Querying DataFrames

In this lecture we're going to talk about querying DataFrames. The first step in the process is to understand Boolean masking. Boolean masking is the heart of fast and efficient querying in numpy and pandas, and its analogous to bit masking used in other areas of computational science. By the end of this lecture you'll understand how Boolean masking works, and how to apply this to a DataFrame to get out data you're interested in.

A Boolean mask is an array which can be of one dimension like a series, or two dimensions like a data frame, where each of the values in the array are either true or false. This array is essentially overlaid on top of the data structure that we're querying. And any cell aligned with the true value will be admitted into our final result, and any cell aligned with a false value will not.

Load Data

In [1]:

```
# Let's start with an example and import our graduate admission dataset. First w
e'll bring in pandas
import pandas as pd
# Then we'll load in our CSV file
df = pd.read_csv('datasets/Admission_Predict.csv', index_col=0)
# And we'll clean up a couple of poorly named columns like we did in a previous
 lecture
df.columns = [x.lower().strip() for x in df.columns]
# And we'll take a look at the results
df.head()
```

Out[1]:

	gre score	toefl score	university rating	sop	lor	cgpa	research	chance of admit
Serial No.								
1	337	118	4	4.5	4.5	9.65	1	0.92
2	324	107	4	4.0	4.5	8.87	1	0.76
3	316	104	3	3.0	3.5	8.00	1	0.72
4	322	110	3	3.5	2.5	8.67	1	0.80
5	314	103	2	2.0	3.0	8.21	0	0.65

Creating Boolean Masks

Broadcasting A Comparison Operator

Boolean masks are created by applying operators directly to the pandas Series or DataFrame objects. For instance, in our graduate admission dataset, we might be interested in seeing only those students that have a chance higher than 0.7.

To build a Boolean mask for this query, we want to project the chance of admit column using the indexing operator and apply the greater than operator with a comparison value of 0.7. This is essentially **broadcasting a comparison operator**, greater than `>`, with the results being returned as **a Boolean Series**. The resultant Series is indexed where the value of each cell is either `True` or `False` depending on whether a student has a chance of admit higher than 0.7

In [2]:

```
admit_mask=df['chance of admit'] > 0.7
admit_mask
```

Out[2]:

```
Serial No.
1       True
2       True
3       True
4       True
5      False
       ...
396     True
397     True
398     True
399    False
400     True
Name: chance of admit, Length: 400, dtype: bool
```

This is pretty fundamental, so take a moment to look at this.

- The **result of broadcasting a comparison operator is a Boolean mask** - true or false values depending upon the results of the comparison.
- Underneath, **pandas** is applying the comparison operator you **specified through vectorization** to all of the values in the array you specified which, in this case, is the chance of admit column of the dataframe.
- The result is a **series**, since **only one column is being operator on**, filled with either True or False values, which is what the comparison operator returns.

Data Manipulation Using Boolean Masks

In [3]:

```
# So, what do you do with the boolean mask once you have formed it? Well, you ca
n just lay it on top of the
# data to "hide" the data you don't want, which is represented by all of the Fal
se values. We do this by using
# the .where() function on the original DataFrame.
df.where(admit_mask).head()
```

Out[3]:

	gre score	toefl score	university rating	sop	lor	cgpa	research	chance of admit
Serial No.								
1	337.0	118.0	4.0	4.5	4.5	9.65	1.0	0.92
2	324.0	107.0	4.0	4.0	4.5	8.87	1.0	0.76
3	316.0	104.0	3.0	3.0	3.5	8.00	1.0	0.72
4	322.0	110.0	3.0	3.5	2.5	8.67	1.0	0.80
5	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

Calling .dropna( )

We see that the resulting data frame keeps the original indexed values, and only data which met the condition was retained. **All of the rows which did not meet the condition have NaN data instead**, but **these rows were not dropped from our dataset.**

The next step is, if we don't want the NaN data, we use the dropna() function

In [4]:

```
df.where(admit_mask).dropna().head()
```

Out[4]:

	gre score	toefl score	university rating	sop	lor	cgpa	research	chance of admit
Serial No.								
1	337.0	118.0	4.0	4.5	4.5	9.65	1.0	0.92
2	324.0	107.0	4.0	4.0	4.5	8.87	1.0	0.76
3	316.0	104.0	3.0	3.0	3.5	8.00	1.0	0.72
4	322.0	110.0	3.0	3.5	2.5	8.67	1.0	0.80
6	330.0	115.0	5.0	4.5	3.0	9.34	1.0	0.90

Shorthand for dropna()

In [5]:

```
# The returned DataFrame now has all of the NaN rows dropped. Notice the index n
ow includes
# one through four and six, but not five.
#
# Despite being really handy, where() isn't actually used that often. Instead, t
he pandas devs
# created a shorthand syntax which combines where() and dropna(), doing both at
 once. And, in
# typical fashion, the just overloaded the indexing operator to do this!

df[df['chance of admit'] > 0.7].head()
```

Out[5]:

	gre score	toefl score	university rating	sop	lor	cgpa	research	chance of admit
Serial No.								
1	337	118	4	4.5	4.5	9.65	1	0.92
2	324	107	4	4.0	4.5	8.87	1	0.76
3	316	104	3	3.0	3.5	8.00	1	0.72
4	322	110	3	3.5	2.5	8.67	1	0.80
6	330	115	5	4.5	3.0	9.34	1	0.90

Sanity Check [ ]

I personally find this much harder to read, but it's also very more common when you're reading other people's code, so it's important to be able to understand it. Just reviewing this indexing operator on DataFrame, it now does three things:

- It can be called with a string parameter to project a single column `df["gre score"]`
- You can send it a list of columns as strings `df[["gre score","toefl score"]]`
- You can send it a Boolean Mask `df[df["gre score"]>320]`

Each of these is mimicing functionality from either .loc() or .where(), or .dropna().

In [6]:

```
# It can be called with a string parameter to project a single column
print(df["gre score"].head())
type(df["gre score"])
```

```
Serial No.
1    337
2    324
3    316
4    322
5    314
Name: gre score, dtype: int64
```

Out[6]:

```
pandas.core.series.Series
```

In [7]:

```
# Or you can send it a list of columns as strings
print(df[["gre score","toefl score"]].head())
type(df[["gre score","toefl score"]])
```

```
            gre score  toefl score
Serial No.                        
1                 337          118
2                 324          107
3                 316          104
4                 322          110
5                 314          103
```

Out[7]:

```
pandas.core.frame.DataFrame
```

In [8]:

```
# Or you can send it a boolean mask
print(df[df["gre score"]>320].head())
type(df[df["gre score"]>320])
```

```
            gre score  toefl score  university rating  sop  lor  cgp
a  \
Serial No.
1                 337          118                  4  4.5  4.5  9.6
5
2                 324          107                  4  4.0  4.5  8.8
7
4                 322          110                  3  3.5  2.5  8.6
7
6                 330          115                  5  4.5  3.0  9.3
4
7                 321          109                  3  3.0  4.0  8.2
0

            research  chance of admit
Serial No.
1                  1             0.92
2                  1             0.76
4                  1             0.80
6                  1             0.90
7                  1             0.75
```

Out[8]:

```
pandas.core.frame.DataFrame
```

Combining Multiple Boolean Masks

Before we leave this, lets talk about combining multiple boolean masks, such as multiple criteria for including. In bitmasking in other places in computer science this is done with "and", if both masks must be True for a True value to be in the final mask), or "or" if only one needs to be True.

Unfortunately, it doesn't feel quite as natural in pandas. For instance, if you want to take two boolean series and and them together.

In [9]:

```
try:
    (df['chance of admit'] > 0.7) and (df['chance of admit'] < 0.9)
except:
    print("ValueError")
```

```
ValueError
```

This doesn't work. And despite using pandas for awhile, I still find I regularly try and do this. The problem is that you have series objects, and **python underneath doesn't know how to compare two series using and or or.**

Using Pipe `|` and Ampersand `&` Operators

Instead, the pandas authors have overwritten the pipe | and ampersand & operators to handle this for us

One thing to watch out for is order of operations! A common error for new pandas users is to try and do boolean comparisons using the & operator **but not putting parentheses around the individual terms you are interested in**

In [10]:

```
(df['chance of admit'] > 0.7) & (df['chance of admit'] < 0.9)
```

Out[10]:

```
Serial No.
1      False
2       True
3       True
4       True
5      False
       ...
396     True
397     True
398    False
399    False
400    False
Name: chance of admit, Length: 400, dtype: bool
```

In [11]:

```
# Example of not putting parentheses around the individual terms you are interes
ted in
try:
    df['chance of admit'] > 0.7 & df['chance of admit'] < 0.9
except:
    print("ValueError & TypeError")
```

```
ValueError & TypeError
```

The problem is that Python is trying to bitwise and a 0.7 and a pandas dataframe, when you really want to bitwise and the broadcasted dataframes together.

Using built-in Functions `.gt()` and `.lt()`

Another way to do this is to just get rid of the comparison operator completely, and instead use the built in functions which mimic this approach

```
In [12]:
```

```python
df['chance of admit'].gt(0.7) & df['chance of admit'].lt(0.9)
```

```
Out[12]:
```

```
Serial No.
1      False
2       True
3       True
4       True
5      False
       ...  
396     True
397     True
398    False
399    False
400    False
Name: chance of admit, Length: 400, dtype: bool
```

```
In [13]:
```

```python
# These functions are build right into the Series and DataFrame objects, so you
 can chain them
# too, which results in the same answer and the use of no visual operators. You
 can decide what
# looks best for you
df['chance of admit'].gt(0.7).lt(0.9)
```

```
Out[13]:
```

```
Serial No.
1      False
2      False
3      False
4      False
5       True
       ...  
396    False
397    False
398    False
399     True
400    False
Name: chance of admit, Length: 400, dtype: bool
```

This only works if you operator, such as less than or greater than, is built into the DataFrame, but I certainly find that last code example much more readable than one with ampersands and parenthesis.

You need to be able to read and write all of these, and **understand the implications of the route** you are choosing. It's worth really going back and rewatching this lecture to make sure you have it. I would say 50% or more of the work you'll be doing in data cleaning involves querying DataFrames.

Summary

In this lecture, we have learned to query dataframe using boolean masking, which is extremely important and often used in the world of data science. With boolean masking, we can select data based on the criteria we desire and, frankly, you'll use it everywhere. We've also seen how there are many different ways to query the DataFrame, and the interesting side implications that come up when doing so.