Chapter 15

More Manipulations and Types

Contents

1 Sorting DataFrames

- If we want to sort a DataFrame or Series then we use the `sort_values` method, which returns the same object with the values sorted.
- This method returns a DataFrame or Series in-which the data is sorted.
- If you are sorting based on a single column you can pass the column name in directly:

```
>>> dfCars.sort_values('annualfee')
```

  If you wish to sort based on multiple columns, you pass them in as a list:

```
>>> dfCars.sort_values(['countyname', 'annualfee'])
```

- The default sort order is ascending (lowest to highest), but we can change that using the "ascending" parameter. This needs to be a boolean (or list of booleans) with a length equal to the number of columns being sorted. Two examples below:

```
>>> dfCars.sort_values(['countyname', 'annualfee'], ascending = [False, True])

>>> dfCars.sort_values('countyname', ascending = False)
```

- The above two queries *do not modify the original DataFrame*, but only return a sorted version. If we wish to modify the original DataFrame we have to either use another assignment operator or use the `inplace` argument:

```
>>> d_1 = dfCars.loc[:,['annualfee', 'registrations']]

>>> d_1.head()
   annualfee  registrations
0      680.0              5
1     1386.0            198
2   309852.0           5020
3    18413.0            366
4   133690.0           2507

>>> d_1.sort_values('annualfee').head()
       annualfee  registrations
37907        0.0              1
34713        0.0              1
4454         0.0              1
38246        0.0              3
30700        0.0              1

>>> d_1.head()
   annualfee  registrations
0      680.0              5
1     1386.0            198
2   309852.0           5020
3    18413.0            366
4   133690.0           2507

>>> d_1.sort_values('annualfee', inplace=True)

>>> d_1.head()
       annualfee  registrations
37907        0.0              1
34713        0.0              1
4454         0.0              1
38246        0.0              3
30700        0.0              1
```

- Before proceeding, take a look at the index numbers on the rows above. The first `d_1.head()` call returns an index of 0 through 4. The next two, however, return an index of the numbers that mapped to the original RangeIndex row location (37907, 34713, ...). This means that the index on the DataFrame returned is no longer in order. If we wish to reorder the index so that it matches the actual row number we will need to change the index, as well will see below.
- Note that the above has important implications for how operations work in pandas. Specifically, operations follow the index – not the row order! This is true *even if we do not explicitly specify an index*.

```
>>> d_1 = pd.DataFrame( [1,2,3], columns=['a'])

>>> d_2 = d_1.sort_values(['a'], ascending=False).copy()

>>> d_1
  a
---
  1
  2
  3

>>> d_2
  a
---
  3
  2
  1

>>> d_1 + d_2
  a
---
  2
  4
  6
```

The important feature of pandas to keep in mind is that all operations are index-based, unless specified otherwise.

- In my experience it is most common to want to sort by the internal values of the data which are not an index. If, however, we wish to sort based on the index values, then we can use the `sort_index` command which will return a DataFrame or Series sorted by the index:

```
>>> d_1 = pd.DataFrame({'A' : [1,2,3], 'B' : [3,2,1]})

>>> d_1 = d_1.sort_values(['B'], ascending = True)

>>> d_1
  A    B
---  ---
  3    1
  2    2
  1    3

>>> d_1.sort_index()
  A    B
---  ---
  1    3
  2    2
  3    1
```

You can see that the DataFrame which was returned by the `sort_index` has now been sorted along

that dimension. This command also has an `inplace` argument.

- How are Nulls handled? Unlike SQL which treats Nulls as the largest value and last alphabetically, pandas treats Nulls as separate condition. Specifically, Nulls are always put last. If you want to change this behavior there is a named argument `na_position` which can be set to either `first` or `last` which determines the position of Nulls in the sort order.

```
>>> dfCars.sort_values(['annualfee'], na_position='first').loc[:, 'annualfee'].head()
11   NaN
14   NaN
22   NaN
23   NaN
44   NaN
Name: annualfee, dtype: float64

>>> dfCars.sort_values(['annualfee'], na_position='last').loc[:, 'annualfee'].head()
37907    0.0
34713    0.0
4454     0.0
38246    0.0
30700    0.0
Name: annualfee, dtype: float64
```

2 Dealing with Duplicates

- A frequent work flow when duplicate values occur within a dataset is removing them. Pandas provides a few useful operations in order to do this: `unique`, `drop_duplicates` and `duplicated`.
- The first of these, `unique` works on a Series and returns the unique values within that Series:

```
>>> dfCars.loc[:, 'vehiclecat'].unique()
['Bus' 'Moped' 'Truck' 'Trailer' 'Multi-purpose' 'Motor Home' 'Automobile'
 'Motorcycle' 'Autocycle']
```

  This operation returns an `array`, not a Series, so be careful!

- The second operation, `drop_duplicates`, works on both Series and DataFrames and returns the same object that it is called upon. For example, the following operation does the same thing as the previous code snippet, but this time returns a Series:

```
>>> dfCars.loc[:, 'vehiclecat'].drop_duplicates()
0                 Bus
1               Moped
2               Truck
3             Trailer
13     Multi-purpose
22         Motor Home
50         Automobile
81         Motorcycle
411         Autocycle
Name: vehiclecat, dtype: object
```

- This operation can also be done on entire DataFrames:

```
>>> dfCars.loc[:, ['vehiclecat', 'year']].drop_duplicates().head()
  vehiclecat  year
0        Bus  2008
1      Moped  2011
2      Truck  2012
3    Trailer  2015
4      Truck  2016
```

This example returns a DataFrame with only 142 rows, which contains the unique values of the two specified columns. As in SQL, this does *not* create data. If there is combination of data points which is not present within the original data that combination will not appear in the result.

- One common operation we want to handle is removing duplicates from a DataFrame based on a subset of the columns in the DataFrame. We can do this using the `drop_duplicates` method with the `subset`argument set to a list of the columns we want to return as unique.

- Dropping duplicates in this matter raises the issue of which, non-de-duplicated rows do we want to keep? Specifically consider the following example below:

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

>>> dfCarsC = ( dfCarsC
 .loc[(dfCarsC.loc[:, 'countyname'] == 'Polk')
      & (dfCarsC.loc[:, 'completecategory'] == 'Motor Home - A')
      , :]
 .sort_values('year')
)

>>> # SPOT

>>> dfCarsC.drop_duplicates(subset='countyname')
  year  countyname    motorvehicle    vehiclecat    vehicletype     tonnage       [...]
------  ------------  --------------  ------------  --------------  ---------  -- [...]
  2005  Polk          Yes             Motor Home    Motor Home - A                [...]

>>> dfCarsC = dfCarsC.sort_values('year', ascending=False)

>>> dfCarsC.drop_duplicates(subset='countyname')
  year  countyname    motorvehicle    vehiclecat    vehicletype     tonnage       [...]
------  ------------  --------------  ------------  --------------  ---------  -- [...]
  2021  Polk          Yes             Motor Home    Motor Home - A                [...]
```

- We can see that in the above example we use the `drop_duplicates` method and, in the first example it returns the row from 2005 while the second returns the row from 2021. If we were doing an analysis where we wanted the result of this command than we have cognizant of the current row order since this method, by default, keeps the *first row* that it encounters within that DataFrame.

- Looking at that example, consider the spot where there is a comment in the code above. If someone later came and did something which switched the sort order in the DataFrame then the analysis could be changed without registering an error – subsequent analysis would therefore be based on a different DataFrame and return a different number.

- Because of this, I strongly recommend (nay, require), all `drop_duplicates` in code bases I oversee to have their sort order fully specified with method chaining, such as in the below.

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

>>> dfCarsC = ( dfCarsC
 .loc[(dfCarsC.loc[:, 'countyname'] == 'Polk')
      & (dfCarsC.loc[:, 'completecategory'] == 'Motor Home - A')
      , :]
 .sort_values('year')
)

>>> (dfCarsC
     .sort_values(['countyname', 'year'], ascending=False)
     .drop_duplicates(subset=['countyname'], keep='first')
     )
  year  countyname    motorvehicle    vehiclecat    vehicletype     tonnage       [...]
------  ------------  --------------  ------------  --------------  ---------  -- [...]
  2021  Polk          Yes             Motor Home    Motor Home - A                [...]
```

- In the above I have also added the `keep` argument to the command. I do this because, while this is the default behavior, I can't (and don't expect) others to remember what is the default. Putting it in works as a reminder.
- A final useful command is `duplicated` which returns a boolean series which identifies if a particular row is a duplicate or not. Just like the `drop_duplicates` method it can take both a subset of columns as well as a `keep` named argument to guide which rows are deemed a duplicate or not.

3 Using Type specific functions

- As can be expected with any data focused library, Pandas has a large number of useful *type specific functions*. Type specific functions are functions which are only allowed to operate on data types which have a particular type. To reach these methods in Pandas we use a set of *Series* methods as accessor attributes.
- In this section we will consider two sets of type specific functions: those that work on Dates and those that work on Strings.

3.1 Dates

- Date and date related objects are built from the standard Python `datatime` library.
- There are two commonly used data types:
  1. The `datetime64` type which represents a discrete date and time.
  2. The `timedelta` type which represents an interval.
- There are four common operations we want to do with dates:
  1. **Convert string to datetime:** The two most common ways of doing this:
     (a) **Upon Loading:** When the data is being loaded, there are ways to tell Pandas that a particular column is a date. When loading CSV files, the `read_csv` method has an argument called `parse_dates` which accepts a list of columns which are converted upon loading. Note that this method tries to infer the date from the data present and frequently fails.
     (b) **Via `to_datetime`:** Using this method entails either replacing a column or assigning a new column using this built-in Pandas method:

```
>>> dfMTA.loc[:, 'mtadt'] = pd.to_datetime( dfMTA.loc[:, 'mtadt'], format='%m/%d/%Y')
```

Both the `to_datetime` method as well as a number of other date methods require the user to specify the explicit format of the date string to be processed using the *strftime/strptime* formatting system. This system is a common method of defining the date type in data processing and is based upon an older unix/C standard.[1] The function strftime function takes a date object and returns a string while the strptime takes a string and returns a date object.

The format uses a set of conversion characters, which begin with a % to represent parts of a date. These conversion characters and then combined with ordinary characters (everything else) to combine to specify the final date format. For example, consider the following specification: `%Y-%m-%d` which would represent the year, month and date, separated by dashes. You can see further examples in Table 15.1 below.

Syntax	Example	Description
`%Y-%m-%d`	2022-02-02	Four-digit year and zero padded month and day
`%m/%d/%Y %H:%M:%S`	03/02/2023 14:55:22	M/D/Y (all zero padded) and hour/min/sec with a 24 hr clock

Table 15.1: strftime examples

The Python documentation has a full list of acceptable codes. That documentation can be found here: `https://docs.python.org/3/library/datetime.html#strftime-and-strptime-for`

2. **Extract a date component:** Once you have a datetime object, you can use the `db` accessor to extract a portion of the date from it. In this case we use the `dt` followed by what we are trying to extract:

```
>>> dfMTA.loc[:, 'mtadt'].dt.year.head()
0    2015
1    2015
2    2015
3    2015
4    2015
Name: mtadt, dtype: int32
```

Options for this include things like `year, month, day` and `dayofweek`.

3. **Convert datetime to string:** As discussed above we can use the `strftime` style formatting with the `dt` accessor function. For example:

```
>>> dfMTA.loc[:, 'mtadt'].dt.strftime('%Y-%m').head()
0    2015-11
1    2015-11
2    2015-11
3    2015-11
4    2015-11
Name: mtadt, dtype: object
```

[1] You can find a link to the man page here: `https://man7.org/linux/man-pages/man3/strftime.3.html`

4. **Basic date math and comparisons:** To add and subtract times we use a `Timedelta` object, which represents a length of time to be applied.

```
>>> (dfMTA.loc[:, 'mtadt'] + pd.Timedelta(days=2)).head()
0   2015-11-30
1   2015-11-30
2   2015-11-30
3   2015-11-30
4   2015-11-30
Name: mtadt, dtype: datetime64[ns]
```

`Timedelta` objects allow us to consistently do math on datetime objects, so use these rather than relying on other methods. Comparison operators $(=, >, <)$ work as expected:

```
>>> dfMTA.loc[(dfMTA.loc[:, 'mtadt'] > '2015-01-01')].head()
   plaza      mtadt  hr direction  vehiclesez  vehiclescash
0      1 2015-11-28   0         I         477           205
1      1 2015-11-28   0         O         486           252
2      1 2015-11-28   1         I         350           171
3      1 2015-11-28   1         O         307           182
4      1 2015-11-28   2         I         280           133
```

- Unlike in SQL there is no obvious `date_trunc` method. While one might expect the `floor` command to do this, there is an open issue on pandas:

  Pandas is broken: `https://github.com/pandas-dev/pandas/issues/15303#`

- The current date and time can be found with the following commands:

```
>>> pd.to_datetime('now')
2023-08-14 20:50:19.553795

>>> pd.to_datetime('today')
2023-08-14 20:50:19.553917
```

These commands return the current date and time at the time that the code is run. This is useful when trying to write code which would analyze a rolling time frame, such as "the last 90 days."

3.2 Strings

- To reference string functions in Pandas, we call the `str` accessor on a column and then reference the string function.
- These methods exclude NaN values and usually match standard string methods in Python and Excel. Table 15.2 contains a list of some useful functions that can be found in Pandas.
- Consider the following example, which uppercases the countyname:

Name	Description
lower / upper	Lower or Upper Cases a string
len	Returns the length of the string
strip	Removes white spaces and new line characters from a string
split	Splits a string into a list from a given pattern
startswith / endswith	Returns a boolean based on if the string follows the pattern
contains	Returns a boolean if the string contains a pattern

Table 15.2: String Functions in Pandas

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

>>> dfCarsC.loc[:, 'UC'] = dfCarsC.loc[:, 'countyname'].str.upper()

>>> dfCarsC.loc[:, ['countyname', 'UC']].head()
  countyname         UC
0        Ida        IDA
1     Jasper     JASPER
2   Harrison   HARRISON
3  Palo Alto  PALO ALTO
4      Adair      ADAIR
```

In this example, the countyname is referenced as a series and then the `str` accessor is called. After the `str` accessor is called then the string function `upper` is called. This returns the uppercased value.

- If we want to refer to specific portions of a string (such as the first or last character), we use the `str` accessor and then apply our normal slicing operations afterwards. For example, to get the first two characters of a string we could do the following:

```
>>> dfCars.loc[:, 'countyname'].str[0:2].head()
0    Id
1    Ja
2    Ha
3    Pa
4    Ad
Name: countyname, dtype: object
```

As with the rest of Python, strings start with zero and a slice is inclusive on the left side, but not on the right. The object returned is another Series.

- To concat strings together we use the `cat` method within the `str` accessor or a `+` :

```
>>> dfCars.loc[:, 'countyname'].str.cat(dfCars.loc[:, 'countyname'])
0                     IdaIda
1               JasperJasper
2           HarrisonHarrison
3         Palo AltoPalo Alto
4                 AdairAdair
                    ...
41197           MarionMarion
41198             WorthWorth
41199     WinnebagoWinnebago
41200       DelawareDelaware
41201       HumboldtHumboldt
Name: countyname, Length: 41202, dtype: object

>>> dfCars.loc[:, 'countyname'] + dfCars.loc[:, 'countyname']
0                     IdaIda
1               JasperJasper
2           HarrisonHarrison
3         Palo AltoPalo Alto
4                 AdairAdair
                    ...
41197           MarionMarion
41198             WorthWorth
41199     WinnebagoWinnebago
41200       DelawareDelaware
41201       HumboldtHumboldt
Name: countyname, Length: 41202, dtype: object
```

- Using the `split` function provides us an interesting application of the object data type. Let's look at the top of the "vehicletype" column in the DataFrame:

```
>>> dfCars.loc[:, 'vehicletype'].head(10)
0               Bus
1             Moped
2             Truck
3    Travel Trailer
4             Truck
5             Truck
6             Truck
7             Truck
8             Moped
9             Truck
Name: vehicletype, dtype: object
```

As can be seen in the result, there are multiple different types of vehicles, some with one word and some with multiple words. If we use the `split` method on this, we will create a list:

```
>>> dfCars.loc[:, 'vehicletype'].str.split(' ').head(10)
0                 [Bus]
1              [Moped]
2              [Truck]
3    [Travel, Trailer]
4              [Truck]
5              [Truck]
6              [Truck]
7              [Truck]
8              [Moped]
9              [Truck]
Name: vehicletype, dtype: object
```

Each of the phrases in the original dataset has been turned into a list – and the lists do not have the same number of items! Surprisingly, we can store this in the DataFrame itself and the series will have type "object", though the contents will be a list!

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

>>> dfCarsC = dfCarsC.assign(newcol = dfCars.loc[:, 'vehicletype'].str.split(' '))

>>> dfCarsC.loc[:, 'newcol'].dtypes
object

>>> type( dfCarsC.loc[:, 'newcol'].iloc[0] )
<class 'list'>
```

- Since the string functions themselves are accessible from any string series, they can be chained together to generate more complex operations:

```
>>> dfCars.loc[:, 'countyname'].str.upper().str.startswith('A')
0        False
1        False
2        False
3        False
4         True
         ...
41197    False
41198    False
41199    False
41200    False
41201    False
Name: countyname, Length: 41202, dtype: bool
```

- We can also use them to locate information via the `loc` function. In the example below we added a `fillna` command since there are some NaNs in the underlying data and Pandas doesn't allow `loc` indexing with NaN's present!

```
>>> dfCars.loc[(dfCars.loc[:, 'tonnage'].str.contains('Tons')).fillna(False), :]
  year  countyname    motorvehicle    vehiclecat    vehicletype    tonnage      r [...]
------  ------------  --------------  ------------  -------------  ---------  --- [...]
  2012  Harrison      Yes             Truck         Truck          3 Tons           [...]
  2016  Adair         Yes             Truck         Truck          3 Tons           [...]
  2016  Van Buren     Yes             Truck         Truck          4 Tons           [...]
  2018  Story         Yes             Truck         Truck          3 Tons           [...]
  2019  Cerro Gordo   Yes             Truck         Truck          5 Tons           [...]
[...]
```

Specifically, if we try the command above without the `fillna` present the result will be:

```
ValueError: cannot index with vector containing NA / NaN values
```

- Finally, the `str` accessory function returns a string, so we can use standard string slicing functions to manipulate strings.

```
>>> dfCars.loc[:, 'countyname'].str[0:3].str.upper().head()
0    IDA
1    JAS
2    HAR
3    PAL
4    ADA
Name: countyname, dtype: object
```

4 CASE style statements and the "isin" operator

- To mimic SQL style CASE statements, the `loc` operator can be used.
- For example, lets say that we wish to create a column ("regsize") which is equal to 'Small', 'Medium' or 'Large', depending on if the number of registrations is less than 200, between 200 and 500 and more than 500. One way to accomplishing this:

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

>>> dfCarsC.loc[(dfCarsC.loc[:, 'registrations'] < 200), 'regsize'] = 'Small'

>>> dfCarsC.loc[(dfCarsC.loc[:, 'registrations'] >= 200)
     & (dfCarsC.loc[:, 'registrations'] < 500), 'regsize'] = 'Medium'

>>> dfCarsC.loc[(dfCarsC.loc[:, 'registrations'] >= 500), 'regsize' ] = 'Large'

>>> dfCarsC.loc[:, ['registrations', 'regsize']].head()
   registrations regsize
0              5   Small
1            198   Small
2           5020   Large
3            366  Medium
4           2507   Large
```

- The `loc` method is a bit overloaded within Pandas in the sense that it can be used in a variety of different ways that can be at times confusing. In this case we are using the method to not only isolate

rows and columns, but also to assign them values.[2]

- The series object has an `isin` method which behaves similarly to the "in" clause in SQL. Provided a list of items to match it will return a True/False value depending on if the value is in it or not. If we want to isolate all the data relating to both Adair and Wright counties then we use it in the following manner:

```
>>> (dfCars
        .loc[(dfCars.loc[:, 'countyname'].isin(['Adair', 'Wright'])), 'countyname']
        .value_counts())
countyname
Adair     410
Wright    410
Name: count, dtype: int64
```

A caveat about `isin` is that will return False when applied to a NaN.

```
>>> (dfCars
 .loc[(dfCars.loc[:, 'tonnage'].isna()), 'tonnage']
 .isin(['list', 'of', 'values'])
 .value_counts())
tonnage
False    22604
Name: count, dtype: int64
```

5 Regex Pattern Matching

- Pandas has a number of built-in, useful string matching methods, such as `startswith`, `endswith` and `contains`, which we mentioned previously.
- For more complex matches, pandas allows you to use what is called Regular Expression (sometimes called *regex*).
- The most common way of doing this is by using the accessor `str` and then applying a regex enabled method, such as `findall` or `match`, though there are others.
- Let's do a simple example:

```
>>> dfCars.countyname.str.findall('Adair').head()
0          []
1          []
2          []
3          []
4    [Adair]
Name: countyname, dtype: object
```

You can see that the object returned a list for row containing the matching substrings. Lets do something a bit more complex:

[2]We will learn more about the issues that this arrises when we get to Section 9

```
>>> dfCars.countyname.str.upper().str.findall('A').head()
0       [A]
1       [A]
2       [A]
3    [A, A]
4    [A, A]
Name: countyname, dtype: object
```

Once again, we can see that the all matching strings were returned, which in this case would be two of the letter "A".

- Regex is *incredibly* powerful and *incredibly* complex. We can do things like case insensitive matching:

```
>>> dfCars.countyname.str.findall('(?i)a').head()
0       [a]
1       [a]
2       [a]
3    [a, A]
4    [A, a]
Name: countyname, dtype: object
```

- The following will identify everything that starts with an upper case "A":

```
>>> (dfCars
        .loc[(dfCars.loc[:, 'countyname'].str.match('^A.*')), 'countyname']
        .value_counts()
        )
countyname
Adair        410
Appanoose    407
Allamakee    403
Audubon      393
Adams        382
Name: count, dtype: int64
```

- An even more complex example is to get everything that starts with "A" and ends with "s", which will find "Adams", but not the rest of the counties above.

```
>>> (dfCars
        .loc[(dfCars.loc[:, 'countyname'].str.match('^A.*s$')), 'countyname']
        .value_counts()
        )
countyname
Adams    382
Name: count, dtype: int64
```

- It makes sense to spend some time (not a lot) reading over some regex and you should use any opportunity on the homework to play around with it. I'm never going to ask you anything more than the simple, built-in, stuff on an exam.

- While regex is incredibly powerful for pattern matching purposes, its Achilles heel is that it isn't a

strong standard. It is actually a collection of different standards that have significant overlap. The implication is that it is entirely possible that code that works with "Regex" in one place will not work in another.