SESUG Paper 152-2019

Integrate Python with SAS® using SASPy for a Simpler, More Effective Script

John Vickery, North Carolina State University

ABSTRACT

Why write two scripts in two different languages when you can get it done in one? By using the SASPy module, you can easily move data between SAS and Python sessions giving you the best of both worlds. At the NC State University Libraries, we need to manage a dynamic, million plus e-book collection with on-demand purchasing and auto-upgrade options. It is common for a publisher to release an e-book across multiple platforms each having differing access rights. In order to prevent duplicate auto-purchases we regularly combine data from SAS data sets and our catalog web services. In this case study, we show how Python handles the web service and SASPy allows us quick access to large data sets on disk. This paper will do a deep dive into the script and will show how effective an open analytics combination can be. In addition to SASPy this paper also shows examples of a few common Python modules such as Pandas, Requests and ElementTree.

INTRODUCTION

Why write two scripts in two different languages when you can get it done in one? By using the SASPy module, you can easily move data between SAS and Python sessions giving you the best of both worlds.

This paper discuses a case study at the North Carolina State University Libraries where we are using SASPy with Python to access a large SAS dataset on disk while other modules of Python handle a call to our catalog web service.

At the Libraries, we need to manage a dynamic, million plus e-book collection with on-demand purchasing and auto-upgrade options. It is common for a publisher to release an e-book across multiple platforms each having differing access rights. In order to prevent duplicate auto-purchases we regularly combine data from SAS data sets and our catalog web services.

In this paper, we show how Python handles the web service call and SASPy allows us quick access to large data sets on disk. This paper begins with a brief introduction of SASPy. Next using code from the complete script, the paper touches on three common Python modules: Pandas, Requests and ElementTree. The final section steps through the script line by line.

While there is much more related to Python and SASPy than is covered here, this paper may appeal to experienced and newer SAS programmers who are interested in integrating Python into their toolkit.

SAS DATA SETS USED IN THE SCRIPT

The script shown in this paper references the following three SAS data sets in a library named SIRSI:

- SIRSI.ITEMS is a 4.2 million observation (1.9 GB) data set of item level detail for each record in the Libraries' catalog. An observation in the ITEMS data set has many-to-one relationship to the TITLES data set.
- SIRSI.TITLES is a 2.9 million observation (6.6 GB) data set of title level bibliographic data. An observation in the TITLES data set has a one-to-many relationship to the ITEMS data set.
- SIRSI.ISBN is a 2.7 million observation (62 MB) data set of print and e-book ISBNs. An observation in

the ISBN data set has a many-to-many relationship to the ITEMS data set.

SASPY BASICS

SASPy is a Python module that allows SAS 9.4 and above to interface with a Python process. In order to use the SASPy module, you must have both Python 3.x or above and SAS 9.4 or above. While it is not the focus of this article, SASPy is included in the current SAS University Edition.

STARTING A SAS SESSION

The first step to using SASPy in a Python script is to import the module and initialize a SAS session. The following code uses Python's `IMPORT` statement and the `saspy.SASsession()` method to get started. We also use the SASPy `SASLIB()` method to assign a SAS LIBREF.

```
❶  import pandas as pd
❷  import saspy
❸  import requests
❹  import xml.etree.ElementTree as ET

❺  sas = saspy.SASsession()

❻  sas.saslib(libref="sirsi", path="D:\\Sirsi_data\\current")
```

In this snippet:

Lines 1 – 4 import each of the modules used in the script. SASPy is imported in line 2. It is standard to include a separate `IMPORT` statement on a new line for each module imported. The Pandas, Requests and ElementTree modules are discussed in sections below.

Line 5 initializes a SAS session named `sas`. The variable `sas` is used throughout the script to refer to the SAS session.

Line 6 uses the `saslib()` method to assign a SAS libref. Note that we reference the SAS session using the `sas` variable name from line 5. The first parameter `libref=` is the valid SAS libref. The second parameter is the path to the library.

SD2DF METHOD

Within the context of the case study and script discussed in this paper, the primary benefit of SASPy is the ability to access SAS data sets from within a Python session. The `sd2df()`method makes this a simple process. Note that `sd2df()` is an alias for `sasdata2dataframe()`. The method returns a Pandas data frame.

The following is an example from the full script using only two parameters:

```
❶  sasdups = sas.sd2df(table='df', libref='work')
❷  sasdups.info()
```

In **line 1**, the returned Pandas data frame is assigned the name `sasdups`. Note that the `sd2df()` method is called on the SAS session named `sas` from line 5 in the previous section. The first parameter `table=` specifies the name of the SAS data set to be exported to a Pandas data frame. The second parameter `libref=` specifies the libref of that SAS data set.

Line 2 calls Pandas' `info()`method. This method is useful as it "prints information about a DataFrame including the index dtype and column dtypes, non-null values and memory usage." Note that this line is not included in the full script detailed below.

The `sd2df()` method also includes parameters that can be useful when working with larger SAS data sets.

The following code (not included in the final script) uses the `dsopts` parameter to keep only those variables in the SAS data set that begin with "itemcat":

```
itemcats = sas.sd2df(table="items", libref="sirsi",
                     dsopts={"keep": "itemcat:"})
```

Note the use of a colon `:` as a variable name wildcard. The dsopts parameter accepts a python dictionary with any of the following SAS data set options: `WHERE, DROP, KEEP, OBS, FIRSTOBS, FORMAT`.

The `sd2df()` method also accepts a method parameter specifying either `memory` (the default) or `csv`. CSV uses an intermediary csv file that can be faster for large data sets.

DF2SD METHOD

In order to move data from Python to SAS, SASPy provides the `df2sd()` method. This method imports a Pandas data frame to a SAS data set. As with `sd2df()`, `df2sd()` is an alias for `dataframe2sasdata().` The method returns a SASdata object.

The following is an example of the method from the full script:

```
sas.df2sd(df, table="df", libref="work")
```

This example specifies three parameters. The first is the Pandas data frame to be imported. In this case, it is a data frame named `"df"`. The second parameter, `table=`, is the name of the SAS data set to create. Finally, `libref=` specifies the libref for the data set that is being created.

Note that the SASdata object is not assigned to a Python variable. In the next section, we show how the `submit()` method can be used to access the SAS data set created with the `df2sd()` method.

SUBMIT METHOD

The submit() method is particularly helpful as it allows for any SAS code to be submitted. The method returns the SAS log and listing output as a Python dictionary.

In the example below, a PROC SQL procedure is submitted to create a data set within the work library of the SAS session:

```
❶  results_dict = sas.submit("""
❷  proc sql;
❸     create table controls as
❹     select distinct i.catkey, i.titlecontrol, i.itemid,
❺                     t.title, t.author, t.pubyear
❻     from sirsi.items as i, sirsi.titles as t
❼     where i.catkey in ( select distinct catkey_ebook from df ) and
❽           i.titlecontrol = t.titlecontrol;
❾  quit;
❿  """)
```

In this example, only **lines 1 and 10** contain SASPy specific code. **Lines 2 through 9** contain standard SAS Proc SQL code. The code to submitted is entered as a Python string hence the triple quotes.

In line 7, note the table named `df`. This is the SAS data set created using the `df2sd()` method from the previous section.

PANDAS, REQUESTS AND ELEMENTTREE

Besides SASPy, Pandas, Requests and ElementTree are the only other Python modules used in this case study script. While it is beyond the scope of this paper to give a complete overview of each of these modules, this section points out a few highlights of each.

PANDAS

In many ways, you can consider Pandas the essential library for data manipulation and analysis with Python. Full documentation on the Pandas library can be found here:

https://pandas.pydata.org/pandas-docs/stable/

Within Pandas, two data structures are particularly helpful. They are as McKinney (2018) states the "two workhorse data structures: *Series* and *DataFrame*. While they are not a universal solution for every problem, they provide a solid, easy-to-use basis for most applications."

The Pandas "Comparison with SAS" documentation section provides a helpful terminology translation. A DataFrame is the Pandas equivalent of a SAS data set. A Series can be considered analogous to a single column of a data set.

Other Pandas to SAS terms from the "Comparison with SAS" documentation include the following:

Pandas	SAS
DataFrame	Data set
Series	Data set column
Column	Variable
Row	Observation
Groupby	By-group
Nan	.

Table 1. Pandas and SAS terms

As with SASPy, the Pandas module must be imported. It is convention to import the Pandas module as follows:

```
import pandas as pd
```

Among the Pandas specific code used in the script described in this paper, two are particularly useful: `DataFrame.loc` and `DataFrame.merge`.

The `DataFrame.loc` method is used to select "a group of rows and columns by label(s) or a boolean array" (Pandas docs). In the example below, the `.loc` method is applied to a DataFrame named "sasdups" to select rows where the "provider_group" column is null. In this case, the input in the `[]` brackets is a boolean series:

```
sasdups.loc[sasdups['provider_group'].isnull()]
```

The `DataFrame.merge` method is used to "merge DataFrame or named Series objects with a database-style join" (Pandas docs). The merge method is analogous to a SAS DATA STEP MERGE or a PROC SQL join.

The following simple example from the full script merges two DataFrames using the how= and on= parameters:

```
dups = pd.merge(EBSISBN, ebookISBN, how="inner", on="isbn")
```

The method can also perform left, right and outer merges by replacing "inner" on the how= parameter.

REQUESTS

A significant portion of the script discussed in this paper involves accessing the libraries' web service. The Requests module for Python makes it very simple to send HTTP/1.1 requests. In fact, the script in this paper only uses five lines of Requests specific code. You can find the full documentation for the Requests module here: https://2.python-requests.org/en/master/

As shown in the "Starting a SAS Session" section above, the first requirement is to import the module:

```
import requests
```

Next, it's as simple as passing a URL to the `get()`method. In the line below, the URL is an XML response from the libraries' web service:

```
r = requests.get('https://sirsi.lib.ncsu.edu/cgi-bin/retrieve_marc?key=2123464')
```

The response object is named "r" and you can use other Requests module methods to get additional information. For example, you can access the response object content as bytes with the `r.content`. In the following line, the response content is used as an argument to a method from the ElementTree module:

```
tree = ET.fromstring(r.content)
```

The web service described in this paper returns XML content. If your response is JSON, the Requests module provides a built in JSON decoder: `r.json()`

The Requests module also makes it simple to handle errors and exceptions:

`r.raise_for_status()` will raise an HTTPError if the HTTP request returns an unsuccessful status code.

Used within a Python Try-Except block, these lines from the full script will print out any HTTP errors to the terminal:

```
except requests.exceptions.HTTPError as e:
    print("Error: " + str(e))
```

ELEMENTTREE

The web service described in this paper only returns an XML response. The xml.etree.ElementTree module is a simple solution to parse XML data. Just as with the Pandas and Requests sections, this section barely scratches the surface of the module. You can access the full documentation for the module API here: https://docs.python.org/3.7/library/xml.etree.elementtree.html

The main requirement for the script described in this paper is to parse an XML document response and access specific elements within its hierarchical structure.

As always, the module must be imported. It is convention to import the ElementTree module as follows:

```
import xml.etree.ElementTree as ET
```

ElementTree is actually a sub-module of the xml package. Additional XML processing interfaces are also grouped in the xml package. You can find those modules here: https://docs.python.org/3.7/library/xml.html

To parse XML content directly from a string, you can use the `ET.fromstring()`function:

```
tree = ET.fromstring(r.content)
```

Note that this is the same line shown above in the Requests section. This function will return an "Element" instance. In the line above, this Element instance is assigned to a variable named "Tree". Various objects can be accessed within the Element instance. In this script, the following objects are used to extract data from the XML response:

```
findall()
```

- Finds all matching subelements by tag name or path and returns a list in document order

`attrib`

- Python dictionary containing the elements attributes this Python.

`text`

- The text attribute holds either the text between the element’s start tag and its first child or end tag, or None

LINE-BY-LINE THROUGH THE CODE

This section puts together the pieces discussed above and looks at the code line by line. Lines that produce output other than the final excel spreadsheet such as the `DataFrame.info()` method have been removed.

The following is the complete script:

```
1.   import pandas as pd
2.   import saspy
3.   import requests
4.   import xml.etree.ElementTree as ET

5.   # start SAS session
6.   sas = saspy.SASsession()

7.   # assign siri libref
8.   sas.saslib(libref="sirsi", path="D:\\Sirsi_data\\current")

9.   # submit SAS PROC SQL to get ISBNS for EBS and NON-EBS items
10.   results_dict = sas.submit("""
11.   proc sql;
12.      /* EBS ISBNs */
13.     create table EBSISBN as
14.     select isbn.*
15.     from sirsi.items as i,
16.          sirsi.isbn
17.     where i.itemcat3 = 'LEASED' and
18.           i.itemtype = 'EBOOK' and
19.           i.catkey = isbn.catkey
20.     order by isbn;

21.     /* NON-EBS ISBNs */
22.     create table ebookISBN as
23.     select isbn.*
24.     from sirsi.items as i,
25.          sirsi.isbn
26.     where i.itemcat3 ^= 'LEASED' and
27.           i.itemtype = 'EBOOK' and
28.           i.catkey = isbn.catkey
```

```
   order by isbn;
quit;
""")

# convert SAS datasets from above to dataframe
EBSISBN = sas.sd2df(table="EBSISBN", libref="work")
ebookISBN = sas.sd2df(table="ebookISBN", libref="work")

# rename catkey columns in dataframes
EBSISBN.rename(columns={"catkey": "catkey_ebs"}, inplace=True)
ebookISBN.rename(columns={"catkey": "catkey_ebook"}, inplace=True)

# merge to get dups
dups = pd.merge(EBSISBN, ebookISBN, how="inner", on="isbn")

# get unique catkeys from dups dataframe
ebookcats = dups["catkey_ebook"].drop_duplicates()

# pass catkeys to MARC retrieve API
# URL for MARC service
serviceurl = 'https://sirsi.lib.ncsu.edu/cgi-bin/retrieve_marc?key='
suffix = '&format=marcxml'

# iterate over catkeys series
lst = list()
for catkey in ebookcats:
    datalst = dict()
    datalst['catkey_ebook'] = catkey
    url = serviceurl + str(catkey) + suffix
    r = requests.get(url)
    try:
        r.raise_for_status()
        tree = ET.fromstring(r.content)
        for l in tree.findall('.//{http://www.loc.gov/MARC21/slim}datafield'):
            for sf in l.findall('.//{http://www.loc.gov/MARC21/slim}subfield'):
                if l.attrib['tag'] == '506':
                    if sf.attrib['code'] == 'a':
                        datalst['UserAccessNote'] = sf.text
                    if l.attrib['tag'] == '856':
                        if sf.attrib['code'] == 'u':
                            datalst['URL'] = sf.text
                        if sf.attrib['code'] == 'z':
                            datalst['provider'] = sf.text
```

```
      lst.append(datalst)
  except requests.exceptions.HTTPError as e:
      print("Error: " + str(e))

# save as dataframe
df = pd.DataFrame(lst)

# convert to SAS dataset and add data
sas.df2sd(df, table="df", libref="work")

results_dict = sas.submit("""
proc sql;
   create table controls as
   select distinct i.catkey,
                   i.titlecontrol,
                   i.itemid,
                   t.title,
                   t.author,
                   t.pubyear
   from sirsi.items as i,
        sirsi.titles as t
   where i.catkey in ( select distinct catkey_ebook
                       from df ) and
         i.titlecontrol = t.titlecontrol
   order by i.catkey;
quit;
""")

results_dict = sas.submit("""
/* merge MARC and sirsi data */
proc sort data=df;
    by catkey_ebook;
run;

data df;
   merge df(in=d rename=(catkey_ebook=catkey))
         controls(in=c);
   by catkey;
   if d;
   providerID = compress(titlecontrol,,'k d');
   label catkey = 'Catkey';
run;
""")

# back to python for output
```

```
105. sasdups = sas.sd2df(table='df', libref='work')

106. # group providers
107. providerpairs = [('EBSCO', 'EBSCO'),
108.                 ('ProQuest|ebrary', 'ProQuest-Ebrary'),
109.                 ('Project', 'PROJECT MUSE')]

110. for pair in providerpairs:
111.     sasdups.loc[sasdups.provider.str.contains(pair[0], na=False),
112.                 'provider_group'] = pair[1]

113. # group null values as “OTHER”
114. sasdups.loc[sasdups['provider_group'].isnull(), 'provider_group'] = 'OTHER'

115. # date variable in YYYY-MM-DD format
116. today = pd.Timestamp('now').strftime('%Y-%m-%d')

117. # variable for output file name
118. fname = 'EBS_dups_' + today + '.xlsx'

119. # create a sorted list of provider groups
120. providergroups = sorted(sasdups.provider_group.unique().tolist())

121. # output to excel with a separate worksheet for each provider group
122. with pd.ExcelWriter(fname, engine='xlsxwriter') as writer:
123.     for group in providergroups:
124.         sasdups.loc[sasdups['provider_group'] == group].to_excel(
125.            writer, sheet_name=group, startrow=1, header=False, index=False)
```

1 – 4. Load the necessary Python modules with the import statement. Note that `Pandas` and `ElementTree` are assigned the names `pd` and `ET`.

6. Start as SAS session with the `saspy.SASsession()` method. The session is assigned the name `sas`.

8. Assign a SAS libref with the `saspy.saslib()` method. Note that saspy is referred to by the name `sas` from line 6. Throughout the script, saspy is referenced using the name `sas`.

9 – 31. Submit SAS PROC SQL code to the SAS session by using the `saspy.submit()` method. Two SAS data sets are accessed from the “sirsi” libref and two SAS data sets are created in the work library of the SAS session.

32 – 34. The two SAS data sets created from the PROC SQL code in lines 9 – 34 are imported to Pandas data frames using the `saspy.sd2df()` method.

35 – 37. Rename the column “catkey” in each of the two DataFrames created in lines 33 and 34. The method used is the `DataFrame.rename()` method. The first parameter `columns=` takes a python dictionary as the argument. This specifies the current name and the new name. The `inplace=` parameter specifies whether to return a new DataFrame or not.

38 – 39. This merges the two DataFrames from lines 32 – 34 using the `pandas.merge()` method. This is equivalent to a SAS DATA STEP MERGE or a PROC SQL join operation. The `merge()` method has many possible configurations. In this case, it is a simple inner join (as specified by the `how=` parameter) on the "isbn" columns (the `on=` parameter) of the two DataFrames.

40 – 41. This creates a Pandas Series of unique values from the "catkey_ebook" column of the "dups" DataFrame created in line 41. The Series is named "ebookcats". The `drop_duplicates()` method creates the unique values.

42 – 45. These lines create two variables, "serviceurl" and "suffix" that will be used to build a valid URL to pass to the libraries' API.

46 – 68. This block loops through each value of the "ebookcats" series from line 41 and passes it to the libraries' catalog API. Lines from the loop are examined in more detail below:

> **47.** Create and empty Python list named "lst". This is used to hold the results of the API calls.
>
> **48.** Begin a Python "for loop" to iterate over each value in the "ebookcats" series. This is analogous to a SAS DO LOOP.
>
> **49.** Create an empty Python dictionary named "datalst" to hold the data elements retrieved from the API. This dictionary is appended to the "lst" list and overwritten with each iteration of the loop.
>
> **50.** Create a key in the "datalst" named "catkey_ebook" and assign it the value of the current catkey in the loop iteration.
>
> **51.** Concatenate the URL elements from lines 44 and 45 using the + operator. The current catkey is added between. Note that it must be converted to a string using the `str()` function to be used in a concatenation operation.
>
> **52.** Send a GET request to the url constructed in line 51.
>
> **53.** Begin a Python TRY block. Python Try – Except blocks are used for exception handling. The Try block tests the code while the Except block handles the error or exception.
>
> **54.** Raise any stored HTTPErrors.
>
> **55.** Parse the API's XML response (`r.content`) with the `ElementTree.fromstring()` method. The parsed XML is named "tree".
>
> **56.** Combine a "for loop" and the ElementTree.findall() method to loop through all matching sub-elements of the "datafield" tag. Note that the XML schema is used to simplify the path.
>
> **57 - 65.** Within the "for loop" from line 56, begin another loop to iterate through all matching "subfield" tags.
>
> > **58.** As in SAS, the `IF` statement is used for conditional execution. In this line, the condition is whether or not the value for the "tag" key in the dictionary of attributes for the "l" elements equals "506".
> >
> > **59.** The same as 58 but checking the "code" key.
> >
> > **60.** Assign the text content of the XML element to the "UserAccessNote" key of the datalst dictionary.
> >
> > **61 – 65.** The same process as lines 58 – 60.
>
> **66.** Append the "datalst" dictionary to the end of the "lst" list. "lst" will become a list of dictionaries.
>
> **67.** This Except statement saves a request HTTPError as a variable named "e".
>
> **68.** Print the error from line 67 if it exists.

70. Construct a Pandas DataFrame named “df” from the list of dictionaries, “lst”.

72. Import the DataFrame from line 70 to a SAS data set named “df”. The data is saved to the “work” library.

73 – 89. Submit SAS PROC SQL code to the SAS session by using the `saspy.submit()` method. Two SAS data sets are accessed from the “sirsi” libref. The SAS data set from line 72 is accessed from the “work” library. A SAS data set is also created in the work library of the SAS session.

90 – 103. This is the same as lines 73 – 89 except that the SAS code a PROC SORT and a DATA STEP MERGE. Note that we are using the “df” data set from line 72.

105. Export the “df” SAS data set back to Python using the `saspy.sd2df()` method.

106 – 109. Create a Python list of tuples named “providerpairs”. The list is used in the following “for loop”. Python tuples are sequences like lists but are immutable.

110 – 112. Loop through each tuple in the “providerpairs” list and use the `DataFrame.loc` method with a conditional. The conditional checks if the “provider” column contains the first element in the tuple pair and assigns the “provider_group” column the value from the second tuple element.

114. This line is essentially the same as line 111. The only difference is that the conditional checks for null values in the “provider_group” column and assigns them the value of “OTHER”.

116. Combine Pandas `Timestamp()` and `strftime()` methods to create a date variable in the YYYY-MM-DD format.

118. Create a variable for the output filename using the + concatenation operator and the date variable from line 116.

120. Create a sorted list of unique values from the “provider_group” column by combining the sorted function with the `unique()` and `tolist()` methods. The `unique()` method returns an array of unique values from a Pandas series.

121 – 125. These lines create the final excel output with a separate worksheet for each “provider_group”.

> **122.** The Pandas ExcelWriter class writes DataFrame objects into excel sheets. Using the `with` statement will automatically close the writer object.
>
> **123.** Start a loop to iterate over each element in the “providergroups” list from like 120.
>
> **124.** Use the `DataFrame.loc` method to subset the “sasdups” DataFrame for the current value of the “providergroups” list. This returns a DataFrame.
>
> **125.** Continuing line 124: the `to_excel()` method is applied to the returned DataFrame. The first parameter “writer” specifies the object from line 122. The `sheet_name=` parameter will label each worksheet according to the current value of the “providergroups” list. `startrow=` indicates the upper left cell row to start the output. The `header=` and `index=` parameters indicate whether to write out the column names and index column.

CONCLUSION

SASPy enables you to move data between SAS and Python sessions. This paper discussed a case study at the North Carolina State University Libraries where we are using SASPy with Python to access a large SAS dataset on disk while other modules of Python handle a call to our catalog web service. This paper began with a brief introduction of SASPy. Next using code from the complete script, the paper touched on three common Python modules: Pandas, Requests and ElementTree. The final section stepped through the script line by line.

While there is much more related to Python and SASPy than can be covered here, this paper may appeal to SAS programmers who are interested in integrating Python into their toolkit with SASPy.

REFERENCES

McKinney, W. (2018). *Python for Data Analysis : Data Wrangling with Pandas, NumPy, and IPython* (Vol. Second edition). Sebastopol, CA: O'Reilly Media. Retrieved from http://search.ebscohost.com/login.aspx?direct=true&db=nlebk&AN=1605925&site=ehost-live

RECOMMENDED READING

- "10 minutes to pandas": https://pandas.pydata.org/pandas-docs/stable/getting_started/10min.html#min
- "How to code in Python with SAS 9.4": https://blogs.sas.com/content/sgf/2018/01/10/come-on-in-were-open-the-openness-of-sas-94/
- "Introducing SASPy: Use Python code to access SAS" on The SAS Dummy blog: https://blogs.sas.com/content/sasdummy/2017/04/08/python-to-sas-saspy/
- McKinney, W. (2018). *Python for Data Analysis : Data Wrangling with Pandas, NumPy, and IPython*
- Pandas "Comparison with SAS" documentation: https://pandas.pydata.org/pandas-docs/stable/getting_started/comparison/comparison_with_sas.html
- SASPy GitHub Repository: https://sassoftware.github.io/saspy/
- Vanderplas, J. T. (2016). *Python Data Science Handbook : Essential Tools for Working with Data*

CONTACT INFORMATION

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

John Vickery
North Carolina State University Libraries
(919) 513-0344
John_vickery@ncsu.edu