Python → Data Scienceβ → Data analysis with pandas → Series

Theory: Series

19 minutes 0 / 5 problems solved

Skip this topic Start practicing

247 users solved this topic. Latest completion was about 10 hours ago.

As you already know, `pandas` is a popular Python library for data manipulation. This topic will introduce you to `Series`, a basic one-dimensional data structure in `pandas`.

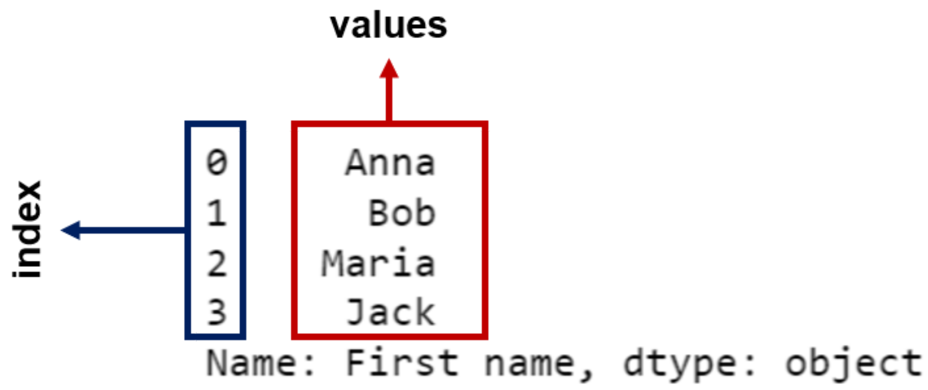
`Series` is a building block to the 2D data structure in pandas, `DataFrame`. The latter is the one you will use the most in practice. So, while it is important to have some idea of `Series`, in-depth knowledge of its functionality is not necessary.

Before we start, do not forget that you need to import `pandas` to be able to use all of its functionality, including `Series`. Note that traditionally, the name of the library is abbreviated as `pd` in the import statement:

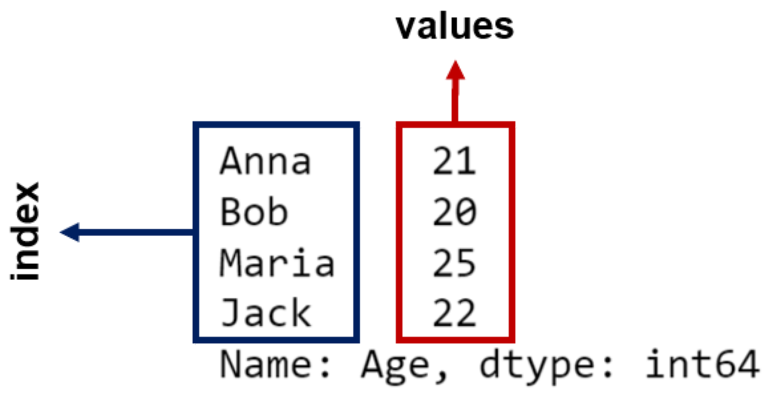
```
import pandas as pd
```

§1. What is Series

`Series` is a one-dimensional array. For example, here is a `Series` that stores names of the students on a machine learning class:



You can notice that each element stored in a `Series` is associated with a label called **index**. By default, this index is just the sequence 0, 1, 2, ... However, any custom values can be used. For example, we can store ages of the students in a `Series`, and set students' names as row identifiers so that we know which student the age corresponds to:



Cool, but how do you create such a `Series` object? There are several ways to do so.

§2. Converting other data structures to Series

If the data is already stored in some other data structure, you can easily convert it to `Series` as shown below:

Current topic:

Series ···

Topic depends on:

✓ List Stage 1 13★ ···

✓ Dictionary Stage 1 6★ ···

✓ Declaring a function Stage 1 10★ ···

✗ Introduction to pandas ···

Topic is required for:

DataFrame ···

Table of contents:

```
ages_list = [21, 20, 25, 22]
names_list = ['Anna', 'Bob', 'Maria', 'Jack']

ages_series = pd.Series(ages_list, index=names_list, name='Age')
print(ages_series)

# Anna     21
# Bob      20
# Maria    25

# Jack     22

# Name: Age, dtype: int64
```

Here, we convert a list of students' ages `ages_list` into a `Series` object `ages_series`. We also assign a custom index, students' names stored in the list `names_list`, using the `index` keyword. If we fail to provide the values for the `index`, values 0, 1, 2, 3 would be assigned. Finally, we can also give our `Series` a name by specifying the optional `name` parameter.

Similarly, you can convert a Python dictionary into a Series object. Note that dictionary keys (students' names in the example below) will automatically become the indexes in the new `Series`. Cool, right?

```
student_ages_dict = {'Anna': 21, 'Bob': 20, 'Maria': 25, 'Jack': 22}

ages_series = pd.Series(student_ages_dict, name='Ages')
print(ages_series)

# Anna     21
# Bob      20
# Maria    25
# Jack     22

# Name: Ages, dtype: int64
```

You can always change index later by modifying the index attribute of the `Series`:

```
ages_series.index = ['A', 'B', 'M', 'J']
print(ages_series)

# A    21
# B    20
# M    25
# J    22
# Name: Ages, dtype: int64
```

> Of course, the length of the data and index you provide should coincide. Otherwise, an error will occur.

§3. Modifying a Series object

`Series` is **value-mutable**; you can easily change the values stored in a `Series`, for example by accessing them by index. To illustrate this, let's update Jack's age:

```
ages_series['Jack'] = 23
print(ages_series)

# Anna     21
# Bob      20
# Maria    25
# Jack     23
# Name: Ages, dtype: int64
```

However, a `Series` object is **size-immutable**, once it's created, no elements can be added to or removed from it. It is made on purpose, to efficiently store `Series` in memory.

But what should you do if you need to add or drop some values to/from a `Series`? No worries, appending, and removing elements is still possible. However, the result of these operations will be a new `Series` object.

For example, let's remove Maria's record from our `Series`. This can be done with the drop method.:

```
new_ages_series = ages_series.drop(index='Maria')
print(new_ages_series)

# Anna    21
# Bob     20
# Jack    23
# Name: Ages, dtype: int64
```

Note that the original `Series` remained unchanged:

```
print(ages_series)

# Anna     21
# Bob      20
# Maria    25
# Jack     23
# Name: Ages, dtype: int64
```

If you want the returned `Series` to be automatically assigned to the original one, you can specify the optional `inplace` parameter to be `True` :

```
ages_series.drop(index='Maria', inplace=True)
print(ages_series)

# Anna    21
# Bob     20
# Jack    23
# Name: Ages, dtype: int64
```

To add new records to `Series`, one can explicitly specify the value for the new index. Let's add Maria's record back to the `ages_series`:

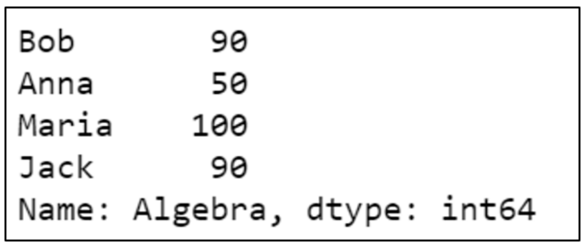
```
ages_series['Maria'] = 25
print(ages_series)

# Anna     21
# Bob      20
# Jack     23
# Maria    25
# Name: Ages, dtype: int64
```

Note that this syntax is introduced for convenience. Since Series is size-immutable, behind the scenes, a new `Series` with the newly added element will be created and automatically assigned to the original one.

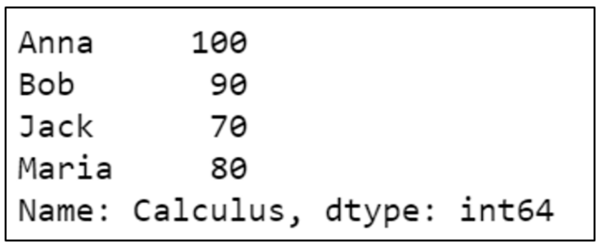
§4. Operations on Series

The key feature of `pandas` is that operations between several Series automatically align the data based on the index.

Let's imagine we have two `Series`, `algebra` and `calculus`, containing students' exam results for Algebra and Calculus courses respectively:



```
Bob       90
Anna      50
Maria    100
Jack      90
Name: Algebra, dtype: int64
```



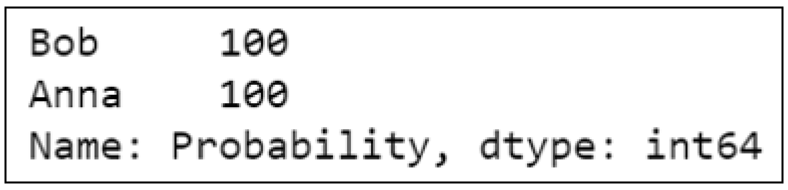
```
Anna     100
Bob       90
Jack      70
Maria     80
Name: Calculus, dtype: int64
```

Suppose we want to compute the average score students got for the two exams.

```
average = 0.5*(algebra + calculus)
print(average)

# Anna     75.0
# Bob      90.0
# Jack     80.0
# Maria    90.0
# dtype: float64
```

Note that the order of the students in the two `Series`, `algebra` and `calculus`, is different, and yet the averages are computed correctly. Very convenient!

Alright, but what happens if some indexes from one Series don't exist in the other one? Let's imagine there was also the third exam on Probability, but only Anna and Bob took it. The results are stored in the `probability` `Series`:



```
Bob      100
Anna     100
Name: Probability, dtype: int64
```

What will happen if we try to compute the average of the three Series now?

```
average = 0.33*(algebra + calculus + probability)
print(average)

# Anna     82.5
# Bob      92.4
# Jack      NaN
# Maria     NaN
# dtype: float64
```

As you can see, the result of an operation between unaligned `Series` has the **union** of the indexes involved. If a label is not found in one of the operands, the result will be marked as a missing value `NaN`.

Writing code without doing any explicit data alignment gives a lot of flexibility in data analysis. The integrated data alignment in `pandas` is what sets the library apart from the majority of related tools for working with labeled data.

§5. Conclusions

- `Series` is a one-dimensional data structure in pandas.
- `Series` store values along with their labels called indexes.
- `Series` is value-mutable but size-immutable: one can modify values stored in it, but cannot add new values to or remove values from it.
- When performing operations on several `Series` objects, they are automatically aligned on the index.

Report a typo

21 users liked this theory. 1 didn't like it. **What about you?**

😍 🙂 😐 🙁 😡

Start practicing

Comments (0) Hints (0) Useful links (0) Show discussion