Querying a Series

A panda.Series can be queried, either by the index position or the index label.
If you don't give an index to the series, 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.
Here's an example using national sporting event data from Wikipedia.
Let's say we want to have a list of all of the sports in our index and a list of countries as values.
You might keep these in a dictionary.

If you wanted to see the fourth country on this, we would use the iloc attribute with the parameter 3.
If you wanted to see which country has golf as its national sport, we would use the loc attribute with parameter golf.
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 we'll call the indexing operator.

```
sports = {'Archery': 'Bhutan',
          'Golf': 'Scotland',
          'Sumo': 'Japan',
          'Taekwondo': 'South Korea'}
s = pd.Series(sports)
s
```

```
Archery           Bhutan
Golf            Scotland
Sumo               Japan
Taekwondo    South Korea
dtype: object
```

```
s.iloc[3]
```

```
'South Korea'
```

```
s.loc['Golf']
```

```
'Scotland'
```

```
s[3]
```

```
'South Korea'
```

```
s['Golf']
```

```
'Scotland'
```

Though in Python, this calls get and set an item methods 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.
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.
If you pass in an object, it will query as if you wanted to use the label based loc attribute.

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.
And the safer option is to be more explicit and use the iloc or loc attributes directly.

Here's an example using some new sports data, where countries are indexed by integer.
If we try and call s[0], we get a key error, because there's no item in the sports list with an index of zero.

```
sports = {99: 'Bhutan',
          100: 'Scotland',
          101: 'Japan',
          102: 'South Korea'}
s = pd.Series(sports)
```

```
s[0] #This won't call s.iloc[0] as one might expect, it generates an error instead
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-30-00fa51989f14> in <module>
----> 1 s[0] #This won't call s.iloc[0] as one might expect, it generates an error instead

~\Anaconda3\lib\site-packages\pandas\core\series.py in __getitem__(self, key)
   1066         key = com.apply_if_callable(key, self)
   1067         try:
-> 1068             result = self.index.get_value(self, key)
   1069
   1070             if not is_scalar(result):

~\Anaconda3\lib\site-packages\pandas\core\indexes\base.py in get_value(self, series, key)
   4728         k = self._convert_scalar_indexer(k, kind="getitem")
   4729         try:
-> 4730             return self._engine.get_value(s, k, tz=getattr(series.dtype, "tz", None))
   4731         except KeyError as e1:
   4732             if len(self) > 0 and (self.holds_integer() or self.is_boolean()):

pandas\_libs\index.pyx in pandas._libs.index.IndexEngine.get_value()
```

Instead we have to call iloc explicitly if we want the first item.

```
s.iloc[0]
```

```
'Bhutan'
```

Okay, so 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 want to do some sort of operation.
This could be trying to find a certain number, summarizing data or transforming the data in some way.

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 data frame of floating point values.

```
s = pd.Series([100.00, 120.00, 101.00, 3.00])
s
```

```
0    100.0
1    120.0
2    101.0
3      3.0
dtype: float64
```

Let's think of these as prices for different products.
We could write a little routine which iterates over all of the items in the series and adds them together to get a total.

```
total = 0
for item in s:
    total+=item
print(total)
```

```
324.0
```

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.

Here's how we would really write the code using the NumPy sum method.
First we need to import the numpy module, and then we just call np.sum and pass in an iterable item.
In this case, our panda series.

```
import numpy as np

total = np.sum(s)
print(total)
```

```
324.0
```

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.
You'll see this used a lot when demonstrating techniques with Pandas.

```
#this creates a big series of random numbers
s = pd.Series(np.random.randint(0,1000,10000))
s.head()
```

```
0    615
1    873
2    656
3    648
4    436
dtype: int32
```

Note that I've just used the head method, which reduces the amount of data printed out by the series to the first five elements.
We can actually verify that length of the series is correct using the len function.

```
len(s)
```

```
10000
```

Magic functions begin with a percentage sign.
If we type this sign and then hit the Tab key, we can see a list of the available magic functions.
You could write your own magic functions too.
We're actually going to use what's called a cellular magic function.
These start with two percentage signs and modify a raptor code in the current Jupyter cell.
The function we're going to use is called timeit.
And as you may have guessed from the name, 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, we'll use 1,000 loops.
I'll ask timeit here to use 100 runs because we're recording this.

```
%%timeit -n 100
summary = 0
for item in s:
    summary+=item
```

```
2.59 ms ± 83.7 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

Not bad.
Timeit ran this code and it doesn't seem like it takes very long at all.
Now let's try with vectorization.

```
%%timeit -n 100
summary = np.sum(s)
```

```
The slowest run took 7.21 times longer than the fastest. This could mean that an intermediate result is being cached.
348 µs ± 394 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

Wow!
This is a pretty shocking difference in the speed and demonstrates why data scientists need to be aware of parallel computing features and start thinking in functional programming terms.

Related feature in Pandas and NumPy 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.
Here I'll just use the head operator to just print out the top five rows in the series.

```
s+=2 #adds two to each item in s using broadcasting
s.head()
```

```
0    617
1    875
2    658
3    650
4    438
dtype: int32
```

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.
But if you find yourself iterating through a series, you should question whether you're doing things in the best possible way.

Here's how we would do this using the series *setvalue* method.

Let's try and time the two approaches.

```
for label, value in s.iteritems():
    s.set_value(label, value+2)
s.head()
```

```
C:\Users\a\Anaconda3\lib\site-packages\ipykernel_launcher.py:2: FutureWarning: set_value is deprecated and will be removed in a
future release. Please use .at[] or .iat[] accessors instead
```

```
0    107
1    703
2    492
3    752
4    990
dtype: int32
```

```
for label, value in s.iteritems():
    s.at[label]= value+2
s.head()
```

```
0    105
1    701
2    490
3    750
4    988
dtype: int32
```

Amazing. Not only is it significantly faster, but it's more concise and maybe even easier to read too.

The typical mathematical operations you would expect are vectorized, and the NumPy documentation outlines what it takes to create vectorized functions of your own.

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.

```
%%timeit -n 10
s = pd.Series(np.random.randint(0,1000,10000))
for label, value in s.iteritems():
    s.loc[label]= value+2
```

```
8.72 s ± 112 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

```
%%timeit -n 10
s = pd.Series(np.random.randint(0,1000,10000))
s+=2
```

```
810 µs ± 205 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

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.

Here's an example using a series of a few numbers.
We could add some new value, maybe an animal, just by calling the .loc indexing operator.
We see that mixed types for data values or index labels are no problem for Pandas.

```
s = pd.Series([1, 2, 3])
s.loc['Animal'] = 'Bears'
s
```

```
0             1
1             2
2             3
Animal    Bears
dtype: object
```

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 data frames different, conceptually, that a relational database might be.

Revisiting the issue of countries and their national sports, it turns out that many countries seem to like this game cricket.
We go back to our original series on sports. It's possible to create a new series object with multiple entries for cricket, and then use *append* to bring these together.

```
original_sports = pd.Series({'Archery': 'Bhutan',
                             'Golf': 'Scotland',
                             'Sumo': 'Japan',
                             'Taekwondo': 'South Korea'})
cricket_loving_countries = pd.Series(['Australia',
                                      'Barbados',
                                      'Pakistan',
                                      'England'],
                                     index=['Cricket',
                                            'Cricket',
                                            'Cricket',
                                            'Cricket'])
all_countries = original_sports.append(cricket_loving_countries)
```

There are a couple of important considerations when using append.
*First, Pandas is going to take your series and try to infer the best data types to use.
In this example, everything is a string, so there's no problem here.

*Second, the append method doesn't actually change the underlying series.
It instead returns a new series which is made up of the two appended together.
We can see this by going back and printing the original series of values and seeing that they haven't changed.

```
original_sports
```

```
Archery           Bhutan
Golf            Scotland
Sumo               Japan
Taekwondo    South Korea
dtype: object
```

This is actually a significant issue for new Pandas users who are used to objects being changed in place.
So watch out for it, not just with append but with other Pandas functions as well.

Finally, we see that when we query the appended series for those who have cricket as their national sport, we don't get a single value, but a series itself.
This is actually very common, and if you have a relational database background, this is very similar to every table query resulting in a return set which itself is a table.

```
cricket_loving_countries
```

```
Cricket    Australia
Cricket     Barbados
Cricket     Pakistan
Cricket      England
dtype: object
```

```
all_countries
```

```
Archery           Bhutan
Golf            Scotland
Sumo               Japan
Taekwondo    South Korea
Cricket        Australia
Cricket         Barbados
Cricket         Pakistan
Cricket          England
dtype: object
```

```
all_countries.loc['Cricket']
```

```
Cricket    Australia
Cricket     Barbados
Cricket     Pakistan
Cricket      England
dtype: object
```

In this lecture, we focused on one of the primary data types of the Pandas library, the series. There are many more methods associated with this 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 data frame.
The data frame 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.