Writing large data sets to Excel with Python and pandas

ON THIS PAGE

After using the API to retrieve a large data set from your Zendesk product, you might want to move the data set to a Microsoft Excel worksheet to more easily view and analyze the data. This tutorial teaches you how to munge the API data and write it to Excel. Data munging is the process of converting, or mapping, data from one format to another to be able to use it in another tool.

The tutorial uses Python 3 and pandas , a data analysis toolkit for Python that's widely used in the scientific and business communities. To install pandas, see the instructions on the pandas website.

You'll also need OpenPyXL , a third-party library that pandas uses for reading and writing Excel files. To install it, see the instructions on the OpenPyXL website.

Sections covered in the tutorial:

- Get the data from the API
- Munge the data
- Write the data to Excel
- Code complete

A note about the code examples : Some lines of code in the examples may wrap to the next line because of the article's page width. When copying the examples in this tutorial, ignore the line wrapping. Line breaks matter in Python.

Disclaimer: Zendesk provides this article for instructional purposes only. Zendesk does not support or guarantee the code. Zendesk also can't provide support for third-party technologies such as Python and pandas.

Get the data from the API

Start with the question you want answered. Determine what data you need to answer it, then get the data from your Zendesk product using the API. To learn how, see Getting large data sets with the Zendesk API and Python .

For all the possible data you can retrieve from your Zendesk product, see the "JSON Format" tables in the API reference . Most APIs have a "List" endpoint for getting multiple records.

Let's say you retrieved all the posts in a community topic and sideloaded the users who wrote the posts. The resulting data structure in Python usually mirrors the JSON data returned by the API. In this case, it would consist of a Python dictionary containing one list of posts and one list of users. Each item in the lists would consist of a dictionary of properties. Example:

```
{
    'posts': [
        {
            'id': 123,
            'title': 'My printer is on fire!',
            ...
        },
        ...
    ],
    'users': [
        {
            'id': 321,
            'name': 'jbloe',
            ...
        },
        ...
    ]
}
```

Also assume that you serialized the data structure in a file named **my_serialized_data** . Serializing a data structure means translating it into a format that can be stored and then reconstituted later in the same environment. Serializing your data is far more efficient than calling the API every time you need the data. Getting a large data set can involve hundreds if not thousands of API requests.

In Python, you can use the built-in pickle module to serialize and deserialize complex data structures such as your dictionary of posts and users. See Serialize the data to reuse it in the tutorial mentioned above.

In your script, the first step is to get the API data and assign it to a variable. Create a file named **write_posts.py** and paste the following code in it:

```
import pandas as pd

topic = pd.read_pickle('my_serialized_data')
```

The serialized data is read from the **my_serialized_data** file, reconstituted as a dictionary, and assigned to a variable named **topic** .

Tip : The code assumes the pickle file is in the same folder as the script. If it's in another folder, import the built-in `os` library and use it to specify the file path:

```
import os
import pandas as pd

topic = pd.read_pickle(os.path.join('..', 'pickle_jar', 'my_serialized_data'))
```

Munge the data

An Excel worksheet consists of a 2-dimensional table of rows and columns. Unfortunately, your data isn't in a neat 2-dimensional structure that can be easily written to Excel. It's in a dictionary consisting of a list of posts and a list of users. Each item in the lists consists of a dictionary of properties. See Get the data from the API above for the structure.

The data also includes a lot of extra information you don't want in your Excel file. For example, each record contains all the attributes listed in the Posts API doc . You want only the following data about each post:

- post id
- title of the post
- date created
- name of the person who created it

This section teaches you how to munge your complex dictionary into a 2-dimensional data structure with 4 columns. It uses pandas DataFrames to do the job. A DataFrame is a fundamental, 2-dimensional data structure in pandas. You can think of it as a spreadsheet or a SQL table.

Topics covered:

- Create the DataFrames
- Convert the ISO 8601 date strings

- Merge the DataFrames
- Clean up after the merge

The section only scratches the surface of how you can use pandas to munge data. To learn more, see the pandas docs .

Create the DataFrames

Add the following lines to your script to convert the posts and users lists in your **topic** dictionary into 2 DataFrames:

```
posts_df = pd.DataFrame(topic['posts'],
                        columns=['id', 'title', 'created_at', 'author_id'])
users_df = pd.DataFrame(topic['users'], columns=['id', 'name'])
```

The `DataFrame()` method in each statement takes the list data from the **topic** dictionary as its first argument. The `columns` parameter specifies the keys of the dictionaries in the list to include as columns in the resulting DataFrame. For more information, see DataFrame in the pandas docs.

You can view the DataFrames created in memory by adding the following temporary print statements:

```
print(posts_df)
print(users_df)
```

Save the file. In your command line tool, navigate to the folder with the script and run the following command:

```
python3 write_posts.py
```

Your data should be written to the console. Additional columns wrap if they don't fit the display width. If you're satisfied everything is working as expected, delete the temporary print statements.

Convert the ISO 8601 date strings

Sometimes you have to clean up data for analysis. For example, the 'created_at' dates returned by the API are ISO 8601 format strings such as `'2015-08-13T16:38:39Z'` . OpenPyXL, the library pandas uses to work with Excel files, writes these dates to Excel as strings. Once they're in Excel, however, you can't easily reformat the strings as dates.

OpenPyXL does write Python date objects as custom formatted dates in Excel. So if you need to use the date format in Excel for your analysis, you could convert each string in the 'created_at' column into a Python date object. You can use the `apply()` method of the column object to specify a Python lambda expression that modifies the data in each row of the column. A lambda expression is a one-line mini function.

Modify your script as follows to import the build-in `dateutil.parser` library and then use it to convert the 'created_at' strings. Ignore the line break caused by the right margin. The statement should be on a single line.

```
import dateutil.parser

... # rest of script

posts_df['created_at'] = posts_df['created_at'].apply(lambda x: dateutil.parser.
parse(x).date())
```

The conversion statement works as follows:

- The expression `posts_df['created_at']` selects the column in the DataFrame
- The lambda expression in the `apply()` method converts each ISO 8601 string in the column into a Python `date` object. It basically says, "For the data in each row, which I'll call `x` , make the following change to `x` "
- The **dateutil** parser converts the ISO 8601 date string into a `datetime` object. Next, the `date()` method converts the `datetime` into a `date` object
- The resulting column is assigned back to the 'created_at' column in **posts_df**

Merge the DataFrames

The **posts_df** DataFrame contains most of the data you want to write to Excel, including the 'id', 'title', and 'created_at' columns. However, the 'author_id' column only lists user ids, not actual user names. The associated user names are contained in **users_df** , which was derived from sideloading users with the API.

In the same way you can join two tables in SQL using a common key in both tables, you can merge two DataFrames using a common key in both DataFrames. The common key in your DataFrames is the user id, which is named 'author_id' in **posts_df** and 'id' in **users_df** .

Add the following statement to merge the DataFrames:

```
merged_df = pd.merge(posts_df, users_df, how='left', left_on='author_id',
                right_on='id')
```

The `merge()` method joins the two DataFrames using user ids as the common key (`left_on='author_id', right_on='id'`).

A 'left' merge (`how='left'`) is the same as a left join in SQL. It returns all the rows from the left DataFrame, **posts_df** , including rows that don't have matching key values in **users_df** .

Because you're using **users_df** as a lookup table, make sure it doesn't have any duplicate records. The **users_df** DataFrame is made up of sideloaded data from the API. Sideloaded data may contain duplicate records because the same record may be saved many times during pagination. To remove duplicate records, you can modify the **users_df** variable declaration as follows (highlighted):

```
users_df = pd.DataFrame(topic['users'],
             columns=['id', 'name']).drop_duplicates(subset=['id'])
```

The `drop_duplicates()` method looks at the values in the DataFrame's 'id' column and deletes any row with a duplicate id.

Clean up after the merge

The two original DataFrames have a column named 'id'. You can't have two columns with the same name in the merged DataFrame, so pandas adds a '_x' and a '_y' suffix to the overlapping column names. For example, if you added the following temporary statement to the script:

```
print(list(merged_df))
```

and then ran the script, the following list of column names would be printed in the console:

```
['id_x', 'title', 'created_at', 'author_id', 'id_y', 'name']
```

Because DataFrame column names are used as column headings in an Excel workbook, you decide to rename the 'id_x' column to 'post_id' as follows:

```
merged_df.rename(columns={'id_x': 'post_id'}, inplace=True)
```

The merged DataFrame also includes the 'id_y' and 'author_id' columns that you don't want in your Excel file. You can drop the columns with the following statement:

```
merged_df.drop(['id_y', 'author_id'], axis=1, inplace=True)
```

The `axis=1` argument specifies that you're referring to a column, not a row.

Write the data to Excel

After you're done munging the data, you can write the data to Excel as follows:

```
merged_df.to_excel('topic_posts.xlsx', index=False)
```

```
print('Spreadsheet saved.')
```

The `index=False` argument prevents the `to_excel()` method from creating labels for the rows in Excel.

Save the file. In your command line tool, navigate to the folder with the script and run the following command:

```
python3 write_posts.py
```

Check for the **topic_posts.xlsx** file in the folder containing your script and open it in Excel. It should contain your post data.

Code complete

```
import dateutil.parser

import pandas as pd

topic = pd.read_pickle('my_serialized_data')
posts_df = pd.DataFrame(topic['posts'], columns=['id', 'title', 'created_at', 'author_id'])
users_df = pd.DataFrame(topic['users'], columns=['id', 'name']).drop_duplicates(subset=['id'])

posts_df['created_at'] = posts_df['created_at'].apply(lambda x: dateutil.parser.parse(x).date())

merged_df = pd.merge(posts_df, users_df, how='left', left_on='author_id', right_on='id')
merged_df.rename(columns={'id_x': 'post_id'}, inplace=True)
merged_df.drop(['id_y', 'author_id'], axis=1, inplace=True)

merged_df.to_excel('topic_posts.xlsx', index=False)
print('Spreadsheet saved.')
```

Join our developer community

Forum Blog Slack