Member-only story

How to use loc and iloc for selecting data in Pandas

Pandas tips and tricks to help you get started with data analysis

B. Chen · Follow

Published in Towards Data Science · 8 min read · Mar 18, 2021

Photo by Clay Banks on Unsplash

When it comes to select data on a DataFrame, Pandas `loc` and `iloc` are two

top favorites. They are quick, fast, easy to read, and sometimes interchangeable.

In this article, we'll explore the differences between `loc` and `iloc`, take a looks at their similarities, and check how to perform data selection with them. We will go over the following topics:

1. Differences between `loc` and `iloc`
2. Selecting via a single value
3. Selecting via a list of values
4. Selecting a range of data via slice
5. Selecting via conditions and callable
6. `loc` and `iloc` are interchangeable when labels are 0-based integers

Please check out Notebook for the source code.

1. Differences between `loc` and `iloc`

The main distinction between `loc` and `iloc` is:

- `loc` is label-based, which means that you have to specify rows and columns based on their row and column **labels.**
- `iloc` is integer position-based, so you have to specify rows and columns by their **integer position values** (0-based integer position).

Here are some differences and similarities between `loc` and `iloc` :

	loc	iloc
A value	A single label or integer e.g. `loc[A]` or `loc[1]`	A single integer e.g. `iloc[1]`
A list	A list of labels e.g. `loc[[A, B]]`	A list of integers e.g. `iloc[[1,2,3]]`
Slicing	e.g. `loc[A:B]`, A and B are included	e.g. `iloc[n:m]`, n is included, m is excluded
Conditions	A bool Series or list	A bool list
Callable function	`loc[lambda x: x[2]]`	`iloc[lambda x: x[2]]`

Differences and Similarities between loc and iloc (image by author)

For demonstration, we create a DataFrame and load it with the **Day** column as the index.

```
df = pd.read_csv('data/data.csv', index_col=['Day'])
```

Day	Weather	Temperature	Wind	Humidity
Mon	Sunny	12.79	13	30
Tue	Sunny	19.67	28	96
Wed	Sunny	17.51	16	20
Thu	Cloudy	14.44	11	22
Fri	Shower	10.51	26	79
Sat	Shower	11.07	27	62
Sun	Sunny	17.50	20	10

image by author

2. Selecting via a single value

Both `loc` and `iloc` allow input to be a single value. We can use the following syntax for data selection:

- `loc[row_label, column_label]`
- `iloc[row_position, column_position]`

For example, let's say we would like to retrieve Friday's temperature value.

With `loc`, we can pass the row label `'Fri'` and the column label `'Temperature'`.

```
# To get Friday's temperature
>>> df.loc['Fri', 'Temperature']

10.51
```

The equivalent `iloc` statement should take the row number `4` and the column number `1`.

```
# The equivalent `iloc` statement
>>> df.iloc[4, 1]

10.51
```

We can also use `:` to return all data. For example, to get all rows:

```
# To get all rows
>>> df.loc[:, 'Temperature']

Day
Mon    12.79
Tue    19.67
Wed    17.51
Thu    14.44
Fri    10.51
Sat    11.07
Sun    17.50
Name: Temperature, dtype: float64

# The equivalent `iloc` statement
>>> df.iloc[:, 1]
```

And to get all columns:

```
# To get all columns
>>> df.loc['Fri', :]

Weather        Shower
Temperature     10.51
Wind               26
Humidity           79
Name: Fri, dtype: object

# The equivalent `iloc` statement
>>> df.iloc[4, :]
```

Note that the above 2 outputs are **Series**. `loc` and `iloc` will return a **Series** when the result is 1-dimensional data.

3. Selecting via a list of values

We can pass a list of labels to `loc` to select multiple rows or columns:

```
# Multiple rows
>>> df.loc[['Thu', 'Fri'], 'Temperature']

Day
Thu    14.44
Fri    10.51
Name: Temperature, dtype: float64
```

```
# Multiple columns
>>> df.loc['Fri', ['Temperature', 'Wind']]

Temperature    10.51
Wind              26
Name: Fri, dtype: object
```

Similarly, a list of integer values can be passed to `iloc` to select multiple rows or columns. Here are the equivalent statements using `iloc`:

```
>>> df.iloc[[3, 4], 1]

Day
Thu    14.44
Fri    10.51
Name: Temperature, dtype: float64

>>> df.iloc[4, [1, 2]]

Temperature    10.51
Wind              26
Name: Fri, dtype: object
```

All the above outputs are **Series** because their results are 1-dimensional data.

The output will be a **DataFrame** when the result is 2-dimensional data, for

example, to access multiple rows and columns

```
# Multiple rows and columns
rows = ['Thu', 'Fri']
cols=['Temperature','Wind']

df.loc[rows, cols]
```

Day	Temperature	Wind
Thu	14.44	11
Fri	10.51	26

The equivalent `iloc` statement is:

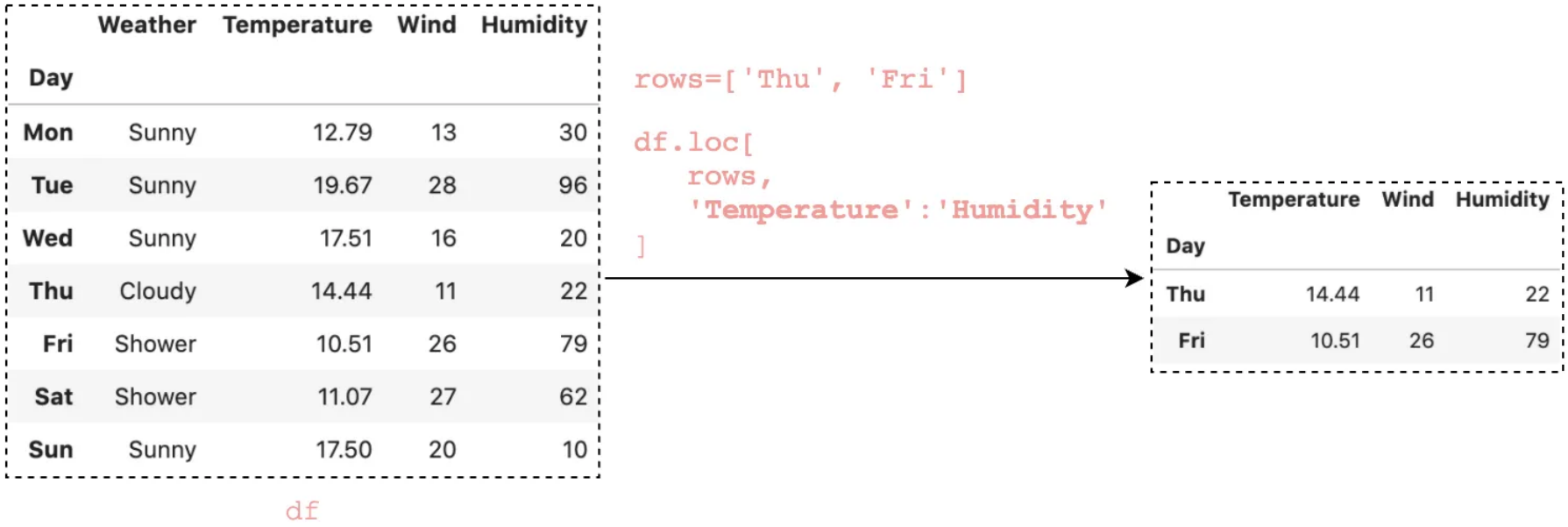
```
rows = [3, 4]
cols = [1, 2]

df.iloc[rows, cols]
```

4. Selecting a range of data via slice

Slice (written as `start:stop:step`) is a powerful technique that allows selecting a range of data. It is very useful when we want to select everything in between two items.

`loc` with slice

With `loc`, we can use the syntax `A:B` to select data from label **A** to label **B** (Both **A** and **B** are included):



Day	Weather	Temperature	Wind	Humidity
Mon	Sunny	12.79	13	30
Tue	Sunny	19.67	28	96
Wed	Sunny	17.51	16	20
Thu	Cloudy	14.44	11	22
Fri	Shower	10.51	26	79
Sat	Shower	11.07	27	62
Sun	Sunny	17.50	20	10

Day	Temperature	Wind	Humidity
Thu	14.44	11	22
Fri	10.51	26	79

image by author

```
# Slicing row labels
cols = ['Temperature', 'Wind']

df.loc['Mon':'Thu', cols]
```

Day	Weather	Temperature	Wind	Humidity
Mon	Sunny	12.79	13	30
Tue	Sunny	19.67	28	96
Wed	Sunny	17.51	16	20
Thu	Cloudy	14.44	11	22
Fri	Shower	10.51	26	79
Sat	Shower	11.07	27	62
Sun	Sunny	17.50	20	10

df

```
cols = ['Temperature', 'Wind']

df.loc['Mon':'Thu', cols]
```

Day	Temperature	Wind
Mon	12.79	13
Tue	19.67	28
Wed	17.51	16
Thu	14.44	11

image by author

We can use the syntax `A:B:S` to select data from label **A** to label **B** with step size **S** (Both **A** and **B** are included):

```
# Slicing with step
df.loc['Mon':'Fri':2 , :]
```

Day	Weather	Temperature	Wind	Humidity
Mon	Sunny	12.79	13	30
Tue	Sunny	19.67	28	96
Wed	Sunny	17.51	16	20
Thu	Cloudy	14.44	11	22
Fri	Shower	10.51	26	79
Sat	Shower	11.07	27	62
Sun	Sunny	17.50	20	10

df

df.loc['Mon':'Fri':2, :]

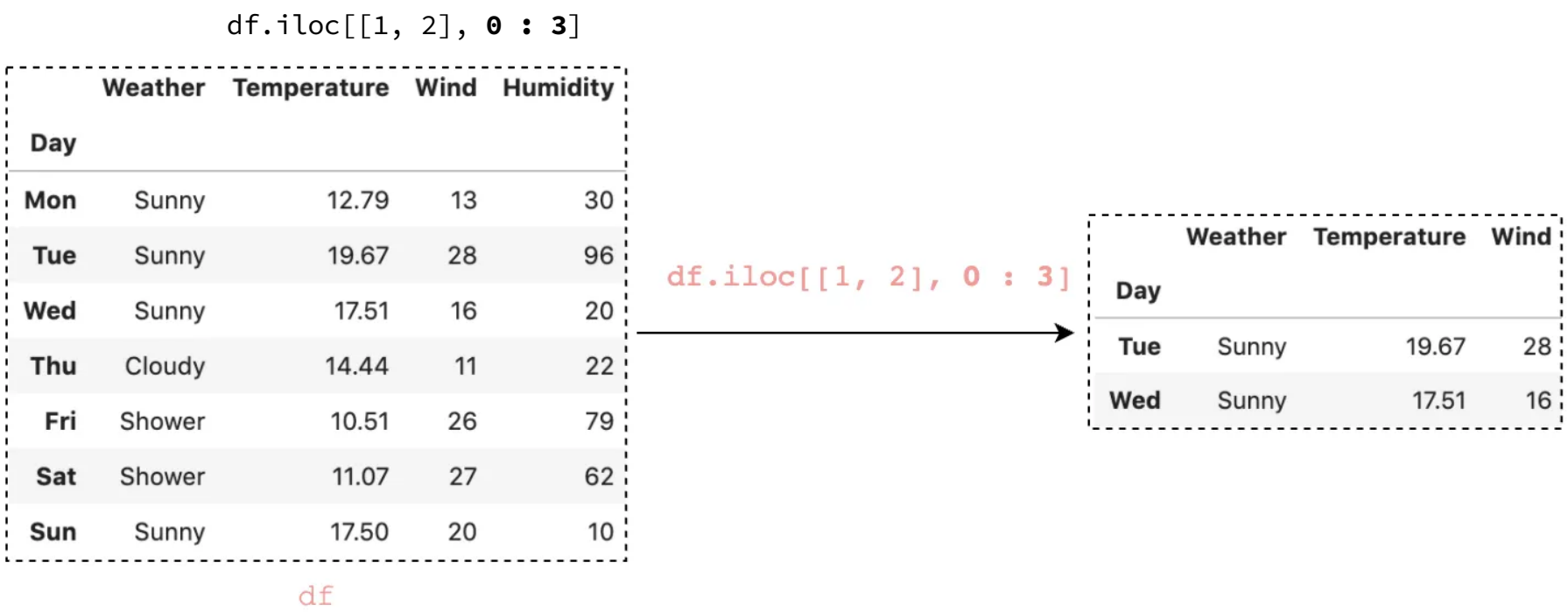
Day	Weather	Temperature	Wind	Humidity
Mon	Sunny	12.79	13	30
Wed	Sunny	17.51	16	20
Fri	Shower	10.51	26	79

image by author

`iloc` with slice

With `iloc`, we can also use the syntax `n:m` to select data from position **n** (included) to position **m** (excluded). However, the main difference here is that the endpoint (**m**) is excluded from the `iloc` result.

For example, selecting columns from position 0 up to 3 (excluded):



Day	Weather	Temperature	Wind	Humidity
Mon	Sunny	12.79	13	30
Tue	Sunny	19.67	28	96
Wed	Sunny	17.51	16	20
Thu	Cloudy	14.44	11	22
Fri	Shower	10.51	26	79
Sat	Shower	11.07	27	62
Sun	Sunny	17.50	20	10

Day	Weather	Temperature	Wind
Tue	Sunny	19.67	28
Wed	Sunny	17.51	16

image by author

Similarly, we can use the syntax `n:m:s` to select data from position **n** (included) to position **m** (excluded) with step size **s**. Notes that the endpoint **m** is excluded.

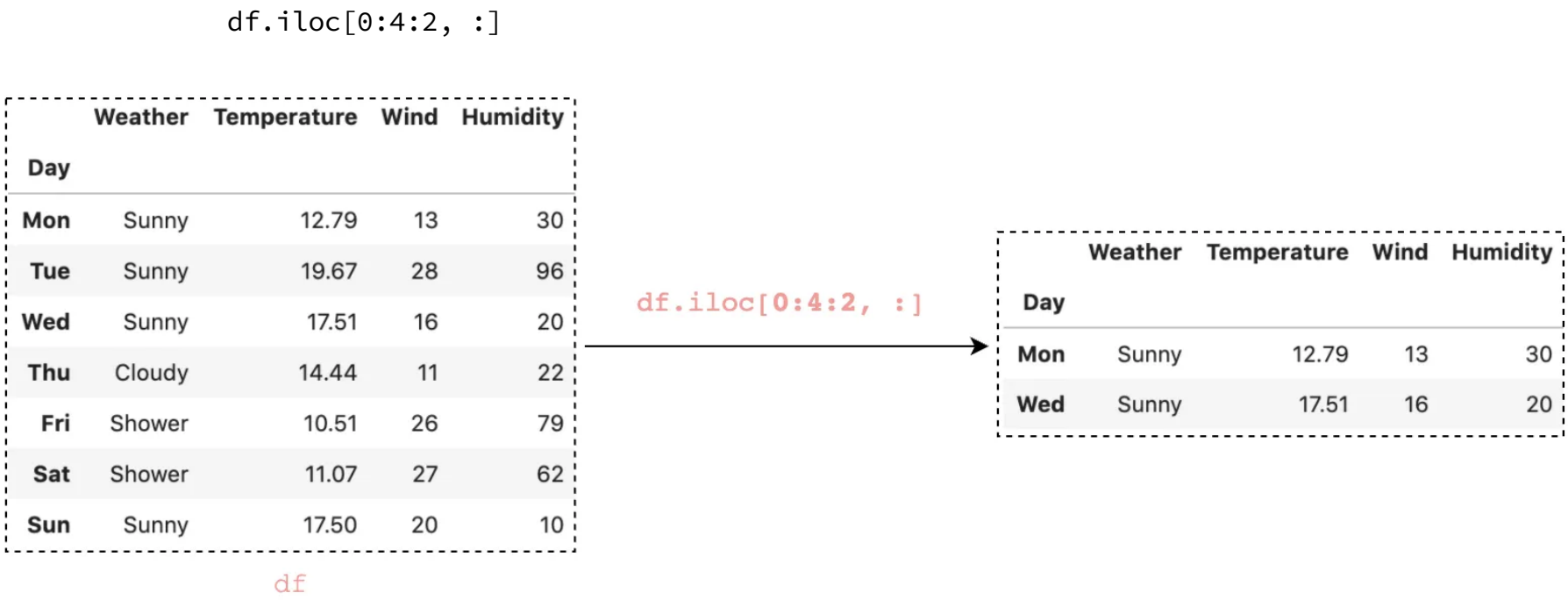


image by author

5. Selecting via conditions and callable

Conditions

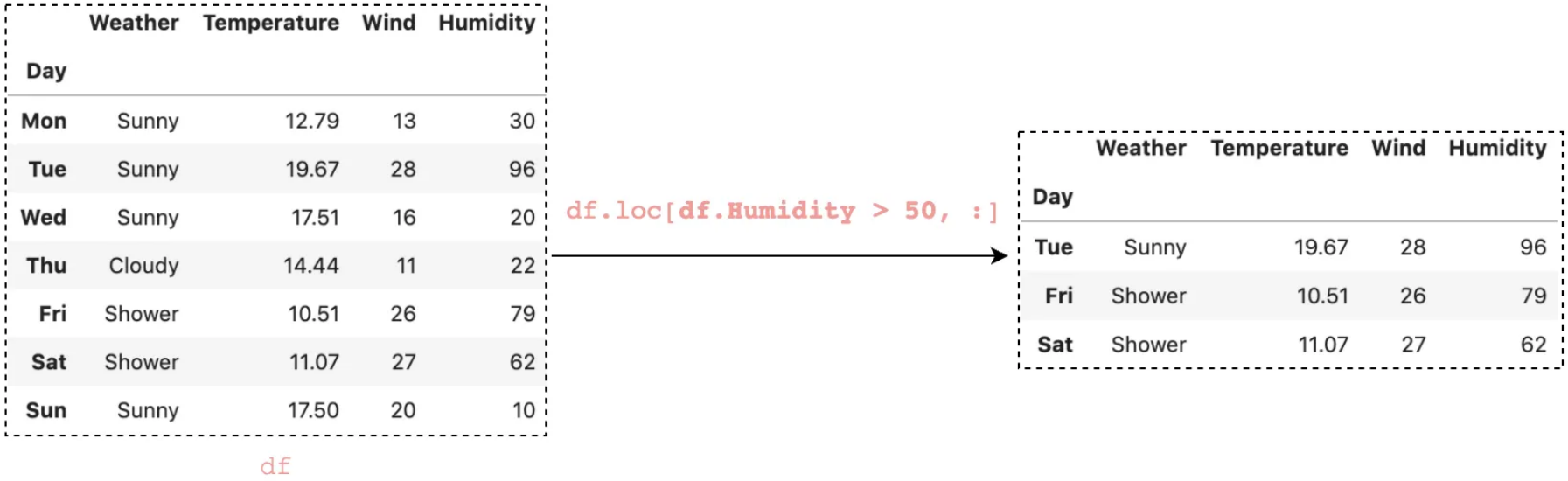
`loc` with conditions

Often we would like to filter the data based on conditions. For example, we

may need to find the rows where humidity is greater than 50.

With `loc`, we just need to pass the condition to the `loc` statement.

```
# One condition
df.loc[df.Humidity > 50, :]
```



Day	Weather	Temperature	Wind	Humidity
Mon	Sunny	12.79	13	30
Tue	Sunny	19.67	28	96
Wed	Sunny	17.51	16	20
Thu	Cloudy	14.44	11	22
Fri	Shower	10.51	26	79
Sat	Shower	11.07	27	62
Sun	Sunny	17.50	20	10

df

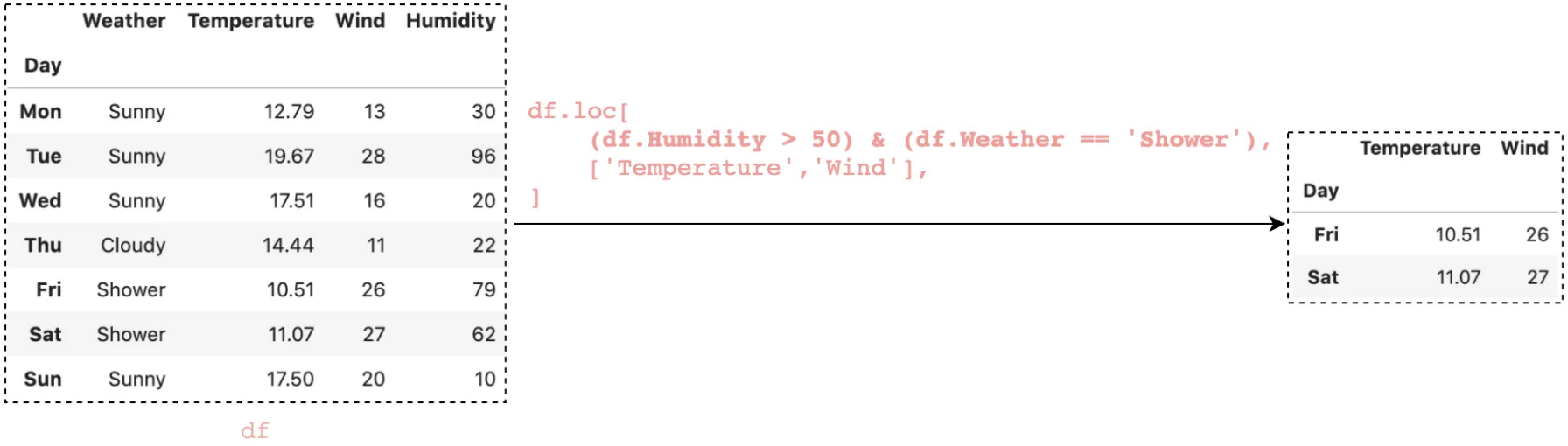
Day	Weather	Temperature	Wind	Humidity
Tue	Sunny	19.67	28	96
Fri	Shower	10.51	26	79
Sat	Shower	11.07	27	62

image by author

Sometimes, we may need to use multiple conditions to filter our data. For example, find all the rows where humidity is more than 50 and the weather

is Shower:

```
## multiple conditions
df.loc[
    (df.Humidity > 50) & (df.Weather == 'Shower'),
    ['Temperature','Wind'],
]
```



Day	Weather	Temperature	Wind	Humidity
Mon	Sunny	12.79	13	30
Tue	Sunny	19.67	28	96
Wed	Sunny	17.51	16	20
Thu	Cloudy	14.44	11	22
Fri	Shower	10.51	26	79
Sat	Shower	11.07	27	62
Sun	Sunny	17.50	20	10

Day	Temperature	Wind
Fri	10.51	26
Sat	11.07	27

image by author

`iloc` with conditions

For `iloc`, we will get a **ValueError** if pass the condition straight into the

statement:

```
# Getting ValueError
df.iloc[df.Humidity > 50, :]
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
~/anaconda3/envs/tf-tutorial/lib/python3.7/site-packages/pandas/core/indexing.py
    701             try:
--> 702                 self._validate_key(k, i)
    703             except ValueError as err:

~/anaconda3/envs/tf-tutorial/lib/python3.7/site-packages/pandas/core/indexing.py
   1343                 raise ValueError(
-> 1344                     "iLocation based boolean indexing cannot use "
   1345                     "an indexable as a mask"

ValueError: iLocation based boolean indexing cannot use an indexable as a mask
```

image by author

We get the error because `iloc` cannot accept a boolean Series. It only accepts a boolean list. We can use the `list()` function to convert a Series into a boolean list.

```
# Single condition
df.iloc[list(df.Humidity > 50)]
```

Similarly, we can use `list()` to convert the output of multiple conditions into a boolean list:

```
## multiple conditions
df.iloc[
    list((df.Humidity > 50) & (df.Weather == 'Shower')),
    :,
]
```

Callable function

`loc` with callable

`loc` accepts a **callable** as an indexer. The callable must be a function with one argument that returns valid output for indexing.

For example to select columns

Day	Weather	Temperature	Wind	Humidity
Mon	Sunny	12.79	13	30
Tue	Sunny	19.67	28	96
Wed	Sunny	17.51	16	20
Thu	Cloudy	14.44	11	22
Fri	Shower	10.51	26	79
Sat	Shower	11.07	27	62
Sun	Sunny	17.50	20	10

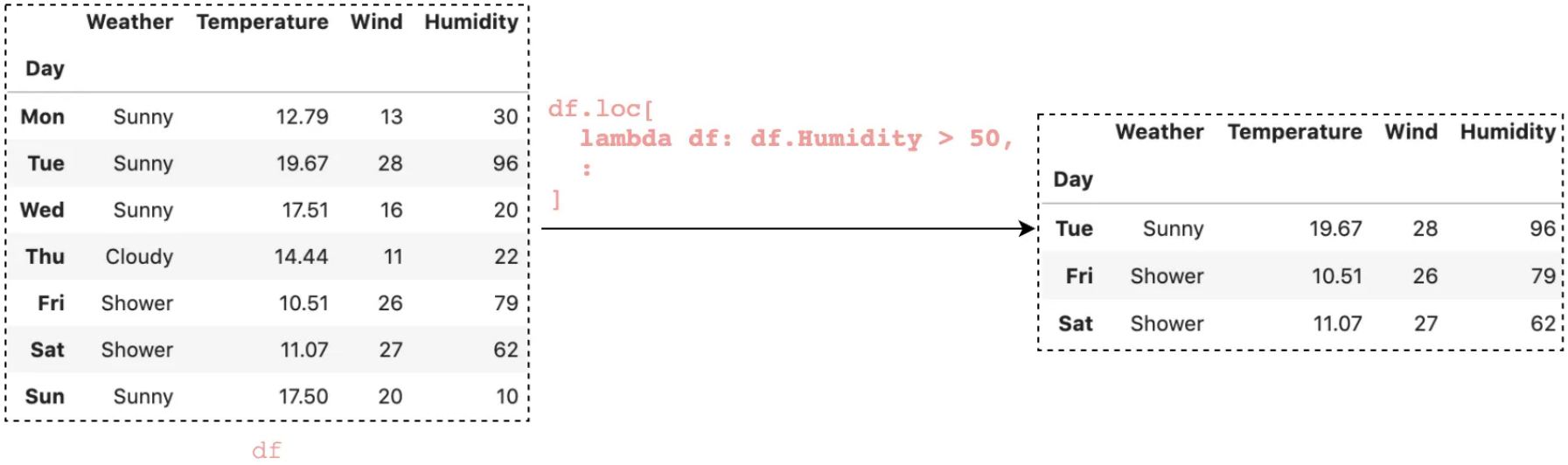
df

```
df.loc[
  :,
  lambda df: ['Humidity', 'Wind']
]
```

Day	Humidity	Wind
Mon	30	13
Tue	96	28
Wed	20	16
Thu	22	11
Fri	79	26
Sat	62	27
Sun	10	20

And to filter data with a callable:

```
# With condition
df.loc[lambda df: df.Humidity > 50, :]
```



Day	Weather	Temperature	Wind	Humidity
Mon	Sunny	12.79	13	30
Tue	Sunny	19.67	28	96
Wed	Sunny	17.51	16	20
Thu	Cloudy	14.44	11	22
Fri	Shower	10.51	26	79
Sat	Shower	11.07	27	62
Sun	Sunny	17.50	20	10

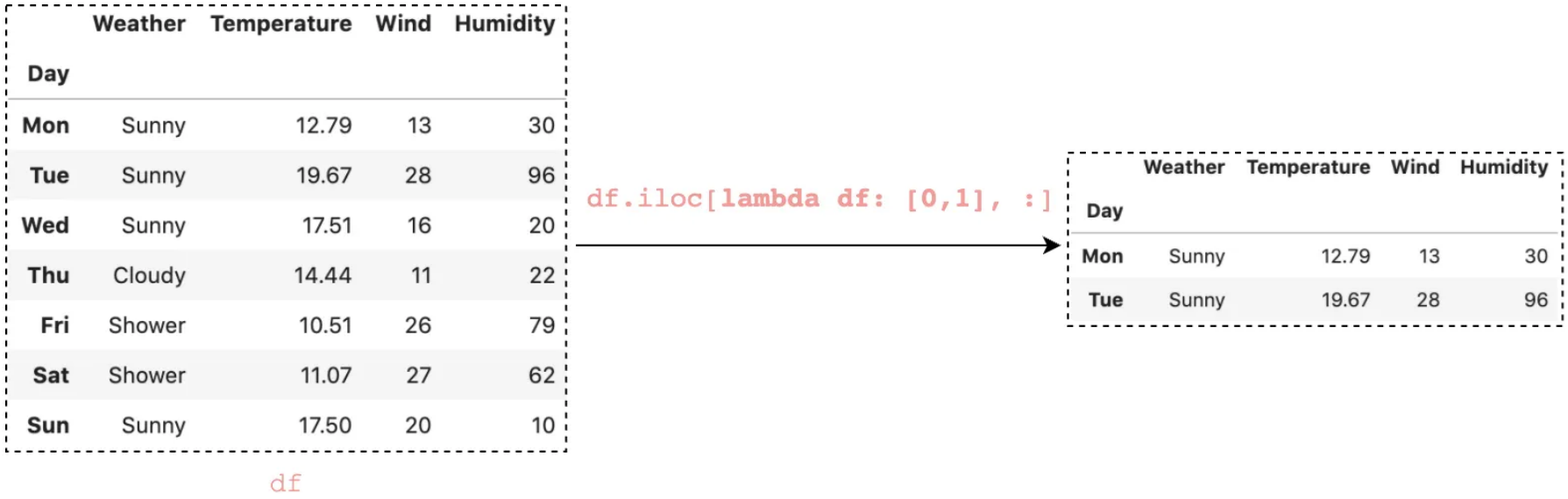
Day	Weather	Temperature	Wind	Humidity
Tue	Sunny	19.67	28	96
Fri	Shower	10.51	26	79
Sat	Shower	11.07	27	62

image by author

`iloc` with callable

`iloc` can also take a **callable** as an indexer.

```
df.iloc[lambda df: [0,1], :]
```



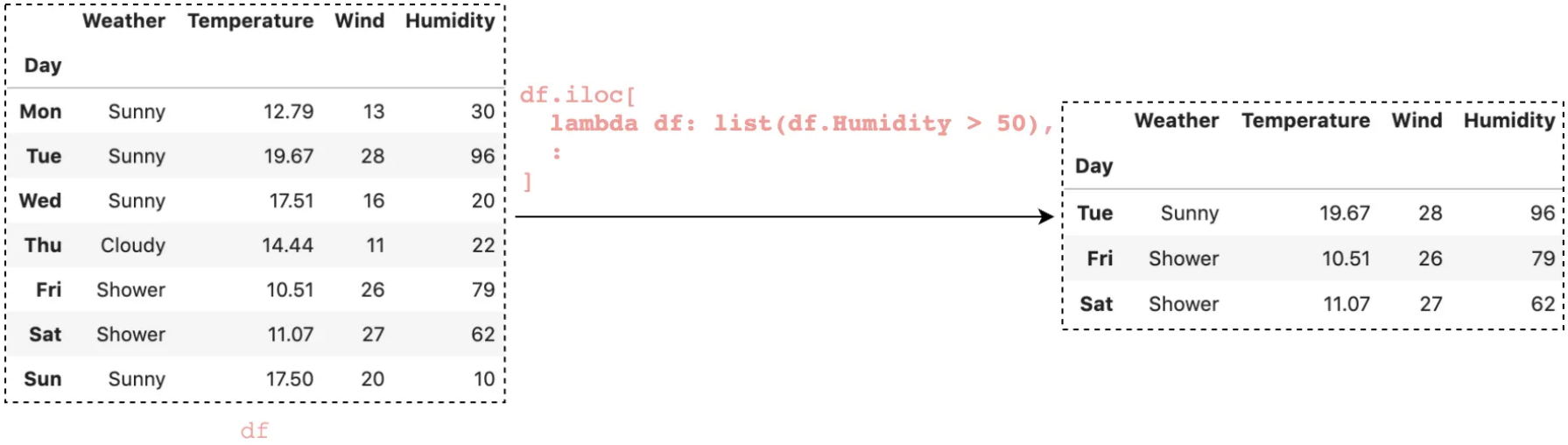
Day	Weather	Temperature	Wind	Humidity
Mon	Sunny	12.79	13	30
Tue	Sunny	19.67	28	96
Wed	Sunny	17.51	16	20
Thu	Cloudy	14.44	11	22
Fri	Shower	10.51	26	79
Sat	Shower	11.07	27	62
Sun	Sunny	17.50	20	10

Day	Weather	Temperature	Wind	Humidity
Mon	Sunny	12.79	13	30
Tue	Sunny	19.67	28	96

image by author

To filter data with callable, `iloc` will require `list()` to convert the output of conditions into a boolean list:

```
df.iloc[lambda df: list(df.Humidity > 50), :]
```



Day	Weather	Temperature	Wind	Humidity
Mon	Sunny	12.79	13	30
Tue	Sunny	19.67	28	96
Wed	Sunny	17.51	16	20
Thu	Cloudy	14.44	11	22
Fri	Shower	10.51	26	79
Sat	Shower	11.07	27	62
Sun	Sunny	17.50	20	10

Day	Weather	Temperature	Wind	Humidity
Tue	Sunny	19.67	28	96
Fri	Shower	10.51	26	79
Sat	Shower	11.07	27	62

image by author

6. `loc` and `iloc` are interchangeable when labels are 0-based integers

For demonstration, let's create a DataFrame with 0-based integers as headers and index labels.

```
df = pd.read_csv(
    'data/data.csv',
    header=None,
    skiprows=[0],
)
```

With `header=None`, the Pandas will generate 0-based integer values as headers. With `skiprows=[0]`, those headers **Weather**, **Temperature,** etc we have been using will be skipped.

	0	1	2	3	4
0	Mon	Sunny	12.79	13	30
1	Tue	Sunny	19.67	28	96
2	Wed	Sunny	17.51	16	20
3	Thu	Cloudy	14.44	11	22
4	Fri	Shower	10.51	26	79
5	Sat	Shower	11.07	27	62
6	Sun	Sunny	17.50	20	10

image by author

Now, `loc`, a label-based data selector, can accept a single integer and a list of integer values. For example:

```
>>> df.loc[1, 2]
19.67
```

```
>>> df.loc[1, [1, 2]]
1    Sunny
2    19.67
Name: 1, dtype: object
```

The reason they are working is that those integer values (`1` and `2`) are interpreted as ***labels*** of the index. This use is **not** an integer position along with the index and is a bit confusing.

In this case, `loc` and `iloc` are interchangeable when selecting via a single value or a list of values.

```
>>> df.loc[1, 2] == df.iloc[1, 2]
True
```

```
>>> df.loc[1, [1, 2]] == df.iloc[1, [1, 2]]
1    True
2    True
Name: 1, dtype: bool
```

Note that `loc` and `iloc` will return different results when selecting via slice and conditions. They are essentially different because:

- slice: endpoint is excluded from `iloc` result, but included in `loc`
- conditions: `loc` accepts boolean Series, but `iloc` can only accept a boolean list.

Conclusion

Finally, here is a summary

`loc` is label based and allowed inputs are:

- A single label `'A'` or `2` (Note that `2` is interpreted as a *label* of the index.)
- A list of labels `['A', 'B', 'C']` or `[1, 2, 3]` (Note that `1, 2, 3` are interpreted as *labels* of the index.)

- A slice with labels `'A':'C'` (Both are included)
- Conditions, a boolean Series or a boolean array
- A `callable` function with one argument

`iloc` is integer position based and allowed inputs are:

- An integer e.g. `2`.
- A list or array of integers `[1, 2, 3]`.
- A slice with integers `1:7` (the endpoint `7` is excluded)
- Conditions, but only accept a boolean array
- A `callable` function with one argument

`loc` and `iloc` are interchangeable when the labels of Pandas DataFrame are 0-based integers

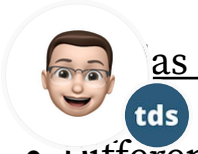
I hope this article will help you to save time in learning Pandas data selection. I recommend you to check out the documentation to know about other things you can do.

Thanks for reading. Please check out the notebook for the source code and stay tuned if you are interested in the practical aspect of machine learning.

You may be interested in some of my other Pandas articles:

Pandas | Python | Data Science | Data Analysis | Data Selection

- Pandas cut() function for transforming numerical data into categorical data
- Using Pandas method chaining to improve code readability
- How to do a Custom Sort on Pandas DataFrame
- All the Pandas shift() you should know for data analysis
- When to use Pandas transform() function
- as concat() tricks you should know
- Difference between apply() and transform() in Pandas
- All the Pandas merge() you should know
- Working with datetime in Pandas DataFrame
- Pandas read_csv() tricks you should know



Written by B. Chen

4.2K Followers · Writer for Towards Data Science

Machine Learning practitioner

Follow

- 4 tricks you should know to parse date columns with Pandas read_csv()

More from B. Chen and Towards Data Science

More tutorials can be found on my Github

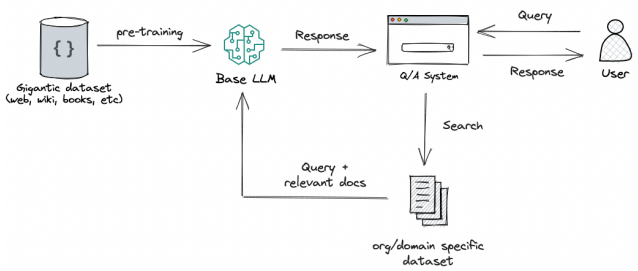
B. Chen in Towards Data Science

How to convert JSON into a Pandas DataFrame

Some of the most useful Pandas tricks

· 6 min read · Dec 21, 2020

-- 5



Heiko Hotz in Towards Data Science

RAG vs Finetuning—Which Is the Best Tool to Boost Your LLM...

The definitive guide for choosing the right method for your use case

· 19 min read · Aug 25

-- 16

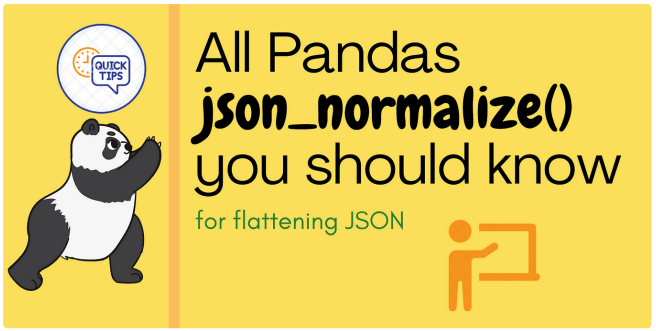
Giuseppe Scalamogna in Towards Data Science

New ChatGPT Prompt Engineering Technique: Program Simulation

A potentially novel technique for turning a ChatGPT prompt into a mini-app.

9 min read · Sep 4

-- 12



B. Chen in Towards Data Science

All Pandas json_normalize() you should know for flattening JSON

Some of the most useful Pandas tricks

· 8 min read · Feb 23, 2021

-- 20

See all from B. Chen

See all from Towards Data Science

Recommended from Medium

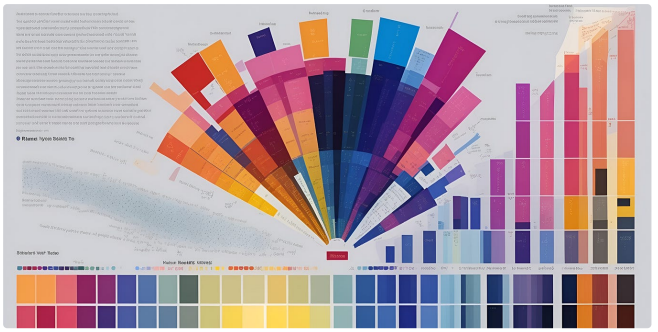
Avi Chawla in Towards Data Science

It's Time to Say GoodBye to pd.read_csv() and pd.to_csv()

Discussing another major caveat of Pandas

4 min read · May 27, 2022

-- 49



Bex T. in Towards AI

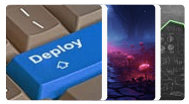
3 Best (Often Better) Alternatives To Histograms

Avoid the most dangerous pitfall of histograms

· 10 min read · Sep 12

-- 5

Lists



Predictive Modeling w/ Python

20 stories · 401 saves

New_Reading_List

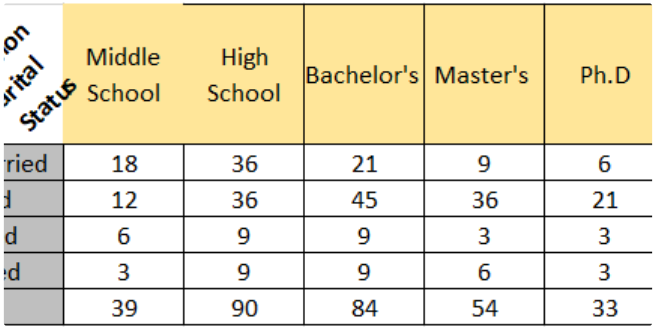
174 stories · 113 saves

Coding & Development

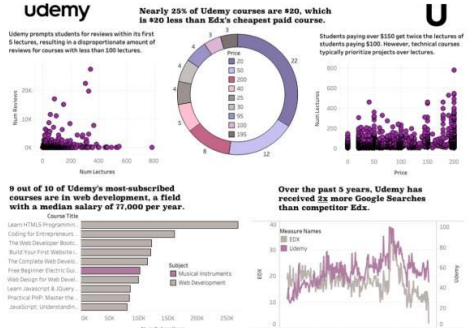
11 stories · 185 saves

Practical Guides to Machine Learning

10 stories · 466 saves



Status	Middle School	High School	Bachelor's	Master's	Ph.D
ried	18	36	21	9	6
d	12	36	45	36	21
d	6	9	9	3	3
d	3	9	9	6	3
	39	90	84	54	33



Maninder Singh

Understanding Categorical Correlations with Chi-Square Test...

In this, I explained about correlation(code) between categorical features which I Learne...

9 min read · Jun 18

--

Zach Qui... in Pipeline: Your Data Engineering Res...

Creating The Dashboard That Got Me A Data Analyst Job Offer

A walkthrough of the Udemy dashboard that got me a job offer from one of the biggest...

· 9 min read · Dec 5, 2022

-- 22

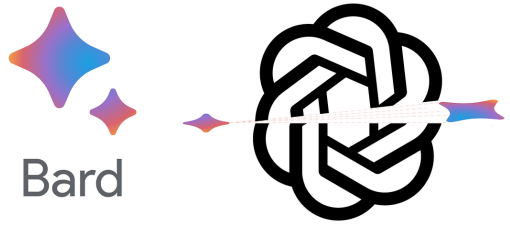
Breno de Jesus Fernandes

Write Better SQL: How to Avoid the Subquery Hell

Make complex sql queries easy to understanding, debugging, and maintaining

6 min read · Jul 11

--



AL Anany

The ChatGPT Hype Is Over—Now Watch How Google Will Kill...

It never happens instantly. The business game is longer than you know.

· 6 min read · Sep 1

-- 252

See more recommendations

Help Status Writers Blog Careers Privacy Terms About Text to speech Teams