QueryingSeries_ed

July 13, 2023

1 Querying Series

In this lecture, we'll talk about one of the primary data types of the Pandas library, the Series. You'll learn about the structure of the Series, how to query and merge Series objects together, and the importance of thinking about parallelization when engaging in data science programming.

[1]:
```
# A pandas Series can be queried either by the index position or the index␣
 ↪label. If you don't give an
# index to the series when querying, the position and the label are effectively␣
 ↪the same values. To
# query by numeric location, starting at zero, use the iloc attribute. To query␣
 ↪by the index label,
# you can use the loc attribute.

# Lets start with an example. We'll use students enrolled in classes coming␣
 ↪from a dictionary
import pandas as pd
students_classes = {'Alice': 'Physics',
                   'Jack': 'Chemistry',
                   'Molly': 'English',
                   'Sam': 'History'}
s = pd.Series(students_classes)
s
```

[1]:
```
Alice      Physics
Jack     Chemistry
Molly      English
Sam        History
dtype: object
```

[2]:
```
# So, for this series, if you wanted to see the fourth entry we would we would␣
 ↪use the iloc
# attribute with the parameter 3.
s.iloc[3]
```

[2]:
```
'History'
```

[3]:
```
# If you wanted to see what class Molly has, we would use the loc attribute␣
 ↪with a parameter
# of Molly.
s.loc['Molly']
```

[3]:
```
'English'
```

[4]:
```
# Keep in mind that iloc and loc are not methods, they are attributes. So you␣
 ↪don't use
# parentheses to query them, but square brackets instead, which is called the␣
 ↪indexing operator.
# In Python this calls get or set for an item depending on the context of its␣
 ↪use.

# This might seem a bit confusing if you're used to languages where␣
 ↪encapsulation of attributes,
# variables, and properties is common, such as in Java.
```

[5]:
```
# Pandas tries to make our code a bit more readable and provides a sort of␣
 ↪smart syntax using
# the indexing operator directly on the series itself. For instance, if you␣
 ↪pass in an integer parameter,
# the operator will behave as if you want it to query via the iloc attribute
s[3]
```

[5]:
```
'History'
```

[6]:
```
# If you pass in an object, it will query as if you wanted to use the label␣
 ↪based loc attribute.
s['Molly']
```

[6]:
```
'English'
```

[7]:
```
# So what happens if your index is a list of integers? This is a bit␣
 ↪complicated and Pandas can't
# determine automatically whether you're intending to query by index position␣
 ↪or index label. So
# you need to be careful when using the indexing operator on the Series itself.␣
 ↪The safer option
# is to be more explicit and use the iloc or loc attributes directly.

# Here's an example using class and their classcode information, where classes␣
 ↪are indexed by
# classcodes, in the form of integers
class_code = {99: 'Physics',
              100: 'Chemistry',
              101: 'English',
              102: 'History'}
s = pd.Series(class_code)
```

[8]:
```
# If we try and call s[0] we get a key error because there's no item in the␣
 ↪classes list with
# an index of zero, instead we have to call iloc explicitly if we want the␣
 ↪first item.

s[0]
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
File /opt/conda/lib/python3.9/site-packages/pandas/core/indexes/base.py:3803, i␣
 ↪Index.get_loc(self, key, method, tolerance)
   3802 try:
-> 3803     return self._engine.get_loc(casted_key)
   3804 except KeyError as err:

File /opt/conda/lib/python3.9/site-packages/pandas/_libs/index.pyx:138, in␣
 ↪pandas._libs.index.IndexEngine.get_loc()

File /opt/conda/lib/python3.9/site-packages/pandas/_libs/index.pyx:165, in␣
 ↪pandas._libs.index.IndexEngine.get_loc()

File pandas/_libs/hashtable_class_helper.pxi:2263, in pandas._libs.hashtable.
 ↪Int64HashTable.get_item()

File pandas/_libs/hashtable_class_helper.pxi:2273, in pandas._libs.hashtable.
 ↪Int64HashTable.get_item()

KeyError: 0

The above exception was the direct cause of the following exception:

KeyError                                  Traceback (most recent call last)
Cell In [8], line 4
      1 # If we try and call s[0] we get a key error because there's no item in␣
 ↪the classes list with
      2 # an index of zero, instead we have to call iloc explicitly if we want␣
 ↪the first item.
----> 4 s[0]

File /opt/conda/lib/python3.9/site-packages/pandas/core/series.py:981, in Serie
 ↪__getitem__(self, key)
    978     return self._values[key]
    980 elif key_is_scalar:
```

```
--> 981     return self._get_value(key)
    983 if is_hashable(key):
    984     # Otherwise index.get_value will raise InvalidIndexError
    985     try:
    986         # For labels that don't resolve as scalars like tuples and␣
 ↪frozensets

File /opt/conda/lib/python3.9/site-packages/pandas/core/series.py:1089, in␣
 ↪Series._get_value(self, label, takeable)
   1086     return self._values[label]
   1088 # Similar to Index.get_value, but we do not fall back to positional
-> 1089 loc = self.index.get_loc(label)
   1090 return self.index._get_values_for_loc(self, loc, label)

File /opt/conda/lib/python3.9/site-packages/pandas/core/indexes/base.py:3805, i␣
 ↪Index.get_loc(self, key, method, tolerance)
   3803     return self._engine.get_loc(casted_key)
   3804 except KeyError as err:
-> 3805     raise KeyError(key) from err
   3806 except TypeError:
   3807     # If we have a listlike key, _check_indexing_error will raise
   3808     #  InvalidIndexError. Otherwise we fall through and re-raise
   3809     #  the TypeError.
   3810     self._check_indexing_error(key)

KeyError: 0
```

[9]:
```
# So, that didn't call s.iloc[0] underneath as one might expect, instead it
# generates an error
```

[10]:
```
# Now we know how to get data out of the series, let's talk about working with␣
 ↪the data. A common
# task is to want to consider all of the values inside of a series and do some␣
 ↪sort of
# operation. This could be trying to find a certain number, or summarizing data␣
 ↪or transforming
# the data in some way.
```

[11]:
```
# A typical programmatic approach to this would be to iterate over all the␣
 ↪items in the series,
# and invoke the operation one is interested in. For instance, we could create␣
 ↪a Series of
# integers representing student grades, and just try and get an average grade

grades = pd.Series([90, 80, 70, 60])
```

```
total = 0
for grade in grades:
    total+=grade
print(total/len(grades))
```

```
75.0
```

[12]:
```
# This works, but it's slow. Modern computers can do many tasks simultaneously,␣
 ↪especially,
# but not only, tasks involving mathematics.

# Pandas and the underlying numpy libraries support a method of computation␣
 ↪called vectorization.
# Vectorization works with most of the functions in the numpy library,␣
 ↪including the sum function.
```

[13]:
```
# Here's how we would really write the code using the numpy sum method. First␣
 ↪we need to import
# the numpy module

import numpy as np

# Then we just call np.sum and pass in an iterable item. In this case, our␣
 ↪panda series.

total = np.sum(grades)
print(total/len(grades))
```

```
75.0
```

[14]:
```
# Now both of these methods create the same value, but is one actually faster?␣
 ↪The Jupyter
# Notebook has a magic function which can help.

# First, let's create a big series of random numbers. This is used a lot when␣
 ↪demonstrating
# techniques with Pandas
numbers = pd.Series(np.random.randint(0,1000,10000))

# Now lets look at the top five items in that series to make sure they actually␣
 ↪seem random. We
# can do this with the head() function
numbers.head()
```

[14]:
```
0    523
1    154
2    181
```

```
3    716
4      1
dtype: int64
```

[15]:
```
# We can actually verify that length of the series is correct using the len␣
  ↪function
len(numbers)
```

[15]:
```
10000
```

[16]:
```
# Ok, we're confident now that we have a big series. The ipython interpreter␣
  ↪has something called
# magic functions begin with a percentage sign. If we type this sign and then␣
  ↪hit the Tab key, you
# can see a list of the available magic functions. You could write your own␣
  ↪magic functions too,
# but that's a little bit outside of the scope of this course.
```

[17]:
```
# Here, we're actually going to use what's called a cellular magic function.␣
  ↪These start with two
# percentage signs and wrap the code in the current Jupyter cell. The function␣
  ↪we're going to use
# is called timeit. This function will run our code a few times to determine,␣
  ↪on average, how long
# it takes.

# Let's run timeit with our original iterative code. You can give timeit the␣
  ↪number of loops that
# you would like to run. By default, it is 1,000 loops. I'll ask timeit here to␣
  ↪use 100 runs because
# we're recording this. Note that in order to use a cellular magic function, it␣
  ↪has to be the first
# line in the cell
```

[18]:
```
%%timeit -n 100
total = 0
for number in numbers:
    total+=number

total/len(numbers)
```

```
1.02 ms ± 8.79 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

[19]:
```
# Not bad. Timeit ran the code and it doesn't seem to take very long at all.␣
  ↪Now let's try with
# vectorization.
```

[20]:
```
%%timeit -n 100
total = np.sum(numbers)
total/len(numbers)
```

```
49 µs ± 2.96 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

[21]:
```
# Wow! This is a pretty shocking difference in the speed and demonstrates why␣
 ↪one should be
# aware of parallel computing features and start thinking in functional␣
 ↪programming terms.
# Put more simply, vectorization is the ability for a computer to execute␣
 ↪multiple instructions
# at once, and with high performance chips, especially graphics cards, you can␣
 ↪get dramatic
# speedups. Modern graphics cards can run thousands of instructions in parallel.
```

[22]:
```
# A Related feature in pandas and nummy is called broadcasting. With␣
 ↪broadcasting, you can
# apply an operation to every value in the series, changing the series. For␣
 ↪instance, if we
# wanted to increase every random variable by 2, we could do so quickly using␣
 ↪the += operator
# directly on the Series object.

# Let's look at the head of our series
numbers.head()
```

[22]:
```
0    523
1    154
2    181
3    716
4      1
dtype: int64
```

[23]:
```
# And now lets just increase everything in the series by 2
numbers+=2
numbers.head()
```

[23]:
```
0    525
1    156
2    183
3    718
4      3
dtype: int64
```

[24]:
```
# The procedural way of doing this would be to iterate through all of the items␣
 ↪in the
# series and increase the values directly. Pandas does support iterating␣
 ↪through a series
# much like a dictionary, allowing you to unpack values easily.

# We can use the iteritems() function which returns a label and value
for label, value in numbers.iteritems():
    # in the early version of pandas we would use the set_value() function
    # in the current version, we use the iat() or at() functions,
    numbers.iat[label]= value+2
# And we can check the result of this computation
numbers.head()
```

[24]:
```
0    527
1    158
2    185
3    720
4      5
dtype: int64
```

[25]:
```
# So the result is the same, though you may notice a warning depending upon the␣
 ↪version of
# pandas being used. But if you find yourself iterating pretty much *any time*␣
 ↪in pandas,
# you should question whether you're doing things in the best possible way.
```

[26]:
```
# Lets take a look at some speed comparisons. First, lets try five loops using␣
 ↪the iterative approach
```

[27]:
```
%%timeit -n 10
# we'll create a blank new series of items to deal with
s = pd.Series(np.random.randint(0,1000,1000))
# And we'll just rewrite our loop from above.
for label, value in s.iteritems():
    s.loc[label]= value+2
```

```
37 ms ± 225 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

[28]:
```
# Now lets try that using the broadcasting methods
```

[29]:
```
%%timeit -n 10
# We need to recreate a series
s = pd.Series(np.random.randint(0,1000,1000))
# And we just broadcast with +=
s+=2
```

```
151 µs ± 40.5 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

[30]:
```
# Amazing. Not only is it significantly faster, but it's more concise and even␣
↪easier
# to read too. The typical mathematical operations you would expect are␣
↪vectorized, and the
# nump documentation outlines what it takes to create vectorized functions of␣
↪your own.
```

[31]:
```
# One last note on using the indexing operators to access series data. The .loc␣
↪attribute lets
# you not only modify data in place, but also add new data as well. If the␣
↪value you pass in as
# the index doesn't exist, then a new entry is added. And keep in mind, indices␣
↪can have mixed types.
# While it's important to be aware of the typing going on underneath, Pandas␣
↪will automatically
# change the underlying NumPy types as appropriate.
```

[32]:
```
# Here's an example using a Series of a few numbers.
s = pd.Series([1, 2, 3])

# We could add some new value, maybe a university course
s.loc['History'] = 102

s
```

[32]:
```
0            1
1            2
2            3
History    102
dtype: int64
```

[33]:
```
# We see that mixed types for data values or index labels are no problem for␣
↪Pandas. Since
# "History" is not in the original list of indices, s.loc['History']␣
↪essentially creates a
# new element in the series, with the index named "History", and the value of␣
↪102
```

[34]:
```
# Up until now I've shown only examples of a series where the index values were␣
↪unique. I want
# to end this lecture by showing an example where index values are not unique,␣
↪and this makes
# pandas Series a little different conceptually then, for instance, a␣
↪relational database.
```

```
# Lets create a Series with students and the courses which they have taken
students_classes = pd.Series({'Alice': 'Physics',
                   'Jack': 'Chemistry',
                   'Molly': 'English',
                   'Sam': 'History'})
students_classes
```

```
[34]: Alice      Physics
      Jack     Chemistry
      Molly      English
      Sam        History
      dtype: object
```

```
[35]: # Now lets create a Series just for some new student Kelly, which lists all of␣
       ↪the courses
      # she has taken. We'll set the index to Kelly, and the data to be the names of␣
       ↪courses.
      kelly_classes = pd.Series(['Philosophy', 'Arts', 'Math'], index=['Kelly',␣
       ↪'Kelly', 'Kelly'])
      kelly_classes
```

```
[35]: Kelly    Philosophy
      Kelly          Arts
      Kelly          Math
      dtype: object
```

```
[36]: # Finally, we can append all of the data in this new Series to the first using␣
       ↪the .append()
      # function.
      all_students_classes = students_classes.append(kelly_classes)

      # This creates a series which has our original people in it as well as all of␣
       ↪Kelly's courses
      all_students_classes
```

```
[36]: Alice       Physics
      Jack      Chemistry
      Molly       English
      Sam         History
      Kelly    Philosophy
      Kelly          Arts
      Kelly          Math
      dtype: object
```

```
[37]: # There are a couple of important considerations when using append. First,␣
       ↪Pandas will take
      # the series and try to infer the best data types to use. In this example,␣
       ↪everything is a string,
      # so there's no problems here. Second, the append method doesn't actually␣
       ↪change the underlying Series
      # objects, it instead returns a new series which is made up of the two appended␣
       ↪together. This is
      # a common pattern in pandas - by default returning a new object instead of␣
       ↪modifying in place - and
      # one you should come to expect. By printing the original series we can see␣
       ↪that that series hasn't
      # changed.
      students_classes
```

```
[37]: Alice      Physics
      Jack     Chemistry
      Molly      English
      Sam        History
      dtype: object
```

```
[38]: # Finally, we see that when we query the appended series for Kelly, we don't␣
       ↪get a single value,
      # but a series itself.
      all_students_classes.loc['Kelly']
```

```
[38]: Kelly    Philosophy
      Kelly          Arts
      Kelly          Math
      dtype: object
```

In this lecture, we focused on one of the primary data types of the Pandas library, the Series. You learned how to query the Series, with .loc and .iloc, that the Series is an indexed data structure, how to merge two Series objects together with append(), and the importance of vectorization.

There are many more methods associated with the Series object that we haven't talked about. But with these basics down, we'll move on to talking about the Panda's two-dimensional data structure, the DataFrame. The DataFrame is very similar to the series object, but includes multiple columns of data, and is the structure that you'll spend the majority of your time working with when cleaning and aggregating data.