# SeriesDataStructure_ed

May 9, 2022

In this lecture we're going to explore the pandas Series structure. By the end of this lecture you should be familiar with how to store and manipulate single dimensional indexed data in the Series object.

The series is one of the core data structures in pandas. You think of it a cross between a list and a dictionary. The items are all stored in an order and there's labels with which you can retrieve them. An easy way to visualize this is two columns of data. The first is the special index, a lot like keys in a dictionary. While the second is your actual data. It's important to note that the data column has a label of its own and can be retrieved using the .name attribute. This is different than with dictionaries and is useful when it comes to merging multiple columns of data. And we'll talk about that later on in the course.

[1]:
```
# Let's import pandas to get started
import pandas as pd
```

[19]:
```
# As you might expect, you can create a series by passing in a list of values.
# When you do this, Pandas automatically assigns an index starting with zero␣
 ↪and
# sets the name of the series to None. Let's work on an example of this.

# One of the easiest ways to create a series is to use an array-like object,␣
 ↪like
# a list.

# Here I'll make a list of the three of students, Alice, Jack, and Molly, all␣
 ↪as strings
students = ['Alice', 'Jack', 'Molly']

# Now we just call the Series function in pandas and pass in the students
a= pd.Series(students)

print(a,"\n")

#Chequeamos el nombre de la serie, como no le asignamos uno se llamará None
print("Nombre de la serie:", a.name)
```

```
0    Alice
1     Jack
2    Molly
```

```
dtype: object

Nombre de la serie: None
```

[3]:
```
# The result is a Series object which is nicely rendered to the screen. We see
↪here that
# the pandas has automatically identified the type of data in this Series as
↪"object" and
# set the dytpe parameter as appropriate. We see that the values are indexed
↪with integers,
# starting at zero
```

[29]:
```
# We don't have to use strings. If we passed in a list of whole numbers, for
↪instance,
# we could see that panda sets the type to int64. Underneath panda stores
↪series values in a
# typed array using the Numpy library. This offers significant speedup when
↪processing data
# versus traditional python lists.

# Lets create a little list of numbers
numbers = [1, 2, 3]
# And turn that into a series

b= pd.Series(numbers)
print(b)

#Algunos atributos de las series de Pandas: nombre de la columna de datos y
↪elementos que componen el index
print(b.name)
print(b.index)
```

```
0    1
1    2
2    3
dtype: int64
None
RangeIndex(start=0, stop=3, step=1)
```

[5]:
```
# And we see on my architecture that the result is a dtype of int64 objects
```

[6]:
```
# There's some other typing details that exist for performance that are
↪important to know.
# The most important is how Numpy and thus pandas handle missing data.

# In Python, we have the none type to indicate a lack of data. But what do we
↪do if we want
# to have a typed list like we do in the series object?
```

```python
# Underneath, pandas does some type conversion. If we create a list of strings␣
↪and we have
# one element, a None type, pandas inserts it as a None and uses the type␣
↪object for the
# underlying array.

# Let's recreate our list of students, but leave the last one as a None
students = ['Alice', 'Jack', None]
# And lets convert this to a series
pd.Series(students)
```

[6]:
```
0    Alice
1     Jack
2     None
dtype: object
```

[7]:
```python
# However, if we create a list of numbers, integers or floats, and put in the␣
↪None type,
# pandas automatically converts this to a special floating point value␣
↪designated as NaN,
# which stands for "Not a Number".

# So lets create a list with a None value in it
numbers = [1, 2, None]
# And turn that into a series
pd.Series(numbers)
```

[7]:
```
0    1.0
1    2.0
2    NaN
dtype: float64
```

[8]:
```python
# You'll notice a couple of things. First, NaN is a different value. Second,␣
↪pandas
# set the dytpe of this series to floating point numbers instead of object or␣
↪ints. That's
# maybe a bit of a surprise - why not just leave this as an integer?␣
↪Underneath, pandas
# represents NaN as a floating point number, and because integers can be␣
↪typecast to
# floats, pandas went and converted our integers to floats. So when you're␣
↪wondering why the
# list of integers you put into a Series is not floats, it's probably because␣
↪there is some
# missing data.
```

[23]:
```
# For those who might not have done scientific computing in Python before, it␣
↪is important
# to stress that None and NaN might be being used by the data scientist in the␣
↪same way, to
# denote missing data, but that underneath these are not represented by pandas␣
↪in the same
# way.

# NaN is *NOT* equivilent to None and when we try the equality test, the result␣
↪is False.

# Lets bring in numpy which allows us to generate an NaN value

# NaN NO ES LO MISMO QUE None!!!!!!

import numpy as np
# And lets compare it to None
np.nan == None
```

[23]: False

[24]:
```
# It turns out that you actually can't do an equality test of NAN to itself.␣
↪When you do,
# the answer is always False.

np.nan == np.nan
```

[24]: False

[11]:
```
# Instead, you need to use special functions to test for the presence of not a␣
↪number,
# such as the Numpy library isnan().

np.isnan(np.nan)
```

[11]: True

[12]:
```
# So keep in mind when you see NaN, it's meaning is similar to None, but it's a
# numeric value and treated differently for efficiency reasons.
```

[28]:
```
# Let's talk more about how pandas' Series can be created. While my list might␣
↪be a common
# way to create some play data, often you have label data that you want to␣
↪manipulate.
# A series can be created directly from dictionary data. If you do this, the␣
↪index is
# automatically assigned to the keys of the dictionary that you provided and␣
↪not just
# incrementing integers.
```

```
# Here's an example using some data of students and their classes.

students_scores = {'Alice': 'Physics',
                   'Jack': 'Chemistry',
                   'Molly': 'English'}
s = pd.Series(students_scores)


print(s)

print(s.name)
```

```
Alice      Physics
Jack     Chemistry
Molly      English
dtype: object
None
```

[14]:
```
# We see that, since it was string data, pandas set the data type of the series␣
 ↪to "object".
# We see that the index, the first column, is also a list of strings.
```

[3]:
```
# Once the series has been created, we can get the index object using the index␣
 ↪attribute.

s.index
```

[3]:
```
Index(['Alice', 'Jack', 'Molly'], dtype='object')
```

[16]:
```
# As you play more with pandas you'll notice that a lot of things are␣
 ↪implemented as numpy
# arrays, and have the dtype value set. This is true of indicies, and here␣
 ↪pandas infered
# that we were using objects for the index.
```

[17]:
```
# Now, this is kind of interesting. The dtype of object is not just for␣
 ↪strings, but for
# arbitrary objects. Lets create a more complex type of data, say, a list of␣
 ↪tuples.
students = [("Alice","Brown"), ("Jack", "White"), ("Molly", "Green")]
pd.Series(students)
```

[17]:
```
0    (Alice, Brown)
1     (Jack, White)
2    (Molly, Green)
dtype: object
```

[18]:
```
# We see that each of the tuples is stored in the series object, and the type␣
 ↪is object.
```

[12]:
```
# You can also separate your index creation from the data by passing in the␣
↪index as a
# list explicitly to the series.

s = pd.Series(['Physics', 'Chemistry', 'English'], index=['Alice', 'Jack',␣
↪'Molly'])
s
```

[12]:
```
Alice      Physics
Jack     Chemistry
Molly      English
dtype: object
```

[20]:
```
# So what happens if your list of values in the index object are not aligned␣
↪with the keys
# in your dictionary for creating the series? Well, pandas overrides the␣
↪automatic creation
# to favor only and all of the indices values that you provided. So it will␣
↪ignore from your
# dictionary all keys which are not in your index, and pandas will add None or␣
↪NaN type values
# for any index value you provide, which is not in your dictionary key list.

# Here's and example. I'll pass in a dictionary of three items, in this case␣
↪students and
# their courses
students_scores = {'Alice': 'Physics',
                   'Jack': 'Chemistry',
                   'Molly': 'English'}
# When I create the series object though I'll only ask for an index with three␣
↪students, and
# I'll exclude Jack
s = pd.Series(students_scores, index=['Alice', 'Molly', 'Sam'])
s
```

[20]:
```
Alice    Physics
Molly    English
Sam          NaN
dtype: object
```

[21]:
```
# The result is that the Series object doesn't have Jack in it, even though he␣
↪was in our
# original dataset, but it explicitly does have Sam in it as a missing value.
```

In this lecture we've explored the pandas Series data structure. You've seen how to create a series from lists and dictionaries, how indicies on data work, and the way that pandas typecasts data including missing values.