PandasIntroduction

November 15, 2021

This week we're going to deepen our investigation to how Python can be used to manipulate, clean, and query data by looking at the Pandas data tool kit. Pandas was created by Wes McKinney in 2008, and is an open source project under a very permissive license. As an open source project it's got a strong community, with over one hundred software developers all committing code to help make it better. Before pandas existed we had only a hodge podge of tools to use, such as numpy, the python core libraries, and some python statistical tools. But pandas has quickly become the defacto library for representing relational data for data scientists.

I want to take a moment here to introduce the question answering site Stack Overflow. Stack Overflow is used broadly within the software development community to post questions about programming, programming languages, and programming toolkits. What's special about Stack Overflow is that it's heavily curated by the community. And the Pandas community, in particular, uses it as their number one resource for helping new members. It's quite possible if you post a question to Stack Overflow, and tag it as being Pandas and Python related, that a core Pandas developer will actually respond to your question. In addition to posting questions, Stack Overflow is a great place to go to see what issues people are having and how they can be solved. You can learn a lot from browsing Stacks at Stack Overflow and with pandas, this is where the developer community is.

A second resource you might want to consider are books. In 2012 Wes McKinney wrote the definitive Pandas reference book called Python for Data Analysis and published by O'Reilly, and it's recently been update to a second edition. I consider this the go to book for understanding how Pandas works. I also appreciate the more brief book "Learning the Pandas Library" by Matt Harrison. It's not a comprehensive book on data analysis and statistics. But if you just want to learn the basics of Pandas and want to do so quickly, I think it's a well laid out volume and it can be had for a good price.

The field of data science is rapidly changing. There's new toolkits and method being created everyday. It can be tough to stay on top of it all. Marco Rodriguez and Tim Golden maintain a wonderful blog aggregator site called Planet Python. You can visit the webpage at planet-python.org, subscribe with an RSS reader, or get the latest articles from the @PlanetPython Twitter feed. There's lots of regular Python data science contributors, and I highly recommend it if you follow RSS feeds.

Here's my last plug on how to deepen your learning. Kyle Polich runs an excellent podcast called Data Skeptic. It isn't Python based per se, but it's well produced and it has a wonderful mixture of interviews with experts in the field as well as short educational lessons. Much of the word he describes is specific to machine learning methods. But if that's something you are planning to explore through this specialization this course is in, I would really encourage you to subscribe to his podcast.

That's it for a little bit of an introduction to this week of the course. Next we're going to dive right into Pandas library and talk about the series data structure.

SeriesDataStructure_ed

November 15, 2021

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
```

[2]:
```
# 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
pd.Series(students)
```

[2]:
```
0    Alice
1     Jack
2    Molly
dtype: object
```

[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
```

[4]:
```
# 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
pd.Series(numbers)
```

[4]:
```
0    1
1    2
2    3
dtype: int64
```

[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?

# 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.
```

[9]:
```python
# 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
import numpy as np
# And lets compare it to None
np.nan == None
```

[9]:
```
False
```

[10]:
```python
# 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
```

[10]:
```
False
```

[11]:
```python
# 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]:
```python
# 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.
```

[13]:
```python
# 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)
s
```

[13]:
```
Alice      Physics
Jack     Chemistry
Molly      English
dtype: object
```

[14]:
```python
# 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.
```

[15]:
```python
# Once the series has been created, we can get the index object using the index␣
↪attribute.

s.index
```

[15]:
```
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.
```

[19]:
```
# 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
```

[19]:
```
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.

QueryingSeries_ed

November 15, 2021

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)

    <ipython-input-8-bd1f5b262fbc> in <module>
      2 # an index of zero, instead we have to call iloc explicitly if we␣
↪want the first item.
      3
  ----> 4 s[0]


    /opt/conda/lib/python3.7/site-packages/pandas/core/series.py in␣
↪__getitem__(self, key)
   1062         key = com.apply_if_callable(key, self)
   1063         try:
-> 1064             result = self.index.get_value(self, key)
   1065
   1066             if not is_scalar(result):


    /opt/conda/lib/python3.7/site-packages/pandas/core/indexes/base.py in␣
↪get_value(self, series, key)
   4721         k = self._convert_scalar_indexer(k, kind="getitem")
   4722         try:
-> 4723             return self._engine.get_value(s, k, tz=getattr(series.
↪dtype, "tz", None))
   4724         except KeyError as e1:
   4725             if len(self) > 0 and (self.holds_integer() or self.
↪is_boolean()):


    pandas/_libs/index.pyx in pandas._libs.index.IndexEngine.get_value()


    pandas/_libs/index.pyx in pandas._libs.index.IndexEngine.get_value()


    pandas/_libs/index.pyx in pandas._libs.index.IndexEngine.get_loc()
```

```
pandas/_libs/hashtable_class_helper.pxi in pandas._libs.hashtable.
↪Int64HashTable.get_item()


pandas/_libs/hashtable_class_helper.pxi in pandas._libs.hashtable.
↪Int64HashTable.get_item()


KeyError: 0
```

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

[ ]:
```
# 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.
```

[ ]:
```
# 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))
```

[ ]:
```
# 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
```

```
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))
```

[ ]:
```
# 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()
```

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

[ ]:
```
# 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.
```

[ ]:
```
# 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
```

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

total/len(numbers)
```

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

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

[ ]:
```
# 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.
```

[ ]:
```
# 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()
```

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

[ ]:
```
# 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():
    # now for the item which is returned, lets call set_value()
    numbers.set_value(label, value+2)
# And we can check the result of this computation
numbers.head()
```

[ ]:
```
# 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.
```

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

[ ]:
```
%%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
```

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

[ ]:
```
%%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
```

[ ]:
```
# 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.
```

[ ]:
```
# 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.
```

[ ]:
```
# 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
```

[ ]:
```
# 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
```

[ ]:
```
# 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
```

[ ]:
```
# 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
```

[ ]:
```
# 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
```

[ ]:
```
# 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
```

[ ]:
```
# 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']
```

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.

DataFrameDataStructure_ed

November 15, 2021

The DataFrame data structure is the heart of the Panda's library. It's a primary object that you'll be working with in data analysis and cleaning tasks.

The DataFrame is conceptually a two-dimensional series object, where there's an index and multiple columns of content, with each column having a label. In fact, the distinction between a column and a row is really only a conceptual distinction. And you can think of the DataFrame itself as simply a two-axes labeled array.

[1]:
```
# Lets start by importing our pandas library
import pandas as pd
```

[2]:
```
# I'm going to jump in with an example. Lets create three school records for␣
↪students and their
# class grades. I'll create each as a series which has a student name, the␣
↪class name, and the score.
record1 = pd.Series({'Name': 'Alice',
                        'Class': 'Physics',
                        'Score': 85})
record2 = pd.Series({'Name': 'Jack',
                        'Class': 'Chemistry',
                        'Score': 82})
record3 = pd.Series({'Name': 'Helen',
                        'Class': 'Biology',
                        'Score': 90})
```

[3]:
```
# Like a Series, the DataFrame object is index. Here I'll use a group of␣
↪series, where each series
# represents a row of data. Just like the Series function, we can pass in our␣
↪individual items
# in an array, and we can pass in our index values as a second arguments
df = pd.DataFrame([record1, record2, record3],
                  index=['school1', 'school2', 'school1'])

# And just like the Series we can use the head() function to see the first␣
↪several rows of the
# dataframe, including indices from both axes, and we can use this to verify␣
↪the columns and the rows
df.head()
```

[3]:
```
          Name      Class  Score
school1  Alice    Physics     85
school2   Jack  Chemistry     82
school1  Helen    Biology     90
```

[4]:
```
# You'll notice here that Jupyter creates a nice bit of HTML to render the␣
 ↪results of the
# dataframe. So we have the index, which is the leftmost column and is the␣
 ↪school name, and
# then we have the rows of data, where each row has a column header which was␣
 ↪given in our initial
# record dictionaries
```

[5]:
```
# An alternative method is that you could use a list of dictionaries, where␣
 ↪each dictionary
# represents a row of data.

students = [{'Name': 'Alice',
              'Class': 'Physics',
              'Score': 85},
            {'Name': 'Jack',
             'Class': 'Chemistry',
             'Score': 82},
            {'Name': 'Helen',
             'Class': 'Biology',
             'Score': 90}]

# Then we pass this list of dictionaries into the DataFrame function
df = pd.DataFrame(students, index=['school1', 'school2', 'school1'])
# And lets print the head again
df.head()
```

[5]:
```
          Name      Class  Score
school1  Alice    Physics     85
school2   Jack  Chemistry     82
school1  Helen    Biology     90
```

[6]:
```
# Similar to the series, we can extract data using the .iloc and .loc␣
 ↪attributes. Because the
# DataFrame is two-dimensional, passing a single value to the loc indexing␣
 ↪operator will return
# the series if there's only one row to return.

# For instance, if we wanted to select data associated with school2, we would␣
 ↪just query the
# .loc attribute with one parameter.
df.loc['school2']
```

[6]:
```
Name            Jack
Class      Chemistry
Score             82
Name: school2, dtype: object
```

[7]:
```
# You'll note that the name of the series is returned as the index value, while
 ↪the column
# name is included in the output.

# We can check the data type of the return using the python type function.
type(df.loc['school2'])
```

[7]:
```
pandas.core.series.Series
```

[8]:
```
# It's important to remember that the indices and column names along either
 ↪axes horizontal or
# vertical, could be non-unique. In this example, we see two records for
 ↪school1 as different rows.
# If we use a single value with the DataFrame lock attribute, multiple rows of
 ↪the DataFrame will
# return, not as a new series, but as a new DataFrame.

# Lets query for school1 records
df.loc['school1']
```

[8]:

	Name	Class	Score
school1	Alice	Physics	85
school1	Helen	Biology	90

[9]:
```
# And we can see the the type of this is different too
type(df.loc['school1'])
```

[9]:
```
pandas.core.frame.DataFrame
```

[10]:
```
# One of the powers of the Panda's DataFrame is that you can quickly select
 ↪data based on multiple axes.
# For instance, if you wanted to just list the student names for school1, you
 ↪would supply two
# parameters to .loc, one being the row index and the other being the column
 ↪name.

# For instance, if we are only interested in school1's student names
df.loc['school1', 'Name']
```

[10]:
```
school1    Alice
school1    Helen
Name: Name, dtype: object
```

[11]:
```
# Remember, just like the Series, the pandas developers have implemented this
 ↪using the indexing
# operator and not as parameters to a function.
```

```
# What would we do if we just wanted to select a single column though? Well,␣
↪there are a few
# mechanisms. Firstly, we could transpose the matrix. This pivots all of the␣
↪rows into columns
# and all of the columns into rows, and is done with the T attribute
df.T
```

[11]:

```
        school1    school2  school1
Name      Alice       Jack    Helen
Class   Physics  Chemistry  Biology
Score        85         82       90
```

[12]:

```
# Then we can call .loc on the transpose to get the student names only
df.T.loc['Name']
```

[12]:

```
school1    Alice
school2     Jack
school1    Helen
Name: Name, dtype: object
```

[13]:

```
# However, since iloc and loc are used for row selection, Panda reserves the␣
↪indexing operator
# directly on the DataFrame for column selection. In a Panda's DataFrame,␣
↪columns always have a name.
# So this selection is always label based, and is not as confusing as it was␣
↪when using the square
# bracket operator on the series objects. For those familiar with relational␣
↪databases, this operator
# is analogous to column projection.
df['Name']
```

[13]:

```
school1    Alice
school2     Jack
school1    Helen
Name: Name, dtype: object
```

[14]:

```
# In practice, this works really well since you're often trying to add or drop␣
↪new columns. However,
# this also means that you get a key error if you try and use .loc with a␣
↪column name
df.loc['Name']
```

```
        ␣
↪---------------------------------------------------------------------------

      KeyError                                  Traceback (most recent call␣
↪last)
```

```
/opt/conda/lib/python3.7/site-packages/pandas/core/indexes/base.py in␣
↪get_loc(self, key, method, tolerance)
   2889             try:
-> 2890                 return self._engine.get_loc(key)
   2891             except KeyError:


pandas/_libs/index.pyx in pandas._libs.index.IndexEngine.get_loc()


pandas/_libs/index.pyx in pandas._libs.index.IndexEngine.get_loc()


pandas/_libs/index.pyx in pandas._libs.index.IndexEngine.
↪_get_loc_duplicates()


pandas/_libs/index.pyx in pandas._libs.index.IndexEngine.
↪_maybe_get_bool_indexer()


KeyError: 'Name'


During handling of the above exception, another exception occurred:


KeyError                                  Traceback (most recent call␣
↪last)

<ipython-input-14-b44ae13e0b9f> in <module>
      1 # In practice, this works really well since you're often trying to␣
↪add or drop new columns. However,
      2 # this also means that you get a key error if you try and use .loc␣
↪with a column name
----> 3 df.loc['Name']


/opt/conda/lib/python3.7/site-packages/pandas/core/indexing.py in␣
↪__getitem__(self, key)
   1408
   1409             maybe_callable = com.apply_if_callable(key, self.obj)
-> 1410             return self._getitem_axis(maybe_callable, axis=axis)
   1411
   1412     def _is_scalar_access(self, key: Tuple):
```

```
    /opt/conda/lib/python3.7/site-packages/pandas/core/indexing.py in␣
↪_getitem_axis(self, key, axis)
   1823         # fall thru to straight lookup
   1824         self._validate_key(key, axis)
-> 1825         return self._get_label(key, axis=axis)
   1826
   1827


    /opt/conda/lib/python3.7/site-packages/pandas/core/indexing.py in␣
↪_get_label(self, label, axis)
    155             raise IndexingError("no slices here, handle elsewhere")
    156
--> 157         return self.obj._xs(label, axis=axis)
    158
    159     def _get_loc(self, key: int, axis: int):


    /opt/conda/lib/python3.7/site-packages/pandas/core/generic.py in␣
↪xs(self, key, axis, level, drop_level)
   3736             loc, new_index = self.index.get_loc_level(key,␣
↪drop_level=drop_level)
   3737         else:
-> 3738             loc = self.index.get_loc(key)
   3739
   3740             if isinstance(loc, np.ndarray):


    /opt/conda/lib/python3.7/site-packages/pandas/core/indexes/base.py in␣
↪get_loc(self, key, method, tolerance)
   2890                 return self._engine.get_loc(key)
   2891             except KeyError:
-> 2892                 return self._engine.get_loc(self.
↪_maybe_cast_indexer(key))
   2893         indexer = self.get_indexer([key], method=method,␣
↪tolerance=tolerance)
   2894         if indexer.ndim > 1 or indexer.size > 1:


    pandas/_libs/index.pyx in pandas._libs.index.IndexEngine.get_loc()


    pandas/_libs/index.pyx in pandas._libs.index.IndexEngine.get_loc()


    pandas/_libs/index.pyx in pandas._libs.index.IndexEngine.
↪_get_loc_duplicates()
```

```
pandas/_libs/index.pyx in pandas._libs.index.IndexEngine.
↪_maybe_get_bool_indexer()


KeyError: 'Name'
```

[ ]:
```
# Note too that the result of a single column projection is a Series object
type(df['Name'])
```

[ ]:
```
# Since the result of using the indexing operator is either a DataFrame or␣
↪Series, you can chain
# operations together. For instance, we can select all of the rows which␣
↪related to school1 using
# .loc, then project the name column from just those rows
df.loc['school1']['Name']
```

[ ]:
```
# If you get confused, use type to check the responses from resulting␣
↪operations
print(type(df.loc['school1'])) #should be a DataFrame
print(type(df.loc['school1']['Name'])) #should be a Series
```

[ ]:
```
# Chaining, by indexing on the return type of another index, can come with some␣
↪costs and is
# best avoided if you can use another approach. In particular, chaining tends␣
↪to cause Pandas
# to return a copy of the DataFrame instead of a view on the DataFrame.
# For selecting data, this is not a big deal, though it might be slower than␣
↪necessary.
# If you are changing data though this is an important distinction and can be a␣
↪source of error.
```

[ ]:
```
# Here's another approach. As we saw, .loc does row selection, and it can take␣
↪two parameters,
# the row index and the list of column names. The .loc attribute also supports␣
↪slicing.

# If we wanted to select all rows, we can use a colon to indicate a full slice␣
↪from beginning to end.
# This is just like slicing characters in a list in python. Then we can add the␣
↪column name as the
# second parameter as a string. If we wanted to include multiple columns, we␣
↪could do so in a list.
# and Pandas will bring back only the columns we have asked for.
```

```
# Here's an example, where we ask for all the names and scores for all schools␣
 ↪using the .loc operator.
df.loc[:,['Name', 'Score']]
```

[ ]:
```
# Take a look at that again. The colon means that we want to get all of the␣
 ↪rows, and the list
# in the second argument position is the list of columns we want to get back
```

[ ]:
```
# That's selecting and projecting data from a DataFrame based on row and column␣
 ↪labels. The key
# concepts to remember are that the rows and columns are really just for our␣
 ↪benefit. Underneath
# this is just a two axes labeled array, and transposing the columns is easy.␣
 ↪Also, consider the
# issue of chaining carefully, and try to avoid it, as it can cause␣
 ↪unpredictable results, where
# your intent was to obtain a view of the data, but instead Pandas returns to␣
 ↪you a copy.
```

[ ]:
```
# Before we leave the discussion of accessing data in DataFrames, lets talk␣
 ↪about dropping data.
# It's easy to delete data in Series and DataFrames, and we can use the drop␣
 ↪function to do so.
# This function takes a single parameter, which is the index or row label, to␣
 ↪drop. This is another
# tricky place for new users -- the drop function doesn't change the DataFrame␣
 ↪by default! Instead,
# the drop function returns to you a copy of the DataFrame with the given rows␣
 ↪removed.

df.drop('school1')
```

[ ]:
```
# But if we look at our original DataFrame we see the data is still intact.
df
```

[ ]:
```
# Drop has two interesting optional parameters. The first is called inplace,␣
 ↪and if it's
# set to true, the DataFrame will be updated in place, instead of a copy being␣
 ↪returned.
# The second parameter is the axes, which should be dropped. By default, this␣
 ↪value is 0,
# indicating the row axis. But you could change it to 1 if you want to drop a␣
 ↪column.

# For example, lets make a copy of a DataFrame using .copy()
copy_df = df.copy()
# Now lets drop the name column in this copy
copy_df.drop("Name", inplace=True, axis=1)
```

```
copy_df
```

```
[ ]: # There is a second way to drop a column, and that's directly through the use␣
     ↪of the indexing
     # operator, using the del keyword. This way of dropping data, however, takes␣
     ↪immediate effect
     # on the DataFrame and does not return a view.
     del copy_df['Class']
     copy_df
```

```
[ ]: # Finally, adding a new column to the DataFrame is as easy as assigning it to␣
     ↪some value using
     # the indexing operator. For instance, if we wanted to add a class ranking␣
     ↪column with default
     # value of None, we could do so by using the assignment operator after the␣
     ↪square brackets.
     # This broadcasts the default value to the new column immediately.

     df['ClassRanking'] = None
     df
```

In this lecture you've learned about the data structure you'll use the most in pandas, the DataFrame. The dataframe is indexed both by row and column, and you can easily select individual rows and project the columns you're interested in using the familiar indexing methods from the Series class. You'll be gaining a lot of experience with the DataFrame in the content to come.

DataFrameIndexingAndLoading_ed

November 15, 2021

In this course, we'll be largely using smaller or moderate-sized datasets. A common workflow is to read the dataset in, usually from some external file, then begin to clean and manipulate the dataset for analysis. In this lecture I'm going to demonstrate how you can load data from a comma separated file into a DataFrame.

[1]:
```
# Lets just jump right in and talk about comma separated values (csv) files.␣
↪You've undoubtedly used these -
# any spreadsheet software like excel or google sheets can save output in CSV␣
↪format. It's pretty loose as a
# format, and incredibly lightweight. And totally ubiquitous.

# Now, I'm going to make a quick aside because it's convenient here. The␣
↪Jupyter notebooks use ipython as the
# kernel underneath, which provides convenient ways to integrate lower level␣
↪shell commands, which are
# programs run in the underlying operating system. If you're not familiar with␣
↪the shell don't worry too much
# about this, but if you are, this is super handy for integration of your data␣
↪science workflows. I want to
# use one shell command here called "cat", for "concatenate", which just␣
↪outputs the contents of a file. In
# ipython if we prepend the line with an exclamation mark it will execute the␣
↪remainder of the line as a shell
# command.  So lets look at the content of a CSV file
!cat datasets/Admission_Predict.csv
```

[2]:
```
# We see from the output that there is a list of columns, and the column␣
↪identifiers are listed as strings on
# the first line of the file. Then we have rows of data, all columns separated␣
↪by commas. Now, there are lots
# of oddities with the CSV file format, and there is no one agreed upon␣
↪specification. So you should be
# prepared to do a bit of work when you pull down CSV files to explore. But␣
↪this lecture isn't focused on CSV
# files, and is more about pandas DataFrames. So lets jump into that.

# Let's bring in pandas to work with
import pandas as pd

# Pandas mades it easy to turn a CSV into a dataframe, we just call read_csv()
df = pd.read_csv('datasets/Admission_Predict.csv')

# And let's look at the first few rows
df.head()
```

[2]:
```
   Serial No.  GRE Score  TOEFL Score  University Rating  SOP  LOR   CGPA  \
0           1        337          118                  4  4.5   4.5  9.65
1           2        324          107                  4  4.0   4.5  8.87
2           3        316          104                  3  3.0   3.5  8.00
3           4        322          110                  3  3.5   2.5  8.67
4           5        314          103                  2  2.0   3.0  8.21

   Research  Chance of Admit
0         1             0.92
1         1             0.76
2         1             0.72
3         1             0.80
4         0             0.65
```

[3]:
```
# We notice that by default index starts with 0 while the students' serial␣
↪number starts from 1. If you jump
# back to the CSV output you'll deduce that pandas has create a new index.␣
↪Instead, we can set the serial no.
# as the index if we want to by using the index_col.
df = pd.read_csv('datasets/Admission_Predict.csv', index_col=0)
df.head()
```

[3]:
```
             GRE Score  TOEFL Score  University Rating  SOP  LOR   CGPA  \
Serial No.
1                  337          118                  4  4.5   4.5  9.65
2                  324          107                  4  4.0   4.5  8.87
3                  316          104                  3  3.0   3.5  8.00
4                  322          110                  3  3.5   2.5  8.67
5                  314          103                  2  2.0   3.0  8.21

             Research  Chance of Admit
Serial No.
1                   1             0.92
2                   1             0.76
3                   1             0.72
4                   1             0.80
5                   0             0.65
```

[4]:
```
# Notice that we have two columns "SOP" and "LOR" and probably not everyone␣
 ↪knows what they mean So let's
# change our column names to make it more clear. In Pandas, we can use the␣
 ↪rename() function It takes a
# parameter called columns, and we need to pass into a dictionary which the␣
 ↪keys are the old column name and
# the value is the corresponding new column name
new_df=df.rename(columns={'GRE Score':'GRE Score', 'TOEFL Score':'TOEFL Score',
                   'University Rating':'University Rating',
                   'SOP': 'Statement of Purpose','LOR': 'Letter of␣
 ↪Recommendation',
                   'CGPA':'CGPA', 'Research':'Research',
                   'Chance of Admit':'Chance of Admit'})
new_df.head()
```

[4]:
```
             GRE Score  TOEFL Score  University Rating  Statement of Purpose  \
Serial No.
1                  337          118                  4                   4.5
2                  324          107                  4                   4.0
3                  316          104                  3                   3.0
4                  322          110                  3                   3.5
5                  314          103                  2                   2.0

             LOR   CGPA  Research  Chance of Admit
Serial No.
1            4.5  9.65         1             0.92
2            4.5  8.87         1             0.76
3            3.5  8.00         1             0.72
4            2.5  8.67         1             0.80
5            3.0  8.21         0             0.65
```

[5]:
```
# From the output, we can see that only "SOP" is changed but not "LOR" Why is␣
↪that? Let's investigate this a
# bit. First we need to make sure we got all the column names correct We can␣
↪use the columns attribute of
# dataframe to get a list.
new_df.columns
```

[5]:
```
Index(['GRE Score', 'TOEFL Score', 'University Rating', 'Statement of Purpose',
       'LOR ', 'CGPA', 'Research', 'Chance of Admit '],
      dtype='object')
```

[6]:
```
# If we look at the output closely, we can see that there is actually a space␣
↪right after "LOR" and a space
# right after "Chance of Admit. Sneaky, huh? So this is why our rename␣
↪dictionary does not work for LOR,
# because the key we used was just three characters, instead of "LOR "

# There are a couple of ways we could address this. One way would be to change␣
↪a column by including the space
# in the name
new_df=new_df.rename(columns={'LOR ': 'Letter of Recommendation'})
new_df.head()
```

[6]:
```
            GRE Score  TOEFL Score  University Rating  Statement of Purpose  \
Serial No.
1                 337          118                  4                   4.5
2                 324          107                  4                   4.0
3                 316          104                  3                   3.0
4                 322          110                  3                   3.5
5                 314          103                  2                   2.0

            Letter of Recommendation  CGPA  Research  Chance of Admit
Serial No.
1                                4.5  9.65         1              0.92
2                                4.5  8.87         1              0.76
3                                3.5  8.00         1              0.72
4                                2.5  8.67         1              0.80
5                                3.0  8.21         0              0.65
```

[7]:
```
# So that works well, but it's a bit fragile. What if that was a tab instead of␣
↪a space? Or two spaces?
# Another way is to create some function that does the cleaning and then tell␣
↪renamed to apply that function
# across all of the data. Python comes with a handy string function to strip␣
↪white space called "strip()".
# When we pass this in to rename we pass the function as the mapper parameter,␣
↪and then indicate whether the
# axis should be columns or index (row labels)
```

```
new_df=new_df.rename(mapper=str.strip, axis='columns')
# Let's take a look at results
new_df.head()
```

[7]:
```
            GRE Score  TOEFL Score  University Rating  Statement of Purpose  \
Serial No.
1                 337          118                  4                   4.5
2                 324          107                  4                   4.0
3                 316          104                  3                   3.0
4                 322          110                  3                   3.5
5                 314          103                  2                   2.0

            Letter of Recommendation  CGPA  Research  Chance of Admit
Serial No.
1                                4.5  9.65         1             0.92
2                                4.5  8.87         1             0.76
3                                3.5  8.00         1             0.72
4                                2.5  8.67         1             0.80
5                                3.0  8.21         0             0.65
```

[8]:
```
# Now we've got it - both SOP and LOR have been renamed and Chance of Admit has␣
↪been trimmed up. Remember
# though that the rename function isn't modifying the dataframe. In this case,␣
↪df is the same as it always
# was, there's just a copy in new_df with the changed names.
df.columns
```

[8]:
```
Index(['GRE Score', 'TOEFL Score', 'University Rating', 'SOP', 'LOR ', 'CGPA',
       'Research', 'Chance of Admit '],
      dtype='object')
```

[9]:
```
# We can also use the df.columns attribute by assigning to it a list of column␣
↪names which will directly
# rename the columns. This will directly modify the original dataframe and is␣
↪very efficient especially when
# you have a lot of columns and you only want to change a few. This technique␣
↪is also not affected by subtle
# errors in the column names, a problem that we just encountered. With a list,␣
↪you can use the list index to
# change a certain value or use list comprehension to change all of the values

# As an example, lets change all of the column names to lower case. First we␣
↪need to get our list
cols = list(df.columns)
# Then a little list comprehenshion
cols = [x.lower().strip() for x in cols]
# Then we just overwrite what is already in the .columns attribute
df.columns=cols
```

```
# And take a look at our results
df.head()
```

[9]:

```
             gre score  toefl score  university rating  sop  lor  cgpa  \
Serial No.
1                  337          118                  4  4.5  4.5  9.65
2                  324          107                  4  4.0  4.5  8.87
3                  316          104                  3  3.0  3.5  8.00
4                  322          110                  3  3.5  2.5  8.67
5                  314          103                  2  2.0  3.0  8.21

            research  chance of admit
Serial No.
1                  1             0.92
2                  1             0.76
3                  1             0.72
4                  1             0.80
5                  0             0.65
```

In this lecture, you've learned how to import a CSV file into a pandas DataFrame object, and how to do some basic data cleaning to the column names. The CSV file import mechanisms in pandas have lots of different options, and you really need to learn these in order to be proficient at data manipulation. Once you have set up the format and shape of a DataFrame, you have a solid start to further actions such as conducting data analysis and modeling.

Now, there are other data sources you can load directly into dataframes as well, including HTML web pages, databases, and other file formats. But the CSV is by far the most common data format you'll run into, and an important one to know how to manipulate in pandas.

QueryingDataFrame_ed

November 15, 2021

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.

[1]:
```
# Let's start with an example and import our graduate admission dataset. First␣
↪we'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()
```

[1]:
```
            gre score  toefl score  university rating  sop  lor  cgpa  \
Serial No.
1                 337          118                  4  4.5  4.5  9.65
2                 324          107                  4  4.0  4.5  8.87
3                 316          104                  3  3.0  3.5  8.00
4                 322          110                  3  3.5  2.5  8.67
5                 314          103                  2  2.0  3.0  8.21

            research  chance of admit
Serial No.
1                  1             0.92
2                  1             0.76
3                  1             0.72
4                  1             0.80
5                  0             0.65
```

[2]:
```
# 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
admit_mask=df['chance of admit'] > 0.7
admit_mask
```

[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
```

[3]:
```
# 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 (so efficiently and in
# parallel) 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.
```

[4]:
```
# So, what do you do with the boolean mask once you have formed it? Well, you␣
↪can just lay it on top of the
# data to "hide" the data you don't want, which is represented by all of the␣
↪False values. We do this by using
```

```
# the .where() function on the original DataFrame.
df.where(admit_mask).head()
```

[4]:
```
            gre score  toefl score  university rating  sop  lor  cgpa  \
Serial No.
1               337.0        118.0                4.0  4.5  4.5  9.65
2               324.0        107.0                4.0  4.0  4.5  8.87
3               316.0        104.0                3.0  3.0  3.5  8.00
4               322.0        110.0                3.0  3.5  2.5  8.67
5                 NaN          NaN                NaN  NaN  NaN   NaN

            research  chance of admit
Serial No.
1                1.0             0.92
2                1.0             0.76
3                1.0             0.72
4                1.0             0.80
5                NaN              NaN
```

[5]:
```
# 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
df.where(admit_mask).dropna().head()
```

[5]:
```
            gre score  toefl score  university rating  sop  lor  cgpa  \
Serial No.
1               337.0        118.0                4.0  4.5  4.5  9.65
2               324.0        107.0                4.0  4.0  4.5  8.87
3               316.0        104.0                3.0  3.0  3.5  8.00
4               322.0        110.0                3.0  3.5  2.5  8.67
6               330.0        115.0                5.0  4.5  3.0  9.34

            research  chance of admit
Serial No.
1                1.0             0.92
2                1.0             0.76
3                1.0             0.72
4                1.0             0.80
6                1.0             0.90
```

[6]:
```
# The returned DataFrame now has all of the NaN rows dropped. Notice the index␣
↪now includes
# one through four and six, but not five.
#
```

```
# Despite being really handy, where() isn't actually used that often. Instead,␣
↪the 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()
```

[6]:
```
            gre score  toefl score  university rating  sop  lor  cgpa  \
Serial No.
1                 337          118                  4  4.5  4.5  9.65
2                 324          107                  4  4.0  4.5  8.87
3                 316          104                  3  3.0  3.5  8.00
4                 322          110                  3  3.5  2.5  8.67
6                 330          115                  5  4.5  3.0  9.34

            research  chance of admit
Serial No.
1                  1             0.92
2                  1             0.76
3                  1             0.72
4                  1             0.80
6                  1             0.90
```

[7]:
```
# 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 two things:

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

[7]:
```
Serial No.
1    337
2    324
3    316
4    322
5    314
Name: gre score, dtype: int64
```

[8]:
```
# Or you can send it a list of columns as strings
df[["gre score","toefl score"]].head()
```

[8]:
```
            gre score  toefl score
Serial No.
1                 337          118
2                 324          107
3                 316          104
```

```
4                322          110
5                314          103
```

[9]:
```
# Or you can send it a boolean mask
df[df["gre score"]>320].head()
```

[9]:
```
            gre score  toefl score  university rating  sop  lor  cgpa  \
Serial No.
1                337          118                  4  4.5  4.5  9.65
2                324          107                  4  4.0  4.5  8.87
4                322          110                  3  3.5  2.5  8.67
6                330          115                  5  4.5  3.0  9.34
7                321          109                  3  3.0  4.0  8.20

            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
```

[10]:
```
# And each of these is mimicing functionality from either .loc() or .where().
 ↪dropna().
```

[11]:
```
# 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.

# Unfortunatly, it doesn't feel quite as natural in pandas. For instance, if␣
 ↪you want to take two boolean
# series and and them together
(df['chance of admit'] > 0.7) and (df['chance of admit'] < 0.9)
```

```
      ␣
↪---------------------------------------------------------------------------

       ValueError                                Traceback (most recent call␣
↪last)

       <ipython-input-11-3d7e76efc1e4> in <module>
         5 # Unfortunatly, it doesn't feel quite as natural in pandas. For␣
↪instance, if you want to take two boolean
         6 # series and and them together
   ----> 7 (df['chance of admit'] > 0.7) and (df['chance of admit'] < 0.9)
```

```
    /opt/conda/lib/python3.7/site-packages/pandas/core/generic.py in␣
↪__nonzero__(self)
   1554             "The truth value of a {0} is ambiguous. "
   1555             "Use a.empty, a.bool(), a.item(), a.any() or a.all().".
↪format(
-> 1556                 self.__class__.__name__
   1557             )
   1558         )


    ValueError: The truth value of a Series is ambiguous. Use a.empty, a.
↪bool(), a.item(), a.any() or a.all().
```

[ ]:
```python
# 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. Instead, the pandas authors have overwritten the pipe | and␣
↪ampersand & operators to handle this
# for us
(df['chance of admit'] > 0.7) & (df['chance of admit'] < 0.9)
```

[ ]:
```python
# 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
df['chance of admit'] > 0.7 & df['chance of admit'] < 0.9
```

[ ]:
```python
# 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
```

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

[ ]:
```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)
```

[ ]:
```
# 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.
```

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.

IndexingDataFrame_ed

November 15, 2021

As we've seen, both Series and DataFrames can have indices applied to them. The index is essentially a row level label, and in pandas the rows correspond to axis zero. Indices can either be either autogenerated, such as when we create a new Series without an index, in which case we get numeric values, or they can be set explicitly, like when we use the dictionary object to create the series, or when we loaded data from the CSV file and set appropriate parameters. Another option for setting an index is to use the set_index() function. This function takes a list of columns and promotes those columns to an index. In this lecture we'll explore more about how indexes work in pandas.

[1]:
```
# The set_index() function is a destructive process, and it doesn't keep the␣
↪current index.
# If you want to keep the current index, you need to manually create a new␣
↪column and copy into
# it values from the index attribute.

# Lets import pandas and our admissions dataset
import pandas as pd
df = pd.read_csv("datasets/Admission_Predict.csv", index_col=0)
df.head()
```

[1]:
```
            GRE Score  TOEFL Score  University Rating  SOP  LOR   CGPA  \
Serial No.
1                 337          118                  4  4.5   4.5  9.65
2                 324          107                  4  4.0   4.5  8.87
3                 316          104                  3  3.0   3.5  8.00
4                 322          110                  3  3.5   2.5  8.67
5                 314          103                  2  2.0   3.0  8.21

            Research  Chance of Admit
Serial No.
1                  1             0.92
2                  1             0.76
3                  1             0.72
4                  1             0.80
5                  0             0.65
```

[2]:
```
# Let's say that we don't want to index the DataFrame by serial numbers, but␣
↪instead by the
```

```python
# chance of admit. But lets assume we want to keep the serial number for later.␣
 ↪So, lets
# preserve the serial number into a new column. We can do this using the␣
 ↪indexing operator
# on the string that has the column label. Then we can use the set_index to set␣
 ↪index
# of the column to chance of admit

# So we copy the indexed data into its own column
df['Serial Number'] = df.index
# Then we set the index to another column
df = df.set_index('Chance of Admit ')
df.head()
```

[2]:

```
                 GRE Score  TOEFL Score  University Rating  SOP  LOR   CGPA  \
Chance of Admit
0.92                   337          118                  4  4.5   4.5  9.65
0.76                   324          107                  4  4.0   4.5  8.87
0.72                   316          104                  3  3.0   3.5  8.00
0.80                   322          110                  3  3.5   2.5  8.67
0.65                   314          103                  2  2.0   3.0  8.21

                 Research  Serial Number
Chance of Admit
0.92                    1              1
0.76                    1              2
0.72                    1              3
0.80                    1              4
0.65                    0              5
```

[3]:

```python
# You'll see that when we create a new index from an existing column the index␣
 ↪has a name,
# which is the original name of the column.

# We can get rid of the index completely by calling the function reset_index().␣
 ↪This promotes the
# index into a column and creates a default numbered index.
df = df.reset_index()
df.head()
```

[3]:

```
   Chance of Admit   GRE Score  TOEFL Score  University Rating  SOP  LOR   \
0              0.92        337          118                  4  4.5   4.5
1              0.76        324          107                  4  4.0   4.5
2              0.72        316          104                  3  3.0   3.5
3              0.80        322          110                  3  3.5   2.5
4              0.65        314          103                  2  2.0   3.0

   CGPA  Research  Serial Number
```

```
0  9.65         1           1
1  8.87         1           2
2  8.00         1           3
3  8.67         1           4
4  8.21         0           5
```

[4]:
```
# One nice feature of Pandas is multi-level indexing. This is similar to␣
↪composite keys in
# relational database systems. To create a multi-level index, we simply call␣
↪set index and
# give it a list of columns that we're interested in promoting to an index.

# Pandas will search through these in order, finding the distinct data and form␣
↪composite indices.
# A good example of this is often found when dealing with geographical data␣
↪which is sorted by
# regions or demographics.

# Let's change data sets and look at some census data for a better example.␣
↪This data is stored in
# the file census.csv and comes from the United States Census Bureau. In␣
↪particular, this is a
# breakdown of the population level data at the US county level. It's a great␣
↪example of how
# different kinds of data sets might be formatted when you're trying to clean␣
↪them.

# Let's import and see what the data looks like
df = pd.read_csv('datasets/census.csv')
df.head()
```

[4]:
```
   SUMLEV  REGION  DIVISION  STATE  COUNTY   STNAME           CTYNAME  \
0      40       3         6      1       0  Alabama           Alabama
1      50       3         6      1       1  Alabama    Autauga County
2      50       3         6      1       3  Alabama    Baldwin County
3      50       3         6      1       5  Alabama    Barbour County
4      50       3         6      1       7  Alabama       Bibb County

   CENSUS2010POP  ESTIMATESBASE2010  POPESTIMATE2010  ...  RDOMESTICMIG2011  \
0        4779736            4780127          4785161  ...          0.002295
1          54571              54571            54660  ...          7.242091
2         182265             182265           183193  ...         14.832960
3          27457              27457            27341  ...         -4.728132
4          22915              22919            22861  ...         -5.527043

   RDOMESTICMIG2012  RDOMESTICMIG2013  RDOMESTICMIG2014  RDOMESTICMIG2015  \
0         -0.193196          0.381066          0.582002         -0.467369
```

```
1         -2.915927          -3.012349           2.265971          -2.530799
2         17.647293          21.845705          19.243287          17.197872
3         -2.500690          -7.056824          -3.904217         -10.543299
4         -5.068871          -6.201001          -0.177537           0.177258

   RNETMIG2011  RNETMIG2012  RNETMIG2013  RNETMIG2014  RNETMIG2015
0     1.030015     0.826644     1.383282     1.724718     0.712594
1     7.606016    -2.626146    -2.722002     2.592270    -2.187333
2    15.844176    18.559627    22.727626    20.317142    18.293499
3    -4.874741    -2.758113    -7.167664    -3.978583   -10.543299
4    -5.088389    -4.363636    -5.403729     0.754533     1.107861

[5 rows x 100 columns]
```

[5]:
```
# In this data set there are two summarized levels, one that contains summary
# data for the whole country. And one that contains summary data for each state.
# I want to see a list of all the unique values in a given column. In this
# DataFrame, we see that the possible values for the sum level are using the
# unique function on the DataFrame. This is similar to the SQL distinct␣
↪operator

# Here we can run unique on the sum level of our current DataFrame
df['SUMLEV'].unique()
```

[5]:
```
array([40, 50])
```

[6]:
```
# We see that there are only two different values, 40 and 50
```

[7]:
```
# Let's exclue all of the rows that are summaries
# at the state level and just keep the county data.
df=df[df['SUMLEV'] == 50]
df.head()
```

[7]:
```
   SUMLEV  REGION  DIVISION  STATE  COUNTY   STNAME           CTYNAME  \
1      50       3         6      1       1  Alabama    Autauga County
2      50       3         6      1       3  Alabama    Baldwin County
3      50       3         6      1       5  Alabama    Barbour County
4      50       3         6      1       7  Alabama       Bibb County
5      50       3         6      1       9  Alabama     Blount County

   CENSUS2010POP  ESTIMATESBASE2010  POPESTIMATE2010  ...  RDOMESTICMIG2011  \
1          54571              54571            54660  ...          7.242091
2         182265             182265           183193  ...         14.832960
3          27457              27457            27341  ...         -4.728132
4          22915              22919            22861  ...         -5.527043
5          57322              57322            57373  ...          1.807375

   RDOMESTICMIG2012  RDOMESTICMIG2013  RDOMESTICMIG2014  RDOMESTICMIG2015  \
1         -2.915927         -3.012349          2.265971         -2.530799
```

```
2         17.647293          21.845705          19.243287          17.197872
3         -2.500690          -7.056824          -3.904217         -10.543299
4         -5.068871          -6.201001          -0.177537           0.177258
5         -1.177622          -1.748766          -2.062535          -1.369970

   RNETMIG2011  RNETMIG2012  RNETMIG2013  RNETMIG2014  RNETMIG2015
1     7.606016    -2.626146    -2.722002     2.592270    -2.187333
2    15.844176    18.559627    22.727626    20.317142    18.293499
3    -4.874741    -2.758113    -7.167664    -3.978583   -10.543299
4    -5.088389    -4.363636    -5.403729     0.754533     1.107861
5     1.859511    -0.848580    -1.402476    -1.577232    -0.884411

[5 rows x 100 columns]
```

[8]:
```python
# Also while this data set is interesting for a number of different reasons,
# let's reduce the data that we're going to look at to just the total␣
 ↪population
# estimates and the total number of births. We can do this by creating
# a list of column names that we want to keep then project those and
# assign the resulting DataFrame to our df variable.

columns_to_keep =␣
 ↪['STNAME','CTYNAME','BIRTHS2010','BIRTHS2011','BIRTHS2012','BIRTHS2013',
                    ␣
 ↪'BIRTHS2014','BIRTHS2015','POPESTIMATE2010','POPESTIMATE2011',
                    ␣
 ↪'POPESTIMATE2012','POPESTIMATE2013','POPESTIMATE2014','POPESTIMATE2015']
df = df[columns_to_keep]
df.head()
```

[8]:
```
    STNAME          CTYNAME  BIRTHS2010  BIRTHS2011  BIRTHS2012  BIRTHS2013  \
1  Alabama  Autauga County         151         636         615         574
2  Alabama  Baldwin County         517        2187        2092        2160
3  Alabama  Barbour County          70         335         300         283
4  Alabama     Bibb County          44         266         245         259
5  Alabama   Blount County         183         744         710         646

   BIRTHS2014  BIRTHS2015  POPESTIMATE2010  POPESTIMATE2011  POPESTIMATE2012  \
1         623         600            54660            55253            55175
2        2186        2240           183193           186659           190396
3         260         269            27341            27226            27159
4         247         253            22861            22733            22642
5         618         603            57373            57711            57776

   POPESTIMATE2013  POPESTIMATE2014  POPESTIMATE2015
1            55038            55290            55347
2           195126           199713           203709
3            26973            26815            26489
```

```
4              22512             22549             22583
5              57734             57658             57673
```

[9]:
```
# The US Census data breaks down population estimates by state and county. We␣
↪can load the data and
# set the index to be a combination of the state and county values and see how␣
↪pandas handles it in
# a DataFrame. We do this by creating a list of the column identifiers we want␣
↪to have indexed. And then
# calling set index with this list and assigning the output as appropriate. We␣
↪see here that we have
# a dual index, first the state name and second the county name.

df = df.set_index(['STNAME', 'CTYNAME'])
df.head()
```

[9]:
```
                        BIRTHS2010  BIRTHS2011  BIRTHS2012  BIRTHS2013  \
STNAME  CTYNAME
Alabama Autauga County         151         636         615         574
        Baldwin County         517        2187        2092        2160
        Barbour County          70         335         300         283
        Bibb County             44         266         245         259
        Blount County          183         744         710         646

                        BIRTHS2014  BIRTHS2015  POPESTIMATE2010  \
STNAME  CTYNAME
Alabama Autauga County         623         600            54660
        Baldwin County        2186        2240           183193
        Barbour County         260         269            27341
        Bibb County            247         253            22861
        Blount County          618         603            57373

                        POPESTIMATE2011  POPESTIMATE2012  POPESTIMATE2013  \
STNAME  CTYNAME
Alabama Autauga County            55253            55175            55038
        Baldwin County           186659           190396           195126
        Barbour County            27226            27159            26973
        Bibb County               22733            22642            22512
        Blount County             57711            57776            57734

                        POPESTIMATE2014  POPESTIMATE2015
STNAME  CTYNAME
Alabama Autauga County            55290            55347
        Baldwin County           199713           203709
        Barbour County            26815            26489
        Bibb County               22549            22583
        Blount County             57658            57673
```

[10]:
```
# An immediate question which comes up is how we can query this DataFrame. We␣
↪saw previously that
# the loc attribute of the DataFrame can take multiple arguments. And it could␣
↪query both the
# row and the columns. When you use a MultiIndex, you must provide the␣
↪arguments in order by the
# level you wish to query. Inside of the index, each column is called a level␣
↪and the outermost
# column is level zero.

# If we want to see the population results from Washtenaw County in Michigan␣
↪the state, which is
# where I live, the first argument would be Michigan and the second would be␣
↪Washtenaw County
df.loc['Michigan', 'Washtenaw County']
```

[10]:
```
BIRTHS2010            977
BIRTHS2011           3826
BIRTHS2012           3780
BIRTHS2013           3662
BIRTHS2014           3683
BIRTHS2015           3709
POPESTIMATE2010    345563
POPESTIMATE2011    349048
POPESTIMATE2012    351213
POPESTIMATE2013    354289
POPESTIMATE2014    357029
POPESTIMATE2015    358880
Name: (Michigan, Washtenaw County), dtype: int64
```

[11]:
```
# If you are interested in comparing two counties, for example, Washtenaw and␣
↪Wayne County, we can
# pass a list of tuples describing the indices we wish to query into loc. Since␣
↪we have a MultiIndex
# of two values, the state and the county, we need to provide two values as␣
↪each element of our
# filtering list. Each tuple should have two elements, the first element being␣
↪the first index and
# the second element being the second index.

# Therefore, in this case, we will have a list of two tuples, in each tuple,␣
↪the first element is
# Michigan, and the second element is either Washtenaw County or Wayne County

df.loc[ [('Michigan', 'Washtenaw County'),
         ('Michigan', 'Wayne County')] ]
```

[11]:

```
                           BIRTHS2010  BIRTHS2011  BIRTHS2012  BIRTHS2013  \
STNAME   CTYNAME
Michigan Washtenaw County         977        3826        3780        3662
         Wayne County            5918       23819       23270       23377

                           BIRTHS2014  BIRTHS2015  POPESTIMATE2010  \
STNAME   CTYNAME
Michigan Washtenaw County        3683        3709           345563
         Wayne County           23607       23586          1815199

                           POPESTIMATE2011  POPESTIMATE2012  POPESTIMATE2013  \
STNAME   CTYNAME
Michigan Washtenaw County           349048           351213           354289
         Wayne County              1801273          1792514          1775713

                           POPESTIMATE2014  POPESTIMATE2015
STNAME   CTYNAME
Michigan Washtenaw County           357029           358880
         Wayne County              1766008          1759335
```

Okay so that's how hierarchical indices work in a nutshell. They're a special part of the pandas library which I think can make management and reasoning about data easier. Of course hierarchical labeling isn't just for rows. For example, you can transpose this matrix and now have hierarchical column labels. And projecting a single column which has these labels works exactly the way you would expect it to. Now, in reality, I don't tend to use hierarchical indicies very much, and instead just keep everything as columns and manipulate those. But, it's a unique and sophisticated aspect of pandas that is useful to know, especially if viewing your data in a tabular form.

MissingValues_ed

November 15, 2021

We've seen a preview of how Pandas handles missing values using the None type and NumPy NaN values. Missing values are pretty common in data cleaning activities. And, missing values can be there for any number of reasons, and I just want to touch on a few here.

For instance, if you are running a survey and a respondant didn't answer a question the missing value is actually an omission. This kind of missing data is called **Missing at Random** if there are other variables that might be used to predict the variable which is missing. In my work when I delivery surveys I often find that missing data, say the interest in being involved in a follow up study, often has some correlation with another data field, like gender or ethnicity. If there is no relationship to other variables, then we call this data **Missing Completely at Random (MCAR)**.

These are just two examples of missing data, and there are many more. For instance, data might be missing because it wasn't collected, either by the process responsible for collecting that data, such as a researcher, or because it wouldn't make sense if it were collected. This last example is extremely common when you start joining DataFrames together from multiple sources, such as joining a list of people at a university with a list of offices in the university (students generally don't have offices).

Let's look at some ways of handling missing data in pandas.

[1]:
```python
# Lets import pandas
import pandas as pd
```

[2]:
```python
# Pandas is pretty good at detecting missing values directly from underlying␣
↪data formats, like CSV files.
# Although most missing valuse are often formatted as NaN, NULL, None, or N/A,␣
↪sometimes missing values are
# not labeled so clearly. For example, I've worked with social scientists who␣
↪regularly used the value of 99
# in binary categories to indicate a missing value. The pandas read_csv()␣
↪function has a parameter called
# na_values to let us specify the form of missing values. It allows scalar,␣
↪string, list, or dictionaries to
# be used.

# Let's load a piece of data from a file called log.csv
df = pd.read_csv('datasets/class_grades.csv')
df.head(10)
```

[2]:

	Prefix	Assignment	Tutorial	Midterm	TakeHome	Final
0	5	57.14	34.09	64.38	51.48	52.50

```
1       8     95.05    105.49   67.50     99.07  68.33
2       8     83.70     83.17     NaN     63.15  48.89
3       7       NaN       NaN   49.38    105.93  80.56
4       8     91.32     93.64   95.00    107.41  73.89
5       7     95.00     92.58   93.12     97.78  68.06
6       8     95.05    102.99   56.25     99.07  50.00
7       7     72.85     86.85   60.00       NaN  56.11
8       8     84.26     93.10   47.50     18.52  50.83
9       7     90.10     97.55   51.25     88.89  63.61
```

[3]:
```
# We can actually use the function .isnull() to create a boolean mask of the␣
↪whole dataframe. This effectively
# broadcasts the isnull() function to every cell of data.
mask=df.isnull()
mask.head(10)
```

[3]:
```
   Prefix  Assignment  Tutorial  Midterm  TakeHome  Final
0   False       False     False    False     False  False
1   False       False     False    False     False  False
2   False       False     False     True     False  False
3   False        True      True    False     False  False
4   False       False     False    False     False  False
5   False       False     False    False     False  False
6   False       False     False    False     False  False
7   False       False     False    False      True  False
8   False       False     False    False     False  False
9   False       False     False    False     False  False
```

[4]:
```
# This can be useful for processing rows based on certain columns of data.␣
↪Another useful operation is to be
# able to drop all of those rows which have any missing data, which can be done␣
↪with the dropna() function.
df.dropna().head(10)
```

[4]:
```
    Prefix  Assignment  Tutorial  Midterm  TakeHome  Final
0        5       57.14     34.09    64.38     51.48  52.50
1        8       95.05    105.49    67.50     99.07  68.33
4        8       91.32     93.64    95.00    107.41  73.89
5        7       95.00     92.58    93.12     97.78  68.06
6        8       95.05    102.99    56.25     99.07  50.00
8        8       84.26     93.10    47.50     18.52  50.83
9        7       90.10     97.55    51.25     88.89  63.61
10       7       80.44     90.20    75.00     91.48  39.72
12       8       97.16    103.71    72.50     93.52  63.33
13       7       91.28     83.53    81.25     99.81  92.22
```

[5]:
```
# Note how the rows indexed with 2, 3, 7, and 11 are now gone. One of the handy␣
↪functions that Pandas has for
```

```
# working with missing values is the filling function, fillna(). This function␣
↪takes a number or parameters.
# You could pass in a single value which is called a scalar value to change all␣
↪of the missing data to one
# value. This isn't really applicable in this case, but it's a pretty common␣
↪use case.

# So, if we wanted to fill all missing values with 0, we would use fillna
df.fillna(0, inplace=True)
df.head(10)
```

[5]:

	Prefix	Assignment	Tutorial	Midterm	TakeHome	Final
0	5	57.14	34.09	64.38	51.48	52.50
1	8	95.05	105.49	67.50	99.07	68.33
2	8	83.70	83.17	0.00	63.15	48.89
3	7	0.00	0.00	49.38	105.93	80.56
4	8	91.32	93.64	95.00	107.41	73.89
5	7	95.00	92.58	93.12	97.78	68.06
6	8	95.05	102.99	56.25	99.07	50.00
7	7	72.85	86.85	60.00	0.00	56.11
8	8	84.26	93.10	47.50	18.52	50.83
9	7	90.10	97.55	51.25	88.89	63.61

[6]:
```
# Note that the inplace attribute causes pandas to fill the values inline and␣
↪does not return a copy of the
# dataframe, but instead modifies the dataframe you have.
```

[7]:
```
# We can also use the na_filter option to turn off white space filtering, if␣
↪white space is an actual value of
# interest. But in practice, this is pretty rare. In data without any NAs,␣
↪passing na_filter=False, can
# improve the performance of reading a large file.

# In addition to rules controlling how missing values might be loaded, it's␣
↪sometimes useful to consider
# missing values as actually having information. I'll give an example from my␣
↪own research.  I often deal with
# logs from online learning systems. I've looked at video use in lecture␣
↪capture systems. In these systems
# it's common for the player for have a heartbeat functionality where playback␣
↪statistics are sent to the
# server every so often, maybe every 30 seconds. These heartbeats can get big␣
↪as they can carry the whole
# state of the playback system such as where the video play head is at, where␣
↪the video size is, which video
# is being rendered to the screen, how loud the volume is.
```

```python
# If we load the data file log.csv, we can see an example of what this might␣
↪look like.
df = pd.read_csv("datasets/log.csv")
df.head(20)
```

[7]:

	time	user	video	playback position	paused	volume
0	1469974424	cheryl	intro.html	5	False	10.0
1	1469974454	cheryl	intro.html	6	NaN	NaN
2	1469974544	cheryl	intro.html	9	NaN	NaN
3	1469974574	cheryl	intro.html	10	NaN	NaN
4	1469977514	bob	intro.html	1	NaN	NaN
5	1469977544	bob	intro.html	1	NaN	NaN
6	1469977574	bob	intro.html	1	NaN	NaN
7	1469977604	bob	intro.html	1	NaN	NaN
8	1469974604	cheryl	intro.html	11	NaN	NaN
9	1469974694	cheryl	intro.html	14	NaN	NaN
10	1469974724	cheryl	intro.html	15	NaN	NaN
11	1469974454	sue	advanced.html	24	NaN	NaN
12	1469974524	sue	advanced.html	25	NaN	NaN
13	1469974424	sue	advanced.html	23	False	10.0
14	1469974554	sue	advanced.html	26	NaN	NaN
15	1469974624	sue	advanced.html	27	NaN	NaN
16	1469974654	sue	advanced.html	28	NaN	5.0
17	1469974724	sue	advanced.html	29	NaN	NaN
18	1469974484	cheryl	intro.html	7	NaN	NaN
19	1469974514	cheryl	intro.html	8	NaN	NaN

[8]:

```python
# In this data the first column is a timestamp in the Unix epoch format. The␣
↪next column is the user name
# followed by a web page they're visiting and the video that they're playing.␣
↪Each row of the DataFrame has a
# playback position. And we can see that as the playback position increases by␣
↪one, the time stamp increases
# by about 30 seconds.

# Except for user Bob. It turns out that Bob has paused his playback so as time␣
↪increases the playback
# position doesn't change. Note too how difficult it is for us to try and␣
↪derive this knowledge from the data,
# because it's not sorted by time stamp as one might expect. This is actually␣
↪not uncommon on systems which
# have a high degree of parallelism. There are a lot of missing values in the␣
↪paused and volume columns. It's
# not efficient to send this information across the network if it hasn't␣
↪changed. So this articular system
# just inserts null values into the database if there's no changes.
```

[9]:
```
# Next up is the method parameter(). The two common fill values are ffill and␣
↪bfill. ffill is for forward
# filling and it updates an na value for a particular cell with the value from␣
↪the previous row. bfill is
# backward filling, which is the opposite of ffill. It fills the missing values␣
↪with the next valid value.
# It's important to note that your data needs to be sorted in order for this to␣
↪have the effect you might
# want. Data which comes from traditional database management systems usually␣
↪has no order guarantee, just
# like this data. So be careful.

# In Pandas we can sort either by index or by values. Here we'll just promote␣
↪the time stamp to an index then
# sort on the index.
df = df.set_index('time')
df = df.sort_index()
df.head(20)
```

[9]:
```
              user          video  playback position paused  volume
time
1469974424  cheryl     intro.html                  5  False    10.0
1469974424     sue  advanced.html                 23  False    10.0
1469974454  cheryl     intro.html                  6    NaN     NaN
1469974454     sue  advanced.html                 24    NaN     NaN
1469974484  cheryl     intro.html                  7    NaN     NaN
1469974514  cheryl     intro.html                  8    NaN     NaN
1469974524     sue  advanced.html                 25    NaN     NaN
1469974544  cheryl     intro.html                  9    NaN     NaN
1469974554     sue  advanced.html                 26    NaN     NaN
1469974574  cheryl     intro.html                 10    NaN     NaN
1469974604  cheryl     intro.html                 11    NaN     NaN
1469974624     sue  advanced.html                 27    NaN     NaN
1469974634  cheryl     intro.html                 12    NaN     NaN
1469974654     sue  advanced.html                 28    NaN     5.0
1469974664  cheryl     intro.html                 13    NaN     NaN
1469974694  cheryl     intro.html                 14    NaN     NaN
1469974724  cheryl     intro.html                 15    NaN     NaN
1469974724     sue  advanced.html                 29    NaN     NaN
1469974754     sue  advanced.html                 30    NaN     NaN
1469974824     sue  advanced.html                 31    NaN     NaN
```

[10]:
```
# If we look closely at the output though we'll notice that the index
# isn't really unique. Two users seem to be able to use the system at the same
# time. Again, a very common case. Let's reset the index, and use some
# multi-level indexing on time AND user together instead,
# promote the user name to a second level of the index to deal with that issue.
```

```python
df = df.reset_index()
df = df.set_index(['time', 'user'])
df
```

[10]:

```
                                 video  playback position paused  volume
time       user
1469974424 cheryl      intro.html                  5  False    10.0
           sue      advanced.html                 23  False    10.0
1469974454 cheryl      intro.html                  6    NaN     NaN
           sue      advanced.html                 24    NaN     NaN
1469974484 cheryl      intro.html                  7    NaN     NaN
1469974514 cheryl      intro.html                  8    NaN     NaN
1469974524 sue      advanced.html                 25    NaN     NaN
1469974544 cheryl      intro.html                  9    NaN     NaN
1469974554 sue      advanced.html                 26    NaN     NaN
1469974574 cheryl      intro.html                 10    NaN     NaN
1469974604 cheryl      intro.html                 11    NaN     NaN
1469974624 sue      advanced.html                 27    NaN     NaN
1469974634 cheryl      intro.html                 12    NaN     NaN
1469974654 sue      advanced.html                 28    NaN     5.0
1469974664 cheryl      intro.html                 13    NaN     NaN
1469974694 cheryl      intro.html                 14    NaN     NaN
1469974724 cheryl      intro.html                 15    NaN     NaN
           sue      advanced.html                 29    NaN     NaN
1469974754 sue      advanced.html                 30    NaN     NaN
1469974824 sue      advanced.html                 31    NaN     NaN
1469974854 sue      advanced.html                 32    NaN     NaN
1469974924 sue      advanced.html                 33    NaN     NaN
1469977424 bob         intro.html                  1   True    10.0
1469977454 bob         intro.html                  1    NaN     NaN
1469977484 bob         intro.html                  1    NaN     NaN
1469977514 bob         intro.html                  1    NaN     NaN
1469977544 bob         intro.html                  1    NaN     NaN
1469977574 bob         intro.html                  1    NaN     NaN
1469977604 bob         intro.html                  1    NaN     NaN
1469977634 bob         intro.html                  1    NaN     NaN
1469977664 bob         intro.html                  1    NaN     NaN
1469977694 bob         intro.html                  1    NaN     NaN
1469977724 bob         intro.html                  1    NaN     NaN
```

[11]:

```python
# Now that we have the data indexed and sorted appropriately, we can fill the
 ↪missing datas using ffill. It's
# good to remember when dealing with missing values so you can deal with
 ↪individual columns or sets of columns
# by projecting them. So you don't have to fix all missing values in one
 ↪command.

df = df.fillna(method='ffill')
```

```
df.head()
```

[11]:
```
                                video  playback position  paused  volume
time       user
1469974424 cheryl      intro.html                  5   False    10.0
           sue      advanced.html                 23   False    10.0
1469974454 cheryl      intro.html                  6   False    10.0
           sue      advanced.html                 24   False    10.0
1469974484 cheryl      intro.html                  7   False    10.0
```

[12]:
```
# We can also do customized fill-in to replace values with the replace()␣
↪function. It allows replacement from
# several approaches: value-to-value, list, dictionary, regex Let's generate a␣
↪simple example
df = pd.DataFrame({'A': [1, 1, 2, 3, 4],
                   'B': [3, 6, 3, 8, 9],
                   'C': ['a', 'b', 'c', 'd', 'e']})
df
```

[12]:
```
   A  B  C
0  1  3  a
1  1  6  b
2  2  3  c
3  3  8  d
4  4  9  e
```

[13]:
```
# We can replace 1's with 100, let's try the value-to-value approach
df.replace(1, 100)
```

[13]:
```
     A  B  C
0  100  3  a
1  100  6  b
2    2  3  c
3    3  8  d
4    4  9  e
```

[14]:
```
# How about changing two values? Let's try the list approach For example, we␣
↪want to change 1's to 100 and 3's
# to 300
df.replace([1, 3], [100, 300])
```

[14]:
```
     A    B  C
0  100  300  a
1  100    6  b
2    2  300  c
3  300    8  d
4    4    9  e
```

[15]:
```
# What's really cool about pandas replacement is that it supports regex too!
# Let's look at our data from the dataset logs again
df = pd.read_csv("datasets/log.csv")
```

```
df.head(20)
```

[15]:

```
          time    user          video  playback position paused  volume
0   1469974424  cheryl     intro.html                  5  False    10.0
1   1469974454  cheryl     intro.html                  6    NaN     NaN
2   1469974544  cheryl     intro.html                  9    NaN     NaN
3   1469974574  cheryl     intro.html                 10    NaN     NaN
4   1469977514     bob     intro.html                  1    NaN     NaN
5   1469977544     bob     intro.html                  1    NaN     NaN
6   1469977574     bob     intro.html                  1    NaN     NaN
7   1469977604     bob     intro.html                  1    NaN     NaN
8   1469974604  cheryl     intro.html                 11    NaN     NaN
9   1469974694  cheryl     intro.html                 14    NaN     NaN
10  1469974724  cheryl     intro.html                 15    NaN     NaN
11  1469974454     sue  advanced.html                 24    NaN     NaN
12  1469974524     sue  advanced.html                 25    NaN     NaN
13  1469974424     sue  advanced.html                 23  False    10.0
14  1469974554     sue  advanced.html                 26    NaN     NaN
15  1469974624     sue  advanced.html                 27    NaN     NaN
16  1469974654     sue  advanced.html                 28    NaN     5.0
17  1469974724     sue  advanced.html                 29    NaN     NaN
18  1469974484  cheryl     intro.html                  7    NaN     NaN
19  1469974514  cheryl     intro.html                  8    NaN     NaN
```

[16]:

```
# To replace using a regex we make the first parameter to replace the regex␣
↪pattern we want to match, the
# second parameter the value we want to emit upon match, and then we pass in a␣
↪third parameter "regex=True".

# Take a moment to pause this video and think about this problem: imagine we␣
↪want to detect all html pages in
# the "video" column, lets say that just means they end with ".html", and we␣
↪want to overwrite that with the
# keyword "webpage". How could we accomplish this?
```

[17]:

```
# Here's my solution, first matching any number of characters then ending in .
↪html
df.replace(to_replace=".*.html$", value="webpage", regex=True)
```

[17]:

```
         time    user    video  playback position paused  volume
0  1469974424  cheryl  webpage                  5  False    10.0
1  1469974454  cheryl  webpage                  6    NaN     NaN
2  1469974544  cheryl  webpage                  9    NaN     NaN
3  1469974574  cheryl  webpage                 10    NaN     NaN
4  1469977514     bob  webpage                  1    NaN     NaN
5  1469977544     bob  webpage                  1    NaN     NaN
6  1469977574     bob  webpage                  1    NaN     NaN
7  1469977604     bob  webpage                  1    NaN     NaN
8  1469974604  cheryl  webpage                 11    NaN     NaN
```

```
9   1469974694  cheryl  webpage                            14    NaN    NaN
10  1469974724  cheryl  webpage                            15    NaN    NaN
11  1469974454     sue  webpage                            24    NaN    NaN
12  1469974524     sue  webpage                            25    NaN    NaN
13  1469974424     sue  webpage                            23  False   10.0
14  1469974554     sue  webpage                            26    NaN    NaN
15  1469974624     sue  webpage                            27    NaN    NaN
16  1469974654     sue  webpage                            28    NaN    5.0
17  1469974724     sue  webpage                            29    NaN    NaN
18  1469974484  cheryl  webpage                             7    NaN    NaN
19  1469974514  cheryl  webpage                             8    NaN    NaN
20  1469974754     sue  webpage                            30    NaN    NaN
21  1469974824     sue  webpage                            31    NaN    NaN
22  1469974854     sue  webpage                            32    NaN    NaN
23  1469974924     sue  webpage                            33    NaN    NaN
24  1469977424     bob  webpage                             1   True   10.0
25  1469977454     bob  webpage                             1    NaN    NaN
26  1469977484     bob  webpage                             1    NaN    NaN
27  1469977634     bob  webpage                             1    NaN    NaN
28  1469977664     bob  webpage                             1    NaN    NaN
29  1469974634  cheryl  webpage                            12    NaN    NaN
30  1469974664  cheryl  webpage                            13    NaN    NaN
31  1469977694     bob  webpage                             1    NaN    NaN
32  1469977724     bob  webpage                             1    NaN    NaN
```

One last note on missing values. When you use statistical functions on DataFrames, these functions typically ignore missing values. For instance if you try and calculate the mean value of a DataFrame, the underlying NumPy function will ignore missing values. This is usually what you want but you should be aware that values are being excluded. Why you have missing values really matters depending upon the problem you are trying to solve. It might be unreasonable to infer missing values, for instance, if the data shouldn't exist in the first place.

DataFrameManipulation_ed

November 15, 2021

Now that you know the basics of what makes up a pandas dataframe, lets look at how we might actually clean some messy data. Now, there are many different approaches you can take to clean data, so this lecture is just one example of how you might tackle a problem.

[4]:
```
import pandas as pd
dfs=pd.read_html("https://en.wikipedia.org/wiki/
↪College_admissions_in_the_United_States")
len(dfs)
```

```
     ␣
↪---------------------------------------------------------------------------

     ImportError                               Traceback (most recent call␣
↪last)

     <ipython-input-4-97ffe7a168ae> in <module>
       1 import pandas as pd
 ----> 2 dfs=pd.read_html("https://en.wikipedia.org/wiki/
↪College_admissions_in_the_United_States")
       3 len(dfs)


     /opt/conda/lib/python3.7/site-packages/pandas/io/html.py in␣
↪read_html(io, match, flavor, header, index_col, skiprows, attrs, parse_dates,␣
↪thousands, encoding, decimal, converters, na_values, keep_default_na,␣
↪displayed_only)
    1103         na_values=na_values,
    1104         keep_default_na=keep_default_na,
 -> 1105         displayed_only=displayed_only,
    1106     )


     /opt/conda/lib/python3.7/site-packages/pandas/io/html.py in␣
↪_parse(flavor, io, match, attrs, encoding, displayed_only, **kwargs)
     886     retained = None
     887     for flav in flavor:
```

```
--> 888         parser = _parser_dispatch(flav)
    889         p = parser(io, compiled_match, attrs, encoding,␣
↪displayed_only)
    890

/opt/conda/lib/python3.7/site-packages/pandas/io/html.py in␣
↪_parser_dispatch(flavor)
    833     if flavor in ("bs4", "html5lib"):
    834         if not _HAS_HTML5LIB:
--> 835             raise ImportError("html5lib not found, please install␣
↪it")
    836         if not _HAS_BS4:
    837             raise ImportError("BeautifulSoup4 (bs4) not found,␣
↪please install it")


ImportError: html5lib not found, please install it
```

[15]:
```
dfs[10]
```

[15]:

	University	St	Ap-plied#	Over-allrate	Earlyrate	Regu-larrate
0	Amherst	MA	5511	12%	NaN	10%
1	Babson	MA	5511	29%	53%	21%
2	Barnard	NY	5440	21%	45%	18%
3	SUNY-Bing.	NY	28174	42%	56%	36%
4	Boston Coll	MA	34000	29%	40%	26%
5	Boston U	MA	43979	45%	47%	44%
6	Bowdoin	ME	6716	16%	25%	14%
7	Brown	RI	28742	10%	19%	NaN
8	Caltech[237]	CA	5225	17%	19%	NaN
9	Carleton	MN	5850	26%	41%	26%
10	Carnegie-Mel	PA	17300	27%	26%	27%
11	Claremont Mc	CA	5056	12%	29%	10%
12	Colby	ME	5241	29%	50%	29%
13	Colgate	NY	7795	29%	51%	27%
14	Columbia	NY	31851	7%	20%	NaN
15	Cooper Union	NY	3556	6%	9%	6%
16	Cornell	NY	37812	16%	33%	NaN
17	Dartmouth	NH	23110	9%	26%	8%
18	Dickinson	PA	5843	40%	53%	28%
19	Duke	NC	31600	12%	25%	11%
20	Elon	NC	10195	51%	86%	29%
21	Emory	GA	17502	26%	38%	25%
22	G Washington	DC	21759	33%	38%	NaN
23	Grinnell	IA	4554	30%	58%	28%
24	Hamilton	NY	5107	27%	44%	NaN

```
25           Hanover  IN       3546          62%     NaN         62%
26      Harvard[238]  MA      34302           6%     18%          4%
27       Harvey Mudd  CA       3591          17%     20%         17%
28     Johns Hopkins  MD      20496          18%     38%         16%
29         Juilliard  NY       2319           7%     NaN          7%
..               ...  ..        ...          ...     ...         ...
51         Texas A&M  TX      31478          59%     NaN         59%
52           Trinity  CT       7716          33%     NaN         NaN
53       UC Berkeley  CA      61702          21%     NaN         NaN
54          UC Davis  CA      49416          46%     NaN         NaN
55         UC Irvine  CA      54532          36%     NaN         NaN
56              UCLA  CA      72657          21%     NaN         NaN
57         UC Merced  CA      13148          75%     NaN         NaN
58      UC Riverside  CA      29888          61%     NaN         NaN
59      UC San Diego  CA      60838          38%     NaN         NaN
60      UC Santa Bar  CA      54831          42%     NaN         NaN
61      UC Santa Cr.  CA      32954          60%     NaN         NaN
62         U Chicago  IL      25277          13%     18%         NaN
63        U Delaware  DE      26534          53%     NaN         53%
64         U Florida  FL      29220          41%     NaN         41%
65      UNC Chapel H  NC      28491          27%     38%         16%
66          U. Penn.  PA      31217          12%     25%         10%
67      U Puget Sou.  WA       6772          53%     88%         53%
68  U Rochester[243]  NY      17428          36%     NaN         NaN
69               USC  CA      46030          18%     NaN         18%
70       U. Virginia  VA      27200          29%     29%         23%
71      U. Wisc-Mad.  WI      29008          54%     54%         NaN
72        Vanderbilt  TN      28335          13%     25%         12%
73            Vassar  NY       7908          22%     43%         21%
74       Wake Forest  NC      11366          32%     43%         31%
75        Wash & Lee  VA       5970          18%     40%         15%
76      Washington U  MO      28826          15%     31%         17%
77         Wesleyan  CT      10503          20%     45%         18%
78       William & M.  VA      13651          31%     48%         29%
79          Williams  MA       7067          17%     NaN         NaN
80              Yale  CT      28974           7%     16%          5%

[81 rows x 6 columns]
```

Python programmers will often suggest that there many ways the language can be used to solve a particular problem. But that some are more appropriate than others. The best solutions are celebrated as Idiomatic Python and there are lots of great examples of this on StackOverflow and other websites.

A sort of sub-language within Python, Pandas has its own set of idioms. We've alluded to some of these already, such as using vectorization whenever possible, and not using iterative loops if you don't need to. Several developers and users within the Panda's community have used the term pandorable for these idioms. I think it's a great term. So, I wanted to share with you a couple of key features of how you can make your code pandorable.

[2]:
```
import pandas as pd
import numpy as np
import timeit

df = pd.read_csv('census.csv')
df.head()
```

```
                                                                          ␣
↪---------------------------------------------------------------------------

      FileNotFoundError                          Traceback (most recent call␣
↪last)

      <ipython-input-2-2d4a7daf9860> in <module>
        3 import timeit
        4
    ----> 5 df = pd.read_csv('census.csv')
        6 df.head()


      /opt/conda/lib/python3.7/site-packages/pandas/io/parsers.py in␣
↪parser_f(filepath_or_buffer, sep, delimiter, header, names, index_col,␣
↪usecols, squeeze, prefix, mangle_dupe_cols, dtype, engine, converters,␣
↪true_values, false_values, skipinitialspace, skiprows, skipfooter, nrows,␣
↪na_values, keep_default_na, na_filter, verbose, skip_blank_lines, parse_dates,␣
↪infer_datetime_format, keep_date_col, date_parser, dayfirst, cache_dates,␣
↪iterator, chunksize, compression, thousands, decimal, lineterminator,␣
↪quotechar, quoting, doublequote, escapechar, comment, encoding, dialect,␣
↪error_bad_lines, warn_bad_lines, delim_whitespace, low_memory, memory_map,␣
↪float_precision)
      683         )
      684
  --> 685         return _read(filepath_or_buffer, kwds)
      686
      687     parser_f.__name__ = name


      /opt/conda/lib/python3.7/site-packages/pandas/io/parsers.py in␣
↪_read(filepath_or_buffer, kwds)
      455
      456     # Create the parser.
  --> 457     parser = TextFileReader(fp_or_buf, **kwds)
      458
      459     if chunksize or iterator:
```

```
    /opt/conda/lib/python3.7/site-packages/pandas/io/parsers.py in␣
↪__init__(self, f, engine, **kwds)
    893             self.options["has_index_names"] = kwds["has_index_names"]
    894
--> 895         self._make_engine(self.engine)
    896
    897     def close(self):


    /opt/conda/lib/python3.7/site-packages/pandas/io/parsers.py in␣
↪_make_engine(self, engine)
   1133     def _make_engine(self, engine="c"):
   1134         if engine == "c":
-> 1135             self._engine = CParserWrapper(self.f, **self.options)
   1136         else:
   1137             if engine == "python":


    /opt/conda/lib/python3.7/site-packages/pandas/io/parsers.py in␣
↪__init__(self, src, **kwds)
   1904         kwds["usecols"] = self.usecols
   1905
-> 1906         self._reader = parsers.TextReader(src, **kwds)
   1907         self.unnamed_cols = self._reader.unnamed_cols
   1908


    pandas/_libs/parsers.pyx in pandas._libs.parsers.TextReader.__cinit__()


    pandas/_libs/parsers.pyx in pandas._libs.parsers.TextReader.
↪_setup_parser_source()


    FileNotFoundError: [Errno 2] File b'census.csv' does not exist: b'census.
↪csv'
```

[ ]:
```
# The first of these is called method chaining.
# The general idea behind method chaining is that every method on an object
# returns a reference to that object. The beauty of this is that you can
# condense many different operations on a DataFrame, for instance, into one␣
↪line
# or at least one statement of code.
# Here's an example of two pieces of code in pandas using our census data.

# The first is the pandorable way to write the code with method chaining. In
```

```
# this code, there's no in place flag being used and you can see that when we
# first run a where query, then a dropna, then a set_index, and then a rename.
# You might wonder why the whole statement is enclosed in parentheses and␣
↪that's
# just to make the statement more readable.
(df.where(df['SUMLEV']==50)
    .dropna()
    .set_index(['STNAME','CTYNAME'])
    .rename(columns={'ESTIMATESBASE2010': 'Estimates Base 2010'}))
```

[3]:
```
# The second example is a more traditional way of writing code.
# There's nothing wrong with this code in the functional sense,
# you might even be able to understand it better as a new person to the␣
↪language.
# It's just not as pandorable as the first example.

df = df[df['SUMLEV']==50]
df.set_index(['STNAME','CTYNAME'], inplace=True)
df.rename(columns={'ESTIMATESBASE2010': 'Estimates Base 2010'})
```

```
      ␣
↪---------------------------------------------------------------------------

       NameError                                 Traceback (most recent call␣
↪last)

       <ipython-input-3-05c796c0e6d0> in <module>
         4 # It's just not as pandorable as the first example.
         5
   ----> 6 df = df[df['SUMLEV']==50]
         7 df.set_index(['STNAME','CTYNAME'], inplace=True)
         8 df.rename(columns={'ESTIMATESBASE2010': 'Estimates Base 2010'})


       NameError: name 'df' is not defined
```

[ ]:
```
# Now, the key with any good idiom is to understand when it isn't helping you.
# In this case, you can actually time both methods and see which one runs␣
↪faster

# We can put the approach into a function and pass the function into the timeit
# function to count the time the parameter number allows us to choose how many
# times we want to run the function. Here we will just set it to 1

def first_approach():
```

```
    global df
    return (df.where(df['SUMLEV']==50)
                .dropna()
                .set_index(['STNAME','CTYNAME'])
                .rename(columns={'ESTIMATESBASE2010': 'Estimates Base 2010'}))

timeit.timeit(first_approach, number=1)
```

[ ]:
```
# Now let's test the second approach. As we notice, we use our global variable
# df in the function. However, changing a global variable inside a function␣
↪will
# modify the variable even in a global scope and we do not want that to happen
# in this case. Therefore, for selecting summary levels of 50 only, I create
# a new dataframe for those records

# Let's run this for once and see how fast it is

def second_approach():
    global df
    new_df = df[df['SUMLEV']==50]
    new_df.set_index(['STNAME','CTYNAME'], inplace=True)
    return new_df.rename(columns={'ESTIMATESBASE2010': 'Estimates Base 2010'})
timeit.timeit(second_approach, number=1)
```

[ ]:
```
# As you can see, the second approach is much faster!
# So, this is a particular example of a classic time readability trade off.

# You'll see lots of examples on stock overflow and in documentation of people
# using method chaining in their pandas. And so, I think being able to read and
# understand the syntax is really worth your time.
# Here's another pandas idiom. Python has a wonderful function called map,
# which is sort of a basis for functional programming in the language.
# When you want to use map in Python, you pass it some function you want␣
↪called,
# and some iterable, like a list, that you want the function to be applied to.
# The results are that the function is called against each item in the list,
# and there's a resulting list of all of the evaluations of that function.

# Python has a similar function called applymap.
# In applymap, you provide some function which should operate on each cell of a
# DataFrame, and the return set is itself a DataFrame. Now I think applymap is
# fine, but I actually rarely use it. Instead, I find myself often wanting to
# map across all of the rows in a DataFrame. And pandas has a function that I
# use heavily there, called apply. Let's look at an example.

# Let's take our census DataFrame.
# In this DataFrame, we have five columns for population estimates.
```

```
# Each column corresponding with one year of estimates. It's quite reasonable␣
↪to
# want to create some new columns for
# minimum or maximum values, and the apply function is an easy way to do this.


# First, we need to write a function which takes in a particular row of data,
# finds a minimum and maximum values, and returns a new row of data nd returns
# a new row of data.  We'll call this function min_max, this is pretty straight
# forward. We can create some small slice of a row by projecting the population
# columns. Then use the NumPy min and max functions, and create a new series
# with a label values represent the new values we want to apply.

def min_max(row):
    data = row[['POPESTIMATE2010',
                'POPESTIMATE2011',
                'POPESTIMATE2012',
                'POPESTIMATE2013',
                'POPESTIMATE2014',
                'POPESTIMATE2015']]
    return pd.Series({'min': np.min(data), 'max': np.max(data)})
```

[ ]:
```
# Then we just need to call apply on the DataFrame.

# Apply takes the function and the axis on which to operate as parameters.
# Now, we have to be a bit careful, we've talked about axis zero being the rows
# of the DataFrame in the past. But this parameter is really the parameter of
# the index to use. So, to apply across all rows, which is applying on all
# columns, you pass axis equal to one.
df.apply(min_max, axis=1)
```

[ ]:
```
# Of course there's no need to limit yourself to returning a new series object.
# If you're doing this as part of data cleaning your likely to find yourself
# wanting to add new data to the existing DataFrame. In that case you just take
# the row values and add in new columns indicating the max and minimum scores.
# This is a regular part of my workflow when bringing in data and building
# summary or descriptive statistics.
# And is often used heavily with the merging of DataFrames.

# Here we have a revised version of the function min_max
# Instead of returning a separate series to display the min and max
# We add two new columns in the original dataframe to store min and max

def min_max(row):
    data = row[['POPESTIMATE2010',
                'POPESTIMATE2011',
                'POPESTIMATE2012',
                'POPESTIMATE2013',
```

```
                'POPESTIMATE2014',
                'POPESTIMATE2015']]
    row['max'] = np.max(data)
    row['min'] = np.min(data)
    return row
df.apply(min_max, axis=1)
```

[1]:
```
# Apply is an extremely important tool in your toolkit. The reason I introduced
# apply here is because you rarely see it used with large function definitions,
# like we did. Instead, you typically see it used with lambdas. To get the most
# of the discussions you'll see online, you're going to need to know how to
# at least read lambdas.

# Here's You can imagine how you might chain several apply calls with lambdas
# together to create a readable yet succinct data manipulation script. One line
# example of how you might calculate the max of the columns
# using the apply function.
rows = ['POPESTIMATE2010',
        'POPESTIMATE2011',
        'POPESTIMATE2012',
        'POPESTIMATE2013',
        'POPESTIMATE2014',
        'POPESTIMATE2015']
df.apply(lambda x: np.max(x[rows]), axis=1)
```

```
      ␣
↪---------------------------------------------------------------------------

      NameError                                 Traceback (most recent call␣
↪last)

      <ipython-input-1-f843db2c999b> in <module>
       15          'POPESTIMATE2014',
       16          'POPESTIMATE2015']
   ---> 17 df.apply(lambda x: np.max(x[rows]), axis=1)


      NameError: name 'df' is not defined
```

[ ]:
```
# The beauty of the apply function is that it allows flexibility in doing
# whatever manipulation that you desire, and the function you pass into apply
# can be any customized function that you write. Let's say we want to divide␣
↪the
# states into four categories: Northeast, Midwest, South, and West
# We can write a customized function that returns the region based on the state
```

```
# the state regions information is obtained from Wikipedia

def get_state_region(x):
    northeast = ['Connecticut', 'Maine', 'Massachusetts', 'New Hampshire',
                 'Rhode Island','Vermont','New York','New␣
↪Jersey','Pennsylvania']
    midwest = ['Illinois','Indiana','Michigan','Ohio','Wisconsin','Iowa',
               'Kansas','Minnesota','Missouri','Nebraska','North Dakota',
               'South Dakota']
    south = ['Delaware','Florida','Georgia','Maryland','North Carolina',
             'South Carolina','Virginia','District of Columbia','West Virginia',
             'Alabama','Kentucky','Mississippi','Tennessee','Arkansas',
             'Louisiana','Oklahoma','Texas']
    west = ['Arizona','Colorado','Idaho','Montana','Nevada','New Mexico','Utah',
            'Wyoming','Alaska','California','Hawaii','Oregon','Washington']

    if x in northeast:
        return "Northeast"
    elif x in midwest:
        return "Midwest"
    elif x in south:
        return "South"
    else:
        return "West"
```

[ ]:
```
# Now we have the customized function, let's say we want to create a new column
# called Region, which shows the state's region, we can use the customized
# function and the apply function to do so. The customized function is supposed
# to work on the state name column STNAME. So we will set the apply function on
# the state name column and pass the customized function into the apply␣
↪function
df['state_region'] = df['STNAME'].apply(lambda x: get_state_region(x))
```

[ ]:
```
# Now let's see the results
df[['STNAME','state_region']]
```

So there are a couple of Pandas idioms. But I think there's many more, and I haven't talked about them here. So here's an unofficial assignment for you. Go look at some of the top ranked questions on pandas on Stack Overflow, and look at how some of the more experienced authors, answer those questions. Do you see any interesting patterns? Chime in on the course discussion forums and let's build some pandorable documents together.

ExampleManipulatingDataFrames

November 15, 2021

In this lecture I'm going to walk through a basic data cleaning process with you and introduce you to a few more pandas API functions.

[1]:
```python
# Let's start by bringing in pandas
import pandas as pd
# And load our dataset. We're going to be cleaning the list of presidents in the US from wikipedia
df=pd.read_csv("datasets/presidents.csv")
# And lets just take a look at some of the data
df.head()
```

[1]:
```
   #          President              Born      Age atstart of presidency  \
0  1  George Washington  Feb 22, 1732[a]  57ăyears, 67ădaysApr 30, 1789
1  2         John Adams  Oct 30, 1735[a]  61ăyears, 125ădaysMar 4, 1797
2  3   Thomas Jefferson  Apr 13, 1743[a]  57ăyears, 325ădaysMar 4, 1801
3  4      James Madison  Mar 16, 1751[a]  57ăyears, 353ădaysMar 4, 1809
4  5        James Monroe     Apr 28, 1758  58ăyears, 310ădaysMar 4, 1817

        Age atend of presidency Post-presidencytimespan          Died  \
0   65ăyears, 10ădaysMar 4, 1797       2ăyears, 285ădays  Dec 14, 1799
1  65ăyears, 125ădaysMar 4, 1801      25ăyears, 122ădays   Jul 4, 1826
2  65ăyears, 325ădaysMar 4, 1809      17ăyears, 122ădays   Jul 4, 1826
3  65ăyears, 353ădaysMar 4, 1817      19ăyears, 116ădays  Jun 28, 1836
4  66ăyears, 310ădaysMar 4, 1825       6ăyears, 122ădays   Jul 4, 1831

                  Age
0  67ăyears, 295ădays
1  90ăyears, 247ădays
2   83ăyears, 82ădays
3  85ăyears, 104ădays
4   73ăyears, 67ădays
```

[2]:
```python
# Ok, we have some presidents, some dates, I see a bunch of footnotes in the "Born" column which might cause
# issues. Let's start with cleaning up that name into firstname and lastname. I'm going to tackle this with
# a regex. So I want to create two new columns and apply a regex to the projection of the "President" column.
```

```
# Here's one solution, we could make a copy of the President column
df["First"]=df['President']
# Then we can call replace() and just have a pattern that matches the last name␣
↪and set it to an empty string
df["First"]=df["First"].replace("[ ].*", "", regex=True)
# Now let's take a look
df.head()
```

[2]:
```
   #          President             Born      Age atstart of presidency  \
0  1  George Washington  Feb 22, 1732[a]  57ăyears, 67ădaysApr 30, 1789
1  2         John Adams  Oct 30, 1735[a]  61ăyears, 125ădaysMar 4, 1797
2  3   Thomas Jefferson  Apr 13, 1743[a]  57ăyears, 325ădaysMar 4, 1801
3  4      James Madison  Mar 16, 1751[a]  57ăyears, 353ădaysMar 4, 1809
4  5       James Monroe     Apr 28, 1758  58ăyears, 310ădaysMar 4, 1817

        Age atend of presidency Post-presidencytimespan          Died  \
0   65ăyears, 10ădaysMar 4, 1797       2ăyears, 285ădays  Dec 14, 1799
1  65ăyears, 125ădaysMar 4, 1801      25ăyears, 122ădays   Jul 4, 1826
2  65ăyears, 325ădaysMar 4, 1809      17ăyears, 122ădays   Jul 4, 1826
3  65ăyears, 353ădaysMar 4, 1817      19ăyears, 116ădays  Jun 28, 1836
4  66ăyears, 310ădaysMar 4, 1825       6ăyears, 122ădays   Jul 4, 1831

                 Age   First
0  67ăyears, 295ădays  George
1  90ăyears, 247ădays    John
2   83ăyears, 82ădays  Thomas
3  85ăyears, 104ădays   James
4   73ăyears, 67ădays   James
```

[3]:
```
# That works, but it's kind of gross. And it's slow, since we had to make a␣
↪full copy of a column then go
# through and update strings. There are a few other ways we can deal with this.␣
↪Let me show you the most
# general one first, and that's called the apply() function. Let's drop the␣
↪column we made first
del(df["First"])

# The apply() function on a dataframe will take some arbitrary function you␣
↪have written and apply it to
# either a Series (a single column) or DataFrame across all rows or columns.␣
↪Lets write a function which
# just splits a string into two pieces using a single row of data
def splitname(row):
    # The row is a single Series object which is a single row indexed by column␣
↪values
    # Let's extract the firstname and create a new entry in the series
```

```python
    row['First']=row['President'].split(" ")[0]
    # Let's do the same with the last word in the string
    row['Last']=row['President'].split(" ")[-1]
    # Now we just return the row and the pandas .apply() will take of merging␣
 ↪them back into a DataFrame
    return row

# Now if we apply this to the dataframe indicating we want to apply it across␣
 ↪columns
df=df.apply(splitname, axis='columns')
df.head()
```

[3]:

```
   #          President             Born      Age atstart of presidency  \
0  1  George Washington  Feb 22, 1732[a]  57ăyears, 67ădaysApr 30, 1789
1  2         John Adams  Oct 30, 1735[a]  61ăyears, 125ădaysMar 4, 1797
2  3   Thomas Jefferson  Apr 13, 1743[a]  57ăyears, 325ădaysMar 4, 1801
3  4      James Madison  Mar 16, 1751[a]  57ăyears, 353ădaysMar 4, 1809
4  5       James Monroe     Apr 28, 1758  58ăyears, 310ădaysMar 4, 1817

         Age atend of presidency Post-presidencytimespan          Died  \
0   65ăyears, 10ădaysMar 4, 1797       2ăyears, 285ădays  Dec 14, 1799
1  65ăyears, 125ădaysMar 4, 1801      25ăyears, 122ădays   Jul 4, 1826
2  65ăyears, 325ădaysMar 4, 1809      17ăyears, 122ădays   Jul 4, 1826
3  65ăyears, 353ădaysMar 4, 1817      19ăyears, 116ădays  Jun 28, 1836
4  66ăyears, 310ădaysMar 4, 1825       6ăyears, 122ădays   Jul 4, 1831

                  Age   First        Last
0  67ăyears, 295ădays  George  Washington
1  90ăyears, 247ădays    John       Adams
2   83ăyears, 82ădays  Thomas   Jefferson
3  85ăyears, 104ădays   James     Madison
4   73ăyears, 67ădays   James      Monroe
```

[4]:

```python
# Pretty questionable as to whether that is less gross, but it achieves the␣
 ↪result and I find that I use the
# apply() function regularly in my work. The pandas series has a couple of␣
 ↪other nice convenience functions
# though, and the next I would like to touch on is called .extract(). Lets drop␣
 ↪our firstname and lastname.
del(df['First'])
del(df['Last'])

# Extract takes a regular expression as input and specifically requires you to␣
 ↪set capture groups that
# correspond to the output columns you are interested in. And, this is a great␣
 ↪place for you to pause the
```

```
# video and reflect - if you were going to write a regular expression that␣
↪returned groups and just had the
# firstname and lastname in it, what would that look like?

# Here's my solution, where we match three groups but only return two, the␣
↪first and the last name
pattern="(^[\w]*)(?:.* )([\w]*$)"

# Now the extract function is built into the str attribute of the Series␣
↪object, so we can call it
# using Series.str.extract(pattern)
df["President"].str.extract(pattern).head()
```

[4]:
```
        0           1
0  George  Washington
1    John       Adams
2  Thomas   Jefferson
3   James     Madison
4   James      Monroe
```

[5]:
```
# So that looks pretty nice, other than the column names. But if we name the␣
↪groups we get named columns out
pattern="(?P<First>^[\w]*)(?:.* )(?P<Last>[\w]*$)"

# Now call extract
names=df["President"].str.extract(pattern).head()
names
```

[5]:
```
    First        Last
0  George  Washington
1    John       Adams
2  Thomas   Jefferson
3   James     Madison
4   James      Monroe
```

[6]:
```
# And we can just copy these into our main dataframe if we want to
df["First"]=names["First"]
df["Last"]=names["Last"]
df.head()
```

[6]:
```
   #          President             Born      Age atstart of presidency  \
0  1  George Washington  Feb 22, 1732[a]  57ăyears, 67ădaysApr 30, 1789
1  2         John Adams  Oct 30, 1735[a]  61ăyears, 125ădaysMar 4, 1797
2  3   Thomas Jefferson  Apr 13, 1743[a]  57ăyears, 325ădaysMar 4, 1801
3  4      James Madison  Mar 16, 1751[a]  57ăyears, 353ădaysMar 4, 1809
4  5       James Monroe     Apr 28, 1758  58ăyears, 310ădaysMar 4, 1817

        Age atend of presidency Post-presidencytimespan          Died  \
0  65ăyears, 10ădaysMar 4, 1797       2ăyears, 285ădays  Dec 14, 1799
```

```
1  65ăyears, 125ădaysMar 4, 1801      25ăyears, 122ădays   Jul 4, 1826
2  65ăyears, 325ădaysMar 4, 1809      17ăyears, 122ădays   Jul 4, 1826
3  65ăyears, 353ădaysMar 4, 1817      19ăyears, 116ădays  Jun 28, 1836
4  66ăyears, 310ădaysMar 4, 1825       6ăyears, 122ădays   Jul 4, 1831

                 Age   First        Last
0  67ăyears, 295ădays  George  Washington
1  90ăyears, 247ădays    John       Adams
2   83ăyears, 82ădays  Thomas   Jefferson
3  85ăyears, 104ădays   James     Madison
4   73ăyears, 67ădays   James      Monroe
```

[7]:
```
# It's worth looking at the pandas str module for other functions which have␣
↪been written specifically
# to clean up strings in DataFrames, and you can find that in the docs in the␣
↪Working with Text
# section: https://pandas.pydata.org/pandas-docs/stable/user_guide/text.html
```

[8]:
```
# Now lets move on to clean up that Born column. First, let's get rid of␣
↪anything that isn't in the
# pattern of Month Day and Year.
df["Born"]=df["Born"].str.extract("([\w]{3} [\w]{1,2}, [\w]{4})")
df["Born"].head()
```

[8]:
```
0    Feb 22, 1732
1    Oct 30, 1735
2    Apr 13, 1743
3    Mar 16, 1751
4    Apr 28, 1758
Name: Born, dtype: object
```

[9]:
```
# So, that cleans up the date format. But I'm going to foreshadow something␣
↪else here - the type of this
# column is object, and we know that's what pandas uses when it is dealing with␣
↪string. But pandas actually
# has really interesting date/time features - in fact, that's one of the␣
↪reasons Wes McKinney put his efforts
# into the library, to deal with financial transactions. So if I were building␣
↪this out, I would actually
# update this column to the write data type as well
df["Born"]=pd.to_datetime(df["Born"])
df["Born"].head()
```

[9]:
```
0   1732-02-22
1   1735-10-30
2   1743-04-13
3   1751-03-16
4   1758-04-28
Name: Born, dtype: datetime64[ns]
```

[10]:

```
# This would make subsequent processing on the dataframe around dates, such as
 ↪getting every President who
# was born in a given time span, much easier.
```

Now, most of the other columns in this dataset I would clean in a similar fashion. And this would be a good practice activity for you, so I would recommend that you pause the video, open up the notebook for the lecture if you don't already have it opened, and then finish cleaning up this dataframe. In this lecture I introduced you to the str module which has a number of important functions for cleaning pandas dataframes. You don't have to use these - I actually use apply() quite a bit myself, especially if I don't need high performance data cleaning because my dataset is small. But the str functions are incredibly useful and build on your existing knowledge of regular expressions, and because they are vectorized they are efficient to use as well.

assignment2

November 15, 2021

1 Assignment 2

For this assignment you'll be looking at 2017 data on immunizations from the CDC. Your datafile for this assignment is in `assets/NISPUF17.csv`. A data users guide for this, which you'll need to map the variables in the data to the questions being asked, is available at `assets/NIS-PUF17-DUG.pdf`. **Note: you may have to go to your Jupyter tree (click on the Coursera image) and navigate to the assignment 2 assets folder to see this PDF file).**

1.1 Question 1

Write a function called `proportion_of_education` which returns the proportion of children in the dataset who had a mother with the education levels equal to less than high school (<12), high school (12), more than high school but not a college graduate (>12) and college degree.

This function should return a dictionary in the form of (use the correct numbers, do not round numbers):

```
{"less than high school":0.2,
"high school":0.4,
"more than high school but not college":0.2,
"college":0.2}
```

[6]:
```
def proportion_of_education():
    # your code goes here
    # YOUR CODE HERE
    import pandas as pd
    df=pd.read_csv('assets/NISPUF17.csv',index_col=0)
    df.columns=[x.lower().strip() for x in df.columns]
    dico={}
    dico["less than high school"]=len(df[df['educ1']==1]['educ1'])/
 ↪len(df['educ1'])
    dico["high school"]=len(df[df['educ1']==2]['educ1'])/len(df['educ1'])
    dico["more than high school but not␣
 ↪college"]=len(df[df['educ1']==3]['educ1'])/len(df['educ1'])
    dico["college"]=len(df[df['educ1']==4]['educ1'])/len(df['educ1'])
    return dico
    raise NotImplementedError()
proportion_of_education()
```

[6]: {'less than high school': 0.10202002459160373,
 'high school': 0.172352011241876,
 'more than high school but not college': 0.24588090637625154,
 'college': 0.47974705779026877}

[7]:
```
assert type(proportion_of_education())==type({}), "You must return a dictionary.
 ↪"
assert len(proportion_of_education()) == 4, "You have not returned a dictionary␣
 ↪with four items in it."
assert "less than high school" in proportion_of_education().keys(), "You have␣
 ↪not returned a dictionary with the correct keys."
assert "high school" in proportion_of_education().keys(), "You have not␣
 ↪returned a dictionary with the correct keys."
assert "more than high school but not college" in proportion_of_education().
 ↪keys(), "You have not returned a dictionary with the correct keys."
assert "college" in proportion_of_education().keys(), "You have not returned a␣
 ↪dictionary with the correct keys."
```

1.2 Question 2

Let's explore the relationship between being fed breastmilk as a child and getting a seasonal influenza vaccine from a healthcare provider. Return a tuple of the average number of influenza vaccines for those children we know received breastmilk as a child and those who know did not.

This function should return a tuple in the form (use the correct numbers:

```
(2.5, 0.1)
```

[8]:
```
def average_influenza_doses():
    # YOUR CODE HERE
    import pandas as pd
    import numpy as np
    df=pd.read_csv('assets/NISPUF17.csv',index_col=0)
    df.columns=[x.lower().strip() for x in df.columns]
    cond=(df['cbf_01']==1)|(df['cbf_01']==2)
    df1=df[cond]
    #df1['p_numhs']=df1['p_numhs'].replace(np.nan,0)
    #df1['p_numhg']=df1['p_numhg'].replace(np.nan,0)
    #df1['numh1']=df1['p_numhs']+df1['p_numhg']
    mean1=df1[df1['cbf_01']==1]['p_numflu']
    mean1=np.nanmean(mean1)
    mean2=df1[df1['cbf_01']==2]['p_numflu']
    mean2=np.nanmean(mean2)
    return (mean1,mean2)
    raise NotImplementedError()
average_influenza_doses()
```

[8]: (1.8799187420058687, 1.5963945918878317)

[7]:
```
assert len(average_influenza_doses())==2, "Return two values in a tuple, the␣
↪first for yes and the second for no."
```

1.3 Question 3

It would be interesting to see if there is any evidence of a link between vaccine effectiveness and sex of the child. Calculate the ratio of the number of children who contracted chickenpox but were vaccinated against it (at least one varicella dose) versus those who were vaccinated but did not contract chicken pox. Return results by sex.

This function should return a dictionary in the form of (use the correct numbers):

```
{"male":0.2,
"female":0.4}
```

Note: To aid in verification, the `chickenpox_by_sex()['female']` value the autograder is looking for starts with the digits `0.0077`.

[12]:
```
def chickenpox_by_sex():
    # YOUR CODE HERE
    import pandas as pd
    df=pd.read_csv('assets/NISPUF17.csv',index_col=0)
    df.columns=[x.lower().strip() for x in df.columns]
    df2=df[df['p_numvrc']>0]
    df2m=df2[df2['sex']==1]
    cond1=df2m['had_cpox']==2
    male=len(df2m[df2m['had_cpox']==1])/len(df2m[cond1])
    df2f=df2[df2['sex']==2]
    cond2=df2f['had_cpox']==2
    female=len(df2f[df2f['had_cpox']==1])/len(df2f[cond2])
    return {'male':male,'female':female}
    raise NotImplementedError()
chickenpox_by_sex()
```

[12]:
```
{'male': 0.009675583380762664, 'female': 0.0077918259335489565}
```

[11]:
```
assert len(chickenpox_by_sex())==2, "Return a dictionary with two items, the␣
↪first for males and the second for females."
```

1.4 Question 4

A correlation is a statistical relationship between two variables. If we wanted to know if vaccines work, we might look at the correlation between the use of the vaccine and whether it results in prevention of the infection or disease [1]. In this question, you are to see if there is a correlation between having had the chicken pox and the number of chickenpox vaccine doses given (varicella).

Some notes on interpreting the answer. The `had_chickenpox_column` is either `1` (for yes) or `2` (for no), and the `num_chickenpox_vaccine_column` is the number of doses a child has been given of the varicella vaccine. A positive correlation (e.g., `corr > 0`) means that an increase in `had_chickenpox_column` (which means more no's) would also increase the values of `num_chickenpox_vaccine_column` (which means more doses of vaccine). If there is a negative correlation (e.g., `corr < 0`), it indicates that having had chickenpox is related to an increase in the number of vaccine doses.

Also, `pval` is the probability that we observe a correlation between `had_chickenpox_column` and `num_chickenpox_vaccine_column` which is greater than or equal to a particular value occurred by chance. A small `pval` means that the observed correlation is highly unlikely to occur by chance. In this case, `pval` should be very small (will end in `e-18` indicating a very small number).

[1] This isn't really the full picture, since we are not looking at when the dose was given. It's possible that children had chickenpox and then their parents went to get them the vaccine. Does this dataset have the data we would need to investigate the timing of the dose?

[5]:
```
def corr_chickenpox():
    import scipy.stats as stats
    import numpy as np
    import pandas as pd

    # this is just an example dataframe
    df=pd.DataFrame({"had_chickenpox_column":np.random.randint(1,3,size=(100)),
                   "num_chickenpox_vaccine_column":np.random.
 ↪randint(0,6,size=(100))})

    # here is some stub code to actually run the correlation
    corr, pval=stats.
 ↪pearsonr(df["had_chickenpox_column"],df["num_chickenpox_vaccine_column"])

    # just return the correlation
    #return corr

    # YOUR CODE HERE
    DF=pd.read_csv('assets/NISPUF17.csv',index_col=0)
    DF.columns=[x.lower().strip() for x in DF.columns]
    cond=(DF['had_cpox']==1)|(DF['had_cpox']==2)
    DF2=DF[cond]
    DF3=DF2[DF2['p_numvrc']>=0]
    CORR, PVAL=stats.pearsonr(DF3['had_cpox'],DF3['p_numvrc'])
    return CORR
    raise NotImplementedError()
corr_chickenpox()
```

[5]: 0.07044873460147986

[30]:
```
assert -1<=corr_chickenpox()<=1, "You must return a float number between -1.0␣
 ↪and 1.0."
```

[ ]:
```
```