

20 Great Pandas Tricks For Data Science

20 Great Pandas tricks that you should use everyday

[Emmett Boudreau](#)



(Image By Author)

Introduction

Unless you are new to Data Science with Python or have some sort of device connected to your router that blocks all incoming packets related to Pandas, you've probably heard of Pandas. Pandas is the go-to library for Pythonic data organization, cleaning, and IPython display that has become the industry standard for reading and processing

data over the past 10 years. Not only does Pandas meet and exceed all expectations for the most part when it comes to working with data, it is also tied to Numpy and works incredibly well with it — cementing it further into the wonderful ecosystem of Pythonic data science packages.

While Pandas is popular, and very frequently used by most scientists working with Python, it is also still a pretty in-depth library with a lot of features. A lot of these features can be easily overlooked, but their value often cannot be overstated. Fortunately, today I have arranged 20 of the lesser-known features in the Pandas library that will help any data scientist on their way to mastering the package!

[notebook](#)

DataFrames

Data frames are the core feature of the Pandas library. Although data frames are a great functionality on their own, without great functions and operations to go with them they just end up being glorified dictionaries.

Fortunately, Pandas has you covered with these really cool ways to explore your data, clean your data, and adjust your data. For all of the examples below, I am going to be working with this data frame:

```
import pandas as pd
import numpy as np

df = pd.read_csv("weatherHistory.csv")

df.head(5)
```

]:

	Formatted Date	Summary	Precip Type	Temperature (C)	Apparent Temperature (C)	Humidity	Wind Speed (km/h)	Wind Bearing (degrees)	Visibility (km)	Loud Cover	Pressure (millibars)	Daily Summary
0	2006-04-01 00:00:00.000 +0200	Partly Cloudy	rain	9.472222	7.388889	0.89	14.1197	251.0	15.8263	0.0	1015.13	Partly cloudy throughout the day.
1	2006-04-01 01:00:00.000 +0200	Partly Cloudy	rain	9.355556	7.227778	0.86	14.2646	259.0	15.8263	0.0	1015.63	Partly cloudy throughout the day.
2	2006-04-01 02:00:00.000 +0200	Mostly Cloudy	rain	9.377778	9.377778	0.89	3.9284	204.0	14.9569	0.0	1015.94	Partly cloudy throughout the day.
3	2006-04-01 03:00:00.000 +0200	Partly Cloudy	rain	8.288889	5.944444	0.83	14.1036	269.0	15.8263	0.0	1016.41	Partly cloudy throughout the day.
4	2006-04-01 04:00:00.000 +0200	Mostly Cloudy	rain	8.755556	6.977778	0.83	11.0446	259.0	15.8263	0.0	1016.51	Partly cloudy throughout the day.

(Image By Author)

The data is historical weather data from dates as early as 1910 and as late as 2006

df.describe()

One of the most underrated features in Pandas is a simple function called describe(). Using the describe function on a data frame yields a very statistical result that will tell you all that you need to know about each column's values independently. This is a great way to understand where most of the data in a given column sits without only needing to consider the mean. Having an idea of the standard deviation, min-max values, and the mean will give a great inclination as to how much variance is in the data, as well. Using this function is a great way to get all of the following statistics for each column incredibly fast:

- mean
- count

- standard deviation
- 1st quartile
- 2nd quartile (median)
- 3rd quartile
- minimum value
- maximum value

`df.describe()`

`df.describe()`

6]:

	Temperature (C)	Apparent Temperature (C)	Humidity	Wind Speed (km/h)	Wind Bearing (degrees)	Visibility (km)	Loud Cover	Pressure (millibars)
count	96453.000000	96453.000000	96453.000000	96453.000000	96453.000000	96453.000000	96453.0	96453.000000
mean	11.932678	10.855029	0.734899	10.810640	187.509232	10.347325	0.0	1003.235956
std	9.551546	10.696847	0.195473	6.913571	107.383428	4.192123	0.0	116.969906
min	-21.822222	-27.716667	0.000000	0.000000	0.000000	0.000000	0.0	0.000000
25%	4.688889	2.311111	0.600000	5.828200	116.000000	8.339800	0.0	1011.900000
50%	12.000000	12.000000	0.780000	9.965900	180.000000	10.046400	0.0	1016.450000
75%	18.838889	18.838889	0.890000	14.135800	290.000000	14.812000	0.0	1021.090000
max	39.905556	39.344444	1.000000	63.852600	359.000000	16.100000	0.0	1046.380000

(Image By Author)

df.groupby()

The `groupby()` function is an awesome function to use to re-organize your observations based on categorical values or continuous ranks by numerical value. The function will simply put identical or similar values closest together.

`df.groupby(['Temperature (C)']).mean()`

```
df.groupby(['Temperature (C)']).mean()
```

5]:

	Apparent Temperature (C)	Humidity	Wind Speed (km/h)	Wind Bearing (degrees)	Visibility (km)	Loud Cover	Pressure (millibars)
Temperature (C)							
-21.822222	-21.822222	0.800000	3.075100	323.000000	1.368500	0.0	1033.660000
-21.111111	-21.111111	0.743333	3.756667	190.000000	2.146667	0.0	1033.400000
-20.783333	-20.783333	0.800000	4.427500	181.000000	1.787100	0.0	1032.330000
-20.555556	-23.238889	0.780000	4.830000	115.000000	2.737000	0.0	1032.950000
-20.277778	-25.072222	0.790000	5.667200	158.000000	1.803200	0.0	1033.530000
...
38.861111	37.327778	0.180000	17.388000	232.000000	9.982000	0.0	1009.890000
38.866667	37.950000	0.210000	12.751200	269.000000	9.982000	0.0	1011.000000
38.983333	36.962963	0.156667	12.354067	158.333333	9.982000	0.0	1012.536667
39.588889	37.600000	0.150000	12.203800	283.000000	10.352300	0.0	1010.240000
39.905556	37.538889	0.130000	23.586500	250.000000	9.982000	0.0	1007.550000

7574 rows × 7 columns

(Image By Author)

df.stack()

The stack function can be used to return both a data frame and series type with more inner-most levels. The new inner-most levels are created by pivoting the data-frame. The first parameter controls which level, or levels are stacked:

df.stack([0])

```
df.head(5)
```

]:

	Formatted Date	Summary	Precip Type	Temperature (C)	Apparent Temperature (C)	Humidity	Wind Speed (km/h)	Wind Bearing (degrees)	Visibility (km)	Loud Cover	Pressure (millibars)	Daily Summary
0	2006-04-01 00:00:00.000 +0200	Partly Cloudy	rain	9.472222	7.388889	0.89	14.1197	251.0	15.8263	0.0	1015.13	Partly cloudy throughout the day.
1	2006-04-01 01:00:00.000 +0200	Partly Cloudy	rain	9.355556	7.227778	0.86	14.2646	259.0	15.8263	0.0	1015.63	Partly cloudy throughout the day.
2	2006-04-01 02:00:00.000 +0200	Mostly Cloudy	rain	9.377778	9.377778	0.89	3.9284	204.0	14.9569	0.0	1015.94	Partly cloudy throughout the day.
3	2006-04-01 03:00:00.000 +0200	Partly Cloudy	rain	8.288889	5.944444	0.83	14.1036	269.0	15.8263	0.0	1016.41	Partly cloudy throughout the day.
4	2006-04-01 04:00:00.000 +0200	Mostly Cloudy	rain	8.755556	6.977778	0.83	11.0446	259.0	15.8263	0.0	1016.51	Partly cloudy throughout the day.

(Image By Author)

Stack really doesn't appear to have done much to this

data. This function is one that can only be felt whenever the table is mostly pivot-able.

df.apply()

The apply function is used to apply arithmetic or logical code across an entire data frame or series type using a Python function. Although this can certainly be applied to a data frame, I am only going to be applying it to a series because this data frame contains both strings and date-time datatypes.

```
df["Temperature (C)"].apply(np.sqrt)
```

df.head(5)

	Formatted Date	Summary	Precip Type	Temperature (C)	Apparent Temperature (C)	Humidity	Wind Speed (km/h)	Wind Bearing (degrees)	Visibility (km)	Loud Cover	Pressure (millibars)	Daily Summary
0	2006-04-01 00:00:00.000 +0200	Partly Cloudy	rain	9.472222	7.388889	0.89	14.1197	251.0	15.8263	0.0	1015.13	Partly cloudy throughout the day.
1	2006-04-01 01:00:00.000 +0200	Partly Cloudy	rain	9.355556	7.227778	0.86	14.2646	259.0	15.8263	0.0	1015.63	Partly cloudy throughout the day.
2	2006-04-01 02:00:00.000 +0200	Mostly Cloudy	rain	9.377778	9.377778	0.89	3.9284	204.0	14.9569	0.0	1015.94	Partly cloudy throughout the day.
3	2006-04-01 03:00:00.000 +0200	Partly Cloudy	rain	8.288889	5.944444	0.83	14.1036	269.0	15.8263	0.0	1016.41	Partly cloudy throughout the day.
4	2006-04-01 04:00:00.000 +0200	Mostly Cloudy	rain	8.755556	6.977778	0.83	11.0446	259.0	15.8263	0.0	1016.51	Partly cloudy throughout the day.

(Image By Author)

df.info()

The info function can be used on a data frame to provide information about the data frame usually relating to performance more than statistics. This is useful if you want to check your memory allocations or the data types of each series inside of a data frame.

`df.info()`

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 96453 entries, 0 to 96452
Data columns (total 12 columns):
Formatted Date          96453 non-null object
Summary                 96453 non-null object
Precip Type             95936 non-null object
Temperature (C)         96453 non-null float64
Apparent Temperature (C) 96453 non-null float64
Humidity                96453 non-null float64
Wind Speed (km/h)       96453 non-null float64
Wind Bearing (degrees)  96453 non-null float64
Visibility (km)         96453 non-null float64
Loud Cover              96453 non-null float64
Pressure (millibars)    96453 non-null float64
Daily Summary           96453 non-null object
dtypes: float64(8), object(4)
memory usage: 8.8+ MB
```

(Image By Author)

`df.query()`

Query is a Pandas function that allows a conditional mask to be applied across an entire data frame. The only significant difference between query and a typical conditional mask is that the query function can take a string that will be interpreted as a conditional statement, whereas a conditional mask will mask over the data with booleans and then return the conditions that are true.

```
df.query('Humidity < .89')
```

df.query('Humidity < .89')

7]:

	Formatted Date	Summary	Precip Type	Temperature (C)	Apparent Temperature (C)	Humidity	Wind Speed (km/h)	Wind Bearing (degrees)	Visibility (km)	Loud Cover	Pressure (millibars)	Daily Summary
1	2006-04-01 01:00:00.000 +0200	Partly Cloudy	rain	9.355556	7.227778	0.86	14.2646	259.0	15.8263	0.0	1015.63	Partly cloudy throughout the day.
3	2006-04-01 03:00:00.000 +0200	Partly Cloudy	rain	8.288889	5.944444	0.83	14.1036	269.0	15.8263	0.0	1016.41	Partly cloudy throughout the day.
4	2006-04-01 04:00:00.000 +0200	Mostly Cloudy	rain	8.755556	6.977778	0.83	11.0446	259.0	15.8263	0.0	1016.51	Partly cloudy throughout the day.
5	2006-04-01 05:00:00.000 +0200	Partly Cloudy	rain	9.222222	7.111111	0.85	13.9587	258.0	14.9569	0.0	1016.66	Partly cloudy throughout the day.
8	2006-04-01 08:00:00.000 +0200	Partly Cloudy	rain	10.822222	10.822222	0.82	11.3183	259.0	9.9820	0.0	1017.37	Partly cloudy throughout the day.
...
96448	2016-09-09 19:00:00.000 +0200	Partly Cloudy	rain	26.016667	26.016667	0.43	10.9963	31.0	16.1000	0.0	1014.36	Partly cloudy starting in the morning.
96449	2016-09-09 20:00:00.000 +0200	Partly Cloudy	rain	24.583333	24.583333	0.48	10.0947	20.0	15.5526	0.0	1015.16	Partly cloudy starting in the morning.
96450	2016-09-09 21:00:00.000 +0200	Partly Cloudy	rain	22.038889	22.038889	0.56	8.9838	30.0	16.1000	0.0	1015.66	Partly cloudy starting in the morning.
96451	2016-09-09 22:00:00.000 +0200	Partly Cloudy	rain	21.522222	21.522222	0.60	10.5294	20.0	16.1000	0.0	1015.95	Partly cloudy starting in the morning.
96452	2016-09-09 23:00:00.000 +0200	Partly Cloudy	rain	20.438889	20.438889	0.61	5.8765	39.0	15.5204	0.0	1016.16	Partly cloudy starting in the morning.

(Image By Author)

df.cumsum()

The cumsum function can be used to get the cumulative sum of both data frames and Pandas series types. This means that numbers will be summed in descending order adding to each-other continuously as they go.

df.cumsum()

Bad news.

The kernel appears to have died, unfortunately. Since my Jupyter kernel is feeling extra emotional today, I will be cutting her (it is a she) a break and instead just running the function on a single series.


```
df["Humidity"].cumsum()
```

```
2]: 0          0.89
    1          1.75
    2          2.64
    3          3.47
    4          4.30
    ...
    96448      70880.96
    96449      70881.44
    96450      70882.00
    96451      70882.60
    96452      70883.21
    Name: Humidity, Length: 96453, dtype: float64
```

(Image By Author)

df[conditional mask]

One of my favorite features inside of the Pandas library is easily the ability to mask data frames with conditions.

Although there is a way to do this with the `filter!()` method in my favorite language, Julia, this is one thing that I truly miss and wish could be brought into the language. Firstly, you need to come up with a condition, for this example we will use wind speed below 14.

```
our_mask = df["Wind Speed (km/h)"] < 14
```

Next, we will apply this mask by simply setting something equal to the booleans as an index:

```
seconddf = df[our_mask]
```

Now our data frame contains no wind speeds greater than

or equal to 14:

```
seconddf.head(5)
```

18]:

	Formatted Date	Summary	Precip Type	Temperature (C)	Apparent Temperature (C)	Humidity	Wind Speed (km/h)	Wind Bearing (degrees)	Visibility (km)	Loud Cover	Pressure (millibars)	Daily Summary
2	2006-04-01 02:00:00.000 +0200	Mostly Cloudy	rain	9.377778	9.377778	0.89	3.9284	204.0	14.9569	0.0	1015.94	Partly cloudy throughout the day.
4	2006-04-01 04:00:00.000 +0200	Mostly Cloudy	rain	8.755556	6.977778	0.83	11.0446	259.0	15.8263	0.0	1016.51	Partly cloudy throughout the day.
5	2006-04-01 05:00:00.000 +0200	Partly Cloudy	rain	9.222222	7.111111	0.85	13.9587	258.0	14.9569	0.0	1016.66	Partly cloudy throughout the day.
6	2006-04-01 06:00:00.000 +0200	Partly Cloudy	rain	7.733333	5.522222	0.95	12.3648	259.0	9.9820	0.0	1016.72	Partly cloudy throughout the day.
8	2006-04-01 08:00:00.000 +0200	Partly Cloudy	rain	10.822222	10.822222	0.82	11.3183	259.0	9.9820	0.0	1017.37	Partly cloudy throughout the day.

(Image By Author)

df.melt()

Melting is the opposite of the pivot function, which will put your data into a sideways format. This is useful if you want to change up what will be the observations and what will be the features. It is important to note, however, that this might not work as well as you expected without either providing parameters or changing up the structure of your data frame in some instances. One example of exactly that is the data frame I am working with:

```
df = df.melt()
```

```
df.melt()
```

]:

	variable	value
0	Formatted Date	2006-04-01 00:00:00.000 +0200
1	Formatted Date	2006-04-01 01:00:00.000 +0200
2	Formatted Date	2006-04-01 02:00:00.000 +0200
3	Formatted Date	2006-04-01 03:00:00.000 +0200
4	Formatted Date	2006-04-01 04:00:00.000 +0200
...
1157431	Daily Summary	Partly cloudy starting in the morning.
1157432	Daily Summary	Partly cloudy starting in the morning.
1157433	Daily Summary	Partly cloudy starting in the morning.
1157434	Daily Summary	Partly cloudy starting in the morning.
1157435	Daily Summary	Partly cloudy starting in the morning.

1157436 rows × 2 columns

(Image By Author)

df.explode()

Have you ever ran into data that looks like this for some reason? :

```
data = pd.DataFrame({"A": ['A', 'B', 'C'], "B": [1, 2, [1, 2, 3, 4]]})
data.head()
```

3]:

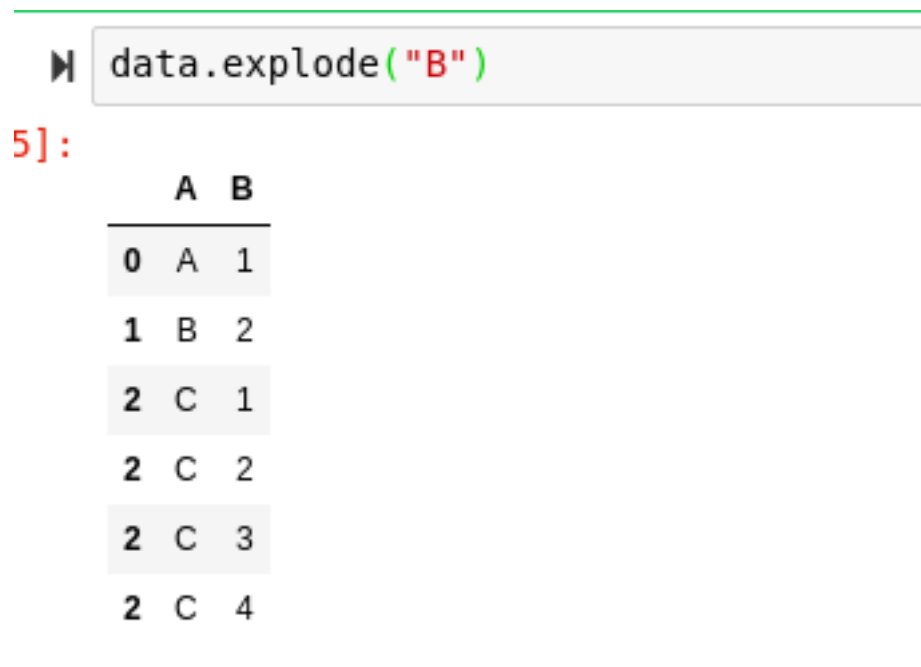
	A	B
0	A	1
1	B	2
2	C	[1, 2, 3, 4]

(Image By Author)

All of the other values are accurate observations with a single integer except for the observations that correspond

to :A on the data frame. Fortunately, we can simply run `explode` on our data frame to correct this. `explode` will “explode” all of the iterable sets inside of the data frame and put them into individual observations.

```
data.explode("B")
```



The image shows a Jupyter Notebook interface. At the top, a code cell contains the command `data.explode("B")`. Below the code cell, the output is displayed, starting with a red prompt character `5]:`. The output is a DataFrame with three columns: an index column, column 'A', and column 'B'. The data is as follows:

	A	B
0	A	1
1	B	2
2	C	1
2	C	2
2	C	3
2	C	4

(Image By Author)

df.nunique()

We have a data frame with a lot of categories and we want to approach a solution to a classification problem. The only issue is that we don't want to use a model that will be prone to over-fitting if we have so few categories that the model ends up being overkill. This would be quite a common scenario in the typical everyday life of a data scientist. Fortunately, the solution to this problem is really simple — a count of all unique values is needed. Even more fortunately, Pandas makes doing so incredibly easy

with the `nunique` function!

`df.nunique()`

```
df.nunique()
|: Formatted Date          96429
   Summary                 27
   Precip Type              2
   Temperature (C)         7574
   Apparent Temperature (C) 8984
   Humidity                 90
   Wind Speed (km/h)       2484
   Wind Bearing (degrees)   360
   Visibility (km)          949
   Loud Cover               1
   Pressure (millibars)     4979
   Daily Summary           214
   dtype: int64
```

(Image By Author)

This will provide a comprehensive list of all of the features in our data frame and their corresponding unique value counts.

`df.infer_objects()`

Another thing that you could potentially run into when working with dirty data is types that are initially presumed to be something that they're not. A great potential time-saver to changing these datatypes inside of your data frame is to use `df.infer_objects()`. The `infer_objects` function takes an educated guess at what datatype each column should be and sets the whole series to that type.

df.infer_objects()

df.infer_objects()

'6]':

	Formatted Date	Summary	Precip Type	Temperature (C)	Apparent Temperature (C)	Humidity	Wind Speed (km/h)	Wind Bearing (degrees)	Visibility (km)	Loud Cover	Pressure (millibars)	Sum
0	2006-04-01 00:00:00.000 +0200	Partly Cloudy	rain	9.472222	7.388889	0.89	14.1197	251.0	15.8263	0.0	1015.13	Partly c througho
1	2006-04-01 01:00:00.000 +0200	Partly Cloudy	rain	9.355556	7.227778	0.86	14.2646	259.0	15.8263	0.0	1015.63	Partly c througho
2	2006-04-01 02:00:00.000 +0200	Mostly Cloudy	rain	9.377778	9.377778	0.89	3.9284	204.0	14.9569	0.0	1015.94	Partly c througho
3	2006-04-01 03:00:00.000 +0200	Partly Cloudy	rain	8.288889	5.944444	0.83	14.1036	269.0	15.8263	0.0	1016.41	Partly c througho
4	2006-04-01 04:00:00.000 +0200	Mostly Cloudy	rain	8.755556	6.977778	0.83	11.0446	259.0	15.8263	0.0	1016.51	Partly c througho
...
95448	2016-09-09 19:00:00.000	Partly	rain	26.016667	26.016667	0.43	10.9063	31.0	16.1000	0.0	1014.36	Partly c starting

(Image By Author)

df.memory_usage()

Memory usage is a function for which the return most might have already assumed. While we did get a broad idea of the memory usage within our data frame using the info() function, memory_usage is far more comprehensive, and will allow us to figure out just which columns are consuming the most of our memory.

df.memory_usage()

```
df.memory_usage()
```

```
7]: Index          128
    Formatted Date  771624
    Summary        771624
    Precip Type    771624
    Temperature (C) 771624
    Apparent Temperature (C) 771624
    Humidity        771624
    Wind Speed (km/h) 771624
    Wind Bearing (degrees) 771624
    Visibility (km)  771624
    Loud Cover      771624
    Pressure (millibars) 771624
    Daily Summary   771624
    dtype: int64
```

(Image By Author)

df.select_dtypes()

The `select_dtypes` function can be used to extract certain columns from a data frame depending on type. A great example of how this could be useful is for something like `apply`, where we wanted to apply `np.sqrt()` to the entire data frame, but we were only able to do so with a single series because of the strings in our data frame.

```
df.select_dtypes(float).apply(np.sqrt)
```

```
df.select_dtypes(float).apply(np.sqrt)
```

]:

	Temperature (C)	Apparent Temperature (C)	Humidity	Wind Speed (km/h)	Wind Bearing (degrees)	Visibility (km)	Loud Cover	Pressure (millibars)
0	3.077698	2.718251	0.943398	3.757619	15.842980	3.978228	0.0	31.861105
1	3.058685	2.688453	0.927362	3.776851	16.093477	3.978228	0.0	31.868950
2	3.062316	3.062316	0.943398	1.982019	14.282857	3.867415	0.0	31.873814
3	2.879043	2.438123	0.911043	3.755476	16.401219	3.978228	0.0	31.881186
4	2.958979	2.641548	0.911043	3.323342	16.093477	3.978228	0.0	31.882754
...
96448	5.100654	5.100654	0.655744	3.316067	5.567764	4.012481	0.0	31.849019
96449	4.958158	4.958158	0.692820	3.177216	4.472136	3.943678	0.0	31.861576
96450	4.694559	4.694559	0.748331	2.997299	5.477226	4.012481	0.0	31.869421
96451	4.639205	4.639205	0.774597	3.244904	4.472136	4.012481	0.0	31.873971
96452	4.520939	4.520939	0.781025	2.424149	6.244998	3.939594	0.0	31.877265

96453 rows × 8 columns

(Image By Author)

df.iterrows()

The iterrows() function is a simple generator type that will generate the a new array of iterations — one for each column. This is great if you need to compress all of your rows into a single iterable object:

```
rowlist = df.iterrows()
```

Now we could iterate over this generator:

```
n [*]: for i in rowlist:
        print(i)

(0, Formatted Date          2006-04-01 00:00:00.000 +0200
Summary                      Partly Cloudy
Precip Type                  rain
Temperature (C)              9.47222
Apparent Temperature (C)     7.38889
Humidity                     0.89
Wind Speed (km/h)           14.1197
Wind Bearing (degrees)       251
Visibility (km)              15.8263
Loud Cover                   0
Pressure (millibars)         1015.13
Daily Summary                Partly cloudy throughout the day.
Name: 0, dtype: object)
(1, Formatted Date          2006-04-01 01:00:00.000 +0200
Summary                      Partly Cloudy
Precip Type                  rain
Temperature (C)              9.35556
Apparent Temperature (C)     7.22778
Humidity                     0.86
```

(Image By Author)

profiling

It's hard to mention Pandas data analysis without talking about Pandas profiling. Although it is going to require another library, it is certainly one of the greatest tools that Pythonic data scientists have at their disposal for a quick overview of data with a single function call. You can check it out here:

[pandas-profiling/pandas-profiling](#)

[Documentation | Slack | Stack Overflow](#)
[Generates profile reports from a pandas DataFrame. The pandas df.describe\(\)...](#)

In order to use it, simply call

```
df.profile_report()
```

This will provide a thorough examination of your data in a convenient HTML leaflet that is easy to digest and perfectly compatible with IPython!

Series

The Pandas series type, though likely not nearly as integral to Pandas as the data frame certainly has its importance. As opposed to the features of a normal list, Pandas series come with a whole lot of different options

that simply aren't supported anywhere else. These typically include the typical quality of life tools that most data scientists have come to expect from working in Python.

Series.isin()

The `isin` function will return a conditional mask that will return true if a given observation in the series is inside of the list of values it is called on. For example, we could create a mask for whether or not each value in our `:PrecipType` column is "rain":

```
df["Precip Type"].isin(["rain"])
```

```
▶ df["Precip Type"].isin(["rain"])
1]: 0         True
     1         True
     2         True
     3         True
     4         True
     ...
    96448      True
    96449      True
    96450      True
    96451      True
    96452      True
     Name: Precip Type, Length: 96453, dtype: bool
```

(Image By Author)

Series.where()

Where will apply a condition to a given series and return values that meet said condition. The `where` function is

pretty much directly named after the SQL equivalent of this functions purpose.

`df["Summary"].where(df["Summary"] != "Mostly Cloudy")`

```
df["Summary"].where(df["Summary"] != "Mostly Cloudy")
]: 0      Partly Cloudy
    1      Partly Cloudy
    2             NaN
    3      Partly Cloudy
    4             NaN
    ...
    96448     Partly Cloudy
    96449     Partly Cloudy
    96450     Partly Cloudy
    96451     Partly Cloudy
    96452     Partly Cloudy
    Name: Summary, Length: 96453, dtype: object
```

(Image By Author)

Series Loc and ILoc

If you aren't familiar with `iloc` and `loc`, then you probably should start practicing — because these two are very important! The rule of thumb is that `loc` is used for labels, and `iloc` is used for integer-based indexing. Firstly, let's try using `iloc` on a series type and see what we get in return.

`df["Precip Type"].iloc[2]`

(Image By Author)

We are getting "rain" as a return here because it is the 2nd index on our "Precip Type" column. Let's take

another look at the data frame head:

df.head(5)

3]:

	Formatted Date	Summary	Precip Type	Temperature (C)	Apparent Temperature (C)	Humidity	Wind Speed (km/h)	Wind Bearing (degrees)	Visibility (km)	Loud Cover	Pressure (millibars)	Daily Summary
0	2006-04-01 00:00:00.000 +0200	Partly Cloudy	rain	9.472222	7.388889	0.89	14.1197	251.0	15.8263	0.0	1015.13	Partly cloudy throughout the day.
1	2006-04-01 01:00:00.000 +0200	Partly Cloudy	rain	9.355556	7.227778	0.86	14.2646	259.0	15.8263	0.0	1015.63	Partly cloudy throughout the day.
2	2006-04-01 02:00:00.000 +0200	Mostly Cloudy	rain	9.377778	9.377778	0.89	3.9284	204.0	14.9569	0.0	1015.94	Partly cloudy throughout the day.
3	2006-04-01 03:00:00.000 +0200	Partly Cloudy	rain	8.288889	5.944444	0.83	14.1036	269.0	15.8263	0.0	1016.41	Partly cloudy throughout the day.
4	2006-04-01 04:00:00.000 +0200	Mostly Cloudy	rain	8.755556	6.977778	0.83	11.0446	259.0	15.8263	0.0	1016.51	Partly cloudy throughout the day.

(Image By Author)

Alternatively, we could use loc instead of iloc to get a value based on the label it holds in the index. Consider the following data frame:

data = pd.DataFrame({"Age": [25, 35,45]}, index =

data.head(5)

]:

	Age
Corgi	25
Maltese	35
Pug	45

(Image By Author)

Now we could call loc on any given index and receive the entire observation set with each feature:

data.loc["Corgi"]

I am just now realizing that 25 is very old for a dog.

I actually am not certain what datatype this will return, so I am curious to check what type this is, as well:

Who would have thought?

Series.rank()

The rank function can be used to provide numerical ranks to values usually based on their position relative to the minimum and maximum values in the data. This is useful for a series of different reasons, but a great application of ranks can be seen in the Wilcox Rank Sum test, a statistical test that uses ranks to return probability.

```
df["Humidity"].rank()
```



Conclusion

Needless to say, Pandas is an amazing library for managing and cleaning up large datasets. Pandas makes a lot of things a lot easier that are frankly quite difficult in other languages. Although there are a lot of great packages in other languages that seek to accomplish similar tasks, most of them make me miss the brilliant ways that Pandas approaches complicated problems.

Using these functions, it is almost guaranteed that most problems a scientist may run into can be solved. Although Pandas is a very in-depth library, it is great to always have a heads up on great functions that can speed up your work and get you to the fun stuff a little bit sooner!