Build your own Pandas Cub

This repository contains a detailed project that teaches you how to build your own Python data analysis library.

Target Student

This project is targeted towards those who understand the fundamentals of Python and wish to build their own data analysis library similar to Pandas from scratch.

Pre-Requisites

- Intermediate knowledge of Python
- Helpful to have heard about special methods
- Helpful to have used NumPy and Pandas before

Objectives

Most data scientists who use Python rely on Pandas. In this assignment, we will build Pandas Cub, a library that implements many of the most common and useful methods found in Pandas. Pandas Cub will:

- Have a DataFrame class with data stored in NumPy arrays
- Use special methods defined in the Python data model
- Have a nicely formatted display of the DataFrame in the notebook
- Select subsets of data with the brackets operator
- Implement aggregation methods - sum, min, max, mean, median, etc...
- Implement non-aggregation methods such as isna, unique, rename, drop
- Group by one or two columns
- Have methods specific to string columns

Setting up the Development Environment

I recommend creating a new environment using the conda package manager. If you do not have conda, you can download it here along with the entire Anaconda distribution. Choose Python 3. When beginning development on a new library, it's a good idea to use a completely separate environment to write your code.

Create the environment with the `environment.yml` file

Conda allows you to automate the environment creation by creating an `environment.yml` file. The contents of the file are minimal and are displayed below.

```
name: pandas_cub
dependencies:
- python=3.6
- pandas
```

```
  - jupyter
  - pytest
```

This file will be used to create a new environment named `pandas_cub`. It will install Python 3.6 in a completely separate folder in your file system along with pandas, jupyter, and pytest. There will actually be many more packages installed as those libraries have dependencies of their own.

Visit this page for more information on conda environments.

Command to create new environment

In the top level directory of this repository, where the `environment.yml` file is located, run the following command from your command line.

```
conda env create -f environment.yml
```

The above command will take some time to complete. Once it completes, the environment will be created.

List the environments

Run the command `conda env list` to show all the environments you have. There will be a `*` next to the active environment, which will likely be `base`, the default environment that everyone starts in.

Activate the pandas_cub environment

Creating the environment does not mean it is active. You must activate in order to use it. Use the following command to activate it.

```
conda activate pandas_cub
```

You should see `pandas_cub` in parentheses preceding your command prompt. You can run the command `conda env list` to confirm that the `*` has moved to `pandas_cub`.

Deactivate environment

You should only use the `pandas_cub` environment to develop this library. When you are done with this session, run the command `conda deactivate` to return to your default conda environment.

Test-Driven Development with pytest

The completion of each part of this project is predicated upon passing the tests written in the `test_dataframe.py` module inside the `tests` folder.

We will rely upon the pytest library to test our code. We installed it along with a command line tool with the same name during our environment creation.

Test-Driven development is a popular approach for development. It involves writing tests first and then writing code that passes the tests.

Testing

All the tests are located in the `test_dataframe.py` module found in the `tests` directory. To run all the tests in this file run the following on the command line.

```
$ pytest tests/test_dataframe.py
```

If you run this command right now, all the tests will fail. As you complete the steps in the project, you will start passing the tests. Once all the tests are passed, the project will be complete.

Automated test discovery

The pytest library has rules for automated test discovery. It isn't necessary to supply the path to the test module if your directories and module names follow those rules. You can simply run `pytest` to run all the tests in this library.

Running specific tests

If you open up one of the test module `test_dataframe.py`, you will see the tests grouped under different classes. Each method of the classes represents exactly one test. To run all the tests within a single class, append two colons followed by the class name. The following is a concrete example:

```
$ pytest tests/test_dataframe.py::TestDataFrameCreation
```

It is possible to run just a single test by appending two more colons followed by the method name. Another concrete example follows:

```
$ pytest tests/test_dataframe.py::TestDataFrameCreation::test_input_types
```

The answer is in pandas_cub_final

The `pandas_cub_final` directory contains the completed `__init__.py` file that contains the code that passes all the tests. Only look at this file after you have attempted to complete the section on your own.

Manually test in a Jupyter Notebook

During development, it's good to have a place to manually experiment with your new code so you can see it in action. We will be using the Jupyter Notebook to quickly see how our DataFrame is changing. Within the `pandas_cub` environment, launch a Jupyter Notebook and open up the `Test Notebook.ipynb` notebook.

Autoreloading

The first cell loads a notebook magic extension which automatically reloads code from files that have changed. Normally, we would have to restart the kernel if we made changes to our code to see it reflect its current state. This magic command saves us from doing this.

Importing pandas_cub

This notebook is at the same level as the inner `pandas_cub` directory. This means that we can import `pandas_cub` directly into our namespace without changing directories. Technically, `pandas_cub` is a Python **package**, which is a directory containing a `__init__.py` file. It is this initialization file that gets run when we write `import pandas_cub as pdc`.

`pandas_cub_final` is also imported so you can see how the completed object is supposed to behave.

A test DataFrame

A simple test DataFrame is created for `pandas_cub`, `pandas_cub_final`, and `pandas`. The output for all three DataFrames are produced in the notebook. There currently is no nice representation for `pandas_cub` DataFrames.

Starting Pandas Cub

You will be editing a single file for this project - the `__init__.py` file found in the `pandas_cub` directory. It contains skeleton code for the entire project. You won't be defining your own classes or methods, but you will be filling out the method bodies.

Open up this file now. You will see many incomplete methods that have the keyword `pass` as their last line. Some methods have code with a comment that says 'your code here'. These are the methods that you will be editing. Other methods are complete and won't need editing.

How to complete the project

Keep the `__init__.py` file open at all times. This is the only file that you will be editing. Read and complete each numbered section below. Edit the method indicated in each section and then run the test. Once you pass that test, move on to the next section.

1. Check DataFrame constructor input types

Our DataFrame class is constructed with a single parameter, `data`. We are going to force our users to set this value as a dictionary that has strings as the keys and one-dimensional NumPy arrays as the values. The keys will eventually become the column names and the arrays will be the values of those columns.

In this step, we will fill out the `_check_input_types` method. This method will ensure that our users have passed us a valid `data` parameter. Notice that this `data` is already assigned to the `_data` instance variable, meaning you will access to within the method with `self._data`.

Specifically, `_check_input_types` must do the following:

- raise a `TypeError` if `data` is not a dictionary
- raise a `TypeError` if the keys of `data` are not strings
- raise a `TypeError` if the values of `data` are not NumPy arrays
- raise a `ValueError` if the values of `data` are not 1-dimensional

Run the following command to test this section:

```
$ pytest tests/test_dataframe.py::TestDataFrameCreation::test_input_types
```

2. Check array lengths

We are now guaranteed that `data` is a dictionary of strings mapped to one-dimensional arrays. Each column of data in our DataFrame must have the same number of elements. In this step, you must ensure that this is the case. Edit the `_check_array_lengths` method and raise a `ValueError` if any of the arrays are differ in length.

Run the following test:

```
$ pytest tests/test_dataframe.py::TestDataFrameCreation::test_array_length
```

3. Change unicode arrays to object

By default, whenever you create a NumPy array of Python strings, it will default the data type of that array to unicode. Unicode arrays are more difficult to manipulate and don't have the flexibility that we desire. So, if our user passes us a Unicode array, we will cover it to a data type called 'object'. This is a flexible type and will help us later when creating methods just for string columns. This type allows any Python objects within the array.

In this step, you will change the data type of Unicode arrays to object. You will do this by checking each arrays data type `kind`. The data type `kind` is a single-character value available by doing `array.dtype.kind`. Use the `astype` array method to change its type.

Edit the `_convert_unicode_to_object` method and verify with the `test_unicode_to_object` test.

4. Find the number of rows in the DataFrame with the `len` function

The number of rows are returned when passing a Pandas DataFrame to the builtin `len` function. We will make pandas_cub behave the same exact way.

To do so we need to implement the special method `__len__`. This is what Python call whenever an object is passed to the `len` function.

Edit the `__len__` method and have it return the number of rows. Test with `test_len`.

5. Return columns as a list

In pandas, calling `df.columns` returns a sequence of the column names. Our column names are currently the keys in our `_data` dictionary. Python provides the `property` decorator which allows us to execute code on something that appears to be just an instance variable.

Edit the `columns` 'method' (really a property) to return a list of the columns in order. Since we are working with Python 3.6, the dictionary keys are internally ordered. Take advantage of this. Validate with the `test_columns` test.

6. Set new column names

In this step, we will be assigning all new columns to our DataFrame by setting the columns property equal to a list. This is the exact same syntax as it is with Pandas. A concrete example below shows how you would set new coulmns for a 3-column DataFrame.

Complete the following tasks:

- Raise a `TypeError` if the object used to set new columns is not a list
- Raise a `ValueError` if the number of column names in the list does not match the current DataFrame
- Raise a `TypeError` if any of the columns are not strings
- Raise a `ValueError` if any of the column names are duplicated in the list
- Reassign the `_data` variable so that all the keys have been updated

```
df.columns = ['state', 'age', 'fruit']
```

Python allows you to set columns by using the decorator `columns.setter`. The value on the right hand side of the assignment statement is passed to the method. Edit the 'column' method decorated by `columns.setter` and test with `test_set_columns`.

7. The `shape` property

The `shape` property in Pandas returns a tuple of the number of rows and columns. The property decorator is used again here. Edit it to have our DataFrame do the same as Pandas. Test with `test_shape`

8. Uncomment `_repr_html_` method

This is a method specifically used by IPython to represent your object in the Jupyter Notebook. This method must return a string of html. This method is fairly complex and you must know some basic html to complete. I decided to implement this method for you. Uncomment it and test the output in the notebook. You should now see a nicely formatted representation of your DataFrame.

9. The `values` property

In Pandas, `values` is a property that returns a single array of all the columns of data. Our DataFrame will do the same. Edit the `values` property and concatenate all the column arrays into a single two-dimensional NumPy array. Return this array. The NumPy `column_stack` function can be helpful here. Test with `test_values`.

10. The `dtypes` property

In Pandas, the `dtypes` property returns a Series containing the data type of each column with the column names in the index. Our DataFrame doesn't have an index. Instead, return a two-column DataFrame. Put the column names under the 'Column Name' column and the data type (bool, int, string, or float) under the column name 'Data Type'.

At the top of the `__init__.py` module there exists a `DTYPE_NAME` dictionary. Use it to convert from array `kind` to the string name of the data type. Test with `test_dtypes`.

11. Select a single column with the brackets

In Pandas, you can select a single with `df['colname']`. Our DataFrame will do the same. To make an object work with the brackets, you must implement the `__getitem__` special method. This method is passed a single parameter, the value within the brackets.

In this step, use `isinstance` to check whether `item` is a string. If it is return a one column DataFrame of that column.

These tests are under a the `TestSelection` class. Run the `test_one_column` test.

12. Select multiple columns with a list

Our DataFrame will also be able to select multiple columns if given a list within the brackets. For example, `df[['colname1', 'colname2']]` will return a two column DataFrame.

Continue editing the `__getitem__` method. If `item` is a list, return a DataFrame of just those columns. Run the `test_multiple_columns`

13. Boolean Selection with a DataFrame

In Pandas, you can filter for specific rows of a DataFrame by passing in a boolean Series/array to the brackets. For instance, the following will select all rows such that `a` is greater than 10.

```
>>> s = df['a'] > 10
>>> df[s]
```

This is called boolean selection. We will make our DataFrame work similarly. Edit the `__getitem__` method and check whether `item` is a DataFrame. If it is then do the following:

- If it is more than one column, raise a `ValueError`
- Extract the underlying array from the single column
- If the underlying array kind is not boolean ('b') raise a `ValueError`
- Use the boolean array to return a new DataFrame with just the rows where the boolean array is `True` along with all the columns.

Run `test_simple_boolean` to test

14.

- A single string selects one column -> `df['colname']`
- A list of strings selects multiple columns -> `df[['colname1', 'colname2']]`
- A one column DataFrame of booleans that filters rows -> `df[df_bool]`
- Row and column selection simultaneously -> `df[rs, cs]`
- cs and rs can be integers, slices, or a list of integers
- rs can also be a one-column boolean DataFrame

Implement the first two items in the list and then copy and paste all the code from pandas_cub_final.

10. Tab Completion

IPython helps us again by providing us with the `_ipython_key_completions_` method. Return a list of the tab completions you would like to have available when inside the brackets operator.

11. Create or overwrite a column

To make an assignment with the brackets operator, Python makes the `__setitem__` method which accepts two values, the `key` and the `value`. We will only implement the simple case of adding a new column or overwriting an old one.

12. `head` and `tail` methods

Have these methods accept a single parameter `n` and return the first/last n rows.

13. Generic aggregation methods

Aggregation methods return a single value for each column. We will only implement column-wise aggregations and not row-wise.

Write a generic method `_agg` that accepts an aggregation function as a string. Use the `getattr` function to get the actual NumPy function.

String columns with missing values will not work. Except this error and don't return columns where the aggregation cannot be found.

Defining the `_agg` method will make all the other aggregation methods work.

14. `isna` method

Return a DataFrame of the same shape that has a boolean for every single value in the DataFrame. Use `np.isnan` except in the case for strings which you can use a vectorized equality expression to `None`

15. `count` method

Return the number of non-missing values for each column

16. `unique` method

Return a list of one-column dataframes of unique values in each column. If there is a single column. Return just the DataFrame

17. `nunique` method

Return the number of unique values for each column

18. `value_counts` method

Return a list of two-column DataFrames with the first column name as the name of the original column and the second column name 'count' containing the number of occurrences for each value.

Use the `Counter` method of the `collections` module. Return the DataFrames with sorted values from greatest to least. You hold off on sorting until you have defined the `sort_values` method.

Accept a boolean parameter `normalize` that returns relative frequencies when `True`.

If the calling DataFrame has a single column, return a single DataFrame.

19. `rename` method

Accept a dictionary of old column names mapped to new column names. Return a new DataFrame

20. `drop` method

Accept a list of column names and return a DataFrame without those columns

21. Non-aggregation methods

There are several non-aggregation methods that function similarly. Create a generic method `_non_agg` that can implement:

- `abs`
- `cummin`
- `cummax`
- `cumsum`
- `clip`
- `round`
- `copy`

The `cummin` and `cummax` functions in NumPy necessitate dot notation to reach. We cannot use `getattr` for this and instead have to use the more specialized `attrgetter` from the `operator` library.

Notice that some of these have parameters. Collect them with `*args`.

22. `diff` and `pct_change` methods

Return the raw difference or percentage change between rows given a distance `n`. You can drop the first n rows.

23. Arithmetic and Comparison Operators

All the arithmetic and comparison operators have special methods available. For instance `__add__` is used for the plus sign, and `__le__` is used for less than or equal to. Each of these methods accepts a single other parameter.

Write a generic method, `_oper` that works with each of these methods.

24. `sort_values` method

This method takes two parameters. The sorting column or columns (as a string or list) and a boolean for the direction to sort. You will need to use NumPy's `argsort` to get the order of the sort for a single column and `lexsort` to sort multiple columns.

25. `sample` method

This method randomly samples the rows of the DataFrame. You can either choose an exact number to sample with `n` or a fraction with `frac`. Sample with replacement by using the boolean `replace`. You can also set the random number seed.

26. `str` accessor

In the `__init__` method, there was a line that created `str` as an instance variable with the `StringMethods` type.

All the string methods use the generic `_str_method` method which accepts the name of the method, the column name and any method-specific parameters.

Modify the generic `_str_method` to make all the other string methods work.

27. `pivot_table` method

This is by far the most complex method to implement. Allow `rows` and `columns` to be column names who's unique values form the groups. Aggregate the column passed to the `values` parameter with the `aggfunc` string.

Allow either `rows` or `columns` to be `None`. If `values` or `aggfunc` is `None` then find the frequency (like in `value_counts`).

28. Automatically add documentation

This method is already completed and automatically adds documentation to the aggregation methods by setting the `__doc__` attribute.

29. Reading simple CSVs

Implement the `read_csv` function by reading through each line. Assume the first line has the column names. Use the second line to assign the data types of each column.