CHAPTER 10

saspy Module

In this chapter we discuss the open source saspy module contributed by SAS Institute. saspy exposes Python APIs to the SAS System. This module allows a Python session to do the following:

- From within a Python session start and connect to a local or remote SAS session
- Enables bi-directional transfer of values between SAS variables and Python objects
- Enables bi-directional exchange of data between pandas DataFrames and SAS datasets
- Integrate SAS and Python program logic within a single execution context executing interactively or in batch (scripted) mode

To get started, you need to take the following steps:

- Install the saspy module
- Modify the saspy.SAScfg configuration file
- Make SAS-supplied Java .jar files available to saspy

Install saspy

On Windows, to install saspy, issue the following command in a Windows terminal session:

```
python -m pip install saspy
```

The installation process downloads any saspy dependent packages. Listing 10.1, saspy Install on Windows displays the output from a Windows terminal for installing saspy.

Listing 10.1. saspy Install on Windows

```
c:\>python -m pip install saspy
Collecting saspy
  Downloading
https://files.pythonhosted.org/packages/bb/07/3fd96b969959ef0e
701e5764f6a239e7bea543b37d2d7a81acb23ed6a0c5/saspy-
2.2.9.tar.gz (97kB)
    100% |████████████████████████████████| 102kB 769kB/s
Successfully built saspy
distributed 1.21.8 requires msgpack, which is not installed.
Installing collected packages: saspy
Successfully installed saspy-2.2.9
```

You should see the statement:

```
Successfully installed saspy-2.2.9
```

Modify the saspy.SASCFG Configuration File

After completing installation, the next step is to modify the `saspy.SAScfg` file to establish which access method Python uses to connect to a SAS session.

In this example we configure an IOM (integrated object model) connection method so that the Python session running on Windows connects to a SAS session running on the same Windows machine. If you have a different set-up, for example, running Python on Windows and connecting to a SAS session executing on Linux, you use the STDIO access method. The detailed instructions are at:

https://sassoftware.github.io/saspy/install.html#configuration

For this step, begin by locating the saspy.SAScfg configuration file created for you during the saspy installation process. Listing 10.2, Locate saspy.SAScfg Configuration File illustrates the Python syntax needed to locate the saspy configuration file.

Listing 10.2. Locate saspy.SAScfg Configuration File

```
>>> import saspy
>>> saspy.SAScfg
<module 'saspy.sascfg' from
'C:\\Users\\randy\\Anaconda3\\lib\\site-
packages\\saspy\\sascfg.py'>
```

As a best practice you should copy the `sascfg.py` configuration file to `sascfg_personal.py`. Doing so ensures that any configuration changes will not be overwritten when subsequent versions of saspy are installed later. The `sascfg_personal.py` can be stored anywhere on the filesystem. If it is stored outside the Python repo then you must always include the fully-qualified path name to the SASSession argument like:

```
sas =
SASSession(cfgfile='C:\\qualified\\path\\sascfg_personal.py)
```

Alternatively, if the `sascfg_personal.py` configuration file is found in the search path defined by the `PYTHONPATH` environment variable, then you can avoid having to supply this argument when invoking saspy. Use the Python `sys.path` statement to return the search-path defined by the PYTHONPATH environment variable as shown in Listing 10.3, Finding the PYTHONPATH Search Paths.

Listing 10.3. Finding the PYTHONPATH Search Paths

```
>>> import sys
>>> sys.path
['', 'C:\\Users\\randy\\Anaconda3\\python36.zip',
'C:\\Users\\randy\\Anaconda3\\DLLs',
'C:\\Users\\randy\\Anaconda3\\lib',
'C:\\Users\\randy\\Anaconda3',
'C:\\Users\\randy\\Anaconda3\\lib\\site-packages',
'C:\\Users\\randy\\Anaconda3\\lib\\site-packages\\win32',
'C:\\Users\\randy\\Anaconda3\\lib\\site-packages\\win32\\lib',
'C:\\Users\\randy\\Anaconda3\\lib\\site-packages\\Pythonwin',
'C:\\Users\\randy\\Anaconda3\\lib\\site-
packages\\IPython\\extensions']
```

In our case, we elect to store the `sascfg_personal.py` configuration file in:

```
C:/Users/randy/Anaconda3/lib/site-packages/
```

directory. Copy:

```
C:/Users/randy/Anaconda3/lib/site-packages/saspy/sascfg.py
```

to

```
C:/Users/randy/Anaconda3/lib/site-packages/personal_sascfg.py
```

Depending on how you connect the Python environment to the SAS session determines the changes needed in the `sascfg_personal.py` configuration file.

In our case, both the Python and SAS execution environments are on the same Windows 10 machine. Accordingly, we modify the following sections of the `sascfg_personal.py` configuration file:

From the original sascfg.py configuration file:

```
SAS_config_names=['default']
```

is changed in the `sascfg_personal.py` configuration file to:

```
SAS_config_names=['winlocal']
```

Make SAS-supplied .jar Files Available

The following four Java .jar files are needed by saspy and are defined by the `classpath` variable in the `sascfg_personal.py` configuration file:

```
sas.svc.connection.jar
log4j.jar
sas.security.sspi.jar
sas.core.jar
```

These four .jar files are part of the existing SAS deployment. Depending on where SAS is installed on Windows, the path will be something like:

```
C:\Program
Files\SASHome\SASDeploymentManager\9.4\products\deploywiz__944
98__prt__xx__sp0__1\deploywiz\<required_jar_file_names.jar>
```


A fifth .jar file which is distributed with the saspy repo, `saspyiom.jar` needs to be defined as part of the `classpath` variable in the `sascfg_personal.py` configuration file as well. In our case this jar file is located at:

```
C:/Users/randy/Anaconda3/Lib/site-packages/saspy/java
```

Once you have confirmed the location of these five .jar files, modify the `sascfg_personal.py` file similar to Listing 10.4 CLASSPATH variable for Windows SAScfg_personal.py File. Be sure to modify the paths sepcific to your environment.

Listing 10.4. CLASSPATH variable for Windows SAScfg_personal.py File

```
# build out a local classpath variable to use below for
Windows clients

cpW  =  "C:\\Program
Files\\SASHome\\SASDeploymentManager\\9.4\\products\\deploywiz
__94498__prt__xx__sp0__1\\deploywiz\\sas.svc.connection.jar"

cpW += ";C:\\Program
Files\\SASHome\\SASDeploymentManager\\9.4\\products\\deploywiz
__94498__prt__xx__sp0__1\\deploywiz\\log4j.jar"

cpW += ";C:\\Program
Files\\SASHome\\SASDeploymentManager\\9.4\\products\\deploywiz
__94498__prt__xx__sp0__1\\deploywiz\\sas.security.sspi.jar"

cpW += ";C:\\Program
Files\\SASHome\\SASDeploymentManager\\9.4\\products\\deploywiz
__94498__prt__xx__sp0__1\\deploywiz\\sas.core.jar"

cpW += ";C:\\Users\\randy\\Anaconda3\\Lib\\site-
packages\\saspy\\java\\saspyiom.jar"
```

The last change needed is to update the dictionary values for the `winlocal` object definition in the `sascfg_personal.py` configuration file similar to Listing 10.5, winlocal Definition for sascfg_personal.py Configuration File.

Listing 10.5. winlocal Definition for sascfg_personal.py Configuration File

```
winlocal = {'java'  : 'C:\\Program
Files\\SASHome\\SASPrivateJavaRuntimeEnvironment\\9.4\\jre\\bi
n\\java',
            'encoding'  : 'windows-1252',
            'classpath' : cpW
            }
```

saspy has a dependency on Java 7 which is met by relying on the SAS Private JRE distributed and installed with SAS software. The SAS Private JRE is part of the existing SAS software installation. Also notice the path filename uses double back-slashes to 'escape' the backslash needed by the Windows path names.

saspy Examples

With the configuration for saspy complete we can begin exploring its capabilities. The goal for these examples is to illustrate the ease by which DataFrame and SAS datasets can be interchanged along with calling Python or SAS methods to act on these data assets. We start with Listing 10.6, Start saspy Session to integrate a Python and SAS session together.

Listing 10.6. Start saspy Session

```
>>> import pandas as pd
>>> import saspy
>>> import numpy as np
>>> from IPython.display import HTML
>>>
>>> sas = saspy.SASsession(cfgname='winlocal', results='TEXT')
SAS Connection established. Subprocess id is 5288
```

In this example the Python `sas` object is created by calling the `saspy.SASsession()` method. The `saspy.SASsession` object is the main object for connecting a Python session with a SAS sub-process. Most of the arguments to the `SASsession` object are set in the `sascfg_personal.py` configuration file discussed at the beginning of this chapter.

In this example, there are two arguments, `cfgname=` and `results=`. The `cfgname=` argument points to the `winlocal` configuration values in the `sascfg_personal.py` configuration file indicating both the Python and the SAS session run locally on

Windows. The `results=` argument has three possible values to indicate how tabular output returned from the `SASsession` object, that is, the execution results from SAS, is rendered. They are:

- pandas, the default value
- TEXT, which is useful when running saspy in batch mode
- HTML, which is useful when running saspy interactively from a Jupyter Notebook

Another useful `saspy.SASsession()` argument is `autoexec=`. In some cases, it is useful to execute a series of SAS statements when the `saspy.SASsession()` method is called and before any SAS statements are executed by the user program. This feature is illustrated in Listing 10.7, Start saspy with Autoexec Processing.

Listing 10.7. Start saspy with Autoexec Processing

```
>>> auto_execsas='''libname sas_data "c:\data";'''
>>>
>>> sas = saspy.SASsession(cfgname='winlocal', results='TEXT',
autoexec=auto_execsas)
SAS Connection established. Subprocess id is 15020
```

In this example, we create the `auto_execsas` object by defining a Python DocString containing the SAS statements used as the statements for the autoexec process to execute. Similar to the behavior for the traditional SAS autoexec processing, the statements defined by the `auto_execsas` object are executed by SAS before executing any subsequent SAS input statements.

To illustrate the integration between Python and SAS using `saspy`, we begin by building the `loandf` DataFrame which is sourced from the Lending Club loan statistics described at:

`https://www.lendingclub.com/info/download-data.action`

This data consists of anonymized loan performance measures from Lending Club which offers personal loans to individuals. We begin by creating the `loandf` DataFrame illustrated in Listing 10.8, Build loandf DataFrame.

Listing 10.8. Build loandf DataFrame

```
>>> url =
"https://raw.githubusercontent.com/RandyBetancourt/PythonForSA
SUsers/master/data/LC_Loan_Stats.csv"
>>>
... loandf = pd.read_csv(url,
...     low_memory=False,
...     usecols=(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 13, 15,
16),
...     names=('id',
...            'mem_id',
...            'ln_amt',
...            'term',
...            'rate',
...            'm_pay',
...            'grade',
...            'sub_grd',
...            'emp_len',
...            'own_rnt',
...            'income',
...            'ln_stat',
...            'purpose',
...            'state',
...            'dti'),
...     skiprows=1,
...     nrows=39786,
...     header=None)
>>> loandf.shape
(39786, 15)
```

The `loandf` DataFrame contains 39,786 rows and 15 columns.

Basic Data Wrangling

In order to effectively analyze the `loandf` DataFrame we must do a bit of data wrangling. Listing 10.9, loandf Initial Attributes returns basic information about the columns and values.

Listing 10.9. loandf Initial Attributes


```
loandf.info()
loandf.describe(include=['O'])
```

The `df.describe()` method accepts the `include=['O']` argument in order to return descriptive information for all columns whose datatype is `object`. Output from the `df.describe()` method is shown in a Jupyter notebook in Figure 10.1, Attributes for Character Value Columns.

```
loandf.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 39786 entries, 0 to 39785
Data columns (total 15 columns):
id          39786 non-null int64
mem_id      39786 non-null int64
ln_amt      39786 non-null int64
term        39786 non-null object
rate        39786 non-null object
m_pay       39786 non-null float64
grade       39786 non-null object
sub_grd     39786 non-null object
emp_len     38705 non-null object
own_rnt     39786 non-null object
income      39786 non-null float64
ln_stat     39786 non-null object
purpose     39786 non-null object
state       39786 non-null object
dti         39786 non-null float64
dtypes: float64(3), int64(3), object(9)
memory usage: 4.6+ MB
```

```
loandf.describe(include=['O'])
```

	term	rate	grade	sub_grd	emp_len	own_rnt	ln_stat	purpose	state
count	39786	39786	39786	39786	38705	39786	39786	39786	39786
unique	2	371	7	35	11	5	7	14	50
top	36 months	10.99%	B	B3	10+ years	RENT	Fully Paid	debt_consolidation	CA
freq	29088	958	12029	2918	8905	18906	33669	18684	7101

Figure 10.1. Attributes for Character Value Columns

The `loandf.info()` method shows the `rate` column has a datatype of `object` indicating they are string values. Similarly, the `term` column has a datatype of `object`.

The `loandf.describe(include=['O'])` method provides further detail revealing the values for the `rate` column having a trailing percent sign (%) and the `term` column values are followed by the string 'months'.

In order to effectively use the `rate` column in any mathematical expression, we need to modify the values by:

1. Stripping the percent sign
2. Mapping, or casting the datatype from character to numeric
3. Dividing the values by 100 to convert from a percent value to a decimal value

In the case of the `term` column values we need to:

1. Strip the string ' months' from the value
2. Map the datatype from character to numeric

Both modifications are shown in Listing 10.10, Basic Data Wrangling.

Listing 10.10 Basic Data Wrangling

```
>>> loandf['rate'] =
loandf.rate.replace('%','',regex=True).astype('float')/100
>>> loandf['rate'].describe()
count    39786.000000
mean         0.120277
std          0.037278
min          0.054200
25%          0.092500
50%          0.118600
75%          0.145900
max          0.245900
Name: rate, dtype: float64
>>> loandf['term'] =
loandf['term'].str.strip('months').astype('float64')
```

```
>>> loandf['term'].describe()
count    39786.000000
mean        42.453325
std         10.641299
min         36.000000
25%         36.000000
50%         36.000000
75%         60.000000
max         60.000000
Name: term, dtype: float64
```

The syntax:

```
loandf['rate'] =
loandf.rate.replace('%','',regex=True).astype('float')/100
```

performs an in-place modification to the `df['rate']` column by calling the `replace()` method to dynamically replace values. In this case, the first argument to the `replace()` method is `'%'`, indicating the percent sign (%) is the source string to replace. The second argument is `''` to indicate the replacement exceprt. Notice there is no space between the quotes, in effect, stripping the percent sign from the string. The third argument, `regex='True'` indicates the `replace()` argument is a string.

The `astype()` attribute is chained to the `replace()` method call and maps or casts the `loandf['rate']` column's datatype from object (strings) to a float (decimal value). The resulting value is then divided by 100.

Next, the `describe()` attribute is attached to the `loandf['rate']` column and returns basic statistics for the newly modified values.

The syntax:

```
loandf['term'] =
loandf['term'].str.strip('months').astype('float64')
```

performs a similar in-place update operation on the `loandf['term']` column. The `strip()` method removes the string 'months' from the values. Chaining the `astype()` method casts this column from an `object` datatype to a `float64` datatype.

Of course, we could have applied the `str.strip()` method to the percent (%) sign for the `df['rate']` column rather than calling the `replace()` method.

Write DataFrame to SAS Dataset

The pandas IO Tools library does not provide a method to export DataFrames to SAS dataset. As of this writing, the saspy module is the only Python module to provide this capability. In order to write a DataFrame to a SAS dataset, we need to define two objects in the Python session.

The first object to make known to Python is the Libref to the target SAS library where the SAS data is to be created from the exported DataFrame. This object is defined by calling the `sas.saslib()` method to establish a SAS Libref to the target SAS data library. This step is not needed if the target SAS data library is the `WORK` library or if there are Librefs defined as part of the autoexec processing described in Listing 10.7, Start saspy with Autoexec Processing. Also note, your site may have Librefs defined with `autoexec.sas` processing independent of the approach described in Listing 10.7. In any of these cases, if a Libref is already known at saspy initialization time, then you can omit this step.

The second step writes the DataFrame rows and columns into the target SAS dataset by calling the `dataframe2sasdataset()` method. Luckily, we can use the alias `df2sas()`.

Define the Libref to Python

With the `loandf` DataFrame shaped appropriately, we can write the DataFrame as a SAS data set illustrating these two steps. Step 1 defines the target Libref by calling the `sas.saslib()` method. This feature is illustrated in Listing 10.11, Define the Libref to Python.

The `sas.saslib()` method accepts four parameters. They are:

1. libref, in this case `sas_data` defining the SAS data library location.

2. engine, or access method. In this case we are using the default `BASE` engine. If accessing a SQL Server table, we would supply `SQLSRV` or `ORACLE` if accessing an Oracle table, etc.

3. path, the file system location for the `BASE` data library, in this case, `C:\data`.

4. options, which are SAS engine or engine supervisor options. In this case, we are not supplying options. An example would be `SCHEMA=` option to define the schema name for SAS/Access to SQL/Server. Any valid SAS `LIBNAME` option can be passed here.

Listing 10.11. Define the Libref to Python

```
>>> sas.saslib('sas_data', 'BASE', 'C:\data')

26    libname sas_data BASE  'C:\data'  ;
NOTE: Libref SAS_DATA was successfully assigned as follows:
      Engine:        BASE
      Physical Name: C:\data
```

Executing this particular call to the `sas.saslib()` method, the `saspy` module forms the SAS `LIBNAME` statement:

```
libname sas_data BASE  'C:\data'  ;
```

and sends the statement for processing to the attached SAS sub-process on your behalf.

Write the DataFrame to a SAS Dataset

Step 2 is to export the DataFrame rows and columns into the target SAS dataset. This is accomplished by calling the `sas.df2sd()` method. This feature is illustrated in Listing 10.12, Write the DataFrame to a SAS Dataset.

The `sas.df2sd()` method has five parameters. They are:

1. The input DataFrame to be written as the output SAS dataset, in this case, the `loandf` DataFrame created previously.

2. `table=` argument which is the name for the output SAS dataset, excluding the `Libref` which is specified as a separate argument.

3. `libref=` argument which, in our case is '`sas_data`' created earlier by calling the `sas.saslib()` method.

4. `results=` argument which in our case uses the default value `PANDAS`. Alternatively, this argument accepts HTML or TEXT as targets.

5. `keep_outer_quotes=` argument which in our case uses the default value `False`, to strip any quotes from delimited data. If you want to keep quotes as part of the delimited data values, set this argument to `True`.

Listing 10.12. Write the DataFrame to a SAS Dataset

```
>>> loansas = sas.df2sd(loandf, table='loan_ds',
libref='sas_data')
>>>
>>> sas.exist(table='loan_ds', libref='sas_data')
1
>>> print(type(loansas))
<class 'saspy.sasbase.SASdata'>
```

The syntax:

```
loansas = sas.df2sd(loandf, table='loan_ds',
libref='sas_data')
```

defines the `loansas` object to Python and writes the DataFrame `loandf` created in Listing 10.8 to the SAS dataset `sas_data.loan_ds`. By assigning the `sas.df2sd()` method call to the `loansas` object, we now have a way to refer to the SAS dataset `sas_data.loan_ds` within the Python context.

The syntax:

```
sas.exist(table='loan_ds', libref='sas_data')
```

is a method call returning a Boolean, except in this case, rather than returning True or False, it returns 1 or 0. This is useful for scripting more complex Python programs using the saspy module for logic branching based on the presence or absence of a particular SAS dataset.

Finally, the `type()` method call shows that the `loansas` object is a SAS Data object which references a SAS dataset or SAS View.

Now that the permanent SAS dataset exists and is mapped to the Python object `loansas`, we can manipulate this object in a more Pythonic fashion. The `loansas` SAS Data Object has several available methods. Some of these methods are displayed in Figure 10.2, SAS Data Object methods.

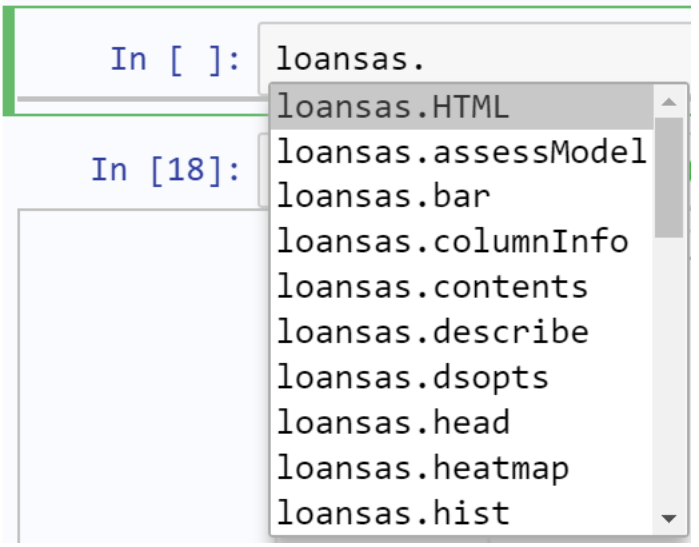

Figure 10.2. SAS Data Object methods

The methods for the SAS Data Object are displayed by entering the syntax:

```
loansas.
```

into the cell of a Jupyter notebook and pressing the <tab> key.

Consider Listing 10.13, Return Column Information.

Listing 10.13. Return Column Information

```
>>> loansas.columnInfo()

                The CONTENTS Procedure
    Alphabetic List of Variables and Attributes
#    Variable    Type    Len


15   dti         Num       8
9    emp_len     Char      9
7    grade       Char      1
1    id          Num       8
11   income      Num       8
3    ln_amt      Num       8
12   ln_stat     Char     18
6    m_pay       Num       8
2    mem_id      Num       8
10   own_rnt     Char      8
13   purpose     Char     18
5    rate        Char      6
14   state       Char      2
8    sub_grd     Char      2
4    term        Char     10
```

The syntax:

```
loansas.columnInfo()
```

returns metadata for the SAS data set by calling `PROC CONTENTS` on your behalf similar to the `loansdf.describe()` method for returning a DataFrame's column attributes. Recall the `loansas` object is mapped to the permanent SAS dataset `sas_data.loan_ds`.

Figure 10.3, Histogram for ln_stat Column illustrates calling the SAS Data Object `bar()` attribute to render a histogram for the loan status variable, in this case, `ln_stat`. For this example, execute the code in Listing 10.13, Loan Status Histogram in a Jupyter notebook. To start a Jupyter notebook on Windows, from a terminal session, enter the command:

```
> python -m notebook
```

To start a Jupyter notebook on Linux, from a terminal session, enter the command:

```
$ jupyter notebook &
```

Copy the program from Listing 10.13, Loan Status Histogram into a cell and press the `>|Run` button.

Listing 10.13 Loan Status Histogram

```
import pandas as pd
import saspy

url = url =
"https://raw.githubusercontent.com/RandyBetancourt/PythonForSA
SUsers/master/data/LC_Loan_Stats.csv"

loandf = pd.read_csv(url,
    low_memory=False,
    usecols=(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 13, 15,
16),
    names=('id',
           'mem_id',
            'ln_amt',
            'term',
             'rate',
              'm_pay',
              'grade',
              'sub_grd',
              'emp_len',
              'own_rnt',
              'income',
              'ln_stat',
              'purpose',
              'state',
              'dti'),
    skiprows=1,
    nrows=39786,
    header=None)

sas = saspy.SASsession(cfgname='winlocal', results='HTML')
sas.saslib('sas_data', 'BASE', 'C:\data')
```

```
loansas = sas.df2sd(loandf, table='loan_ds',
libref='sas_data')
loansas.bar('ln_stat')
```

You may need to supply a differrent value for the `saspy.SASsession(cfgname=` parameter to indentify your specific congiguration file. If you encounter errors related to configuration file set-up, see the SASpy Troubleshooting page at:

https://sassoftware.github.io/saspy/troubleshooting.html

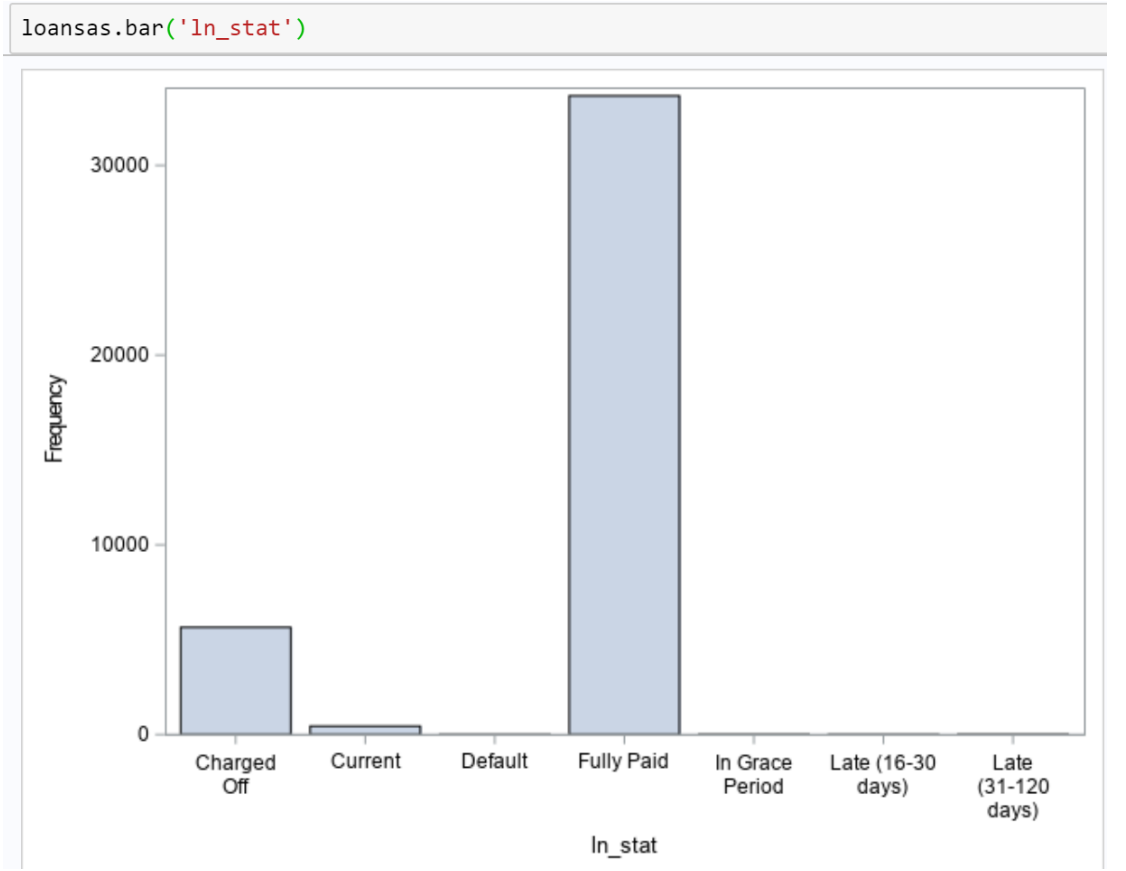

Figure 10.3. Histogram for ln_stat Column

We can see from the histogram that approximately 5,000 loans are charged off, meaning the customer defaulted. Since there are 39,786 rows in the dataset, this represents a charge-off rate of roughly 12.6%.

During development of Python scripts used to call into the saspy module, you will want access to the SAS log for debugging.

The syntax:

```
print(sas.saslog())
```

returns the Log for the entire SAS sub-process which is truncated here. Figure 10.4, saspy Returns Log shows part of the SAS log executing the statement from a Jupyter notebook.

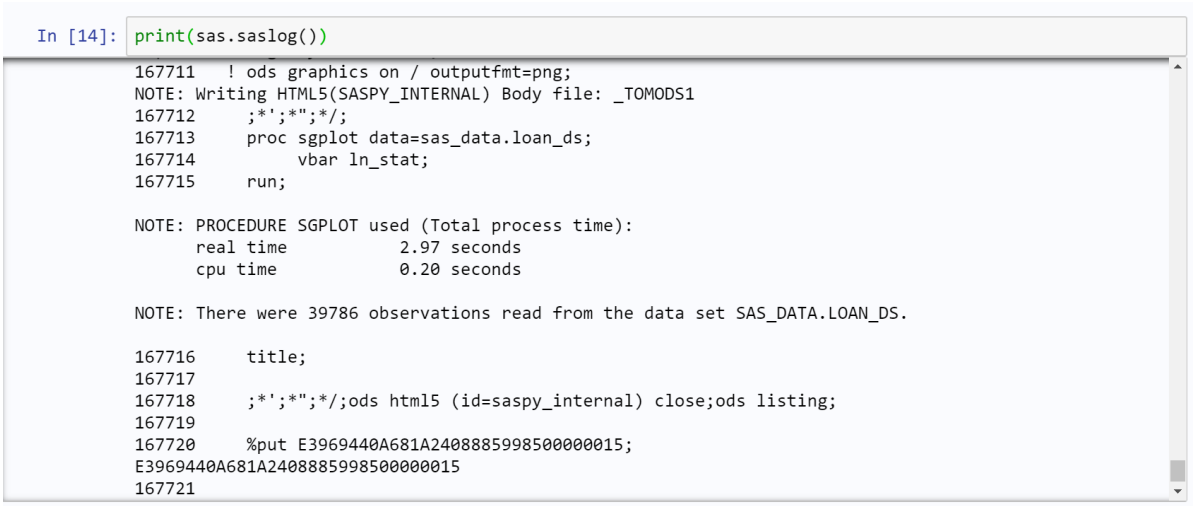

Figure 10.4. saspy Returns Log

The `saspy.SASsession` object has the `.teach_me_SAS()` attribute when set to True, returns the generated SAS code from any method that is called. Listing 10.14, Teach Me SAS, illustrates this capability.

Listing 10.14. Teach Me SAS

```
sas.teach_me_SAS(True)
loansas.bar('ln_stat')
sas.teach_me_SAS(False)
```

Figure 10.5, Teach_me_SAS Attribute displays the output executed in a Jupyter notebook.

```
sas.teach_me_SAS(True)
```

```
loansas.bar('ln_stat')
```

```
proc sgplot data=sas_data.loan_ds;
        vbar ln_stat;
run;
title;
```

```
sas.teach_me_SAS(False)
```

Figure 10.5. Teach_me_SAS Attribute

Execute SAS Code

Another useful feature for the `saspy.SASsession` object is the `submit()` method. This feature enables you to submit arbitrary blocks of SAS code and assign the results to a Python object. Consider Listing 10.15, SAS submit() Method.

Listing 10.15. SAS submit() Method

```
sas_code='''options nodate nonumber;
proc print data=sas_data.loan_ds (obs=5);
var id;
run;'''
results = sas.submit(sas_code, results='TEXT')
print(results['LST'])
```

The `sas_code` object is defined as a Python DocString using three quotes `'''` to mark the begin and end of the DocString. In our case, the DocString holds the text for a valid block of SAS code. The syntax:

```
results = sas.submit(sas_code, results='TEXT')
```

calls the `sas.submit()` method by passing the `sas_code` object containing the SAS statements to be executed by the SAS sub-process. The `results` object receives the output, either in text or html form created by the SAS process.

In our case, we assign the output from `PROC PRINT` to the `results` object and call the `print()` method as:

```
print(results['LST'])
```

The other value for `results` object can be '`LOG`' which returns the section of the SAS log (rather than the entire log) associated with the block of submitted code. These examples are displayed in Figure 10.6, SAS.submit() Method Output from a Jupyter notebook.

```
sas_code='''option nodate nonumber;
proc print data=sas_data.loan_ds (obs=5);
var id;
run;'''
```

```
results = sas.submit(sas_code, results='TEXT')
```

```
print(results['LST'])
```

```
                                                    Obs      id

                                                      1  872482
                                                      2  872482
                                                      3  878770
                                                      4  878701
                                                      5  878693
```

Figure 10.6, SAS.submit() Method Output

You can render SAS output (the listing file) with HTML as well. This capability is illustrated in Listing 10.16, SAS Submit() Method Using HTML.

Listing 10.16, SAS Submit() Method Using HTML

```
from IPython.display import HTML
results = sas.submit(sas_code, results='HTML')
```

```
HTML(results['LST'])
```

In this example, the same `sas_code` object created in Listing 10. 15, SAS submit() Method, is passed to the `sas.submit()` method using the argument `results='HTML'`.

The HTML results from a Jupyter notebook is rendered in Figure 10.7, SAS.submit() Method with HTML Output.

```
from IPython.display import HTML
results = sas.submit(sas_code, results='HTML')
HTML(results['LST'])
```

The SAS System

Obs	id
1	872482
2	872482
3	878770
4	878701
5	878693

Figure 10.7. SAS.submit() Method with HTML Output

Write SAS Dataset to DataFrame

saspy provides the `sas.sasdata2dataframe()` method to write a SAS Dataset to a pandas Dataframe. The alias is `sas.sd2df()`. The `sas.sd2df()` method portents numerous possibilities since a SAS dataset itself is a logical reference mapped to any number of physical data sources across the organization. Depending on which products you license from SAS, a SAS dataset can refer to SAS datasets on a local file system, on a remote file system, or SAS/Access Views attached to RDBMS tables, views, files, etc.

The `sas.sd2df()` method has five parameters. They are:

1. table, the name of the SAS dataset to export to a target DataFrame

2. libref, the `libref` for the SAS dataset

3. dsopts, a Python dictionary containing the following SAS dataset options:

    a. `WHERE` clause
    b. `KEEP` list
    c. `DROP` list
    d. `OBS`
    e. `FIRSTOBS`
    f. `FORMAT`

4. method. The default is `MEMORY`. If the SAS dataset is large, you may get better performance using the `CSV` method.

5. kwargs, which indicates the `sas.sd2df()` method can accept any number of valid parameters passed to it.

Consider Listing 10.17, Export SAS dataset to DataFrame.

Listing 10.17. Export SAS Dataset to DataFrame

```
>>> import pandas as pd
>>> import saspy
>>> sas = saspy.SASsession(cfgname='winlocal')
SAS Connection established. Subprocess id is 17876

>>> ds_options = {'where'   : 'make = "Ford"',
...               'keep'    : ['msrp enginesize Cylinders
Horsepower Make'],
...               }
>>> cars_df = sas.sd2df(table='cars', libref='sashelp',
dsopts=ds_options, method='CSV')
>>> print(cars_df.shape)
(23, 5)
>>> cars_df.head()
   Make   MSRP  EngineSize  Cylinders  Horsepower
0  Ford  41475         6.8         10         310
1  Ford  34560         4.6          8         232
2  Ford  29670         4.0          6         210
3  Ford  22515         3.0          6         201
```

```
4   Ford   13270          2.0           4          130
```

The `ds_options` object uses a dictionary to pass valid SAS dataset options, in this case, a `WHERE` clause and a `KEEP` list. Notice the quoting needed for the value associated with the `where` key:

```
'make = "Ford"'
```

The outer pair of single quotes is required since this string is a value defined for the Python dictionary. The inner pair of double quotes are required since the SAS `WHERE` clause is applied to the character variable `make`.

The syntax:

```
cars_df = sas.sd2df(table='cars', libref='sashelp',
dsopts=ds_options, method='CSV')
```

calls the `sas.sd2df()` method and exports the SAS dataset `SASHELP.CARS` to the pandas DataFrame called `cars_df`. Both the SAS KEEP list and the WHERE clause are applied when the call is made, thus sub setting variables and observations as part of creating the output `cars_df` DataFrame.

Up to this point, the examples have focused on the saspy methods. The next example illustrates a simple pipeline to integrate SAS and Python processing logic in a single script.

The goal for this example is to illustrate the following steps:

1. Use SAS to perform an aggregation on an existing SAS dataset with the `sas.submit()` method. The dataset IN.LOAN_DS

2. Export the SAS dataset to a Dataframe

3. Call the pandas `.plot.bar()` method to create a histogram of credit risk grades from the resulting DataFrame.

This logic is illustrated in Listing 10.18, SAS Python Pipeline.

Listing 10.18 SAS Python Pipeline

```
>>> import pandas as pd
>>> import saspy
>>> sas = saspy.SASsession(cfgname='winlocal', results='Text')
```

```
SAS Connection established. Subprocess id is 13540

>>> sascode='''libname sas_data "c:\data";
... proc sql;
... create table grade_sum as
... select grade
...        , count(*) as grade_ct
... from sas_data.loan_ds
... group by grade;
... quit;'''
>>>
>>> run_sas = sas.submit(sascode, results='TEXT')
>>> df = sas.sd2df('grade_sum')
>>> df.head(10)
  grade  grade_ct
0     A     10086
1     B     12029
2     C      8114
3     D      5328
4     E      2857
5     F      1054
6     G       318
>>> df.plot.bar(x='grade', y='grade_ct', rot=0,
...         title='Histogram of Credit Grades')
```

In this example, the `sas_code` object is a DocString containing the SAS statements:

```
libname sas_data "c:\data";
proc sql;
   create table grade_sum as
   select grade
        , count(*) as grade_ct
   from in.loan_ds
group by grade;
quit;
```

These statements perform a `group by` aggregating the `grade` column in the `sas_data.loan_ds` SAS dataset and outputs the summarized results set to the SAS dataset `WORK.grade_sum`.

The syntax:

```
df = sas.sd2df('grade_sum')
```

creates the `df` DataFrame by calling the `sas.sd2df()` method. The argument to the call is name of the SAS dataset opened on input, in this example we specify the `WORK.grade_sum` dataset.

With the `WORK.grade_sum` dataset written as the `df` DataFrame, we can utilize any of the Python or panda methods for subsequent processing.

Next the `df` DataFrame created from the SAS dataset `WORK.grade_sum` calls the `plot.bar()` method to produce a simple histogram. The y-axis column, `grade_ct` was created as part of the `PROC SQL group by` logic. The resulting histogram is displayed in Figure 10.8, Credit Risk Grades.

```
df.plot.bar(x='grade', y='grade_ct', rot=0,
            title='Histogram of Credit Grades')
```

Figure 2

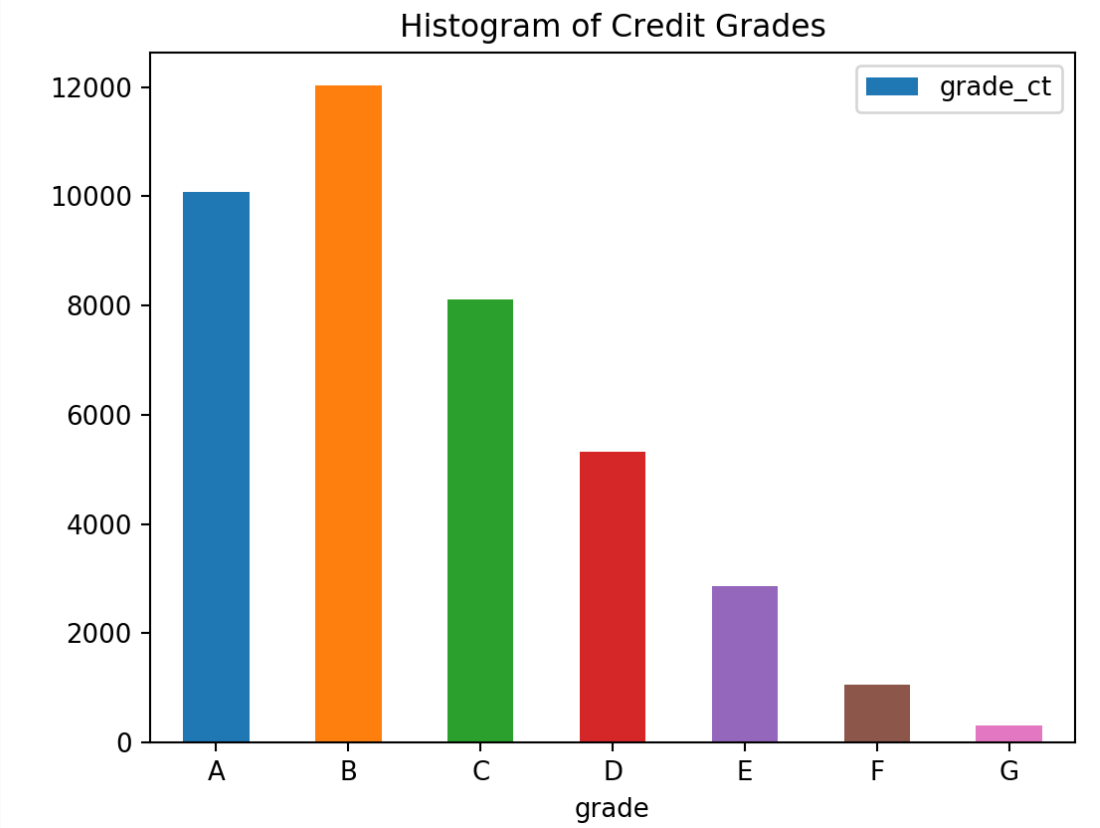

Figure 10.8. Credit Risk Grades

Passing SAS MACRO Variables to Python Objects

Another useful feature of saspy is the ability to pass SAS MACRO variable values to and from Python objects. The saspy SASsession object has two methods for passing values. The `sas.symget()` method assigns the value of a SAS Macro variable to a Python object using the syntax:

```
py_obj = sas.symget(existing_sas_macro_var)
```

Use the `sas.symput()` method to pass the value of a Python object to a SAS Macro variable value using the syntax:

```
sas.symput(sas_macro_variable, py_obj)
```

Consider Listing 10.19, sas.symget() Method.

In this example the `sas_code` object is a Python Doc string containing statements for a SAS Data Step and a SAS PROC Step. The `sas_code` Doc string is passed to the `sas.submit()` method for execution by SAS. In the Data Step code the value from the automatic SAS Macro variable `&syserr` is assigned to the Macro variable `&step1_rc`. Similarly, in the PROC step `&syserr` is assigned to the SAS Macro variable `&step2_rc`. `&syserr` is a read-only macro variable containing the return code from the Data Step and most of the PROC steps.

Listing 10.19 sas.symget() Method

```
>> import pandas as pd
>>> import saspy
>>> sas = saspy.SASsession(cfgname='winlocal')
SAS Connection established. Subprocess id is 16836

>>>
>>> sas_code='''data _null_;
... yes = exist('sashelp.class');
... if yes  = 1 then put
...    'Table Exists';
... else put
```

```
...    'Table Does NOT Exist';
... %let step1_rc = &syserr;
... run;
...
... proc summary data=sashelp.class;
... var weight;
... run;
... %let step2_rc = &syserr;
... '''
>>> run_sas = sas.submit(sas_code, results='TEXT')
>>>
>>> rc = []
>>> rc.append(sas.symget('step1_rc'))
>>> rc.append(sas.symget('step2_rc'))
>>> for i, value in enumerate(rc):
...     if value==0:
...         print ("Normal Run for Step {}.".format(i, value))
...     else:
...         print("Abnormal Run for Step {}. RC={}".format(i,
value))
...
Normal Run for Step 0.
Abnormal Run for Step 1. RC=1012
>>> print(run_sas[('LOG')])
```

Passing the SAS Macro variable return codes from each SAS execution step to a Python object is accomplished calling the `sas.symget()` method. The syntax:

```
rc = []
rc.append(sas.symget('step1_rc'))
rc.append(sas.symget('step2_rc'))
```

creates an empty list and then the `append()` method calls the `sas.symget()` method. The `sas.symget()` method call returns the codes and appends them to the `rc` list. The `for` loop itertates through the `rc` list using `IF/ELSE` logic to print the SAS return codes.

The statement:

```
print(run_sas[('LOG')])
```

returns the SAS Log fragment associated with just those statements executed by the last `sas.submit()` method. This is in contrast to the statement:

```
print(sas.saslog())
```

which returns the SAS Log for the entire session. Examining the Log we can easily see the error.

```
35          proc summary data=sashelp.class;
36          var weight;
37          run;

ERROR: Neither the PRINT option nor a valid output statement
has been given.
NOTE: The SAS System stopped processing this step because of
errors.
```

Prompting

saspy supports interactive prompting. The first type of prompting is done implicitly. For example, when running the `SASsession()` method, if any required argument to the connection method is not specified in the `SAScfg_personal.py` configuration file, then the connection process is halted, and the user is prompted for the missing argument(s).

The second form of prompting is explicit, meaning you as the application builder control where and how prompts are displayed and processed. Both the `sas.submit()` method and the `sas.saslib()` method accept an additional prompt argument. The prompt arguments are presented to the user at run time and are connected to SAS Macro variables you supply either directly in your SAS code or as arguments to the method calls.

Prompt arguments are supplied as a Python dictionary. The keys are the SAS Macro variable names and the values are the Boolean values True or False. The user is prompted for the key values and these entered values are assigned to the Macro variable. The Macro variable name is taken from the dictionary key.

At SAS execution time, the Macro variables are resolved. If you specified False as the value for the key/value pair, then the user sees the input value as it is entered into the prompt area. On the SAS Log, the user-entered values are rendered in clear text.

If you specified True as the value for the key/value pair, then the user does not see the input values; nor is the Macro variable value rendered on the SAS Log.

This feature is useful for obscuring password strings when assigning a `LIBNAME` statement to connect to a relational database. Figure 10.9, saspy Prompting illustrates this feature.

```
In [44]: sas.saslib('sqlsrvr', engine='odbc', options='user=&user pw=&pw datasrc=AdventureWorksDW',
                    prompt={'user': False, 'pw': True})
Please enter value for macro variable user Randy
Please enter value for macro variable pw ••••••••
65                                                          The SAS System
4:37 Thursday, January 3, 2019

732
733         options nosource nonotes;
736         %let user=Randy;
737         libname sqlsrvr odbc user=&user pw=&pw datasrc=AdventureWorksDW;
NOTE: Libref SQLSRVR was successfully assigned as follows:
      Engine:        ODBC
      Physical Name: AdventureWorksDW
738         options nosource nonotes;
741
742
```

Figure 10.9 saspy Prompting

In this example, the prompt= argument is a Python Dictionary:

```
{'user' : False, 'pw' : True}
```

Below the cell in the Jupyter notebook, you can observe the end-user prompt inputs. In this case, it defines two SAS Macro variables, `&user` whose value is Randy and `&pw` whose value is the password string needed by the SAS ODBC connection to the SQL Server Database, AdventureWorksDW. Notice also on the SAS Log, the `%let` assignment for `&pw` is not displayed.

Scripting saspy

Up until this point, all of the examples encountered in this chapter are executed interatively, writing its output to either the Python console or in a Jupyter notebook. Once a Python script goes from development and testing into the production, we need the ability to make the script callable. In other words, execute the script in 'batch' mode. saspy provisions the `set_batch()` method to automate Python script execution making calls into saspy. Consider Listing 10.20, Automating Python Scripts Calling saspy.

Combining some of the examples created previously in this chapter, this Python script executes in non-iteractive mode with the following logic:

1. Creates the `loandf` DataFrame using the `pd.read_csv()` method

2. Performs basic Python data wranging explained in Listing10.10 Basic Data Wrangling

3. Calls the `sas.saslib()` method to expose the SAS libref to the Python environment

4. Calls the `sas.df2sd()` method to convert the `loandf` DataFrame to the SAS dataset `sas_data.loan_ds`

5. Sets the `saspy` execution to batch with the syntax:

```
sas.set_batch(True)
```

6. Calls the saspy SAS Data Object `bar()` method to generate a histogram with the syntax:

```
out=loansas.bar('ln_stat', title="Historgram of Loan Status", label='Loan Status')
```

The `out` object is a dictionary containing two key/value pairs. The first key/value pair is the `key` 'LOG' whose `value` is the contents of the SAS Log. The second pair is the `key` 'LST' whose value is the contents of the SAS Listing. The SAS Listing holds the .html statements used to render the histogram. Since `sas.set_batch()` is set to True, this html is not render and instead is routed to a file. In this case, we are using:

```
C:\data\saspy_batch\sas_output.html
```

as the target file location for the SAS-generated html output.

7. Assigns the SAS Listing (html source statements) to the object `html_out` object.

```
html_out = out['LST']
```

We only want the html souce. Recall the `out` dictionary has two key/value pairs. If both the SAS Log statements and the SAS Listing output were written to the output file, our .html output file will be invalid.

8. Uses the Python Standard Library, `open`, `write`, and `close` calls to write the .html sources statements held in the `html_out` object to a file on the filesystem.

Listing 10.20. Automating Python Scripts Calling saspy

```
>>> #! /usr/bin/python3.5
>>> import pandas as pd
>>> import saspy
>>> url = url =
"https://raw.githubusercontent.com/RandyBetancourt/PythonForSA
SUsers/master/data/LC_Loan_Stats.csv"
>>>
... loandf = pd.read_csv(url,
...     low_memory=False,
...     usecols=(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 13, 15,
16),
...     names=('id',
...            'mem_id',
...            'ln_amt',
...            'term',
...            'rate',
...            'm_pay',
...            'grade',
...            'sub_grd',
...            'emp_len',
...            'own_rnt',
...            'income',
...            'ln_stat',
...            'purpose',
...            'state',
...            'dti'),
...     skiprows=1,
...     nrows=39786,
...     header=None)
>>> loandf['rate'] =
loandf.rate.replace('%','',regex=True).astype('float')/100
```

```
>>> loandf['term'] =
loandf['term'].str.strip('months').astype('float64')
>>>
>>> sas = saspy.SASsession(cfgname='winlocal')
SAS Connection established. Subprocess id is 1164

>>> sas.saslib('sas_data', 'BASE', 'C:\data')
25
26         libname sas_data BASE  'C:\data'  ;
NOTE: Libref SAS_DATA was successfully assigned as follows:
      Engine:        BASE
      Physical Name: C:\data
27
28
>>> loansas = sas.df2sd(loandf, table='loan_ds',
libref='sas_data')
>>>
>>> sas.set_batch(True)
>>> out=loansas.bar('ln_stat', title="Historgram of Loan
Status", label='Loan Status')
>>> html_out = out['LST']
>>> f = open('C:\\data\\saspy_batch\\sas_output.html','w')
>>>
... f.write(html_out)
49354
>>> f.close()
```

This Python script is now callable using any number of methods such as a Windows Shell, Powershell, Bash shell, or a being executed by a scheduler. On Windows the command to run the script is:

```
> python Listing10.20_saspy_set_batch.py
```

For Linux the command to run the script is:

```
$ python Listing10.20_saspy_set_batch.py
```

Summary

In this chapter we discuss the ability to integrate SAS and Python code together in a single Python execution context enabled by the saspy module. We also present

examples on bi-directional interchange of data between a pandas DataFrame and SAS dataset as well as exchanging SAS and Python variable values. The saspy module offers a compelling set of methods to integrate SAS and Python processing logic both interactively and through scripting methods.