Chapter 14

# Introduction

DRAFT

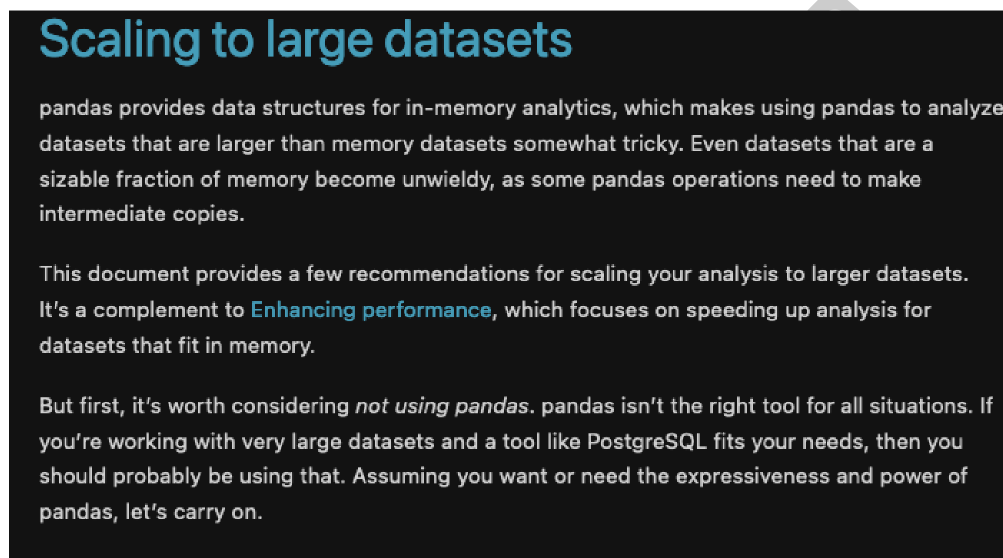
## Contents


1 What is Pandas . . . . . . 225
2 Data structures . . . . . . 226
3 Selecting Columns and Rows . . . . . . 231
4 Column Types Conversion . . . . . . 237
5 Dealing with NaN . . . . . . 237
6 Choosing the largest and smallest values . . . . . . 239
7 Manipulating Data & Method Chaining . . . . . . 240
8 Indexes: Creating and Dropping . . . . . . 244
9 Views and Copies . . . . . . 246


DRAFT

# 1 What is Pandas

- Pandas is a Python library for manipulating data developed by Wes McKinney.
- Pandas itself is a front-end API for manipulating data that is stored *in-memory*. This type of tool is often called an *in-memory database* and, when appropriate, this type of database is the far superior than the relational databases that we have been talking about previously.
- The limitation of this type of database is that the data needs to be small enough to fit into your computer's RAM. For modern data analysis this is a major limitation as many data sets are far larger than the present memory.
- The pandas documentation provides a nice summary of this:[1]



## Scaling to large datasets

pandas provides data structures for in-memory analytics, which makes using pandas to analyze datasets that are larger than memory datasets somewhat tricky. Even datasets that are a sizable fraction of memory become unwieldy, as some pandas operations need to make intermediate copies.

This document provides a few recommendations for scaling your analysis to larger datasets. It's a complement to Enhancing performance, which focuses on speeding up analysis for datasets that fit in memory.

But first, it's worth considering *not using pandas*. pandas isn't the right tool for all situations. If you're working with very large datasets and a tool like PostgreSQL fits your needs, then you should probably be using that. Assuming you want or need the expressiveness and power of pandas, let's carry on.

Figure 14.1: Pandas limitations

- Mentally, pandas consists of a front-end API and a backend in-memory data structure. Before version 2.0 the only backend that was in use was numpy. Specifically complex pandas data structures were built upon numpy based objects.[2]
- The (still default) backend required significant memory use to store data. Per Wes McKinney, *pandas rule of thumb: have 5 to 10 times as much RAM as the size of your dataset*, which is a pretty rough standard to hew to.
- The latest release of pandas, version 2.0, which will be released in *March, 2023* creates a more significant abstraction between the back- and front-end API and allows for the use of Apache Arrow as a backend storage structure in place of numpy.
- Apache Arrow is a columnar in-memory data structure which is optimized for vectorization via SIMD (single instruction, multiple data) routines. By using the Apache Arrow backend, rather than numpy, the ratio to dataset size to memory size is much more manageable.
- The Apache Arrow API is very similar to numpy, but has a number of additional types and also has better support for missing values across types.

[1] Taken from `https://pandas.pydata.org/docs/user_guide/scale.html`

[2] Wes McKinney's blog has a bunch of information about the decision making at the time, you can find it here: `https://wesmckinney.com/blog/apache-arrow-pandas-internals/`

- Pandas is both powerful and useful, especially when doing data analysis. However it comes with some significant downsides:

    1. Pandas is more imperative than SQL which means you tell the interpreter what to do, step-by-step. For data operations there are often many ways to undertake the same operation which means that two people can write two very different Pandas expressions and generate the same results.

    2. Pandas is *not* persistent. It stores data in memory and, if the computer crashes, you lose it.

    3. Pandas is only efficient when the size of data is small enough to "fit" into memory.

    4. Pandas is *not* easily scalable across multiple computers.

- **Since, at the time of the start of this course, version 2.0 has *NOT* been released, we will still be leveraging version 1.5 in the course of this class.**

- To begin we import the Pandas and NumPy packages. By convention, we usually do it as follows:

```
>>> import numpy as np

>>> import pandas as pd
```

- So, when should you use Pandas? You should use it when you have data that easily fits into your computer's memory and you wish to explore or perform straightforward analysis on that data.

## 2 Data structures

- Starting from the bottom there are a number of data *types*. The chart below shows how they line up with underlying Python and Numpy types.

| Description | Pandas Type | Numpy Type | Python (built-in) |
|---|---|---|---|
| Integers | int64 | int8, int16, int32, int64, etc. | int |
| Text | object | string, unicode | str |
| Double/Float | float64 | float16, float32, float64 | float |
| Boolean (T/F) | bool | bool | bool |
| DateTime | datetime64 | datetime64 | N/A |

Table 14.1: Common Data Types

- The types above are how the underlying data is stored, what operations are allowed on it and how those operations act. For example a "+" will do string concatenation when paired with two objects, but will undertake mathematical addition when paired with floats and integers.

- In Pandas these underlying data types are what we put together to build the two basic data structures that we find in Pandas: DataFrames and Series.

- To determine the type of a particular python object we use the `type` command:

```
>>> type( [1,2,3])
```

which will return `list` or `<class 'list'>`.

- The type function works on both underlying data types as well as the larger data structures.

DRAFT

- **An important thing to remember:** Pandas is a bit inconsistent in its design and it is frequently the case that a function, operation or something you do will inadvertently change your data. Whenever diagnosing an issue in Pandas, make sure to check the type. You'll often be surprised about what you are working with.
- The way to think about both a Series and a DataFrame is that they are two objects (index and values), combined. The value component contains the actual data while the index component is its own object which has operations that are allowed on it. Note that index objects are complex and can exist in both *rows* and *columns*. We will touch upon this a bit later.

### Series

- A Series is one dimensional data object with an associated *index*.
- There are a number of different way to create a list. For example, consider the following two commands which initialize two variables ("x_1" and "x_2") which contain the same values.
- The "name" input defines what the column name is while the list at the start are the values. In the second example, we also have an official "index" which creates labels for the rows of the data.

```
>>> x_1 = pd.Series( [1,2,3,4], name="v1")

>>> x_2 = pd.Series( [1,2,3,4], index = ['a', 'b', 'c', 'd'], name="v1")
```

- Printing the values yields the following:

```
>>> x_1
0    1
1    2
2    3
3    4
Name: v1, dtype: int64

>>> x_2
a    1
b    2
c    3
d    4
Name: v1, dtype: int64
```

- Note that `x_1` has an index starting from zero and ending at 3 while the second has an index of a, b, c, d.
- We can access the values and index of a series using the following attributes:

```
>>> x_2.values
[1 2 3 4]

>>> type(x_2.values)
<class 'numpy.ndarray'>
```

The returned object is an *array*, not a series! When we ask for the index, we return an "Index" object, which is a type of Pandas data structure that is similar to a data frame.

```
>>> x_1.index
RangeIndex(start=0, stop=4, step=1)

>>> x_2.index
Index(['a', 'b', 'c', 'd'], dtype='object')
```

- To get the size of a Series we can use the "size" or "shape" attributes. Shape is used more frequently as it will return the *shape* in the case of multidimensional objects (size only returns the *number* of objects).

```
>>> x_2.size
4

>>> x_2.shape
(4,)
```

The "shape" command returns a tuple.

- Not only can the series have a name (such as "v1" in our above examples), but the index can also have a name. This is not something frequently encountered.

```
>>> x_2.name

>>> x_2.index.name = 'alpha'

>>> print(x_2)
alpha
a    1
b    2
c    3
d    4
Name: v1, dtype: int64
```

- Math can be done between Pandas objects:

```
>>> x_1 + x_1
0    2
1    4
2    6
3    8
Name: v1, dtype: int64

>>> x_2 + x_2
alpha
a    2
b    4
c    6
d    8
Name: v1, dtype: int64
```

- Indexes are *incredibly* important because they strongly effect how operators work. In particular, indexes in Pandas align data and when series are added together real effects occur. For example, consider the following:

```
>>> ans = x_1 + x_2
  v1
----
 nan
 nan
 nan
 nan
 nan
[...]
```

  If we look at what is returned it is another Pandas Series, with a single column which has the name "v1" and eight values, all of them NaN.[3] Why did this occur? Two reasons:

  1. Since the indexes didn't align, the addition operator assumed that the rows did not align and thus the resulting dataset had 8 rows $(4 + 4)$.
  2. When adding objects in Pandas, anything added to NaN is equal to NaN.

- If we only want to display the first few data points in a series, we can use the "head" method which can be used as follows:

```
>>> x_2.head(2)
alpha
a    1
b    2
Name: v1, dtype: int64
```

  There is also a "tail" method which returns the last rows:

```
>>> x_2.tail(2)
alpha
c    3
d    4
Name: v1, dtype: int64
```

  If no integer argument is provided the above methods return 5 values.

- If we want to find out information about what data types are in the series we can use the the `dtypes` variable associated with the object:

```
>>> x_2.dtypes
int64
```

### DataFrame

In this section we are going to use the Iowa Cars data (as in the previous SQL section). To use this data, we load the information in a DataFrame, as the command below does:

[3]Interestingly, NaA is from the numpy library, so can be called as "nupy.NaN"



DRAFT

```
>>> dfCars = pd.read_csv('<FILEPATH>/iowa_cars.tdf'
                 ,sep='\t', engine='python', names=['year', 'countyname'
                          , 'motorvehicle' ,'vehiclecat', 'vehicletype'
                          , 'tonnage', 'registrations', 'annualfee'
                          , 'completecategory'])
```

Note that <FILEPATH> needs to be set to your local copy of file.

- A DataFrame is two dimensional data object (think of a table) and an associated index.
- *head, tail, shape, size, index* and *dtypes* are all available and behave as you'd expect:

```
>>> dfCars.head()
  year  countyname    motorvehicle    vehiclecat    vehicletype     tonnage       [...]
------  ------------  --------------  ------------  --------------  ---------  -- [...]
  2008  Ida           Yes             Bus           Bus                           [...]
  2011  Jasper        Yes             Moped         Moped                         [...]
  2012  Harrison      Yes             Truck         Truck           3 Tons        [...]
  2015  Palo Alto     No              Trailer       Travel Trailer                [...]
  2016  Adair         Yes             Truck         Truck           3 Tons        [...]

>>> dfCars.shape
(41202, 9)

>>> dfCars.size
370818

>>> dfCars.dtypes
year                  int64
countyname           object
motorvehicle         object
vehiclecat           object
vehicletype          object
tonnage              object
registrations         int64
annualfee           float64
completecategory     object
dtype: object

>>> dfCars.index
RangeIndex(start=0, stop=41202, step=1)
```

- Outside of `dtypes`, which describes the type of data in each column, the method `describe()` creates a set of summary statistics for all columns:

```
>>> dfCars.describe()
               year  registrations     annualfee
count  41202.000000   41202.000000  3.825800e+04
mean    2013.724504    1767.522135  2.122614e+05
std        4.697944    7417.077934  1.181381e+06
min     2005.000000       1.000000  0.000000e+00
25%     2010.000000      25.000000  3.020000e+03
50%     2014.000000     183.000000  2.464500e+04
75%     2018.000000     989.750000  9.136975e+04
max     2021.000000  218975.000000  4.964507e+07
```

- Note that addition with DataFrames and Series can behave unexpectedly. Consider the following

examples:

```
>>> x_2 = pd.Series( [1,2,3,4], index = ['a', 'b', 'c', 'd'], name="v1")

>>> x_3 = pd.Series( [1,2,3,4], index = ['a', 'b', 'c', 'd'], name="v2")

>>> d_2 = x_2.to_frame()

>>> d_3 = x_3.to_frame()
```

We now have two series (`x_2` and `x_3`) which are the same except for the name ("v1" vs. "v2") and we have two DataFrames based off of those series. Let's do some math:

```
>>> x_2 + x_3
a    2
b    4
c    6
d    8
dtype: int64

>>> d_2 + d_3
   v1  v2
a NaN NaN
b NaN NaN
c NaN NaN
d NaN NaN
```

In the first example we can see that the columns are combined, even though their names are different, but in the second we up with a DataFrame with two NaN columns. In other words, operations on a Series ignore the name while in the DataFrame they do not!

## 3 Selecting Columns and Rows

- There are a *ton* of ways to select rows and columns using Pandas.
- When I first learned Pandas this was the most frustrating part of the experience. In order to minimize the frustration, I recommend focusing on identifying exactly what data object you have and what data object you want returned and then learning only one method for accessing data in that fashion.
- To specify rows and columns, from a DataFrame, we will use `loc` and `iloc` methods.[4]
- To specify objects, there are a number of different notational devices that can be used:
  - For a single column we can use "dot" notation, with a column name after the dot, which returns a series. Note that this only works if the column name is *not* a reserved word for a DataFrame.

    ```
    >>> type( dfCars.tonnage )
    <class 'pandas.core.series.Series'>
    ```

  - Note that a "dot" can be used with any named object, such as an index:

[4] Another one, "ix", was common but is now deprecated and will not be acceptable in this course.

```
>>> x_2.b
2
```

- We can also specify a column using square brackets:

```
>>> type( dfCars['tonnage'] )
<class 'pandas.core.series.Series'>
```

- We can also use double square brackets:

```
>>> type( dfCars[['tonnage']] )
<class 'pandas.core.frame.DataFrame'>
```

Look at the difference between the previous two examples: the first returned a Series while the second returned a DataFrame. We will find that this convention routinely reappears when using pandas – supplying an itterable as a selector returns a DataFrame while an atomic value returns a Series.

- We can also use `loc`, though to select all rows we need to prepend a colon:

```
>>> type( dfCars.loc[:, 'tonnage'] )
<class 'pandas.core.series.Series'>

>>> type( dfCars.loc[:, ['tonnage']] )
<class 'pandas.core.frame.DataFrame'>
```

- Keep in mind is that using `loc` is the preferred method for accessing data based on contents. One issue that can occur when using alternative methods, such as only using the `[[column]]` is that Pandas can interpret this as a *row* specification, rather than column name. No error is returned when this occurs – instead an empty DataFrame is returned as no rows match that definition.
- In this course **you should always use `loc` when selecting rows and columns without leveraging an index**.
- If there is an index, we can specify rows by using loc with the index value:

```
>>> x_2 = pd.Series( [1,2,3,4], index = ['a', 'b', 'c', 'd'], name="v1")

>>> x_2['b']
2
```

- **Strong recommendation: specify both rows and columns when selecting.** The reason for this is that when reading something like `df.loc['value']` or `df['value']` it's difficult to know if this is referring to a row with an index or a column with a name.
- We can use `iloc`, which is an *integer* based access for rows and columns, though this requires knowing that tonnage is the 7th column:

DRAFT

```
>>> type( dfCars.iloc[:, 7] )
<class 'pandas.core.series.Series'>

>>> type( dfCars.iloc[:, [7]] )
<class 'pandas.core.frame.DataFrame'>
```

– The `iloc` method takes any standard *slice* when it is used and can refer to both rows and columns. For example, the code below selects the first 10 columns and rows 20 through 50.

```
>>> d_1 = dfCars.iloc[20:50, 0:10]

>>> type(d_1)
<class 'pandas.core.frame.DataFrame'>

>>> d_1.shape
(30, 9)
```

When looking at the above, some of the methods return DataFrames while others return series. **Keep in mind *what* is being returned! Many of the issues with Pandas that user's encounter is due to an unexpected return type!**

- When you start with a Series there is no need to select columns because there is only a single column.
- Now that we know how to convert from a DataFrame to a Series, how do we go the other direction? To do this we, we use the command `to_frame` on the series:

```
>>> type(dfCars.loc[:, 'year'])
<class 'pandas.core.series.Series'>

>>> type(dfCars.loc[:, 'year'].to_frame())
<class 'pandas.core.frame.DataFrame'>
```

## Value Counts

- A really useful *Series* method is `value_counts` which returns another series containing the unique values and the number of occurrences of that value within the series:

```
>>> dfCars.loc[:, 'tonnage'].value_counts()
tonnage
<10000 lbs                    3401
6+ Tons Non-Special Usage     2970
6+ Tons Special Usage         2970
3 Tons                        2747
>10000 lbs                    2217
5 Tons                        2089
4 Tons                        1769
0 Tons                         429
2 Tons                           6
Name: count, dtype: int64
```

DRAFT

- By default the `value_counts` method ignores missing values. If we want to include missing values, the parameter `dropna` is set to False:

```
>>> dfCars.loc[:, 'tonnage'].value_counts(dropna=False)
tonnage
None                         22604
<10000 lbs                    3401
6+ Tons Non-Special Usage     2970
6+ Tons Special Usage         2970
3 Tons                        2747
>10000 lbs                    2217
5 Tons                        2089
4 Tons                        1769
0 Tons                         429
2 Tons                           6
Name: count, dtype: int64
```

## Content-based selections

- Probably the most common operation done when analyzing data is selecting rows based on a criteria. When doing this with `loc` we provide a boolean object of the same length as the DataFrame. For example, let's consider the following few lines of code:

```
>>> dfCarsC = dfCars.copy()

>>> dfCarsC = dfCarsC.loc[ (dfCarsC.loc[:, 'vehicletype'] == 'Semi Trailer') , ['countyname']]

>>> dfCarsC.head()
countyname
------------
Appanoose
Jefferson
Mahaska
Clay
Greene
```

The first line takes our original cars DataFrame and makes a "copy" of it. Why would we do this? Two reasons:

1. Because we may possibly change our original data and, rather than reload it each time, we can make a copy so that we can always restart.[5]
2. Get in the habit of making copies. As we will see later, unexpected issues with pandas can arise because pandas often creates "views" of a DataFrame, rather than make a copy. It is not uncommon for analysis to be incorrect because a user fails to account for views vs. copies.

The second line has three components:

1. `(dfCarsC.loc [:, 'vehicletype'] == 'Semi Trailer')` This creates an array of True and False values based on the boolean condition. In other words, this returns a series of 41,202 items.
2. `dfCarsC.loc[]` The `loc` method, when used in this manner, only returns those rows which the resulting condition evaluate True, so this is going to keep 1,683 rows.

[5]This is good practice when doing exploratory data analysis as it allows you to quickly restart if you do something incorrect.



DRAFT

3. `['countyname']` This portion of the code keeps only the countyname.

Note the following:

- Pandas uses the double equal "==" to do *comparison*. Keep this in mind!
- Why did we have to re-assign the variable (`dfCarsC = `) in the second line? We had to reassign this because the loc command does not do *in-place* changes. If we didn't do this re-assignment `dfCarsC` would not have changed.
- This returns a DataFrame and not a Series since we selected the column with a list. If we, instead, used `'countryname'` not in a list form we would get a 1 by 1,683 Series instead of a series of 1,683.
- Put all boolean objects in parenthesis when using Pandas!!! Just get used to it, evaluation precedence is AND/OR above equality and inequality, meaning that things will go badly if you don't.
- This example has nested a number of different "things" together. One way to make sure that your Pandas code is readable is by making sure not to overload too many operations into a single line of code.

- Let's do another one! How about we select all columns and only those rows with registrations larger than 10,000? Unsurprisingly, basic row-by-row math operations also work fine.

```
>>> dfCarsC = dfCars.copy()

>>> dfCarsC = dfCarsC.loc[(dfCarsC.loc[:, 'registrations'] > 10000), :]

>>> dfCarsC
  year  countyname    motorvehicle    vehiclecat    vehicletype            tonnag [...]
------  ------------  --------------  ------------  ---------------------  ------ [...]
  2010  Johnson       Yes             Truck         Truck                  3 Tons [...]
  2008  Cerro Gordo   Yes             Automobile    Automobile                    [...]
  2006  Wapello       Yes             Automobile    Automobile                    [...]
  2009  Black Hawk    No              Trailer       Small Regular Trailer         [...]
  2012  Plymouth      Yes             Automobile    Automobile                    [...]
[...]
```

- We can combine multiple conditions using "and" (&), "or" (|) and not (∼). For example, if we want to get all data from Wright county with more than 1,200 registrations we can run the following commands:

```
>>> dfCarsC = dfCars.copy()

>>> dfCarsC = dfCarsC.loc[(dfCarsC.loc[:, 'registrations'] > 1200)
        & (dfCarsC.loc[:, 'countyname'] == 'Wright'), :]

>>> dfCarsC.head()
  year  countyname    motorvehicle    vehiclecat     vehicletype            tonna [...]
------  ------------  --------------  -------------  ---------------------  ----- [...]
  2015  Wright        Yes             Truck          Truck                  3 Ton [...]
  2005  Wright        Yes             Automobile     Automobile                   [...]
  2015  Wright        Yes             Multi-purpose  Multi-purpose                [...]
  2008  Wright        Yes             Truck          Truck                  3 Ton [...]
  2011  Wright        No              Trailer        Small Regular Trailer        [...]
```

Annoyingly these lines are getting longer and longer which makes them more and more difficult to read. To get around this problem, we can add parenthesis around the expression. Adding a

parenthesis allows us to arbitrarily put hard returns and tabs in the expression for organizational purposes:

```
>>> dfCarsC = dfCars.copy()

>>> dfCarsC = (dfCarsC
                .loc[(dfCarsC.loc[:, 'registrations'] > 1200)
                    & (dfCarsC.loc[:, 'countyname'] == 'Wright'), :])

>>> dfCarsC.head()
  year  countyname    motorvehicle    vehiclecat     vehicletype             tonna [...]
------  ------------  --------------  -------------  ---------------------   ----- [...]
  2015  Wright        Yes             Truck          Truck                   3 Ton [...]
  2005  Wright        Yes             Automobile     Automobile                    [...]
  2015  Wright        Yes             Multi-purpose  Multi-purpose                 [...]
  2008  Wright        Yes             Truck          Truck                   3 Ton [...]
  2011  Wright        No              Trailer        Small Regular Trailer         [...]
```

- We can also nest multiple layers of logic using parenthesis:

```
>>> dfCarsC = dfCars.copy()

>>> dfCarsC = (dfCarsC.loc[(
        ((dfCarsC.loc[:, 'registrations'] > 1200) & (dfCarsC.loc[:, 'registrations'] <= 3000))
        |
        ((dfCarsC.loc[:, 'registrations'] > 4000) & (dfCarsC.loc[:, 'registrations'] <= 4200))
        )
        & (dfCarsC.loc[:, 'countyname'] == 'Wright'), :])
```

which will return all rows from Wright county with between 1,200 and 3,000 registrations or 4,000 and 4,200 registrations.

- Keep in mind when writing these commands is that the result of what is inside `loc` needs to be a list of true/false boolean expressions *of the same length as the DataFrame*.

- We can combine these operations to filter both on rows and columns:

```
>>> dfCarsC = dfCars.copy()

>>> dfCarsC = (dfCarsC.loc[(
        ((dfCarsC.loc[:, 'registrations'] > 1200) & (dfCarsC.loc[:, 'registrations'] <= 3000))
        |
        ((dfCarsC.loc[:, 'registrations'] > 4000) & (dfCarsC.loc[:, 'registrations'] <= 4200))
        )
        & (dfCarsC.loc[:, 'countyname'] == 'Wright'), 'countyname'])
```

The above will return a Series only containing the `countyname` while the below will be a DataFrame with `countyname` and the number of registrations:

```
>>> dfCarsC = dfCars.copy()

>>> dfCarsC = dfCarsC.loc[(
        (dfCarsC.loc[:, 'registrations'] > 1200) & (dfCarsC.loc[:, 'registrations'] <= 3000)
        |
        (dfCarsC.loc[:, 'registrations'] > 4000) & (dfCarsC.loc[:, 'registrations'] <= 4200)
        )
        & (dfCarsC.loc[:, 'countyname'] == 'Wright'), ['countyname','registrations']]
```

# 4 Column Types Conversion

- We occasionally need to switch types on columns. To do this we use the `astype` method. Consider the following example below:

```
>>> dfCars.loc[:, 'registrations'].dtype
int64

>>> dfCars.loc[:, 'registrations'].astype(str).dtype
object

>>> dfCars.loc[:, 'registrations'].astype(float).dtype
float64
```

- The `astype` is most commonly used when converting between integers, floats and strings. While there is some ability for the `astype` method to be used when dealing with dates and date related objected it is not recommend and there are better, most consistent options available.

# 5 Dealing with NaN

- Consider the following code:

```
>>> dfCarsC = dfCars.copy()

>>> dfCars217 = dfCarsC.loc[(dfCarsC.loc[:, 'registrations'] > 217000)
    ,['year', 'registrations', 'annualfee']]

>>> print(dfCars217)
       year  registrations   annualfee
2213   2016         217540  32035419.0
2372   2006         218883         NaN
12890  2015         218975  32058351.0
15971  2014         218211  31790136.0
21352  2005         218235         NaN
25896  2008         217073  24160802.0
```

We can see that in this DataFrame we have encountered a number of NaN values which is how missing values are identified in Pandas. To locate these types of values we *do not* use equality operators, but instead the Series method `isna` as shown below:

```
>>> dfCars217.loc[:, 'annualfee'].isna()
2213     False
2372      True
12890    False
15971    False
21352     True
25896    False
Name: annualfee, dtype: bool

>>> dfCars217.loc[:, 'annualfee'] > 1
2213      True
2372     False
12890     True
15971     True
21352    False
25896     True
Name: annualfee, dtype: bool

>>> dfCars217.loc[:, 'annualfee'] < 1
2213     False
2372     False
12890    False
15971    False
21352    False
25896    False
Name: annualfee, dtype: bool
```

- isna can also be applied to entire DataFrames:

```
>>> dfCars.isna()
  year    countyname    motorvehicle    vehiclecat    vehicletype    tonnage    r [...]
------  ------------  --------------  ------------  -------------  ---------  --- [...]
     0             0               0             0              0          1      [...]
     0             0               0             0              0          1      [...]
     0             0               0             0              0          0      [...]
     0             0               0             0              0          1      [...]
     0             0               0             0              0          0      [...]
[...]
```

As with SQL, NaN values evaluate false when given mathematical comparisons. In the example above the NaN values were false for both less-than and greater-than one.

- To identify rows which are not NaN we can use a "not" operator in the following manner:

```
>>> dfCars217.loc[ ~(dfCars217.loc[:, 'annualfee'].isna()), :]
       year  registrations   annualfee
2213   2016         217540  32035419.0
12890  2015         218975  32058351.0
15971  2014         218211  31790136.0
25896  2008         217073  24160802.0
```

- **There is no NaN value for integers**. Any operation on an integer column which generates a NaN

will automatically turn it into a float.

```
>>> x_1
0    1
1    2
2    3
3    4
Name: v1, dtype: int64

>>> x_1.dtypes
int64

>>> x_1[3] = np.nan

>>> print(x_1)
0    1.0
1    2.0
2    3.0
3    NaN
Name: v1, dtype: float64

>>> x_1.dtypes
```

- For NaN values we can use `fillna` in order to replace those values with another:

```
>>> dfCars.loc[:, 'countyname'].fillna('No County')
```

`fillna` can also be used with two columns.

```
>>> dfCars.loc[:, 'tonnage'].fillna(dfCars.loc[:, 'vehiclecat'])
tonnage
---------
Bus
Moped
3 Tons
Trailer
3 Tons
[...]
```

# 6 Choosing the largest and smallest values

- In the next module we will talk about sorting DataFrames, but for now we are going to cover how to return only certain rows of a DataFrame based on their order. Aay that we want to return the largest annualfee value in Scott county. To do this we use the `nlargest` method which takes two input variables: the number to return and the *series* to sort by.

```
>>> dfCarsC = dfCars.copy()

>>> (dfCarsC.loc[(dfCarsC.loc[:, 'countyname'] == 'Scott'), :]
    .nlargest(1, 'annualfee').loc[:, 'annualfee'])
33497    19235858.0
Name: annualfee, dtype: float64
```

We can do the same thing with smallest values:

```
>>> dfCarsC = dfCars.copy()

>>> (dfCarsC.loc[(dfCarsC.loc[:, 'countyname'] == 'Scott'), :]
    .nsmallest(1, 'annualfee').loc[:, 'annualfee'])
2595    0.0
Name: annualfee, dtype: float64
```

Two important features of the above:

1. The column `'annualfee'` has NaN values in it, but those values were *ignored* in both the smallest and largest operator.
2. We were able to append `.loc[:, 'annualfee']` to the end of the statement in both cases *after* the largest/smallest method. How could we do that? Once again, it has to do with understanding what is being returned. In this case, the method returns a DataFrame which we can then apply any of our selection methods.

# 7 Manipulating Data & Method Chaining

- So far we have only taken data, filtered rows and selected columns. In this section we are going to do some basic manipulation of the underlying DataFrame.
- Let's start by creating a column, which we can do by using some of the previous selection operators, but this time with a new name:

```
>>> dfCarsC = dfCars.copy()

>>> dfCarsC.loc[:, 'newcol1'] = 5

>>> dfCarsC.loc[:, 'newcol1'].head()
0    5
1    5
2    5
3    5
4    5
Name: newcol1, dtype: int64
```

- Mathematical operations work on a row-by-row basis:

DRAFT

```
>>> dfCarsC.loc[:, 'newcol2'] = dfCarsC.loc[:, 'registrations'] + 5

>>> dfCarsC.loc[:, ['registrations', 'newcol2']].head()
   registrations  newcol2
0              5       10
1            198      203
2           5020     5025
3            366      371
4           2507     2512
```

- Creating columns based on other columns (of the same shape) is also straightforward:

```
>>> dfCarsC.loc[:, 'newcol3'] = dfCarsC.loc[ :, 'registrations'] / dfCarsC.loc[:, 'annualfee']

>>> dfCarsC.loc[:, 'newcol4'] = dfCarsC.loc[:, 'registrations'] * dfCarsC.loc[:, 'annualfee']

>>> dfCarsC.loc[~(dfCarsC.annualfee.isna())
    ,['registrations', 'newcol3', 'newcol4',  'annualfee']].head()
   registrations   newcol3       newcol4  annualfee
0              5  0.007353  3.400000e+03      680.0
1            198  0.142857  2.744280e+05     1386.0
2           5020  0.016201  1.555457e+09   309852.0
3            366  0.019877  6.739158e+06    18413.0
4           2507  0.018752  3.351608e+08   133690.0
```

- Lets take a look at that special case where we divide by zero:

```
>>> dfCarsC = dfCars.copy()

>>> dfCarsC = dfCarsC.loc[(dfCars.loc[:, 'annualfee'] == 0), ['annualfee', 'registrations']].head()

>>> dfCarsC.loc[:, 'registrations'] / dfCarsC.loc[:, 'annualfee']
2365    inf
2595    inf
2819    inf
2842    inf
3087    inf
dtype: float64
```

We can see that no error was generated *but* a new value `np.inf` was used. Unlike missing values, standard operators can be used on this special value:

```
>>> np.NaN == np.NaN
False

>>> np.inf == np.inf
True

>>> np.inf > 1
True

>>> np.inf < 1
False
```

- The other way to create a new column is by using the `assign` method. Using the `assign` method, rather than the `loc` syntax above allows us to "method chain", because it returns a DataFrame

which has all DataFrame methods available to it:

```
>>> dfCarsC = dfCars.copy()

>>> dfCarsC = (dfCarsC
    .assign(newcol3=dfCarsC.loc[:, 'registrations']/dfCarsC.loc[:, 'annualfee']))
```

In the example above we use the assign operator to create a new column which is the registrations divided by the annual fee. This is exactly the same result as previously, but this time we applied a more functional method to complete this task.

- The `assign` method also works when setting static values, such as in the following example, which creates a column of zeros. It can also be used with multiple column assignments, as shown below, by separating them with commas.

```
>>> dfCarsC = dfCars.copy()

>>> dfCarsC = (dfCarsC
    .assign(newcol3=dfCarsC.loc[:, 'registrations']/dfCarsC.loc[:, 'annualfee'], newcol4 = 0)
    .nlargest(5, 'newcol3')
    )
```

- In general we will try to use method chaining methods to manipulate our data because it leads to much more readable code.

## Dropping, Renaming and Inplace methods

- We can also drop, or remove, columns from a DataFrame using the `drop` method on a DataFrame. There are two common ways of using this method, both shown below:

```
>>> dfCarsC = (dfCars
                .copy()
                .assign(newcol1 = 1, newcol2=2, newcol3=3, newcol4=4)
                )

>>> dfCarsC = dfCarsC.drop(['newcol1', 'newcol2'], axis=1)

>>> dfCarsC = dfCarsC.drop(columns=['newcol3', 'newcol4'])

>>> dfCarsC.columns
Index(['year', 'countyname', 'motorvehicle', 'vehiclecat', 'vehicletype',
       'tonnage', 'registrations', 'annualfee', 'completecategory'],
      dtype='object')
```

The first is to provide a list and se the `axis` option to 1 lets Pandas know that we are dropping columns and not indexed rows. The `axis` variable is used to specify if an operation should be done along rows or columns.[6]

The second way of using this method is by specifying the named parameter `columns` and providing a list of strings.

The `drop` method returns a DataFrame with the chosen objects removed and does not modify the

[6] While it seems obvious that we are trying to drop columns in the example, it's just as possible that we have an index and we could be trying to drop rows instead. This variable allows us to be certain which we are doing.

current DataFrame. If we wish to change the current DataFrame (and return `None`) then we use the `inplace` option to the method:

```
>>> dfCarsC = (dfCars
                .copy()
                .assign(newcol1 = 1, newcol2=2, newcol3=3, newcol4=4)
                )

>>> dfCarsC.drop(['newcol1', 'newcol2'], axis=1, inplace=True)

>>> dfCarsC.drop(columns=['newcol3', 'newcol4'], inplace=True)

>>> dfCarsC.columns
Index(['year', 'countyname', 'motorvehicle', 'vehiclecat', 'vehicletype',
       'tonnage', 'registrations', 'annualfee', 'completecategory'],
      dtype='object')
```

- DataFrame methods commonly have an `inplace` option, which can be helpful to use and easier to read. Be wary however, as it is easy to run into "weird" errors when mixing both `inplace` and not `inplace` commands. In my experience you should either always use `inplace` or never. Mixing them causes problems.
- There are two common ways to rename columns. We can use the `rename` method with the `column` parameter set or we can reassign the labels by setting the `columns` attribute on the DataFrame object.
- The second, setting the `columns` attribute, works because the attribute itself is accessible and while it is shown as an *index* type, it can be set by using any itterable object, such as a list.

```
>>> dfCarsC = dfCars.copy()

>>> dfCarsC.columns
Index(['year', 'countyname', 'motorvehicle', 'vehiclecat', 'vehicletype',
       'tonnage', 'registrations', 'annualfee', 'completecategory'],
      dtype='object')
```

- The columns attribute itself is also an *index* type, but it can be overwritten with any list type to rename the columns. For example:

```
>>> dfCarsC = dfCars.copy()

>>> dfCarsC.columns = ['year2', 'countyname', 'motorvehicle', 'vehiclecat', 'vehicletype',
       'tonnage', 'registrations', 'annualfee', 'completecategory']

>>> dfCarsC.loc[:, 'year2'].head()
  year2
-------
   2008
   2011
   2012
   2015
   2016
```

The column "year" was renamed by changing the name in the list. The object associated with `columns` is *not* mutable and thus must be overwritten and not modified.

DRAFT

- The other way to rename a column is via the `rename` method. `rename` takes a parameter named `column` which we set equal to a dictionary of the form `{old_column : new_column}`

```
>>> dfCarsC = dfCars.copy()

>>> dfCarsC = dfCarsC.rename(columns={'tonnage' : 'tonnage2'})
```

One truly annoying thing about the Pandas rename function is that if you forget the `columns=` parameter, then no error is returned and no column is renamed! In this situation, the command looks for an index value to rename and doesn't find it, so no rename occurs. In other words, the `columns` parameter works the same way that the `axis` parameter did in other commands.

Like many of the other methods, this one also has an "inplace" operator, so the following is equivalent to the previous operation:

```
>>> dfCarsC = dfCars.copy()

>>> dfCarsC.rename(columns={'tonnage' : 'tonnage2'}, inplace=True)
```

- We can use dictionary notation to do multiple renames at the same time using this method:

```
>>> dfCarsC = dfCars.copy()

>>> dfCarsC = dfCarsC.rename(columns={'year' : 'year2', 'tonnage' : 'tonnage2'})
```

# 8 Indexes: Creating and Dropping

- So far our discussion has avoided any mention of how indexes are used in Pandas. In this section we discuss some of the basic use of indexes on the row-level.
- The most common type of index is a `RangeIndex`, which is a simple integer increment. This is the default index set upon loading a dataset:

```
>>> dfCars.index
RangeIndex(start=0, stop=41202, step=1)
```

- To convert a column into an index we use the `set_index` method which takes in a list of columns and creates an index based upon it. For example, the following returns a dataset with an index associated with the "countyname" (which has many repeating values):

```
>>> dfCarsC = dfCars.set_index('countyname')
```

As with many statements in pandas you can have the effect occur in place, rather than returning a new value:

```
>>> dfCarsC = dfCars.copy()

>>> dfCarsC.set_index('countyname', inplace=True)
```

- Index values need not be unique in pandas!

DRAFT

- Indexes can also be multi-level (or hierarchal). We will talk more about this in 4. To add a multi-level index to a DataFrame we supply a list to the `set_index()` method:

```
>>> dfCarsC = dfCars.copy()

>>> dfCarsC = dfCarsC.set_index(['countyname', 'year'])

>>> ans = dfCarsC.head()
                     motorvehicle    vehiclecat    vehicletype     tonnage      r [...]
-------------------  --------------  ------------  --------------  ---------  --- [...]
('Ida', 2008)        Yes             Bus           Bus                            [...]
('Jasper', 2011)     Yes             Moped         Moped                          [...]
('Harrison', 2012)   Yes             Truck         Truck           3 Tons         [...]
('Palo Alto', 2015)  No              Trailer       Travel Trailer                 [...]
('Adair', 2016)      Yes             Truck         Truck           3 Tons         [...]
```

- Multi-level index are represented (when printing) as tuples. We also use tuple-like notation to access them, as we will discuss later.
- We can also remove the index (in that we return it to a column value) by using the `reset_index` method:

```
>>> dfCars.reset_index()
```

When using this method, the column name of the former index is equal to the name of the index (e.g. what can be found in `.index.name` on the DataFrame.)

- If we want to reset the index and then not have the index returned as a column, we use the `drop` parameter:

```
>>> dfCarsC = dfCars.copy()

>>> dfCarsC = (dfCarsC
         .set_index(['countyname', 'year'])
         .reset_index(drop=True)
         )

>>> dfCarsC.dtypes
motorvehicle         object
vehiclecat           object
vehicletype          object
tonnage              object
registrations         int64
annualfee           float64
completecategory     object
dtype: object
```

In the above example, the original two columns (countyname and year) which we turned into the index no longer appear.

- One caveat is that even if you use `reset_index`, this doesn't create a DataFrame without an index. Instead, a standard range index will be created with each row having an index equal to the current row number.
- Finally, we can assign index values by simply assigning them using an object of equal shape.

```
>>> t = dfCars.head()

>>> t.index = np.arange(0, 2*t.shape[0],2)

>>> t.index
Index([0, 2, 4, 6, 8], dtype='int64')
```

# 9 Views and Copies

- The second most common problem that people have (after not knowing what is being returned) is failing to be explicit about creating copies of the data and instead operate on a view.
- What do I mean by "view" vs. "copy"? A "view", sometimes called a "shallow copy" is when we create an object based on another object by creating a pointer to a memory space, rather than creating an explicit copy of that data in memory.
- Why would we do this? Namely speed and memory. A "shallow copy", sometimes called a "zero copy" is much, much faster and places less of a burden on the CPU.
- The downside of this is that it places a burden on the user to understand when a copy or a view is created. Unfortunately in pandas, those rules are not as explicit as they should be and it is common for "weird" behavior to occur because of this.
- We are going to look at five examples of how this behavior can arise.

  1. **Simple View:** In the following example we can see that pandas has created a view of the underlying object.

     ```
     >>> df = pd.DataFrame({"a": np.arange(2), "b": np.arange(2)})

     >>> print(df)
        a  b
     0  0  0
     1  1  1

     >>> df2 = df

     >>> df3 = df.iloc[0:1, 0:1]

     >>> df.loc[:, 'a'] = 5

     >>> print(df2)
        a  b
     0  5  0
     1  5  1

     >>> print(df3)
        a
     0  5
     ```

  2. **Simple Copy:** In the following example we can see that a copy has been created. We use the `.copy()` method in order to explicitly make a copy of the DataFrame.

DRAFT

```
>>> df = pd.DataFrame({"a": np.arange(2), "b": np.arange(2)})

>>> print(df)
   a  b
0  0  0
1  1  1

>>> df2 = df.copy()

>>> df.loc[:, 'a'] = 5

>>> print(df2)
   a  b
0  0  0
1  1  1
```

3. **Wut? (pt. 1):**

```
>>> df = pd.DataFrame({"a": np.arange(2), "b": np.arange(2)})

>>> print(df)
   a  b
0  0  0
1  1  1

>>> df2 = df

>>> df3 = df.iloc[0:1, 0:1]

>>> df.loc[:, 'a'] = 5.1

>>> print(df2)
     a  b
0  5.1  0
1  5.1  1

>>> print(df3)
   a
0  0
```

4. **Wut? (pt. 2):**

```python
df = pd.DataFrame({"a": np.arange(2), "b": np.arange(2)})
print(df, '\n'*2)

column_a = df.loc[:, "a"]

df.iloc[0, 0] = 5
# column_a is a view so it keeps the 5
print(column_a, '\n'*2)

# Lets change column b in the original df
df.loc[:, "b"] = "new entry"
print(df, '\n'*2)

# Now lets change column A in the df
df.iloc[0, 0] = 10
print(df, '\n'*2)

# no longer a view!
print(column_a)
```

```
   a  b
0  0  0
1  1  1


0    5
1    1
Name: a, dtype: int64


   a          b
0  5  new entry
1  1  new entry


    a          b
0  10  new entry
1   1  new entry


0    5
1    1
Name: a, dtype: int64
```

5. **Wut? (pt.3):** In this example you will get a `SettingWithCopyWarning`. This is called "index chaining" and is one of the reasons that we require use `loc` in this course. It is way too easy to end up in situations where this occurs if you use non-`loc` based access methods.

```
>>> df = pd.DataFrame({"a": np.arange(2), "b": np.arange(2)})

>>> df[df.a == 0]['b'] = 100

>>> print(df)
   a  b
0  0  0
1  1  1

>>> df.loc[ (df.loc[:, 'a'] == 0), 'b']= 100

>>> print(df)
   a    b
0  0  100
1  1    1
```

- What have we learned from the above? You have to be careful with views vs. copies. Assuming one or the other can lead to significant problems and unexpected behavior.
- Use `loc` as a way to access data. It's more likely to put you on a path toward a copy.
- The method `_is_view` should return a boolean on if the object is a view or not. I've had varying degrees of luck with it and don't recommend relying on it.
- Use `copy` frequently when you are not intentionally maintaining a view – and maybe most importantly, never assume what you are working on.

DRAFT

DRAFT