SESUG Paper 135-2020

# Handling complex data sources in Python™ and SAS® using Jupyter® Notebook in SAS University Edition VM

David B. Horvath, MS, CCP

## ABSTRACT

This paper (and the resulting session) is a focused on the techniques required to read complex data types including XML, JSON, and even CSV using Python. The underlying tool we will use will be Jupyter Notebook and SAS (both provided as part of the SAS University Edition VM). In the live session, as time allows, some of the available UNIX/Linux tools are discussed along with their data analysis capabilities (but not in this paper – these have been covered in other papers.

## INTRODUCTION

Life would be so much easier if everything was in a database or pulled via API. But that is not the case. All too often we get data files (or have to send them) in various formats. SAS and Python provide significant capabilities.

One of the problems frequently encountered is not knowing much about the file before we get them. That's why I also cover UNIX/Linux tools in the live session – to help you understand the actual contents. Once we know something about the file then we look into processing them

## SAS UNIVERSITY EDITION

In case you aren't aware of it already (and feel free to skip over this paragraph), here is some background about the tool. This is free to download and install for any non-commercial purpose. A training session or personal growth (learning) would certainly fit that definition. You do not need to be attending a formal college/university course to make use of this tool.

At time of writing, the location to download the tool is https://www.sas.com/en_ca/software/university-edition/download-software.html

The software installs in a virtual machine (that runs under Windows). I'll leave the setup instructions to others.

### STARTING UP SAS UNIVERSITY EDITION

The first thing you need to do is to start up the Oracle's (Sun Microsystems) Virtual Box. Once in that application, you select SAS University Edition:

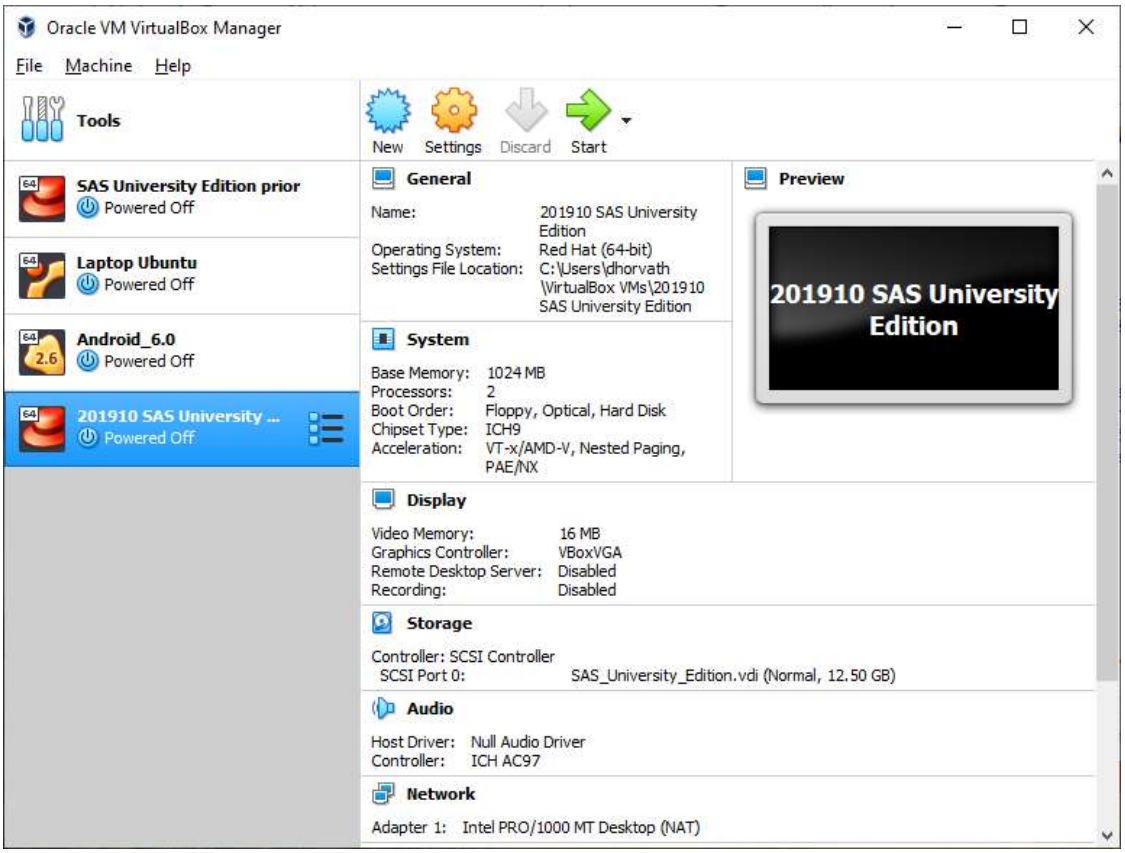


**Figure 1 Virtual Machines within VirtualBox (your experience may vary)**

Once you start SAS University Edition, you will see something like:

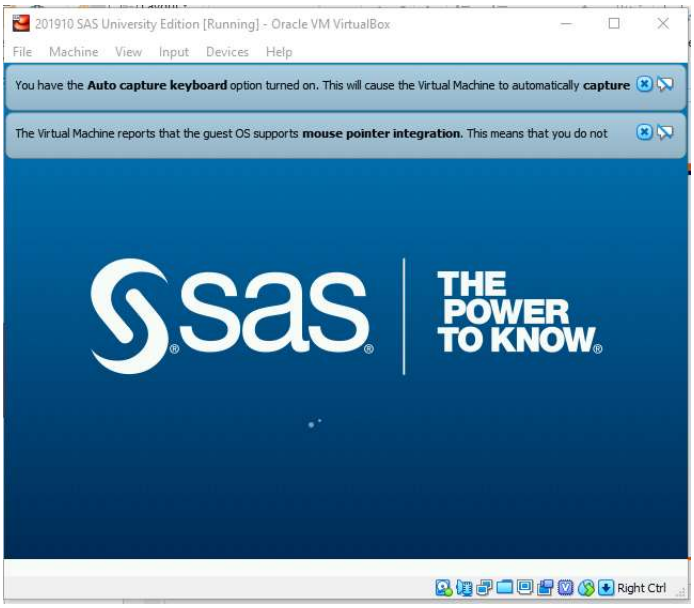


**Figure 2 Initial startup screen**

Startup will continue as Red Hat Linux loads and starts up the services for SAS, looking something like:

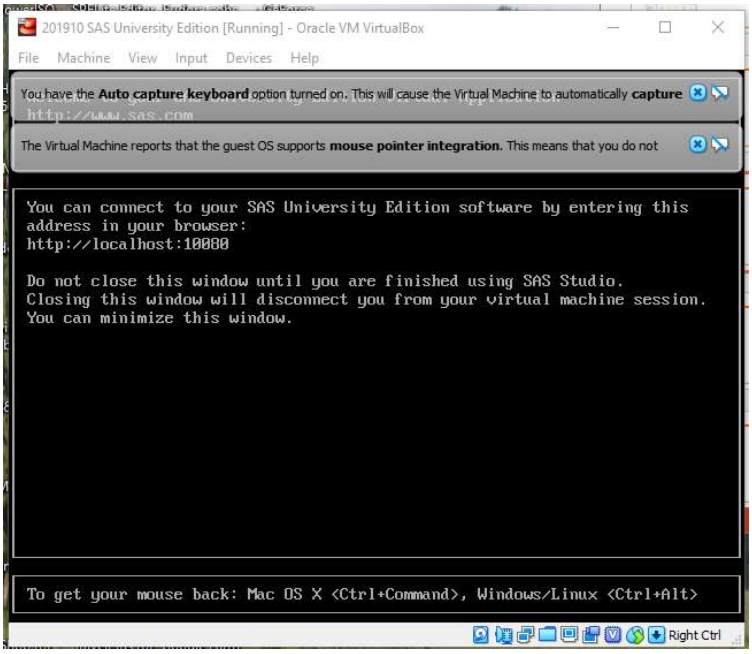


**Figure 3 Startup is now Complete**

As the instructions suggest, open a browser and enter `http://127.0.0.1:10080`

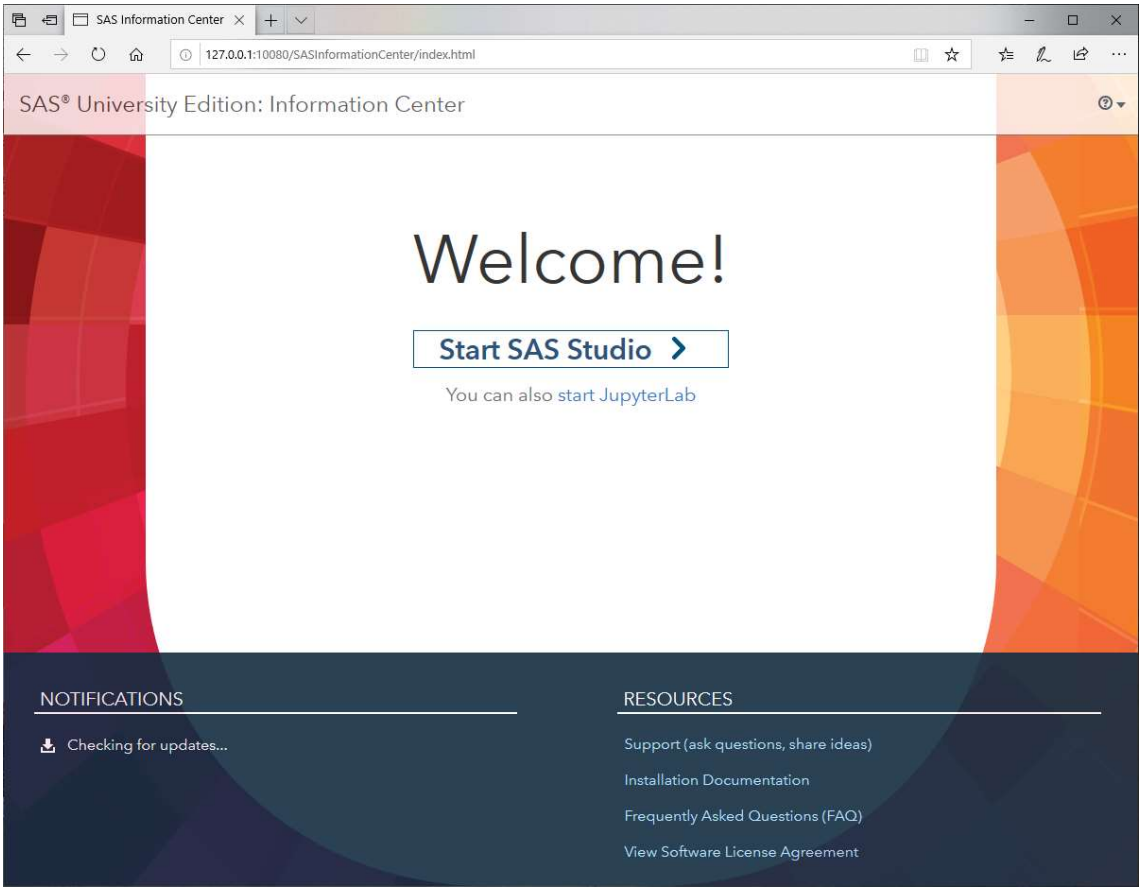


**Figure 4 Execution Options: SAS Studio and JupyterLab**

## STARTING JUPYTERLAB

Select “You can also start JupyterLab” and you will see something like the following:

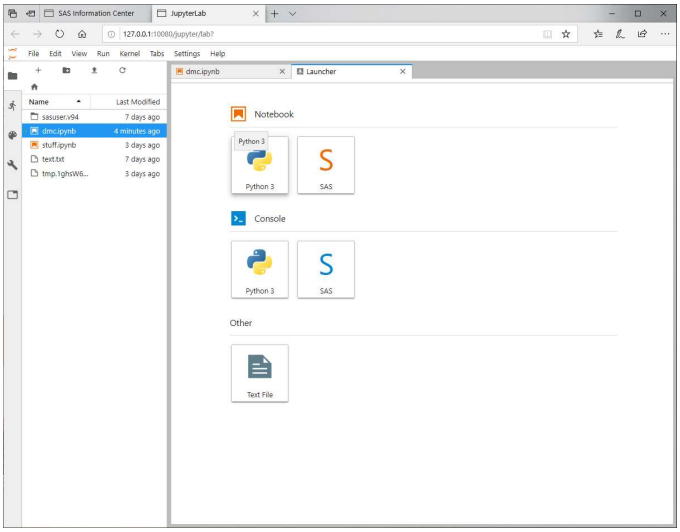


**Figure 5 JupyterLab**

Select Python 3 under Notebook next, enter the following, and run:

```
%%bash
pwd
ls -al /media
```

This will show that the lab is properly configured and able to execute UNIX/Linux shell commands (a neat little trick).

## STARTING SAS STUDIO

Open another browser tab or window, enter http://127.0.0.1:10080, select ““Start SAS Studio >”, and you will see something like the following:

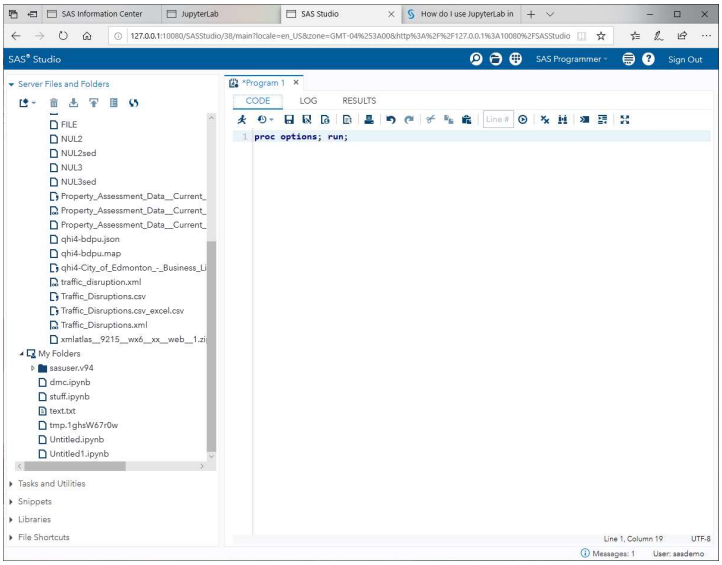


**Figure 6 SAS Studio under University Edition**

We want to test the Studio to make sure everything is configured correctly. Enter the following little program and run:

```
Proc setinit;
Run;
```

Now you can see that we can run SAS programs as well.

Did you notice the “SAS Notebook” on the Jupyter window? We can also run SAS there!

## PROCESSING COMPLEX DATA

There are three main data types that will be reviewed:

- CSV – comma separated value
- XML – eXtensible Markup Language
- JSON – JavaScript Object Notation

And how we can process those within SAS or Python. SAS is often the easier method but not always.

The City of Edmonton, Alberta, Canada is very nice in that they publish large volumes of data about city operations in these formats; at time of writing, https://data.edmonton.ca is the location. The examples make use of their data.

## PROCESSING XML FILES

For basic XML files, SAS is amazingly easy to use. You set up two filename and one libname reference and the engine takes care of the rest. It will even generate a map based on the data provided. It will generate dataset and field names, assign data types, and even provide connections between the various objects within the file as best it can.

```
# Example 1
/* XML example -- problems with file */
filename xmlref  "/folders/myshortcuts/DMC_Code/City_of_Edmonton_-
_Business_Licenses.xml";
filename xmlmref "/folders/myshortcuts/DMC_Code/City_of_Edmonton_-
_Business_Licenses.map";
libname xmlin xmlv2 fileref=xmlref map=xmlmref automap=replace; * don't need
anything after fileref ;

proc datasets lib=xmlin; run;
proc contents data=xmlin.location; run;
data results; set xmlin.location; run;
proc print data=results(obs=10); run;
```

That is, assuming the file meets the criteria that the SAS XML engine requires. In this case, there is a problem with the file no matter which “engine” I use:

```
 78         libname xmlref xmlv2 ;
 ERROR: XML data is not in a format supported natively by the XML libname
engine. Files of this type may require an XMLMap to be
        input properly.

 79         libname xmlref xml ;
 82         proc datasets library=xmlref;
 ERROR: There is an illegal character in the name.
```

```
 ERROR: Encountered during XMLMap parsing at or near line 1, column 1514.
 ERROR: XML describe error: Internal processing error.
```

Looking at the data, I see the following field at column 1514:

```
unt>1</count><:-computed_region_7ccj_gre3>338</:-computed_region_7ccj_gre3><:-com
```

This is no way for SAS to create a variable "**:-computed_region_7ccj_gre3**" and it won't even try. To resolve the issue, you can edit the file (this is where the UNIX/Linux tools come in handy):

```
#Example 1.1
# XML example -- fixing the file
$ cut -b 1500-1580 City*.xml
unt>1</count><:-computed_region_7ccj_gre3>338</:-computed_region_7ccj_gre3><:-com

$ awk '{gsub(":-","X");print $0;'} City*.xml | cut -b 1500-1580
unt>1</count><Xcomputed_region_7ccj_gre3>338</Xcomputed_region_7ccj_gre3><Xcomput

$ awk '{gsub(":-","X");print $0;'} City*.xml > City_fixed.xml

$ cut -b 1500-1580 City_fixed.xml
unt>1</count><Xcomputed_region_7ccj_gre3>338</Xcomputed_region_7ccj_gre3><Xcomput
```

In this case, I use common UNIX/Linux commands to find the ":-" and convert them to "X" in the actual data file. Now the file extracts properly.

## SAS XMLMAPPER TOOL

SAS provides a handy tool to pre-generate the map before execution time. You provide XMLMapper with a sample file and it generates the map. Neither the automatically generated map nor that created by XMLMapper are perfect – they look at a few rows (like proc import) to determine data types and field sizes. At time of writing, it was available for download at https://support.sas.com/downloads/package.htm?pid=486

We did run into an issue with one of the fields. The generated maps determined the proper size as 32 characters but later records were even longer:

```
 81          data results; set xmlin.location; run;

 WARNING: Data truncation occurred on variable human_address Column length=32
Additional length=51.
 WARNING: Data truncation occurred on variable human_address Column length=32
Additional length=51.
 WARNING: Data truncation occurred on variable human_address Column length=32
Additional length=51.
```

This issue can be fixed. We can go into XMLMapper or manually edit the map itself. The important change is to extend the field length.

**Figure 7 City_fixed_for_xmlmapper_fixed.map**

```
      <COLUMN name="human_address">
            <PATH syntax="XPath">/response/row/row/location/@human_address
</PATH>
            <TYPE>character</TYPE>
            <DATATYPE>string</DATATYPE>
            <LENGTH>100</LENGTH>
        </COLUMN>
```

On execution, the error is gone:

```
81          data results; set xmlin.location; run;

 NOTE: There were 24121 observations read from the data set XMLIN.location.
 NOTE: The data set WORK.RESULTS has 24121 observations and 5 variables.
 NOTE: DATA statement used (Total process time):
       real time           9.36 seconds
       cpu time            8.88 seconds
```

## PROCESSING XML IN PYTHON

Because Python reads the entire file into memory, I want to use a smaller example. In this case, it is the GPX file from a Garmin Portable Automobile GPS. Another downside (in addition to the memory consumption) is that you have to know a lot more about the file with Python. It will not generate the translation for you.

```
#Example 2:
# Python XML Parse -- a much smaller example because Python parses the file into memory
#Python code to illustrate parsing of XML files
# importing the required modules
import xml.etree.ElementTree as ET

def parseXML(xmlfile):

    # create element tree object
    tree = ET.parse(xmlfile)

    print ('parsed tree', tree)

    # get root element
    root = tree.getroot()

    print ('root ', root)

    for child in root:
       print ('tag: ', child.tag, ' attrib: ', child.attrib, ' text: ', child.text)
        for grandchild in child:
          print ('\ttag: ', grandchild.tag, ' attrib: ', grandchild.attrib, ' text: ',
grandchild.text)
            for great in grandchild:
             print('\t\ttag: ', great.tag, ' attrib: ', great.attrib, ' text: ', great.text)


def main():
    print('main starting')
    # parse xml file
    newsitems = parseXML('/folders/myshortcuts/DMC_Code/0905.gpx')

    print('parse done')

    # calling main function
print('start')
main()
```

```
start
main starting
parsed tree <xml.etree.ElementTree.ElementTree object at 0x7f2c60988ba8>
root  <Element '{http://www.topografix.com/GPX/1/1}gpx' at 0x7f2c5c1cf978>
tag:  {http://www.topografix.com/GPX/1/1}metadata  attrib:  {}  text:  None
        tag:  {http://www.topografix.com/GPX/1/1}link  attrib:  {'href': 'http://www.garmin.c
om'}  text:  None
                tag:  {http://www.topografix.com/GPX/1/1}text  attrib:  {}  text:  Garmin Int
ernational
        tag:  {http://www.topografix.com/GPX/1/1}time  attrib:  {}  text:  2015-09-05T20:45:3
1Z
tag:  {http://www.topografix.com/GPX/1/1}wpt  attrib:  {'lat': '40.224917', 'lon': '-75.77415
4'}  text:  None
        tag:  {http://www.topografix.com/GPX/1/1}name  attrib:  {}  text:  031201 CAR
        tag:  {http://www.topografix.com/GPX/1/1}sym  attrib:  {}  text:  Pin, Blue
        tag:  {http://www.topografix.com/GPX/1/1}extensions  attrib:  {}  text:  None
                tag:  {http://www.garmin.com/xmlschemas/GpxExtensions/v3}WaypointExtension  a
ttrib:  {}  text:  None
tag:  {http://www.topografix.com/GPX/1/1}wpt  attrib:  {'lat': '40.443999', 'lon': '-76.63169
1'}  text:  None
        tag:  {http://www.topografix.com/GPX/1/1}name  attrib:  {}  text:  041204 CAR
        tag:  {http://www.topografix.com/GPX/1/1}sym  attrib:  {}  text:  Pin, Blue
        tag:  {http://www.topografix.com/GPX/1/1}extensions  attrib:  {}  text:  None
                tag:  {http://www.garmin.com/xmlschemas/GpxExtensions/v3}WaypointExtension  a
ttrib:  {}  text:  None
tag:  {http://www.topografix.com/GPX/1/1}wpt  attrib:  {'lat': '40.446294', 'lon': '-76.64191
6'}  text:  None
        tag:  {http://www.topografix.com/GPX/1/1}name  attrib:  {}  text:  041204 DOE
        tag:  {http://www.topografix.com/GPX/1/1}sym  attrib:  {}  text:  Pin, Blue
        tag:  {http://www.topografix.com/GPX/1/1}extensions  attrib:  {}  text:  None
                tag:  {http://www.garmin.com/xmlschemas/GpxExtensions/v3}WaypointExtension  a
ttrib:  {}  text:  None
tag:  {http://www.topografix.com/GPX/1/1}wpt  attrib:  {'lat': '40.432354', 'lon': '-75.52452
4'}  text:  None
        tag:  {http://www.topografix.com/GPX/1/1}name  attrib:  {}  text:  050117 CAR
        tag:  {http://www.topografix.com/GPX/1/1}sym  attrib:  {}  text:  Pin, Blue
        tag:  {http://www.topografix.com/GPX/1/1}extensions  attrib:  {}  text:  None
                tag:  {http://www.garmin.com/xmlschemas/GpxExtensions/v3}WaypointExtension  a
ttrib:  {}  text:  None
tag:  {http://www.topografix.com/GPX/1/1}wpt  attrib:  {'lat': '39.836276', 'lon': '-76.33904
4'}  text:  None
        tag:  {http://www.topografix.com/GPX/1/1}name  attrib:  {}  text:  050410BRDG
        tag:  {http://www.topografix.com/GPX/1/1}sym  attrib:  {}  text:  Pin, Blue
        tag:  {http://www.topografix.com/GPX/1/1}extensions  attrib:  {}  text:  None
                tag:  {http://www.garmin.com/xmlschemas/GpxExtensions/v3}WaypointExtension  a
ttrib:  {}  text:  None
tag:  {http://www.topografix.com/GPX/1/1}wpt  attrib:  {'lat': '39.834845', 'lon': '-76.37587
3'}  text:  None
        tag:  {http://www.topografix.com/GPX/1/1}name  attrib:  {}  text:  050410DEER
        tag:  {http://www.topografix.com/GPX/1/1}sym  attrib:  {}  text:  Pin, Blue
        tag:  {http://www.topografix.com/GPX/1/1}extensions  attrib:  {}  text:  None
                tag:  {http://www.garmin.com/xmlschemas/GpxExtensions/v3}WaypointExtension  a
```

**Figure 8 Output of Python XML parsing code (continues for many pages)**

Note the ‘for’ loops that pull the children, grandchildren, and great grandchildren in the network within the file. The SAS map handles this work for you.

## PROCESSING JSON FILES

For basic JSON files, SAS is amazingly easy to use. You set up two filename and one libname reference and the engine takes care of the rest. It will even generate a map based on the data provided. It will generate dataset and field names, assign data types, and even provide connections between the various objects within the file as best it can.

```
#Example 3:
/* SAS JSON read */
filename jsonref  "/folders/myshortcuts/DMC_Code/qhi4-bdpu.json";
filename jmap "/folders/myshortcuts/DMC_Code/qhi4-bdpu.map";
libname jsonin json fileref=jsonref map=jmap automap=create; * don't need
anything after jsonref ;

proc datasets lib=jsonin; run;

proc contents data=jsonin.root; run;

data results; set jsonin.root; run;

proc print data=results(obs=10); run;
```

The map is optional and you can even edit it later if you wish (as with XML). The only real change between the SAS XML and JSON examples is the engine specified on the libname statement.

## PROCESSING JSON IN PYTHON

As with XML files, you do need to know a bit more about the JSON file in Python in order to convert it into the resulting format. In this case, I allowed them to be parsed into individual columns that I would then have to handle for final processing.

```
#Example 4:
#Python JSON parse
import json

with open('/folders/myshortcuts/DMC_Code/qhi4-bdpu.json') as json_file:
    json_reader = json.load(json_file)
    print(' jason_reader ', json_reader);

    line_count = 0
    for row in json_reader:
        if line_count == 0:
            print('Column names are {', row,'}')
            line_count += 1
        else:
            print ('\t row ', row, ' EOR.')
            print ('\t row by step: ', end='')
            for col in row:
                print(col, ' ', end='')
            print()
        #    print('\t{', row[0], row[1], row[2], '}.')
            line_count += 1
            if line_count > 10: break
    print('Processed {line_count} lines.')
```

```
{'neighbourhood': 'Gold Bar', 'date_of_issue': '2019-10-21T00:00:00.000', 'expiry_date': '202
0-11-23T00:00:00.000', 'externalid': '164463740-002', 'status': 'ISSUED', 'neighbourhood_id':
'6270', 'business_category': 'General Business', 'trade_name': 'DUNNE 2 CODE', 'count': '1',
'ward': 'Ward 8'}
<class 'list'> <class 'dict'>
keys  dict_keys(['neighbourhood', 'date_of_issue', 'expiry_date', 'externalid', 'status', 'ne
ighbourhood_id', 'business_category', 'trade_name', 'count', 'ward'])
values  dict_values(['Gold Bar', '2019-10-21T00:00:00.000', '2020-11-23T00:00:00.000', '16446
3740-002', 'ISSUED', '6270', 'General Business', 'DUNNE 2 CODE', '1', 'Ward 8'])
items  dict_items([('neighbourhood', 'Gold Bar'), ('date_of_issue', '2019-10-21T00:00:00.00
0'), ('expiry_date', '2020-11-23T00:00:00.000'), ('externalid', '164463740-002'), ('status',
'ISSUED'), ('neighbourhood_id', '6270'), ('business_category', 'General Business'), ('trade_n
ame', 'DUNNE 2 CODE'), ('count', '1'), ('ward', 'Ward 8')])
cols  neighbourhood <class 'str'>
cols  date_of_issue <class 'str'>
cols  expiry_date <class 'str'>
cols  externalid <class 'str'>
cols  status <class 'str'>
cols  neighbourhood_id <class 'str'>
cols  business_category <class 'str'>
cols  trade_name <class 'str'>
cols  count <class 'str'>
cols  ward <class 'str'>
item  neighbourhood  value  Gold Bar
item  date_of_issue  value  2019-10-21T00:00:00.000
item  expiry_date  value  2020-11-23T00:00:00.000
item  externalid  value  164463740-002
item  status  value  ISSUED
item  neighbourhood_id  value  6270
item  business_category  value  General Business
item  trade_name  value  DUNNE 2 CODE
item  count  value  1
item  ward  value  Ward 8
{'latitude': '53.545940626971365', ':@computed_region_eq8d_jmrp': '114', 'address': '10363 -
104 STREET NW', 'neighbourhood': 'Downtown', ':@computed_region_7ccj_gre3': '338', 'externali
d': '343811266-001', 'business_category': 'Travelling or Temporary Sales (1 - 3 Days)', 'busi
ness_improvement_area': 'Downtown', ':@computed_region_mnf4_kaez': '9', 'trade_name': 'ALEXAN
DER AND ROSE', ':@computed_region_izdr_ja4x': '4', ':@computed_region_ecxu_fw7u': '242', 'loc
ation': {'latitude': '53.545940626971365', 'human_address': '{"address": "", "city": "", "sta
te": "", "zip": ""}', 'longitude': '-113.49870079470182'}, 'date_of_issue': '2019-10-20T00:0
0:00.000', 'expiry_date': '2019-10-22T00:00:00.000', 'longitude': '-113.49870079470182', 'sta
tus': 'ISSUED', 'neighbourhood_id': '1090', ':@computed_region_5jki_au6x': '5', 'count': '1',
'ward': 'Ward 6'}
<class 'list'> <class 'dict'>
keys  dict_keys(['latitude', ':@computed_region_eq8d_jmrp', 'address', 'neighbourhood', ':@co
mputed_region_7ccj_gre3', 'externalid', 'business_category', 'business_improvement_area', ':@
computed_region_mnf4_kaez', 'trade_name', ':@computed_region_izdr_ja4x', ':@computed_region_e
cxu_fw7u', 'location', 'date_of_issue', 'expiry_date', 'longitude', 'status', 'neighbourhood_
id', ':@computed_region_5jki_au6x', 'count', 'ward'])
values  dict_values(['53.545940626971365', '114', '10363 - 104 STREET NW', 'Downtown', '338',
'343811266-001', 'Travelling or Temporary Sales (1 - 3 Days)', 'Downtown', '9', 'ALEXANDER AN
D ROSE', '4', '242', {'latitude': '53.545940626971365', 'human_address': '{"address": "", "ci
```

**Figure 9 Output of Python JSON parsing code (Continues for pages even limited)**

This example shows the column names and first few rows of data (controlled by line_count) – as an example process it makes sense to limit the amount of information flying by.

## PROCESSING CSV FILES

SAS provides multiple options for processing of CSV files. ‘Proc import’ can determine how to process the file for you automatically – it actually generates code that you can look at.

```
#Example 5:
/* SAS CSV Proc Import */
proc import
datafile='/folders/myshortcuts/DMC_Code/Property_Assessment_Data__Current_Cal
endar_Year_.csv'
out=result_file replace; run;

proc print data=result_file (obs=10); run;
```

You can alter how proc import handles your file through a series of options:

You can change the DELIMITER (but only if you override the default DBMS=CSV with DBMS=DLM). You can even specify multiple delimiter values with DELIMTER='!|'.

By default, the variable names come from the column names at the top of the file – sasifying the names by replacing spaces with underscores, truncating length, etc. You can disable this with GETNAMES=NO. In that case plain sequential names are generated.

By default, the second row (DATAROW=2) is the first row that is used for actual data. In many ways, it behaves like FIRSTOBS=. When GETNAMES=NO, it defaults to DATAROW=1.

You even get to decide how much time proc import spends looking at the data. By default, it looks at the first 20 data rows (GUESSINGROWS=20). The higher the row count, the greater accuracy – but the more time required. You can set that to any value you consider appropriate.

The generated code can be used as-is or grabbed from the log, saved into the code, and edited as you find appropriate. Example 6 shows the generated code that can be edited as needed (for instance, if street_name needs to be extended to 50 characters).

```
#Example 6:
/* SAS CSV programattic read */
data WORK.RESULT_FILE    ;
   %let _EFIERR_ = 0; /* set the ERROR detection macro variable */
   infile
'/folders/myshortcuts/DMC_Code/Property_Assessment_Data__Current_Calendar_Yea
r_.csv' delimiter = ',' MISSOVER
      DSD lrecl=32767 firstobs=2 ;
   informat Account_Number best32. ;
   informat Suite $1. ;
   informat House_Number best32. ;
   informat Street_Name $14. ;
   informat Assessed_Value best32. ;
   informat Assessment_Class $11. ;
   informat Neighbourhood_ID best32. ;
   informat Neighbourhood $8. ;
   informat Ward $6. ;
   informat Garage $1. ;
   informat Latitude best32. ;
   informat Longitude best32. ;
   format Account_Number best12. ;
   format Suite $1. ;
   format House_Number best12. ;
   format Street_Name $14. ;
   format Assessed_Value best12. ;
   format Assessment_Class $11. ;
   format Neighbourhood_ID best12. ;
```

```
   format Neighbourhood $8. ;
   format Ward $6. ;
   format Garage $1. ;
   format Latitude best12. ;
   format Longitude best12. ;
   input
      Account_Number
      Suite  $
      House_Number
      Street_Name  $
      Assessed_Value
      Assessment_Class  $
      Neighbourhood_ID
      Neighbourhood  $
      Ward  $
      Garage  $
      Latitude
      Longitude
      ;
   if _ERROR_ then call symputx('_EFIERR_',1);  /* set ERROR detection macro
variable */
run;

proc print data=result_file (obs=10);
run;
```

## PROCESSING CSV IN PYTHON

As with XML and JSON files, you do need to know a bit more about the CSV file in Python in order to convert it into the resulting format. In this case, I allowed them to be parsed into individual columns that I would then have to handle for final processing – assigning the individual fields to the column names..

```
#Example 7:
#Python CSV
import csv

with
open('/folders/myshortcuts/DMC_Code/Property_Assessment_Data__Current_Calenda
r_Year_.csv') as csv_file:
    csv_reader = csv.reader(csv_file, delimiter=',')
    line_count = 0
    for row in csv_reader:
        if line_count == 0:
            print('Column names are {', row,'}')
            line_count += 1
        else:
            print ('\t row ', row, ' EOR.')
            print ('\t row by step: ', end='')
            for col in row:
                print(col, ' ', end='')
            print()
        #    print('\t{', row[0], row[1], row[2], '}.')
            line_count += 1
            if line_count > 10: break
    print('Processed {line_count} lines.')
```

```
Column names are { ['Account Number', 'Suite', 'House Number', 'Street Name', 'Assessed Valu
e', 'Assessment Class', 'Neighbourhood ID', 'Neighbourhood', 'Ward', 'Garage', 'Latitude', 'L
ongitude'] }
         row  ['9999999', '', '18226', '107A STREET NW', '458000', 'Residential', '3120', 'CH
AMBERY', 'Ward 3', 'Y', '53.6443745291161', '-113.510584027909']  EOR.
         row by step: 9999999    18226  107A STREET NW  458000  Residential  3120  CHAMBERY
Ward 3  Y  53.6443745291161  -113.510584027909
         row  ['9999998', '', '18222', '107A STREET NW', '445500', 'Residential', '3120', 'CH
AMBERY', 'Ward 3', 'Y', '53.6442576719968', '-113.510573993528']  EOR.
         row by step: 9999998    18222  107A STREET NW  445500  Residential  3120  CHAMBERY
Ward 3  Y  53.6442576719968  -113.510573993528
         row  ['9999997', '', '18218', '107A STREET NW', '444000', 'Residential', '3120', 'CH
AMBERY', 'Ward 3', 'Y', '53.6441365391259', '-113.510617088828']  EOR.
         row by step: 9999997    18218  107A STREET NW  444000  Residential  3120  CHAMBERY
Ward 3  Y  53.6441365391259  -113.510617088828
         row  ['9999995', '', '18210', '107A STREET NW', '510000', 'Residential', '3120', 'CH
AMBERY', 'Ward 3', 'Y', '53.6439069966381', '-113.510518455932']  EOR.
         row by step: 9999995    18210  107A STREET NW  510000  Residential  3120  CHAMBERY
Ward 3  Y  53.6439069966381  -113.510518455932
         row  ['9999994', '', '18206', '107A STREET NW', '490500', 'Residential', '3120', 'CH
AMBERY', 'Ward 3', 'Y', '53.6438117918478', '-113.51030978001']  EOR.
         row by step: 9999994    18206  107A STREET NW  490500  Residential  3120  CHAMBERY
Ward 3  Y  53.6438117918478  -113.51030978001
         row  ['9999993', '', '18205', '106A STREET NW', '408500', 'Residential', '3120', 'CH
AMBERY', 'Ward 3', 'Y', '53.6446612734278', '-113.507102600897']  EOR.
         row by step: 9999993    18205  106A STREET NW  408500  Residential  3120  CHAMBERY
Ward 3  Y  53.6446612734278  -113.507102600897
         row  ['9999992', '', '18209', '106A STREET NW', '412000', 'Residential', '3120', 'CH
AMBERY', 'Ward 3', 'Y', '53.644769098446', '-113.507101328517']  EOR.
         row by step: 9999992    18209  106A STREET NW  412000  Residential  3120  CHAMBERY
Ward 3  Y  53.644769098446  -113.507101328517
         row  ['9999991', '', '18213', '106A STREET NW', '423000', 'Residential', '3120', 'CH
AMBERY', 'Ward 3', 'Y', '53.6448679381013', '-113.507100176022']  EOR.
         row by step: 9999991    18213  106A STREET NW  423000  Residential  3120  CHAMBERY
Ward 3  Y  53.6448679381013  -113.507100176022
         row  ['9999990', '', '18217', '106A STREET NW', '402500', 'Residential', '3120', 'CH
AMBERY', 'Ward 3', 'Y', '53.6449667166816', '-113.507084188726']  EOR.
         row by step: 9999990    18217  106A STREET NW  402500  Residential  3120  CHAMBERY
Ward 3  Y  53.6449667166816  -113.507084188726
         row  ['9999989', '', '18221', '106A STREET NW', '435000', 'Residential', '3120', 'CH
AMBERY', 'Ward 3', 'Y', '53.6451191944828', '-113.507015764117']  EOR.
         row by step: 9999989    18221  106A STREET NW  435000  Residential  3120  CHAMBERY
Ward 3  Y  53.6451191944828  -113.507015764117
Processed  11  lines.
```

**Figure 10 Output of Python CSV parsing code (full output)**

This example also shows the column names and first few rows of data (controlled by line_count) – as an example process it makes sense to limit the amount of information flying by.

## CONCLUSION

The SAS University Edition provides capabilities with both SAS and Python allowing you to learn (and play around – or create examples for SESUG papers) in multiple technologies. Each has advantages and disadvantages. The key really is: what does the employer want? The more skills you have, the better your opportunities.

## REFERENCES


- XML
  - https://www.geeksforgeeks.org/xml-parsing-python/
  - https://docs.python.org/2/library/xml.etree.elementtree.html
  - https://support.sas.com/resources/papers/proceedings/proceedings/sugi29/119-29.pdf

    - https://support.sas.com/resources/papers/proceedings17/1318-2017.pdf
    - http://support.sas.com/documentation/cdl//en/engxml/64990/HTML/default/viewer.htm#n1p6kbmn43fz0en1tajxf3y7karg.htm#n0cioh5o1phs5jn1nh0f634dsngv
- XML Mapper
    - https://support.sas.com/downloads/package.htm?pid=486
- JSON
    - https://realpython.com/python-json/
    - https://support.sas.com/resources/papers/proceedings17/0856-2017.pdf
    - https://blogs.sas.com/content/sasdummy/2016/12/02/json-libname-engine-sas/
- CSV
    - http://www2.sas.com/proceedings/sugi30/038-30.pdf
    - http://support.sas.com/documentation/cdl/en/proc/61895/HTML/default/viewer.htm#a000314361.htm
    - RFC 4180, IETF 2005, http://tools.ietf.org/html/rfc4180
    - http://en.wikipedia.org/wiki/Comma-separated_values
    - http://support.sas.com/kb/3/610.html
- Edmonton Open Government data: https://data.edmonton.ca/
    - Property Assessment Data
        - https://data.edmonton.ca/City-Administration/Property-Assessment-Data-Current-Calendar-Year-/q7d6-ambg
        - EXPORT, scroll down for data dictionary -- CSV, dictionary in JSAON
    - Traffic Distruption Map
        - https://data.edmonton.ca/Transportation/Traffic-Disruptions-Map/mhbf-f3zb
        - EXPORT, scroll down for actual data, no data dictionary -- CSV and XML and CSV for Excel
    - Property Assessment Data
        - https://data.edmonton.ca/City-Administration/Property-Assessment-Data-Current-Calendar-Year-/q7d6-ambg
        - EXPORT, scroll down for data dictionary -- CSV, dictionary in JSAON
    - Traffic Distruption Map
        - https://data.edmonton.ca/Transportation/Traffic-Disruptions-Map/mhbf-f3zb
        - EXPORT, scroll down for actual data, no data dictionary -- CSV and XML and CSV for Excel


## ACKNOWLEDGMENTS


I want to thank the organizers of this great conference, my employer for their willingness to allow me to expand my horizons through events like this, and, last but certainly not least, my spouse Mary who doesn't complain when I spend so much time at the keyboard working on documents like this.


## CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:


David B. Horvath, CCP
+1-610-859-8826
dhorvath@cobs.com
http://www.cobs.com
LinkedIn: https://www.linkedin.com/in/dbhorvath/