Querying WRDS Data using Python

Learn how to use Python to view dataset metadata and query WRDS data

Introduction

Now that you've learned how to connect to WRDS using the Python **wrds** module, you're ready to begin querying WRDS data. This document will begin by examining the **wrds** module, presenting its included methods and covering the basics of its use. Afterwards, it will cover how to explore WRDS metadata to find the particular WRDS data that you're looking for, and then how to use that information to perform actual research queries against that data.

Additional Resources:

- OLS Regression in Python

Top of Section

Using the wrds Module

All Python programs intending to access WRDS data must include the following two lines at the beginning:

```
import wrds
db = wrds.Connection()
```

With the first line importing the Python **wrds** module for use in your program, and the second establishing a new connection to the WRDS database backend and saving that connection as `db`.

This connection ideally requires that you have set up your `.pgpass` file as covered in the previous documentation. If you have not yet done this, the instructions are here:

- PYTHON: From Your Computer
- PYTHON: On the WRDS Cloud

Otherwise you will be prompted for your WRDS username and password each time you connect to WRDS in this fashion.

Note: Class accounts and IPAuth / Daypass accounts are not permitted to access WRDS in this manner and will receive an error if trying this connection. You must have your own, dedicated WRDS account in order to access WRDS from MATLAB.

Methods in the wrds Module

The **wrds** module supports several different methods of accessing WRDS data. The best way to learn about them is using the inline documentation available within a Python console (available in Spyder or Jupyter notebooks on your computer, or via an interactive job on the WRDS Cloud). You can see what functions are available by entering `db` and then pressing Tab key to expand the list, as follows:

To see the list of available db functions

```
db.close()
db.connection()
db.describe_table()
db.get_table()
db.list_tables()
db.raw_sql()
db.get_row_count()
db.list_libraries()
```

NOTE: There are a few other methods beyond the above that you'll see when looking at the contextual help list. Those methods are either internal methods and do not provide any functionality on their own, or are part of the setup tools discussed earlier in this documentation, such as `create_pgpass_file()`.

For an online description of each, use the `help()` function as follows:

To use the help function

```
help(db.get_table)
help(db.raw_sql)
```

Pandas and Numpy

No matter which **wrds** method you use, your data results are always returned as a Pandas DataFrame. Pandas is a popular Python module that, together with NumPy, forms the basis for most numeric data manipulation in Python. The pandas module provides powerful data manipulation capabilities at a granular degree of control over your output.

In all examples provided in this document, the returned results variable **data** is a Pandas DataFrame.

If you are unfamiliar with pandas or numpy, see the following resources:

- 10 Minutes to Pandas
- Pandas: Essential Basic Functionality
- Numpy: Quickstart Tutorial

Limiting the Number of Records Returned

When working with large data sources, it is important to begin your research with small subsets of the data you eventually want to query. Limiting the number of returned records (also called observations) is essential while developing your code, as queries that involve large date ranges or query a large number of variables (column fields) could take a long time and generate large output files.

Generally, until you are sure that you're getting exactly the data you're looking for, you should limit the number of observations returned to a sensible maximum such as 10 or 100. Remember, much of the data provided at WRDS is huge! It is highly recommended to develop your code using such a limit, then simply remove that limit when you are ready to run your final code.

IMPORTANT: This is especially important if you are running Python locally from your computer, as the returned query output data is downloaded from WRDS to your computer. Even if you have a fast computer, a slow or intermittent Internet connection could cripple your research if you don't perform your queries carefully.

There are two ways of limiting the number of records (to say 10), depending on which method you use to get data:

- `get_table()` - for this method, you would use `obs=10`
- `raw_sql()` - for this method, you would use `LIMIT 10`

Here is an example of each:

```
db.get_table('djones', 'djdaily', columns=['date', 'dji'], obs=10)


db.raw_sql('SELECT date,dji FROM djones.djdaily LIMIT 10;', date_cols=['date'])
```

Each of these will be explored in more detail below.

Top of Section

Querying the Dataset Structure (Metadata)

Data at WRDS is organized in a hierarchical manner by vendor (e.g. **crsp**), referred to at the top-level as *libraries*. Each library contains a number of component tables or *datasets* (e.g. **dsf**) which contain the actual data in tabular format, with column headers called *variables* (such as *date*, *askhi*, *bidlo*, etc).

You can analyze the structure of the data through its metadata using the **wrds** module, as outlined in the following steps:

1. List all available *libraries* at WRDS using `list_libraries()`
2. Select a library to work with, and list all available *datasets* within that library using `list_tables()`
3. Select a dataset, and list all available *variables* (column headers) within that dataset using `describe_table()`

NOTE: When referencing library and dataset names, you must use all *lowercase*.

Alternatively, a comprehensive list of all WRDS libraries is available at the Dataset List. This resource provides a listing of each library, their component datasets and variables, as well as a tabular database preview feature, and is helpful in establishing the structure of the data you're looking for in an easy manner from a Web browser.

1. Determine the *libraries* available at WRDS:

```
sorted(db.list_libraries())
```

This will list all libraries available at WRDS in alphabetical order. Though all libraries will be shown, you must have a valid, current subscription for a library in order to access it via Python, just as with SAS or any other supported programming language at WRDS. You will receive an error message indicating this if you attempt to query a table to which your institution does not have access.

2. To determine the *datasets* within a given library:

```
db.list_tables(library="library")
```

Where 'library; is a dataset, such as **crsp** or **comp,** as returned from step 1 above.

3. To determine the column headers (*variables*) within a given dataset:

```
db.describe_table(library="library", table="table")
```

Where 'library' is a dataset such as **crsp** as returned from #1 above and 'table' is a component database within that library, such as **msf**, as returned from query #2 above. Remember that both the library and the dataset are case-sensitive, and must be all-lowercase.

Alternatively, a comprehensive list of all WRDS libraries is available via the WRDS Dataset List. This online resource provides a listing of each library, their component datasets and variables, as well as a tabular database preview feature, and is helpful in establishing the structure of the data you're looking for in an easy, web-friendly manner.

By examining the metadata available to us -- the structure of the data -- we've determined how to reference the data we're researching, and what variables are available within that data. We can now perform our actual research, creating *data queries*, which are explored in depth in the next section.

Top of Section

Querying WRDS Data

Now that you know how to query the metadata and understand the structure of the data, you are ready to query WRDS data directly. The **wrds** module provides several methods that are useful in gathering data:

- `get_table()` - fetches data by matching library and dataset, with the ability to filter using different parameters. This is the easiest method of accessing data.
- `raw_sql()` - executes a SQL query against the specified library and dataset, allowing for highly-granular data queries.
- `get_row_count()` - returns the number of rows in a given dataset.

Each of these is discussed below.

Using get_table()

The method `get_table()` accepts the following parameters:

- **library** - the library to query
- **table** - the dataset to query
- **columns** - the columns (variables) to include in the query (optional)
- **obs** - the number of observations (rows) to return (optional)
- **offset** - the starting point of for the query (optional)

For additional parameters, and further explanation of each, use the built-in help: `help(db.get_table)`

Here's an example that returns the first 10 rows of only the **date** and **dji** columns from the Dow Jones Index:

```
data = db.get_table(library='djones', table='djdaily', columns=['date', 'dji'], obs=10)
data
```

NOTE: The **library** and **table** parameters are required, and are also positional. Therefore, should you chose, you may omit the label for these two parameters as long as you supply them in order.

The following example illustrates this:

```
data = db.get_table('djones', 'djdaily', columns=['date', 'dji'], obs=10)
data
```

Using raw_sql()

The method `raw_sql()` accepts the following parameters:

- **sql** - the SQL string to query
- **date_cols** - a list or dict of column names to parse as date (optional)

- **index_col** - a string or list of column(s) to set as index(es) (optional)

For additional parameters, and further explanation of each, use the built-in help: `help(db.raw_sql)`

Here's an example that does the same thing as `get_table()` example above, but uses SQL to select the data instead:

```
data = db.raw_sql('SELECT date, dji FROM djones.djdaily LIMIT 10', date_cols=['date'])
data
```

The `raw_sql()` method is by far the most useful and popular of the **wrds** module's methods, allowing powerful and granular control over your data processing. Writing SQL for use with this method of fairly straightforward. All data queries are constructed in SQL the following generic manner:

`select columns from library.dataset where column1 = value`

Notice the *dot notation* for the library and dataset. Unlike the other **wrds** methods, where *library* and *table* are specified separately, SQL queries instead use the two together to identify the data location. So, for example, a data query for the *dataset* **msf** within the *library* **crsp** would use the syntax **crsp.msf**, and the same goes for **djones.djdaily**.

You'll likely be doing most of your work using the `raw_sql()` method.

Using get_row_count()

The method `get_row_count()` only accepts the following two parameters:

- **library** - the library to query
- **table** - the dataset to query

For additional information, use the built-in help: **help(db.get_row_count)**

Here is an example that returns the row count for the Dow Jones Index:

```
data = db.get_row_count('djones', 'djdaily')
data
```

Top of Section

Joining Data from Separate Datasets

Data from separate datasets can be joined and analyzed together. The following example will join the Compustat Fundamentals data set (**comp.funda**) with Compustat's pricing dataset (**comp.secm**), and then query for total assets and liabilities mixed with monthly close price and shares outstanding.

To join and query two Compustat datasets:

```
db.raw_sql("""
    SELECT a.gvkey, a.datadate, a.tic, a.conm, a.at, a.lt, b.prccm, b.cshoq
    FROM comp.funda a
    JOIN comp.secm b ON a.gvkey = b.gvkey AND a.datadate = b.datadate
    WHERE a.tic = 'IBM' AND a.datafmt = 'STD' AND a.consol = 'C' AND a.indfmt = 'INDL'
""")
```

The code joins both datasets using a common **gvkey** identifier and date, querying IBM with a frequency of one year, resulting in a result of 55 observations (as of 2017). Running joined queries between large datasets can require large amounts of memory and execution time. It is recommended you limit the scope of your queries to reasonable sizes when performing joins.

NOTE: For this example, you would need an active subscription to both datasets.

Top of Section

Managing your Connections

WRDS users are permitted up to 5 simultaneous connections to our Postgres data backend. For Python users, that means you may use `wrds.Connection()` up to five times before being denied additional connections.

The best way to manage this, is to properly close out your connection to WRDS once you are done with it. With the Python **wrds** module, you simply use the `close()` method like so:

```python
import wrds
db = wrds.Connection()
data = db.raw_sql("SELECT * FROM djones.djdaily")
db.close()
```

Using the above, we've connected to WRDS, downloaded the data we need, and disconnected. We can continue to use the results from our query (**data**) even after disconnecting and the previous connection now longer counts against us when we go to connect again.

You should always disconnect using `close()` when you:

- Exit your Python environment
- Finish running your program
- Complete your data query download step, and want to move onto another.

WRDS is currently working on a facility to allow you to cancel any connections you may have accidentally left open, but in the meantime, if you are getting an error message indicate that you have too many connections open to WRDS, try to close any existing connection you may have with the `close()` method, wait for your existing connections to timeout and close on their own, or reach out to us at WRDS Support for assistance.

Top of Section

Passing Parameters to SQL

The `raw_sql()` method now also supports parameterized SQL, allowing you to pass variables or lists from elsewhere in your Python code to your SQL statement. This is great for large lists of company codes or identifiers, or an array of specific trading days. Here is an example where a dictionary of tickers is passed through to a raw_sql() SQL statement:

```python
params = {"tickers": ("0015B", "0030B", "0032A", "0033A", "0038A")}
data = db.raw_sql(
    "SELECT datadate, gvkey, cusip FROM comp.funda WHERE tic IN %(tickers)s",
    params=params,
)
```

This allows for a great deal of flexibility in terms of your SQL queries. Common use cases might include building out a list of tickers, CUSIPS, etc programmatically or from an external file; re-using the same code list over multiple queries that adjust other parameters, such as date range; or matching based on specified trading days.

This functionality uses Panda's native SQL capabilities, and thus requires SQL that conforms to a slightly different standard than the generalized SQL you otherwise use with `raw_sql()`. Specifically, it uses the Python DB-API 2.0 specification, which conforms to PEP 249.

For more information, please se one of the following links:

Psycopg: Passing Parameters to SQL Queries

Pandas: read_sql_query()

Python: PEP 249

This feature was added in version 3.0.6 of the Python **wrds** module.

Top of Section

Next Steps

Now that you have learned how to how to use Python to view metadata and query data, take a look at a series of examples in our Example Python Data Workflow.

Additional Resources:

- Use SAS in Python: SASPy - Introduction 1
- Use SAS in Python: SASPy - Introduction 2
- Use SAS in Python: SASPy - Introduction 3
- Use SAS in Python: SASPy - Introduction 4

Top of Section

Top

Table of Contents

Related Information

» PYTHON: From Your Computer

» PYTHON: On the WRDS Cloud

» Graphing Data

» Example Workflow