CHAPTER 9

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:

- Start and connect to a local or remote SAS session
- Exchange data between pandas DataFrames and SAS dataset
- Integrate both SAS and Python program logic within a single execution context

To get started, you install and configure the saspy module. 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 9.1, SASPy Install on Windows displays the output from a Windows terminal for installing saspy.

Listing 9.1. 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
```

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 such 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 on Linux, you use the STDIO access method. The detailed instructions are at:

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

Listing 9.2, Locate SASPy.sascfg Configuration File illustrates the syntax needed to locate the saspy configuration file.

Listing 9.2. 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 a new version of saspy is installed. 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 9.3, Finding the PYTHONPATH Search Paths.

Listing 9.3. 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 we are running both a Python and a SAS session are on the same Windows machine. Calling SASPy requires the IOM access method be appropriately defined in the `personal_saspy.cfg` 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 altered in the `sascfg_personal.py` configuration file to:

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

The following four Java jar files are defined in a classpath variable in the `sascfg_personal.py` configuration file:

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

These jar files are a part of the SAS Deployment Manager. 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. In our case this jar file is located at:

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

The last change we need is an update to the dictionary values for the `winlocal` object definition in the `sascfg_personal.py` configuration file as:

```
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 Base SAS software. Also notice the path filename uses a double back-slash to ‘escape’ the backslash needed by the Windows path names.

SASPy Examples

With the configuration for saspy complete we can begin. 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 9.4, Start SASPy Session to integrate a Python and SAS session together.

Listing 9.4. 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` object. 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, we have 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 values to indicate how tabular output returned from the `SASsession` object 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 `SASsession` argument is autoexec. In some cases, it is useful to execute a series of SAS statements when the `SASsession` object is called. This feature is illustrated in Listing 9.4, Start SASPy with Autoexec Processing.

Listing 9.5. 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 SAS 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 build the `loandf` DataFrame which is sourced from the Lending Club loan statistics described at:

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

The data consist of anonymized loan performance data from Lending Club which offers personal loans to individuals. We begin by creating the `loandf` DataFrame illustrated in Listing 9.6, Build loandf DataFrame.

Listing 9.6. 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 9.7, loandf Initial Attributes returns basic information about the columns and values.

Listing 9.7. 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 9.1, Attributes for Character Value Columns.

The `loandf.info()` method shows the `rate` column has a datatype of `object` indicating these 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’.

```
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 9.1. Attributes for Character Value Columns

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

1. Strip the percent sign
2. Map the datatype from character to numeric
3. Divide 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’

2. Map the datatype from character to numeric

Both modifications are shown in Listing 9.10, Basic Data Wrangling.

Listing 9.10 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.replace('%','',regex=True).astype('float')/100
```

calls the `pd.replace()` method used to dynamically replace values. In this case, the first argument is `to_replace='%'`, the second argument is `value=''`, (since there are no spaces between the quote marks, this becomes a null value). The `regex='True'` argument indicates the `to_replace=` argument is a string value.

The `.astype()` attribute maps the `rate` column's datatype from object (strings) to a float (decimal value). The value is then divided by 100.

Chaining the `.describe()` method to the `rate` column returns basic statistics for the values.

Similarly, the syntax:

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

performs a similar operation on the `loandr['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.

Write DataFrame to SAS Dataset

With the `loandf` DataFrame shaped appropriately, we can write the DataFrame as a SAS data set. SASPy provides the `sas.df2sd()` method to write a DataFrame to a SAS dataset. The SAS dataset can either by a temporary dataset written to the current WORK library or a permanent dataset on any location of the filesystem. This feature is illustrated in Listing 9.11, Write a DataFrame as a SAS Dataset.

Listing 9.11, Write a DataFrame as a SAS Dataset

```
>>> 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
27
28
>>> loansas = sas.df2sd(loandf, table='loan_ds',
libref='sas_data')
>>> 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

>>> print(sas.saslog())

                        The SAS System
                                16:01 Monday, November 26, 2018

NOTE: Copyright (c) 2016 by SAS Institute Inc., Cary, NC, USA.
NOTE: SAS (r) Proprietary Software 9.4 (TS1M5)
NOTE: This session is executing on the X64_10PRO  platform.

NOTE: Updated analytical products:

      SAS/STAT 14.3

NOTE: Additional host information:

 X64_10PRO WIN 10.0.17134  Workstation

NOTE: SAS Initialization used (Total process time):
      real time           0.01 seconds
      cpu time            0.00 seconds

1          ;*';*";*/;
2          options svgtitle='svgtitle'; options
validvarname=any pagesize=max nosyntaxcheck; ods graphics on;
```

The syntax:

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

calls the `sas.saslib()` method from `saspy` to expose a SAS library to the current Python session. This method accepts four arguments. They are:

1. Libref, in this case `sas_data`
2. engine, or access method, in this case the default `BASE` engine
3. path, the path to the BASE data library, in this case, `C:\data`
4. options which can be SAS engine or engine supervisor options. In this case, we are not supplying options.

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

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

and sends this statement for processing to the attached SAS sub-process.

In order to write the `loandf` DataFrame as a SAS dataset, call the `sas.df2sd()` method. In this example, the syntax:

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

creates the `loansas` SASdata object and calls the `sas.df2sd()` method to create a new SAS dataset from the `loandf` DataFrame. The `loansas` object becomes a Python reference to the newly created SAS dataset, in this example, the permanent SAS dataset, `sas_data.loan_ds`.

In other words, the current Python execution context has the `loandf` DataFrame defined. In addition, the `loansas` object is defined which is mapped to the permanent SAS dataset `sas_data.loan_ds` created from the `loandf` DataFrame.

The `sas.df2sd()` method reads a DataFrame and writes it as a SAS dataset. This method has five arguments. They are:

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

2. `table=` argument which is the name for the output SAS dataset.

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`.

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`.

The syntax:

```
loansas.columnInfo()
```

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

The syntax:

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

returns the Log for the entire SAS sub-process which is truncated here.

The `loansas` SAS Data Object has several available methods. Some of these methods are displayed in Figure 9.2, SAS Data Object methods.

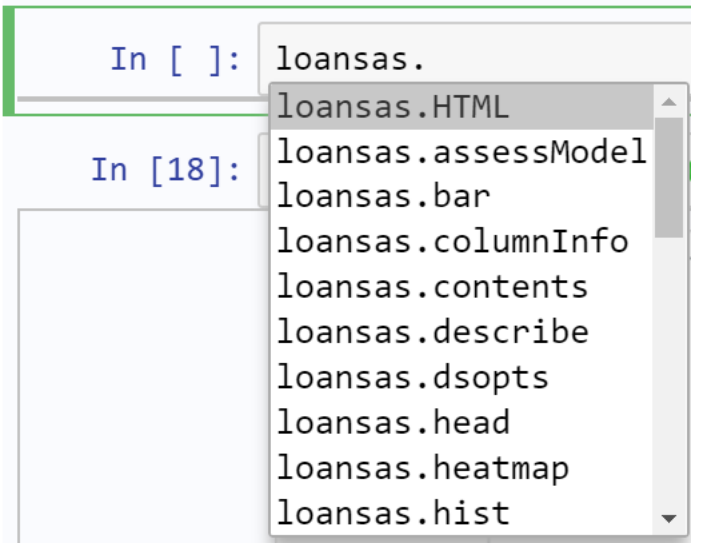

Figure 9.2. SAS Data Object methods

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.

Figure 9.3, Heatmap for ln_stat Column illustrates calling the `.bar()` method to render a histogram for the loan status variable (`ln_stat`). For this example to work, you need to execute the code in Listing 9.12, Loan Status Histogram in a Jupyter notebook. On Windows, from a terminal session, enter the command:

```
python -m notebook
```

to launch a Jupyter notebook. Copy the program from Listing 9.12 into a cell and press the `>|Run` button.

Listing 9.12 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')
```

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

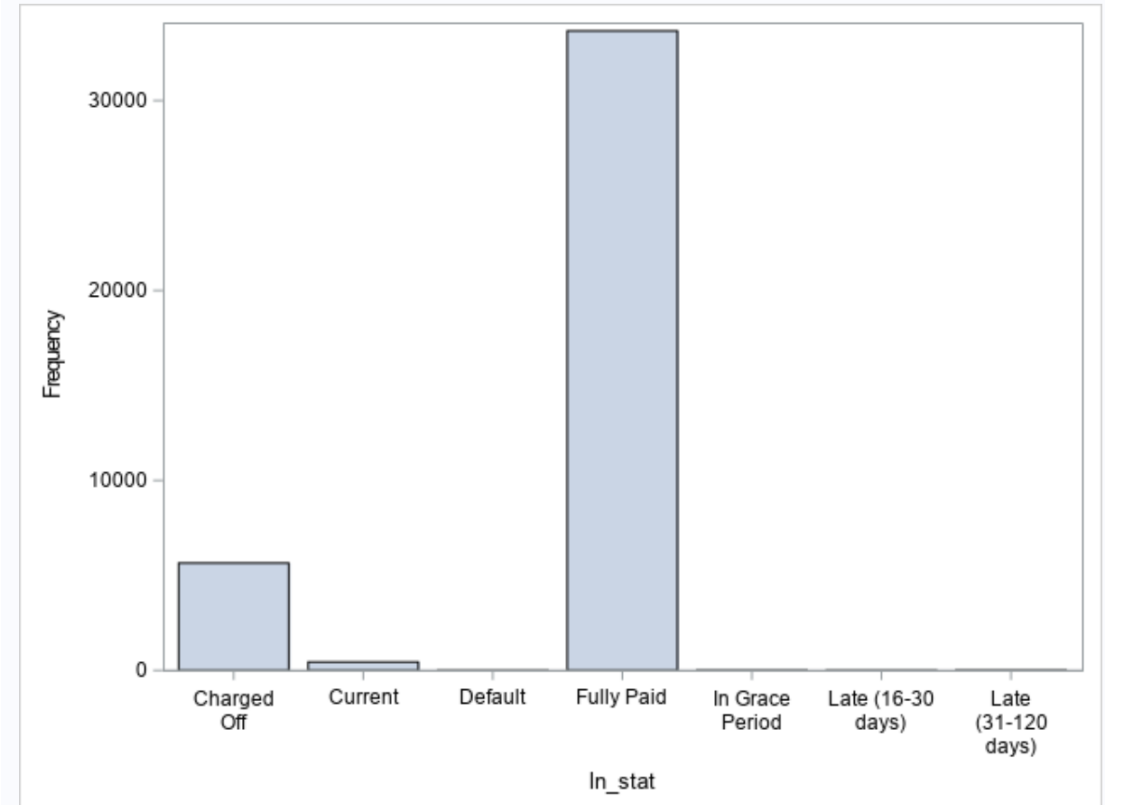

Figure 9.3. Histogram for ln_stat Column

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 12.6%.

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 9.13, Teach Me SAS, illustrates this capability.

Listing 9.13. Teach Me SAS

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

Figure 9.4, 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 9.4. Teach_me_SAS Attribute

Execute SAS Code

By far, the most powerful features of the `saspy.SASsession` object is the `.submit()` attribute. This feature enables you to submit any arbitrary block of SAS code and assign the results to a Python object. Consider Listing 9.14, SAS submit() Method.

Listing 9.14. 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 for 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 log output (rather than the entire log output) associated with the block of code submitted to SAS. These examples are displayed in Figure 9.5, 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 9.5, SAS.submit() Method Output

You can render SAS output (the listing file) with HTML as well. This capability is illustrated in Listing 9.15, SAS Submit() Method Using HTML.

Listing 9.15, 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 9. 14, 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 9.6, 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 9.6. SAS.submit() Method with HTML Output

Write SAS Dataset to DataFrame

SASPy provides the `sas.sd2df()` method to write a SAS Dataset to a Dataframe. The pandas IO Tools library does not provide a method to write SAS dataset to DataFrames. As of this writing, the saspy module is the only Python library to provide this capability.

The goal for this example is to illustrate the use of SAS to perform an aggregation, using the `sas.submit()` method followed by a call to the `pd.plot.bar()` method creating a histogram from the resulting DataFrame. One can easily imagine a Python-driven pipeline incorporating Python and SAS program logic together to achieve the desired outcome.

The ability to create a SAS dataset from an existing dataframe is illustrated in Listing 9.16, SAS Dataset to DataFrame.

Listing 9.16 SAS Dataset to DataFrame

```
>>> import pandas as pd
>>> import saspy
>>> sas = saspy.SASsession(cfgname='winlocal', results='Text')
SAS Connection established. Subprocess id is 13540

>>> sascode='''libname in "c:\data";
... proc sql;
... create table grade_sum as
... select grade
...        , count(*) as grade_ct
... from in.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
```

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

```
proc sql;
 create table grade_sum as
 select grade
        , count(*) as grade_ct
 from in.loan_ds
 group by grade;
```

used to perform a `group by` on the `grade` column and output the 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 parameter to the call is name of the SAS dataset opened on input, in this example, it is `WORK.grade_sum`.

This open ups a large number of possibilities here since a SAS dataset is a logical reference which can map to any number of physical data sources across the organization. Depending on which products you license from SAS, a SAS data set can refer to SAS datasets on a local filesystem, on a remote filesystem, or SAS/Access Views attached to RDBMS tables, views, files, etc.

With the `WORK.grade_sum` dataset written as the `df` DataFrame, we can utilize any of the Python or panda method for further processing. For example, consider Listing 9.17, Histogram of Credit Risk Grades.

Listing 9.17. Histogram of Credit Risk Grades

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

The `df` DataFrame created from the SAS dataset `WORK.grade_sum` in the previous step calls the `plot.bar()` method to produce a simple histogram. The results are displayed in Figure 9.7, Credit Risk Grades. This example was created in a Jupyter Notebook.

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

Figure 2