Build a Data Analysis Library from Scratch

This repository contains a detailed project that teaches you how to build your own Python data analysis library, pandas_cub, from scratch. The end result will be a fully-functioning library similar to pandas.

YouTube Video Series

A detailed video series is available at the Dunder Data YouTube channel that walks you through the entire project.

Target Student

This project is targeted towards those who understand the fundamentals of Python and would like to immerse themselves into a larger, highly structured project that covers some advanced topics. It also touches upon a few crucial areas of software development.

Pre-Requisites

This is not a project suitable for beginning Python users. At a minimum you will need to have a solid understanding of the fundamentals such as:

- Basic types and common data structures (lists, tuples, sets, and dictionaries)
- Control flow with if/else statements and for loops (especially when iterating through lists or dictionaries)
- Raising and handling exceptions
- You will need to have covered the basics of classes and object-oriented programming. If you have never defined a class before, I strongly recommend going through an introductory tutorial on them first. This one from Corey Shafer is good.

In addition to those Python basics, the main workhorse is the numpy library. The project will be difficult to complete without prior exposure to numpy. This quickstart guide might be beneficial for those needing to catch up quickly.

We will not be using the pandas library within our code, but will be implementing many of the same method names with similar parameters and functionality. It will be very beneficial to have some exposure to pandas before beginning.

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
- Select subsets of data with the brackets operator
- Use special methods defined in the Python data model
- Have a nicely formatted display of the DataFrame in the notebook

- 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
- Read in data from a comma-separated value file

In addition to these items specific to data analysis, you will also learn about:

- Creating a development environment
- Test-Driven Development

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 with 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 directory 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 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 developing software. It involves writing tests first and then writing code that passes the tests.

Testing

All the tests have already been written and are located in the `test_dataframe.py` module found in the `tests` directory. There are about 100 tests that you will need to pass to complete the project. 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. You should see a line of red capital 'F's. As you complete the steps in the project, you will start passing the tests. There are about 100 total 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 the `test_dataframe.py` file, 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
```

Installing an IPython Kernel for Jupyter

Although we have set up our development environment to work on the command line, we need to make a few more steps to hook it up with Jupyter Notebooks correctly.

This is important, because Jupyter Notebooks are good for manually testing code as you will see below.

Launch a Jupyter Notebook

Within the `pandas_cub` environment, launch a Jupyter Notebook with the command `jupyter notebook`. When the home page finishes loading in your browser open up the `Test Notebook.ipynb` notebook.

Changing the environment within Jupyter

Although we launched our Jupyter Notebook within the `pandas_cub` environment, our code may not be executed within the `pandas_cub` environment at first. If that sounds bizarre and non-intuitive then you have reached the same conclusion as me. It is possible to run any python executable from a Jupyter Notebook regardless of the environment that it was launched from.

If you run the first cell of the notebook (shown below) you can verify the location in your file system where Python is getting executed.

One of two possibilities can happen

If the value outputted from `sys.executable` is in the pandas_cub environment, it will have a path that ends like this:

```
anaconda3/envs/pandas_cub/bin/python
```

If this is your output, you can skip the rest of this step.

If the value outputted from `sys.executable` ends with the following:

```
anaconda3/bin/python
```

then you are actually not executing python from the `pandas_cub` environment. You need to complete the rest of the step.

```
In [1]: import sys
        sys.executable

Out[1]: '/Users/Ted/anaconda3/bin/python3'
```

Exit out of Jupyter and return to the command line. We need to create a new Kernel, a program that "runs and introspects the user's code"

Thankfully there is a command we can run with the `ipykernel` package to automatically create a new kernel. The `ipykernel` package should get installed during environment creation.

The following command creates the Kernel. Make sure you have activated the `pandas_cub` environment first. You can read more about this command in the documentation.

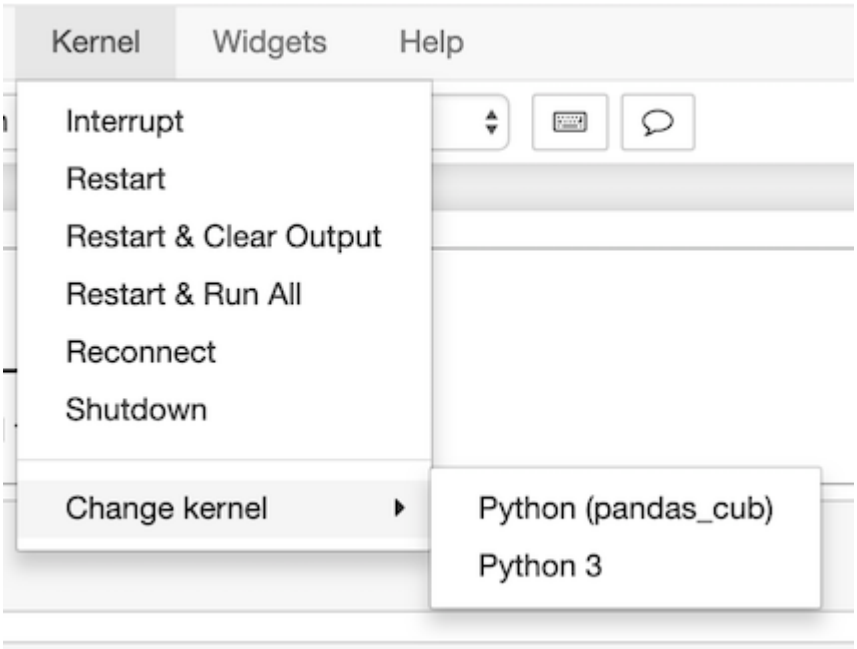
```
python -m ipykernel install --user --name pandas_cub --display-name
"Python (pandas_cub)"
```

You may verify that the `pandas_cub` Kernel was created with the following command:

```
jupyter kernelspec list
```

Launch Jupyter Again

Launch Jupyter again and open up the `Test Notebook.ipynb` notebook. You will still NOT be in the `pandas_cub` environment. You need to navigate inside the 'Kernel' menu above and into 'Change kernel'. Finally, you can select the 'Python (pandas_cub)' Kernel which will place you in the right environment. The kernel will restart once you choose this option.



Run the first cell of the notebook again and you should see that the Python executable is coming from the `pandas_cub` environment directory.

```
In [1]: import sys
        sys.executable

Out[1]: '/Users/Ted/anaconda3/envs/pandas_cub/bin/python'
```

You don't have to do this procedure again or this notebook. From now on, it will open up using the `pandas_cub` Kernel that was created. You can of course change the Kernel again but this is its new default. To verify this, shutdown the notebook and restart it.

If you start a new notebook, you will have the option to decide which Kernel you would like to run it with.

Inspecting the `__init__.py` File

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. These are the methods that you will be editing. A few methods are complete and won't need editing.

Docstrings

You'll notice that all the methods have triple quoted strings directly beneath them. These strings are the documentation or 'docstrings'. All docstrings begin with a short summary of what the method does. A Parameters section follows thats lists each parameter, its type, and a description of how its used. The docstrings end with a Returns section that informs the user of what type of object is returned. It's important to read them as they contain information on how to complete the methods.

There are many ways you can write docstrings, but these follow the numpy docstring guide. There are many other sections you may add to them as well.

Importing pandas_cub

Return back to the Jupyter notebook, which should already be open. 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`.

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.

Autoreloading

The second 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.

Imports

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

We import the `pandas` library so that you can compare and contrast its functionality.

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 visual representation for `pandas_cub` DataFrames.

Starting Pandas Cub

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

The answer is in pandas_cub_final

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

1. Check DataFrame constructor input types

Our DataFrame class is constructed with a single parameter, `data`. Python will call the special `__init__` method when first constructing our DataFrame. This method has already been completed for you and you will not need to edit it. Within the `__init__` method, several more methods are called that check to see if the user has passed it valid data. You will be editing these methods during the next few steps.

In this step, you will only be editing the `_check_input_types` method. This is the first method called within the `__init__` method. It will ensure that our users have passed us a valid `data` parameter.

We are going to force our users to set `data` 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.

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

Edit this method now. Use the `isinstance` function to help you determine the type of an object.

Run the following command to test this step. Once you have passed this test move on to the next step.

```
$ 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 differ in length.

Run the following test:

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

3. Convert unicode arrays to object

Whenever you create a numpy array of Python strings, it will default the data type of that array to unicode. Take a look at the following simple numpy array created from strings. Its data type, found in the `dtype` attribute is shown to be 'U' plus the length of the longest string.

```
>>> a = np.array(['cat', 'dog', 'snake'])
>>> a.dtype
dtype('<U5')
```

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 convert it to a data type called 'object'. This is a flexible type and will help us later when creating methods just for string columns. Technically, this data 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`. See the numpy docs for a list of all the available kinds. Let's retrieve the kind of our array from above.

```
>>> a.dtype.kind
'U'
```

Pass the `astype` array method the correct kind character to change its type.

Edit the `_convert_unicode_to_object` method and fill the dictionary `new_data` with the converted arrays. The result of this method will be returned and assigned as the `_data` instance

variable.

Run `test_unicode_to_object` to test.

A note on names that begin with a single underscore

So far we have seen a few examples of attribute and method names within our DataFrame class that begin with a single underscore. These names are intended to be 'private' and not directly accessed by our users. This doesn't prevent our users from accessing these names as nothing is technically private in Python, but it is common convention and discussed in this section of the PEP8 style guide.

Most IDEs will not show these private methods as choices to the users which is a good thing. They are not at all meant to be accessed by them.

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 calls 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`.

Special Methods

Step 4 introduced us to the `__len__` 'special method'. Python has over 100 special methods that allow you to define how your class behaves when it interacts with a builtin function or operator. In the above example, if `df` is a DataFrame and a user calls `len(df)` then internally the `__len__` method will be called. All special methods begin and end with two underscores.

Let's see a few more examples:

- `df + 5` calls the `__add__` special method
- `df > 5` calls the `__lt__` special method
- `-df` calls the `__neg__` special method
- `round(df)` calls the `__round__` special method

We've actually already seen the special method `__init__` which is used to initialize an object and called when a user calls `DataFrame(data)`.

The Python documentation has good (though complex) coverage of all the special methods. We will be implementing many more special methods. I strongly recommend to reference the documentation to learn more.

5. Return columns as a list

In this step you will make `df.columns` return a list of the column names. Notice that `df.columns` is not a method here. There will be no parentheses that follow it.

Looking at the source code, you will see that `columns` appears to be defined as if it is a method. But, directly above it is the property decorator. The `property` decorator will make `df.columns` work just like a method.

Currently the keys in our `_data` dictionary refer to the columns in our DataFrame. 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.

The property decorator

There is quite a bit more to the property decorator, including how its used to set attributes as is done in the next step. This Stack Overflow question contains a good examples that will explain more.

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. A concrete example below shows how you would set new columns for a 3-column DataFrame.

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

There are three parts to properties in Python; the getter, setter, and deleter. In the previous step, we defined the getter. In this step we will define the setter with the `columns.setter` decorator. The value on the right hand side of the assignment statement is passed to the method decorated by `columns.setter`. Edit this method and 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

Test with `test_set_columns`.

7. The `shape` property

The `shape` property will return a two-item tuple of the number of rows and columns. The property decorator is used again here so that `df.shape` can execute code like a method. We could just make it a normal method and invoke it with `df.shape()` but we are following pandas lead and keeping `shape` as a property.

Test with `test_shape`.

8. Visual HTML representation in the notebook with the `_repr_html_` method

Currently we have no representation of our DataFrame. If you try and output your DataFrame, you'll just get its location in memory and it will look something like this:

```
>>> df
<pandas_cub.DataFrame at 0x116d405c0>
```

The `_repr_html_` method is made available to developers by iPython so that your objects can have nicely formatted HTML displays within Jupyter Notebooks. Read more on this method here in the iPython documentation along with other similar methods for different representations.

This method must return a string of html. This method is fairly complex and you must know some basic html to complete it. I recommend copying and pasting the implementation from pandas_cub_final instead of doing it yourself.

If you do know HTML and are seeking a greater challenger use the docstrings to give you an idea of how the HTML may be formatted. There are no tests for this method.

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`.

Hint when returning a DataFrame from a property/method

Many of the next steps require you to return a DataFrame as the result of the property/method. To do so, you will use the DataFrame constructor like this.

```
return DataFrame(new_data)
```

where `new_data` is a dictionary mapping the column names to a one-dimensional numpy array. It is your job to create the `new_data` dictionary correctly.

10. The `dtypes` property

The `dtypes` property will return a two-column DataFrame with the column names in the first column and their data type as a string in the other. Use 'Column Name' and 'Data Type' as column names.

Use the `DTYPE_NAME` dictionary 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 column with `df['colname']`. Our DataFrame will do the same. To make an object work with the brackets, you must implement the `__getitem__` special method. See the official documentation for more.

This special method is always passed a single parameter, the value within the brackets. We use `item` as the parameter name.

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

These tests are under 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 `test_multiple_columns` to test.

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 only the 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

(Optional) Simultaneous selection of rows and column

Steps 14-18 are optional and fairly difficult. The outcome of these steps is to simultaneously select both rows and columns in the DataFrame. The syntax uses the brackets operator like the previous three steps and looks like this:

```
df[rs, cs]
```

where `rs` is the row selection and `cs` is the column selection.

14. (Optional) Check for simultaneous selection of rows and columns

When you pass the brackets operator a sequence of comma separated values with `df[rs, cs]`, Python passes the `__getitem__` special method a tuple of all the values.

To get started coding, within the `__getitem__` special method check whether `item` is a tuple instance. If is not, raise a `TypeError` and inform the user that they need to pass in either a string (step 11), a list of strings (step 12), a one column boolean DataFrame (step 13) or both a row and column selection (step 14).

If `item` is a tuple, return the result of a call to the `_getitem_tuple` method.

Edit the `_getitem_tuple` method from now through step 18.

Within the `_getitem_tuple` method, raise a `ValueError` if it is not exactly two items in length.

Run `test_simultaneous_tuple` to test.

15. (Optional) Select a single cell of data

In this step, we will select a single cell of data with `df[rs, cs]`. We will assume `rs` is an integer and `cs` is either an integer or a string.

To get started, assign the first element of `item` to the variable `row_selection` and the second element of `item` to `col_selection`. From step 14, we know that `item` must be a two-item tuple.

If `row_selection` is an integer, reassign it as a one-element list of that integer.

Check whether `col_selection` is an integer. If it is, reassign to a one-element list of the string column name it represents.

If `col_selection` is a string, assign it to a one-element list of that string.

Now both `row_selection` and `col_selection` are lists. You will return a single-row, single-column DataFrame. This is different than pandas, which just returns a scalar value.

Write a for loop to iterate through each column in the `col_selection` list to create the `new_data` dictionary. Make sure to select just the row that is needed.

This for-loop will be used for the steps through 18 to return the desired DataFrame.

Run `test_single_element` to test.

16. (Optional) Simultaneously select rows as booleans, lists, or slices

In this step, we will again be selecting rows and columns simultaneously with `df[rs, cs]`. We will allow `rs` to be either a single-column boolean DataFrame, a list of integers, or a slice. For now, `cs` will remain either an integer or a string. The following selections will be possible after this step.

```
df[df['a'] < 10, 'b']
df[[2, 4, 1], 'e']
df[2:5, 3]
```

If `row_selection` is a DataFrame, raise a `ValueError` if it is not one column. Reassign `row_selection` to the values (numpy array) of its column. Raise a `TypeError` if it is not a boolean data type.

If `row_selection` is not a list or a slice raise a `TypeError` and inform the user that the row selection must be either an integer, list, slice, or DataFrame. You will not need to reassign `row_selection` for this case as it will select properly from a numpy array.

Your for-loop from step 15 should return the DataFrame.

Run `test_all_row_selections` to test.

17. (Optional) Simultaneous selection with multiple columns as a list

The `row_selection` variable is now fully implemented. It can be either an integer, list of integers, a slice, or a one-column boolean DataFrame.

As of now, the `col_selection` can only be an integer or a string. In this step, we will handle the case when it is a list.

If `col_selection` is a list, create an empty list named `new_col_selection`. Iterate through each element of `col_selection` and check if it is an integer. If it is, append the string column name to `new_col_selection`. If not, assume it is a string and append it as it is to `new_col_selection`.

`new_col_selection` will now be a list of string column names. Reassign `col_selection` to it.

Again, your for-loop from step 15 will return the DataFrame.

Run `test_list_columns` to test.

18. (Optional) Simultaneous selection with column slices

In this step, we will allow our columns to be sliced with either strings or integers. The following selections will be acceptable.

```
df[rs, :3]
df[rs, 1:10:2]
df[rs, 'a':'f':2]
```

Where `rs` is any of the previously acceptable row selections.

Check if `col_selection` is a slice. Slice objects have `start`, `stop`, and `step` attributes. Define new variables with the same name to hold those attributes of the slice object.

If `col_selection` is not a slice raise a `TypeError` informing the user that the column selection must be an integer, string, list, or slice.

If `start` is a string, reassign it to its integer index amongst the columns.

If `stop` is a string, reassign it to its integer index amongst the columns **plus 1**. We add one here so that we include the last column.

`start`, `stop`, and `step` should now be integers. Use them to reassign `col_selection` to a list of all the column names that are to be selected. You'll use slice notation to do this.

The for-loop from 15 will still work to return the desired DataFrame.

Run `test_col_slice` to test.

19. Tab Completion for column names

It is possible to get help completing column names when doing single-column selections. For instance, let's say we had a column name called 'state' and began making a column selection with `df['s]`. If we press tab right here iPython can show us a dropdown list of all the column names beginning with 's'.

We do this by returning the list of possible values we want to see from the `_ipython_key_completions_` method. Complete that method now.

Run `test_tab_complete` to test.

20. Create a new column or overwrite an old column

We will now have our DataFrame create a single new column or overwrite an existing one. Pandas allows for setting multiple columns at once, and even setting rows and columns simultaneously. Doing such is fairly complex and we will not implement those cases and instead focus on just single-column setting.

Python allows setting via the brackets with the `__setitem__` special method. It receives two values when called, the `key` and the `value`. For instance, if we set a new column like this:

```
df['new col'] = np.array([10, 4, 99])
```

the `key` would be 'new col' and the `value` would be the numpy array.

If the `key` is not a string, raise a `NotImplementedError` stating that the DataFrame can only set a single column.

If `value` is a numpy array, raise a `ValueError` if it is not 1D. Raise a different `ValueError` if the length is different than the calling DataFrame.

If `value` is a DataFrame, raise a `ValueError` if it is not a single column. Raise a different `ValueError` if the length is different than the calling DataFrame. Reassign `value` to the underlying numpy array of the column.

If `value` is a single integer, string, float, or boolean, use the numpy `repeat` function to reassign `value` to be an array the same length as the DataFrame with all values the same. For instance, the following should work.

```
>>> df['new col'] = 85
```

Raise a `TypeError` if `value` is not one of the above types.

After completing the above, `value` will be a one-dimensional array. If it's data type `kind` is the string 'U', change its type to object.

Finally, assign a new column by modifying the `_data` dictionary.

Run `test_new_column` to test.

21. `head` and `tail` methods

The `head` and `tail` methods each accept a single integer parameter `n` which is defaulted to 5. Have them return the first/last n rows.

Run `test_head_tail` to complete this.

22. Generic aggregation methods

We will now implement several methods that perform an aggregation. These methods all return a single value for each column. The following aggregation methods are defined.

- min
- max
- mean
- median
- sum
- var
- std
- all
- any
- argmax - index of the maximum
- argmin - index of the minimum

We will only be performing these aggregations column-wise and not row-wise. Pandas enables users to perform both row and column aggregations.

If you look at our source code, you will see all of the aggregation methods already defined. You will not have to modify any of these methods individually. Instead, they all call the underlying `_agg` method passing it the numpy function.

Complete the generic method `_agg` that accepts an aggregation function.

Iterate through each column of your DataFrame and pass the underlying array to the aggregation function. Return a new DataFrame with the same number of columns, but with just a single row, the value of the aggregation.

String columns with missing values raise a `TypeError`. Except this error and don't return columns where the aggregation cannot be found.

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

All the aggregation methods have their own tests in a separate class named `TestAggregation`. They are all named similarly with 'test_' preceding the name of the aggregation. Run all the tests at once.

23. `isna` method

The `isna` method will return a DataFrame the same shape as the original but with boolean values for every single value. Each value will be tested whether it is missing or not. Use `np.isnan` except in the case for strings which you can use a vectorized equality expression to `None`.

Test with `test_isna` found in the `TestOtherMethods` class.

24. `count` method

The `count` method returns a single-row DataFrame with the number of non-missing values for each column. You will want to use the result of `isna`.

Test with `test_count`

25. `unique` method

This method will return the unique values for each column in the DataFrame. Specifically, it will return a list of one-column DataFrames of unique values in each column. If there is a single column, just return the DataFrame.

The reason we use a list of DataFrames is that each column may contain a different number of unique values. Use the `unique` numpy function.

Test with `test_unique`

26. `nunique` method

Return a single-row DataFrame with the number of unique values for each column.

Test with `test_nunique`

27. `value_counts` method

Return a list of DataFrames, unless there is just one column and then just return a single DataFrame. Each DataFrame will be two columns. The first column name will be the name of the original column. The second column name will be 'count'. The first column will contain the unique values in the original DataFrame column. The 'count' column will hold the frequency of each of those unique values.

Use the numpy `unique` function with `return_counts` set to `True`. Return the DataFrames with sorted counts from greatest to least. Use the numpy `argsort` to help with this.

Use the `test_value_counts` test within the `TestGrouping` class.

28. Normalize options for `value_counts`

We will modify the `value_counts` method to return relative frequencies. The `value_counts` method also accepts a boolean parameter `normalize` that by default is set to `False`. If it is `True`, then return the relative frequencies of each value instead.

Test with `test_value_counts_normalize`

29. `rename` method

The `rename` method renames one or more column names. Accept a dictionary of old column names mapped to new column names. Return a DataFrame. Raise a `TypeError` if `columns` is not a dictionary.

Test with`test_rename` within the `TestOtherMethods` class

30. `drop` method

Accept a single string or a list of column names as strings. Return a DataFrame without those columns. Raise a `TypeError` if a string or list is not provided.

Test with `test_drop`

31. Non-aggregation methods

There are several non-aggregation methods that function similarly. All of the following non-aggregation methods return a DataFrame that is the same shape as the origin.

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

All of the above methods will be implemented with the generic `_non_agg` method. This method is sent the numpy function name of the non-aggregating method.

Pass only the boolean, integer, and float columns to this non-aggregating numpy function.

Keep the string columns (only other data type) in your returned DataFrame. Use the `copy` array method to make an independent copy of them.

Notice that some of these non-aggregating methods have extra keyword arguments. These are passed to `_non_agg` and collected with `**kwargs`. Make sure to pass them to the numpy function as well.

There is a different test for each method in the `TestNonAgg` class.

Update after videos

If you are watching my videos for the course, I updated the pandas_cub_final init file to contain a better solution. The `round` method should ignore boolean columns. The original solution applied had each non-aggregation method work on boolean, integer, and float columns.

32. `diff` method

The `diff` method accepts a single parameter `n` and takes the difference between the current row and the `n` previous row. For instance, if a column has the values [5, 10, 2] and `n=1`, the `diff` method would return [NaN, 5, -8]. The first value is missing because there is no value preceding it.

The `diff` method is a non-aggregating method as well, but there is no direct numpy function that computes it. Instead, we will define a function within this method that computes this difference.

Complete the body of the `func` function.

Allow `n` to be either a negative or positive integer. You will have to set the first or last n values to `np.nan`. If you are doing this on an integer column, you will have to convert it to a float first as integer arrays cannot contain missing values. Use `np.roll` to help shift the data in the arrays.

Test with `test_diff`

33. `pct_change` method

The `pct_change` method is nearly identical to the `diff` method. The only difference is that this method returns the percentage change between the values and not the raw difference. Again, complete the body of the `func` function.

Test with `test_pct_change`

34. Arithmetic and Comparison Operators

All the common arithmetic and comparison operators will be made available to our DataFrame. For example, `df + 5` uses the plus operator to add 5 to each element of the DataFrame. Take a look at some of the following examples:

```
df + 5
df - 5
df > 5
df != 5
5 + df
5 < df
```

All the arithmetic and comparison operators have corresponding special methods that are called whenever the operator is used. For instance `__add__` is called when the plus operator is used, and

`__le__` is called whenever the less than or equal to operator is used. See the full list in the documentation.

Each of these methods accepts a single parameter, which we have named `other`. All of these methods call a more generic `_oper` method which you will complete.

Within the `_oper` method check if `other` is a DataFrame. Raise a `ValueError` if this DataFrame not one column. Otherwise, reassign `other` to be a 1D array of the values of its only column.

If `other` is not a DataFrame do nothing and continue executing the rest of the method. We will not check directly if the types are compatible. Instead we will pass this task onto numpy. So, `df + 5` should work if all the columns in `df` are booleans, integers, or floats.

Iterate through all the columns of your DataFrame and apply the operation to each array. You will need to use the `getattr` function along with the `op` string to retrieve the underlying numpy array method. For instance, `getattr(values, '__add__')` returns the method that uses the plus operator for the numpy array `values`. Return a new DataFrame with the operation applied to each column.

Run all the tests in class `TestOperators`

35. `sort_values` method

This method will sort the rows of the DataFrame by one or more columns. Allow the parameter `by` to be either a single column name as a string or a list of column names as strings. The DataFrame will be sorted by this column or columns.

The second parameter, `asc`, will be a boolean controlling the direction of the sort. It is defaulted to `True` indicating that sorting will be ascending (lowest to greatest). Raise a `TypeError` if `by` is not a string or list.

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.

Run the following tests in the `TestMoreMethods` class.

- `test_sort_values`
- `test_sort_values_desc`
- `test_sort_values_two`
- `test_sort_values_two_desc`

36. `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`. The `seed` parameter will be used to set the random number seed.

Raise a `ValueError` if `frac` is not positive and a `TypeError` if `n` is not an integer.

You will be using numpy's random module to complete this method. Within it are the `seed` and `choice` functions. The latter function has a `replace` parameter that you will need to use. Return a

new DataFrame with the new random rows.

Run `test_sample` to test.

37. `pivot_table` method

This is a complex method to implement. This method allows you to create a pivot table from your DataFrame. The following image shows the final result of calling the pivot table on a DataFrame. It summarizes the mean salary of each gender for each race.

	dept	race	gender	salary
0	Houston Police Department-HPD	White	Male	45279
1	Houston Fire Department (HFD)	White	Male	63166
2	Houston Police Department-HPD	Black	Male	66614
3	Public Works & Engineering-PWE	Asian	Male	71680
4	Houston Airport System (HAS)	White	Male	42390
5	Public Works & Engineering-PWE	White	Male	107962
6	Houston Fire Department (HFD)	Hispanic	Male	52644
7	Health & Human Services	Black	Male	180416
8	Public Works & Engineering-PWE	Black	Male	30347
9	Health & Human Services	Black	Male	55269

	race	Female	Male
0	Asian	59529.761	60178.565
1	Black	48388.608	51868.905
2	Hispanic	44393.107	55381.608
3	Native American	60014.657	68561.416
4	White	66491.838	63405.818

A typical pivot table uses two columns as the **grouping columns** from your original DataFrame. The unique values of one of the grouping columns form a new column in the new DataFrame. In the example above, the race column had five unique values.

The unique values of the other grouping column now form the columns of the new DataFrame. In the above example, there were two unique values of gender.

In addition to the grouping columns is the **aggregating column**. This is typically a numeric column that will get summarized. In the above pivot table, the salary column was aggregated.

The last component of a pivot table is the **aggregating function**. This determines how the aggregating columns get aggregated. Here, we used the `mean` function.

The syntax used to produce the pivot table above is as follows:

```
df.pivot_table(rows='race', columns='gender', values='salary',
aggfunc='mean')
```

`rows` and `columns` will be assigned the grouping columns. `values` will be assigned the aggregating column and `aggfunc` will be assigned the aggregating function. All four parameters will be strings. Since `aggfunc` is a string, you will need to use the builtin `getattr` function to get the correct numpy function.

There are several approaches that you can take to implement this. One approach involves using a dictionary to store the unique combinations of the grouping columns as the keys and a list to store the values of the aggregative column. You could iterate over every single row and then use a two-

item tuple to hold the values of the two grouping columns. A `defaultdict` from the collections module can help make this easier. Your dictionary would look something like this after you have iterated through the data.

```
{('black', 'male'): [50000, 90000, 40000],
 ('black', 'female'): [100000, 40000, 30000]}
```

Once you have mapped the groups to their respective values, you would need to iterate through this dictionary and apply the aggregation function to the values. Create a new dictionary for this.

From here, you need to figure out how to turn this dictionary into the final DataFrame. You have all the values, you just need to create a dictionary of columns mapped to values. Use the first column as the unique values of the rows column.

Other features:

- Return a DataFrame that has the rows and columns sorted
- You must make your pivot table work when passed just one of `rows` or `columns`. If just `rows` is passed return a two-column DataFrame with the first column containing the unique values of the rows and the second column containing the aggregations. Title the second column the same name as `aggfunc`.
- If `aggfunc` is `None` and `values` is not None then raise a `ValueError`.
- If `values` is `None` and `aggfunc` is not then raise a `ValueError` as there are no values to be aggregated.
- If `aggfunc` and `values` are both `None` then set `aggfunc` equal to the string 'size'. This will produce a contingency table (the raw frequency of occurrence). You might need to create an empty numpy array to be a placeholder for the values.

Run `test_pivot_table_rows_or_cols` and `test_pivot_table_both` in the `TestGrouping` class.

38. Automatically add documentation

All docstrings can be retrieved programmitcally with the `__doc__` special attribute. Docstrings can also be dynamically set by assigning this same special attribute a string.

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

39. String-only methods with the `str` accessor

Look back up at the `__init__` method. One of the last lines defines `str` as an instance variable assigned to a new instance of `StringMethods`. Pandas uses the same variable name for its DataFrames and calls it a string 'accessor'. We will also refer to it as an accessor as it gives us access to string-only methods.

Scroll down below the definition of the `DataFrame` class. You will see the `StringMethods` class defined there. During initialization it stores a reference to the underlying DataFrame with `_df`.

There are many string methods defined in this class. The first parameter to each string method is the name of the column you would like to apply the string method to. We will only allow our accessor to work on a single column of the DataFrame.

You will only be modifying the `_str_method` which accepts the string method, the name of the column, and any extra arguments.

Within `_str_method` select the underlying numpy array of the given `col`. Raise a `TypeError` if it does not have kind 'O'.

Iterate over each value in the array and pass it to `method`. It will look like this: `method(val, *args)`. Return a one-column DataFrame with the new data.

Test with class `TestStrings`

40. Reading simple CSVs

It is important that our library be able to turn data in files into DataFrames. The `read_csv` function, at the very end of our module, will read in simple comma-separated value files (CSVs) and return a DataFrame.

The `read_csv` function accepts a single parameter, `fn`, which is a string of the file name containing the data. Read through each line of the file. Assume the values in each line are separated by commas. Also assume the first line contains the column names.

Create a dictionary to hold the data and return a new DataFrame. Use the file `employee.csv` in the `data` directory to test your function manually.

Run all the tests in the `TestReadCSV` class.