 'one', 'dict2': 'two', 'dict3': 'three',
'dict4': 'four'}

~$ ./yaml_reader.py data_dict_compressed.yaml
<class 'dict'> {'dict1': 'one', 'dict2': 'two', 'dict3': 'three',
'dict4': 'four'}
```

Figure 12: Compressed YAML

8.6 YAML network file

The following is an example YAML network description file for two interfaces. The example program in Figure 13 will reconfigure the **ens160** interface to a static address and generate a new configuration file **network2.yaml**.

```
~$ cat network_compressed.yaml
---
# network.yaml file

{network: {ethernets: {ens160: {dhcp4: True}, ens192: {addresses:
[192.168.0.2/24], nameservers: {addresses: [8.8.8.8]}}}, version: 2,
renderer: networkd}}
...

~$ ./yaml_processing.py
<class 'dict'> {'network': {'ethernets': {'ens160': {'dhcp4': True}, 'ens192':
{'addresses': ['192.168.0.2/24'], 'nameservers': {'addresses': ['8.8.8.8']}}}},
'version': 2, 'renderer': 'networkd'}}

{'network': {'ethernets': {'ens160': {'dhcp4': True},
                           'ens192': {'addresses': ['192.168.0.2/24'],
                                      'nameservers': {'addresses': ['8.8.8.8']}}}},
             'renderer': 'networkd',
             'version': 2}}

ens160: {'dhcp4': True}

ens160: {'addresses': ['10.10.10.100/16'], 'nameservers': {'addresses':
['8.8.8.8']}}

# New network2.yaml file

~$ cat network2.yaml
---
# New network2.yaml file

network:
  ethernets:
    ens160:
      addresses:
      - 10.10.10.100/16
      nameservers:
        addresses:
        - 8.8.8.8
    ens192:
      addresses:
      - 192.168.0.2/24
      nameservers:
        addresses:
        - 8.8.8.8
  renderer: networkd
  version: 2
...
```

```
~$ cat yaml_processing.py
#! /usr/bin/env python3
""" Yaml processing """

import sys
import yaml
from pprint import pprint

# Variables
file = "network.yaml"
new_file = "network2.yaml"

# // Open a yaml file for reading //
with open(file, mode="r") as fh:
    head_ = fh.readline()
    net_list = yaml.safe_load(fh)

print(type(net_list), net_list)
print()
pprint(net_list)
print()

# // Print the 'ens160' interface //
print(f"ens160: {net_list['network']['ethernets']['ens160']}\n")

# Add static address on 'ens160'
#   Address: '10.10.10.100/16
#   Nameserver:'8.8.8.8'

new_addr = {"addresses": ["10.10.10.100/16"],
            "nameservers": {"addresses": ["8.8.8.8"]}}

# // Update the 'net_list' dictionary //
net_list["network"]["ethernets"]["ens160"] = new_addr

print(f"ens160: {net_list['network']['ethernets']['ens160']}\n")

# // Write a new network yaml file //
head2_ = "# New network2.yaml file"
print(f"{head2_}\n")
with open(new_file, "w") as fh:
    fh.write(f"---\n{head2_}\n")
    fh.write("\n")
    yaml.dump(net_list, fh, default_flow_style=False)
    fh.write("...\n")

# End
```

Figure 13: Processing YAML

9. Processing JSON files

9.1 JSON

JavaScript Object Notation (JSON) is a format inspired by a subset of the JavaScript programming language dealing with object literal syntax. JSON is a lightweight format that is popular for storing and transporting data. JSON format is relatively easy to understand.

JSON supports primitive types, like **strings** and **numbers** (int, floats), as well as **arrays** (lists, tuples) and **objects** (dictionaries).

9.2 JSON data file

Here is an example JSON dataset

```
~$ cat data_set.json
{
    "firstName": "Tom",
    "lastName": "Clarke",
    "hobbies": ["sailing", "swimming", "hillwalking"],
    "age": 40,
    "children": [
        {
            "firstName": "Cian",
            "age": 12
        },
        {
            "firstName": "Conor",
            "age": 13
        }
    ]
}
```

9.3 JSON/Python translation

Python	JSON
dict	object
list, tuple	array
str	string
int, long, float	number
True	true
False	false
None	null

```
~$ cat json_processing.py
#! /usr/bin/env python3
""" JSON processing """

import json
import pprint

# Variables //
file = "data_set.json"
new_file = "data_set2.json"

# // De-serialise data from the file (read it) //
print(f"De-serialising data from '{file}'")
with open(file, mode="r") as fh:
    data = json.load(fh)

print(f"\n{type(data)}, {data}\n")

# // Add 'rugby' as a hobby //
print("Adding 'rugby' as a new hobby")
data["hobbies"].append("rugby")

# // Add a new child //
print("Adding a new child called 'Orla'")
_ = {"firstName": "Orla", "age": 1}
data["children"].append(_)

# // Serialise data to the file (write it) //
print(f"Serialising data to '{new_file}'\n")
with open(new_file, mode="w") as fh2:
    json.dump(data, fh2, indent=("  "))
    # json.dump(data, fh2)

# End

```

Figure 14: JSON Processing

The program in Figure 14 will de-serialise data, read data from the JSON file and convert it to a Python dictionary with lists. It will change the data by adding a hobby and another child. Finally it dumps the data to a new JSON file.

```
~$ ./json_processing.py
De-serialising data from 'data_set.json'

<class 'dict'>, {'firstName': 'Tom', 'lastName': 'Clarke', 'hobbies':
['sailing', 'swimming', 'hillwalking'], 'age': 40, 'children':
[{'firstName': 'Cian', 'age': 12}, {'firstName': 'Conor', 'age':
13}]}

Adding 'rugby' as a new hobby
Adding a new child called 'Orla'
Serialising data to 'data_set2.json'
```

Note the additions to the dataset.

```
~$ cat data_set.json
{
  "firstName": "Tom",
  "lastName": "Clarke",
  "hobbies": [
    "sailing",
    "swimming",
    "hillwalking",
    "rugby"
  ],
  "age": 40,
  "children": [
    {
      "firstName": "Cian",
      "age": 12
    },
    {
      "firstName": "Conor",
      "age": 13
    },
    {
      "firstName": "Orla",
      "age": 1
    }
  ]
}
```

Note also that by removing `indent=('    ')` from the `json.dump` command the data is serialised to the file in a more compact fashion.

```
~$ cat data_set2.json
{"firstName": "Tom", "lastName": "Clarke", "hobbies": ["sailing",
"swimming", "hillwalking", "rugby"], "age": 40, "children":
[{"firstName": "Cian", "age": 12}, {"firstName": "Conor", "age": 13},
{"firstName": "Orla", "age": 1}]}
```

9.4 json.dump() versus json.dumps()

`json.dump()`: method can be used for writing to JSON file.

`json.dumps()`: method converts a Python object into a JSON string (**s** is for string).

```
>>> import json
>>> dict_ = {'one': 1, 'two': 2, 'three': 3}
>>> print(type(dict_), dict_)
<class 'dict'> {'one': 1, 'two': 2, 'three': 3}

>>> json_obj = json.dumps(dict_)
>>> print(type(json_obj), json_obj)
<class 'str'> {"one": 1, "two": 2, "three": 3}
```

10. Exercise #5.2

Write a program called “**exercise_5.2.py**”.

1. Add a shebang line and a document string “**Exercise #5.2 in Python3**”.
2. Import both **yaml** and **json** modules.
3. Read from the **network.yaml** file already used.
4. Print each interface and the IP address on it.
5. Serialise the data to JSON format into a file called **network.json**.
6. Output from the file should look like this:

```
~$ ./exercise_8.2.py
Reading the 'network.yaml' YAML file
ens192: 192.168.0.2/24
ens224: 192.168.1.2/24
Serialising data to 'network.json'
```

7. **network.json** file should look like this.

```
~$ cat network.json
{
  "network": {
    "ethernets": {
      "ens160": {
        "dhcp4": true
      },
      "ens192": {
        "addresses": [
          "192.168.0.2/24"
        ],
        "nameservers": {
          "addresses": [
            "8.8.8.8"
          ]
        }
      },
    },
    "version": 2,
    "renderer": "networkd"
  }
}
```

This page is intentionally blank