Photo by Sarah_Loetscher on Pixabay

All the Core Functions of Python Pandas You Need to Know

All the Pandas functions you need to nail to become an eligible Python Data Analyst.

Christopher Tao
Apr 17 · 12 min read ★

As one of the most popular library in the Python programming language, Pandas is a "must-learn" library for Data I/O, cleansing, transforming and aggregation.

In `read_excel` , we can specify which spreadsheet to load by giving the `sheet_name` , if multiple sheets are existing.

For `read_json` , it is important to use the `orient` parameter correctly. The most commonly used are `records` when the JSON document is an array, and `index` if you want to use the root keys as indices.

The `to_csv` , `to_excel` and `to_json` are the corresponding writing functions. The important difference is that we need to call these functions from a Data Frame object rather than the Pandas object.

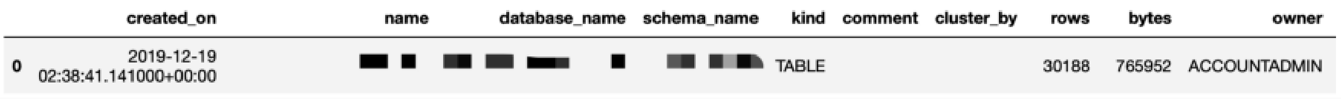
```
df1.to_csv('./data.csv')
df2.to_excel('./data.xlsx')
df3.to_json('./data.json')
```

Reading and Writing directly from/to database is a bit more tricky. Pandas support multiple libraries such as `pymssql` for SQL Server and `pymysql` for MySQL. However, the one I like the most is `sqlalchemy` which supports most popular databases even including cloud databases such as Snowflake DB.

```
from sqlalchemy import create_engine

db_engine = create_engine(
    'snowflake://{user}:{password}@{account}/'.format(
    user='...',
    password='...',
    account='...',
))

df = pd.read_sql("SHOW TABLES", db_engine)
df.head()
```



	created_on	name	database_name	schema_name	kind	comment	cluster_by	rows	bytes	owner
0	2019-12-19 02:38:41.141000+00:00	[redacted]	[redacted]	[redacted]	TABLE			30188	765952	ACCOUNTADMIN

Data Preview

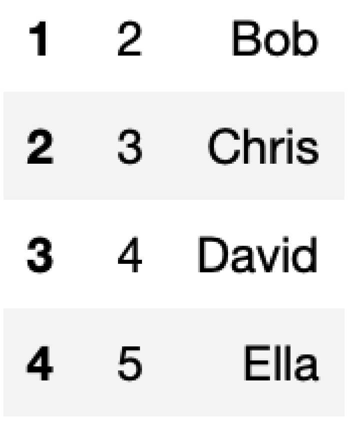
Photo by Christian Wiediger on Unsplash

Very often, we want to get a general picture of the dataset we have, or just want to check whether the data has been loaded into Pandas data frame correctly or not. We'll need to know several functions to do such.

df.head()

```
df = pd.DataFrame({
    'id': [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
    'name': ['Alice', 'Bob', 'Chris', 'David', 'Ella', 'Frank',
'Grace', 'Hellen', 'Iva', 'Jack']
})

df.head()
```



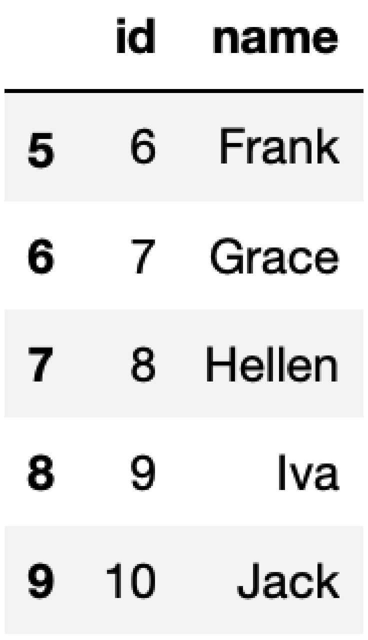
1	2	Bob
2	3	Chris
3	4	David
4	5	Ella

For demonstrating purpose, I firstly created a data frame with two columns — "id" and "name".

The `df.head()` function shows the top 5 rows by default.

However, you can always specify how many rows to show such as `df.head(10)` to show 10 rows.

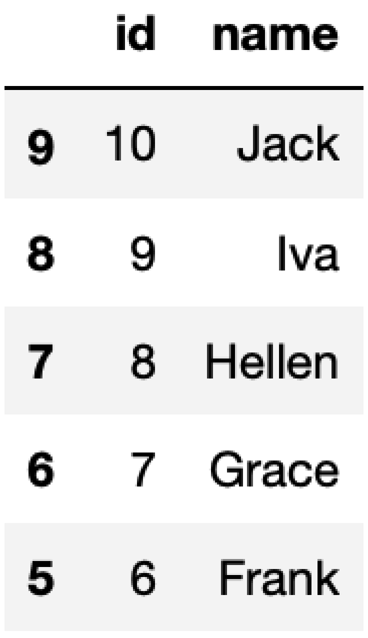
df.tail()



	id	name
5	6	Frank
6	7	Grace
7	8	Hellen
8	9	Iva
9	10	Jack

This will be helpful when your dataset is sorted, and you want to check the results.

df.sort_values()



	id	name
9	10	Jack
8	9	Iva
7	8	Hellen
6	7	Grace
5	6	Frank

```
df.sort_values(by='name', ascending=False, inplace=True)

df.head()
```

We can sort by either ascending or descending. Note that `inplace` needs to be set to `True` if you want to assign the sorted data frame back to your variable. Otherwise, your data frame `df` will not be changed.

df.columns

Please note that this is not a function, but a property of the data frame.

```
df.columns
```

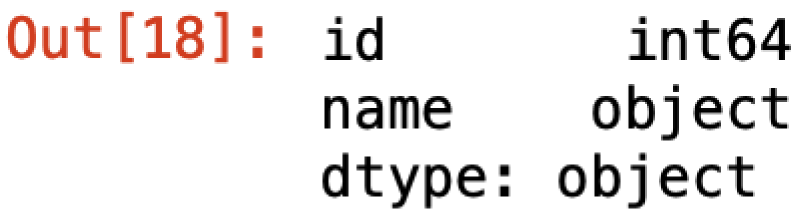
Also, you can assign a list to this property to rename all the columns of the data frame. For example:

```
df.columns = ['no', 'firstname']
```

df.dtypes

This is also a property of the data frame that returns all the data types for the columns. It is very useful when you want to check the data types, especially dealing with `datetime` columns.

```
df.dtypes
```

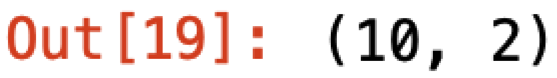


```
Out[18]: id         int64
         name      object
         dtype: object
```

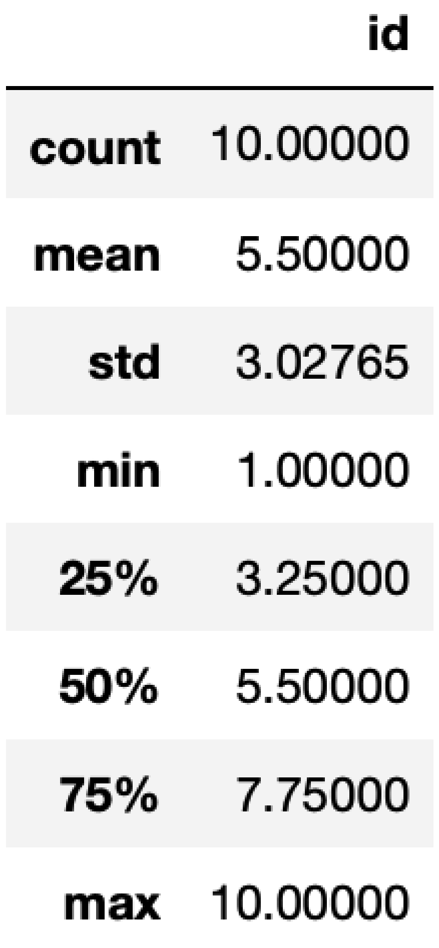
df.shape

This is probably the property that I use the most. Very frequently we may want to check the number of rows and columns of our data frame.

```
df.shape
```



```
Out[19]: (10, 2)
```



	id
count	10.00000
mean	5.50000
std	3.02765
min	1.00000
25%	3.25000
50%	5.50000
75%	7.75000
max	10.00000

```
df.describe()
```

When we are dealing with some measures, we may want to have a picture of the data distribution. `df.describe()` gives this rough statistics.

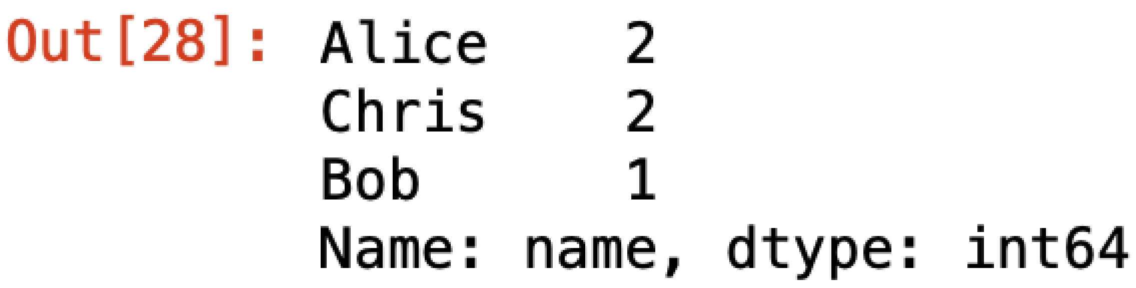
When this function is called for a data frame with multiple columns, the non-numeric ones will be ignored.

Note that the stats for `id` columns doesn’t make any sense, but it demonstrates the function.

s.value_counts()

frequencies in this data frame. Please note that we use `df.name` or `df['name']` to get the name column of the data frame as a series on which the `value_counts()` function can be applied.

```
df = pd.DataFrame({'id': [1, 2, 3, 4, 5],
                   'name': ['Alice', 'Bob', 'Chris', 'Chris',
'Alice']})
df.name.value_counts()
```



```
Out[28]: Alice    2
         Chris    2
         Bob      1
         Name: name, dtype: int64
```

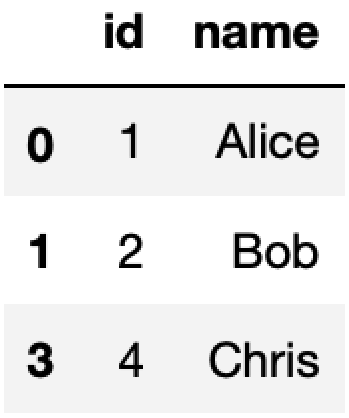
Data Cleansing

It is quite common that the raw dataset we got is not perfect. So, we need to clean the dataset before use. Here are the related Pandas functions.

s.isna()

When we want to filter "NULL" values in the data frame, this function will help. Let's create another data frame with some NULL values in the name column. Note that we use `None` in Python for null objects.

```
df = pd.DataFrame({'id': [1, 2, 3, 4, 5],
                   'name': ['Alice', 'Bob', None, 'Chris', None]})
```



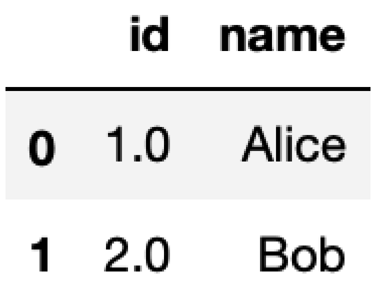
	id	name
0	1	Alice
1	2	Bob
3	4	Chris

Then, suppose we want to filter all the NULL values out.

```
df[~df.name.isna()]
```

`df.name` helps to get the "name" column of the data frame as a series, and then `isna()` returns a series of boolean values representing whether the name is null. After that, the `~` sign at the beginning reversed the boolean value, since we want to reserve the rows that NOT having NULL values. Finally, the data frame `df` will be filtered by this boolean series, where the row with "False" boolean value will be discarded.

```
df.dropna()
```



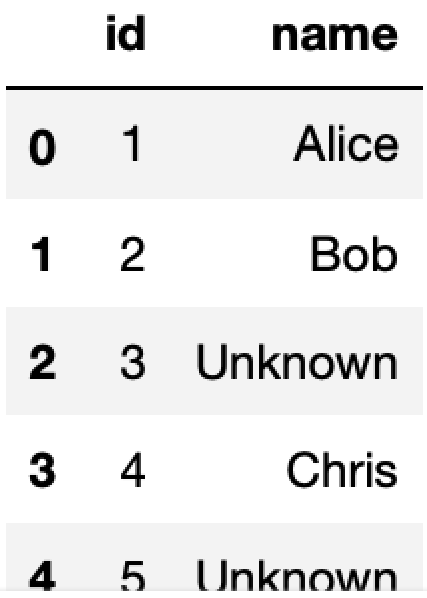
	id	name
0	1.0	Alice
1	2.0	Bob

What if we have multiple columns that have null values and we want to filter out all the rows having at least one null values? Well, we could still use the above method but you'll need to repeat many times for every column.

df.fillna()

```
df = pd.DataFrame({'id': [1, 2, 3, 4, 5],
                   'name': ['Alice', 'Bob', None, 'Chris', None]})

df.fillna('Unknown')
```



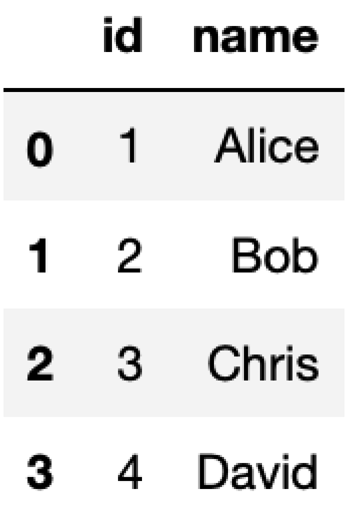
	id	name
0	1	Alice
1	2	Bob
2	3	Unknown
3	4	Chris
4	5	Unknown

In this example, the name column with null values are replaced with "Unknown" string, and we still have these rows.

df.drop_duplicates()

```
df = pd.DataFrame({'id': [1, 2, 3, 4, 3],
                   'name': ['Alice', 'Bob', 'Chris', 'David',
'Chris']})

df.drop_duplicates()
```



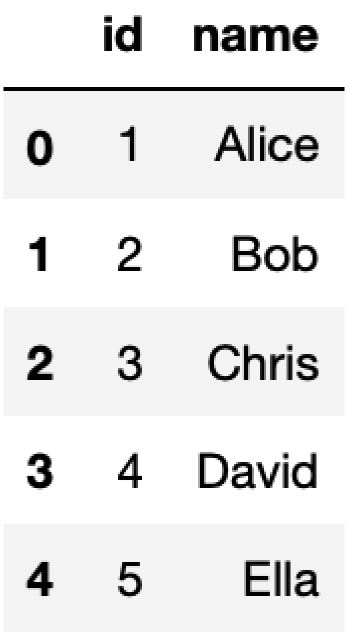
	id	name
0	1	Alice
1	2	Bob
2	3	Chris
3	4	David

Sometimes the raw dataset may have some duplicated rows which we don't actually want them.

In this example, we have two "Chris" with id = 3. So, the function dropped the second.

df.drop()

```
df = pd.DataFrame({'id': [1, 2, 3, 4, 3],
                   'name': ['Alice', 'Bob', 'Chris', 'David',
```



	id	name
0	1	Alice
1	2	Bob
2	3	Chris
3	4	David
4	5	Ella

In this example, the data frame has 3 columns.

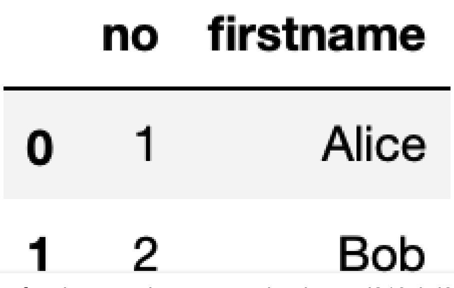
Suppose that we don't need the "comments" column, we could use `df.drop()` function to drop it.

This function can also be used to drop rows.

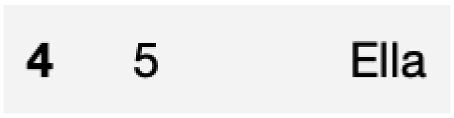
df.rename()

```
df = pd.DataFrame({'id': [1, 2, 3, 4, 5],
                   'name': ['Alice', 'Bob', 'Chris', 'David',
'Ella']})

df.rename(columns={'id': 'no', 'name': 'firstname'})
```



	no	firstname
0	1	Alice
1	2	Bob



4	5	Ella

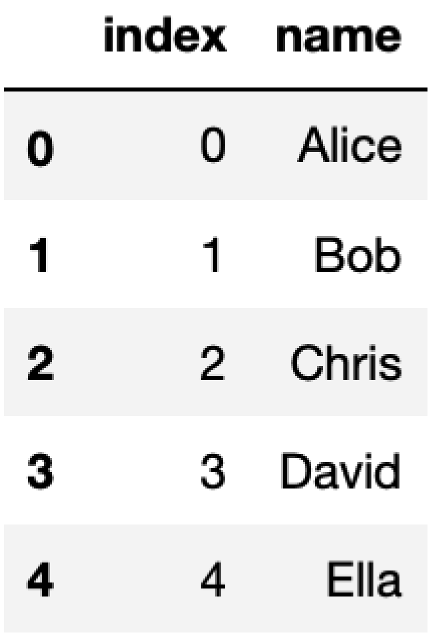
In this example, the data frame is still created with the column names "id" and "name". However, the `df.rename()` function helped to rename the column headers.

Please note that it takes a dictionary as the parameter, where the keys are the old headers and the values are the new headers.

df.reset_index()

```
df = pd.DataFrame({'name': ['Alice', 'Bob', 'Chris', 'David',
'Ella']})

df.reset_index()
```



	index	name
0	0	Alice
1	1	Bob
2	2	Chris
3	3	David
4	4	Ella

In this example, the data frame has only the "name" column. So, what if we want to create another column as identity?

dataset that some duplicated rows were discarded. However, the index will not be continuous any more. If you want to have a continuous index again, this function helps too.

Data Transformation

Photo by Swapnll Dwivedi on Unsplash

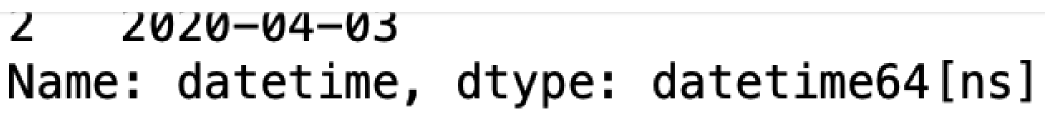
After the data cleansing, we may need to transform the data.

pd.to_datetime()

Commonly, we have our raw dataset with all the dates or time in string format. For the later on analytics purposes such as sorting, we may want to convert these strings into the `datetime` objects.

```
df = pd.DataFrame({'datetime': ['01/04/2020', '02/04/2020',
'03/04/2020']})

pd.to_datetime(df.datetime, format='%d/%m/%Y')
```



```
2   2020-04-03
Name: datetime, dtype: datetime64[ns]
```

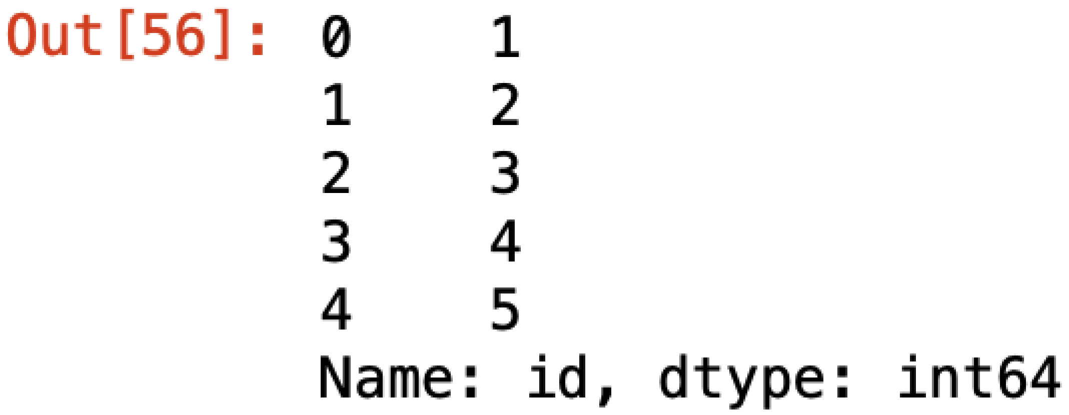
Please note that this function returns a series, so we can assign it back to the column of the data frame.

s.astype()

This function helps us to convert the data type of a column easily. In the example, the data frame was created its "id" field with all string type. Then, the `s.astype()` function helped to convert them into integers.

```
df = pd.DataFrame({'id': ['1', '2', '3', '4', '5']})

df.id.astype(int)
```



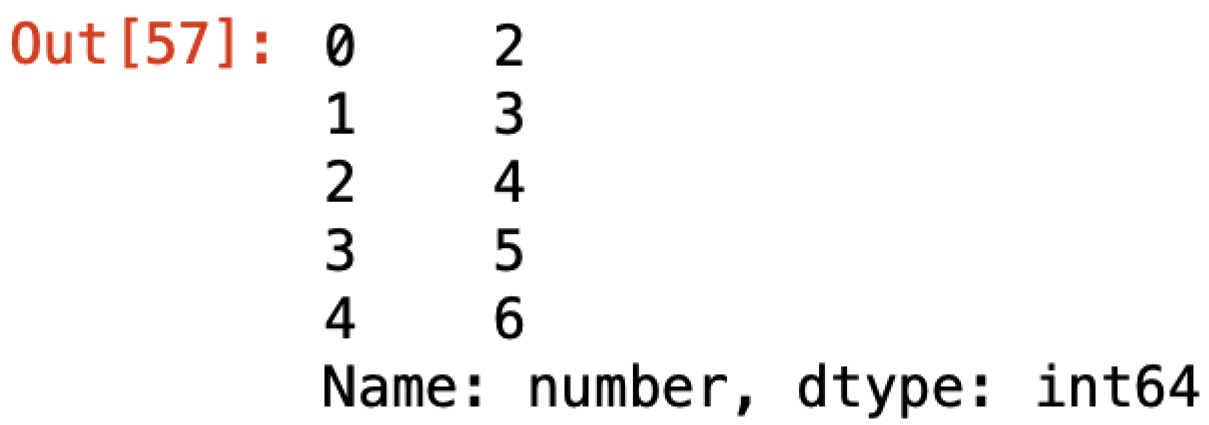
```
Out[56]: 0    1
         1    2
         2    3
         3    4
         4    5
         Name: id, dtype: int64
```

s.apply()

This is probably the function that I used the most. Although Pandas allows us to easily perform some transformation to a whole column of a data frame, for example, `df.col + 1` will add 1 to all the values of this column. However, sometimes we may need to do something quite unique which are not supported by Pandas built-in functions. In this case, the `apply()` function helps.

```
df = pd.DataFrame({'number': [1, 2, 3, 4, 5]})

df.number.apply(lambda n: n+1)
```



```
Out[57]: 0    2
         1    3
         2    4
         3    5
         4    6
         Name: number, dtype: int64
```

In some special cases, the lambda function may not be enough. For example, we have a whole bunch of logic to apply to a column, which can only be put in a customised function. So, the `apply()` function can also be used along with customised functions.

The example below uses a customised function that does exactly the same thing.

```
def add1(n):
    return n+1

df.number.apply(add1)
```

df.apply()

What if we need to use multiple columns in an `apply()` function? In fact, the `apply()` function can be used on a data frame object as well.

```
df = pd.DataFrame({'num1': [1, 2, 3, 4, 5],
                   'num2': [5, 4, 3, 2, 1]})

df.apply(lambda row: row['num1'] + row['num2'], axis=1)
```

```
3    6
4    6
dtype: int64
```

In the above example, the data frame is created with two numeric columns. Then, we use the `lambda` function to get each row with all the cells. After that, we can still use `['col_name']` to access the values. So, `row['num1'] + row['num2']` will return the sum of the values from both of the columns.

Very importantly, the parameter `axis=1` must be specified here, because the `apply()` function on a data frame object will be applied on row indices by default.

Similarly, we can also use a customised function.

```
def sum_cols(row):
    return row['num1'] + row['num2']

df.apply(sum_cols, axis=1)
```

df.explode()

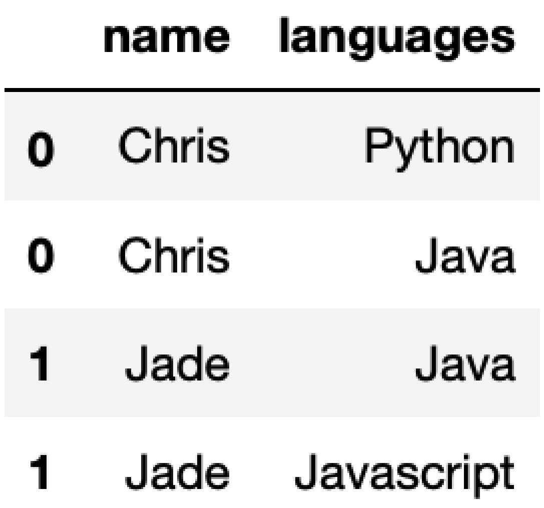
I used to use this function when dealing with JSON documents. Because of the style of JSON, sometimes we have one key with an array value. In this case, we can easily flatten the array.

```
df = pd.DataFrame([{
    'name': 'Chris',
    'languages': ['Python', 'Java']
},{
    'name': 'Jade',
    'languages': ['Java', 'Javascript']
}])
```

```
       name          languages
```

In this example, the languages that directly loaded from a JSON document are still arrays. Then, let's use `df.explode()` function to flatten it out.

```
df.explode('languages')
```



	name	languages
0	Chris	Python
0	Chris	Java
1	Jade	Java
1	Jade	Javascript

Data Aggregation

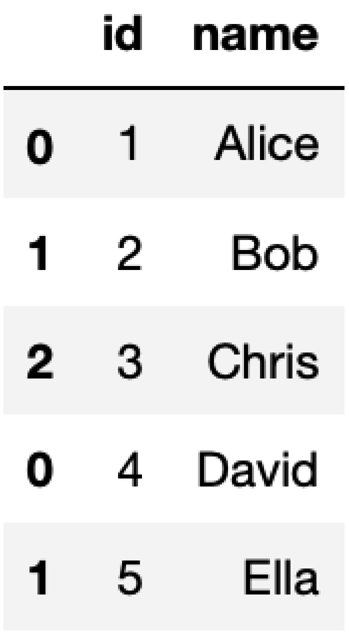
Photo by Martin Sanchez on Unsplash

Data Aggregation plays an important role in Data Analytics. Pandas provide many ways to perform data aggregation. Here I organised some functions that you must know as basics.

pd.concat()

```
df1 = pd.DataFrame({'id': [1, 2, 3],
                    'name': ['Alice', 'Bob', 'Chris']})
df2 = pd.DataFrame({'id': [4, 5],
                     'name': ['David', 'Ella']})

pd.concat([df1, df2])
```



	id	name
0	1	Alice
1	2	Bob
2	3	Chris
0	4	David
1	5	Ella

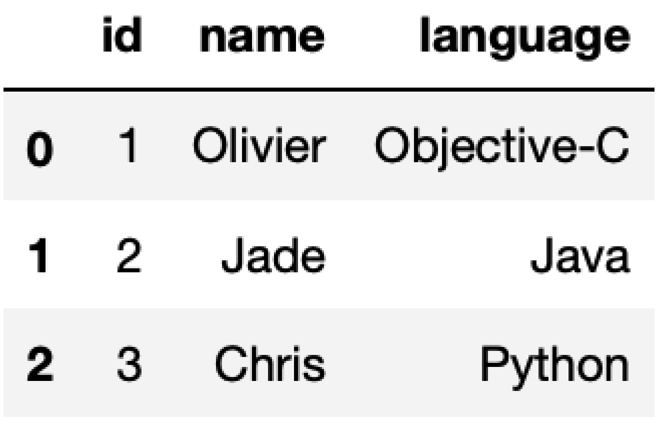
In this example, two data frames are created.

Then, we can use the `pd.concat()` function to concatenate them together as one.

This is another one that I used a lot in practice. If you have experience in SQL queries, this is just like joining two tables.

```
df1 = pd.DataFrame({'id': [1, 2, 3],
                    'name': ['Olivier', 'Jade', 'Chris']})
df2 = pd.DataFrame({'id': [1, 2, 3],
                     'language': ['Objective-C', 'Java', 'Python']})

pd.merge(df1, df2, on='id', how='inner')
```



	id	name	language
0	1	Olivier	Objective-C
1	2	Jade	Java
2	3	Chris	Python

As shown in the code block above, there are two data frames created with the "name" column and the "language" column respectively. Then, we can use this function to "join" them together. Please note that we need to specify which column that is used to join on with `on='id'` and specify how the two data frames are joined `how='inner'` .

df.groupby() and df.groupby().agg()

These two functions would be better demonstrated together, as the function `df.groupby()` cannot produce meaningful results by itself. It has to be used together with other functions that apply to the groups, which I believe `df.groupby().agg()` is the most common one.

```
df = pd.DataFrame({'id': [1, 2, 3, 4, 5],
                   'name': ['Alice', 'Bob', 'Chris', 'David',
'Ella'],
```

	name	age
language		
COBOL	1	68.0
Java	2	30.5
Python	2	26.0

In this example, we created a data frame with "name", "language" and "age".

Then, the data frame is grouped by the languages, and we count the number of names and averaging the ages among these persons. Well, the column headers don't make sense now, but we can use the `df.rename()` function to fix them.

pd.pivot_table()

For the above example, we can also implement it using the `pd.pivot_table()` function. That is, we need to specify the group keys and the measure values as follows (using the same data frame in the previous example):

```
pd.pivot_table(df,
               values=['name', 'age'],
               index=['language'],
               aggfunc={'name': 'count', 'age': ['min', 'max',
'mean']})
```

	age			name
	max	mean	min	count

Java	32.0	30.5	29.0	2
Python	31.0	26.0	21.0	2

Therefore, it is obvious that the `pd.pivot_table()` function is more convenient when we have multiple levels of group keys and measure values.

Summary

Photo by Aaron Burden on Unsplash

Indeed, the Pandas library of Python has a lot more functions that makes it such a flexible and powerful data analytics tool in Python. In this article, I just organised the basic ones that I believe are the most useful. If one can nail all of them, definitely can

Sign up for The Daily Pick

By Towards Data Science

Hands-on real-world examples, research, tutorials, and cutting-edge techniques delivered Monday to Thursday. Make learning your daily ritual. Take a look

Get this newsletter Create a free Medium account to get The Daily Pick in your inbox.

Python Pandas Data Analytics Data Science Data Analysis

- How to convert column types from … to …
- How to merge two data frames …
- and so on

In my opinion, all the newbies of Pandas should know at least the basic Pandas functions and practice them before starting to actually use it. There are not too many, but can help you to solve most of the regular problems.

Functions Lists

What are these functions? OK. In this article, I've organised all of these functions into different categories with separated tables. If you believe that you may already know some (If you have ever used Pandas you must know at least some of them), the tables below are **TD; DL** for you to check your knowledge before you read through.

Notations in the tables:

- **pd**: Pandas
- **df**: Data Frame Object
- **s**: Series Object (a column of Data Frame)

Data I/O

Data Preview

Download CSV View larger version

Data Cleansing

Data Transformation

Airtable Download CSV View larger version

Data Aggregation

Next, let me demonstrate these functions. I will give sample usage of them, but of course, I can't enumerate all the scenarios that these functions might be used. So, it is highly recommended to practice them by yourselves.

Data I/O

Photo by Road Trip with Raj on Unsplash

Reading and Writing from CSV, Excel and JSON document are used very similarly. Note that you can either read from a local path or an URL.