tu13_DataWrangling

February 28, 2023

1 Data Wrangling

Data wrangling generally refers to the process of getting a data set ready for analysis. Why would we need to do that?

Real-world data can be messy. Data sets are recorded and assembled by humans, and humans make mistakes. A single data set might created and updated by multiple people who may decide to do things in slightly different ways. On a spreadsheet, one person might might decide to leave cells with missing data blank, another might enter "NaN", while a third may enter "missing". If the data has many many rows, one person might decide to repeat the column headers partway down so they don't have to scroll up to see them. Any of these things mean that the data set cannot be analyzed "as is" and wrangling will be required.

Even in a tightly controlled laboratory setting in which data are collected via computer and automatically written out to data files, some data wrangling might be required. There might be a separate data file for each subject or experimental session, meaning that these separate files will have to be combined into a single data set before analysis.

Our main wrangling tool is pandas, so we can go ahead and import it.

[1]:
```
import pandas as pd
```

1.1 Loading

For our wrangling practice today, we'll look at a data set containing various measurements on breast cancer patients. The file is called `breast_cancer_data.csv`, and you should place it in the "data" folder you should already have in the same directory as this notebook.

Let's import it as a pandas dataframe.

[2]:
```
bcd = pd.read_csv('./data/breast_cancer_data.csv')
bcd
```

[2]:
```
     patient_id  clump_thickness  cell_size_uniformity  cell_shape_uniformity  \
0       1000025              5.0                   1.0                      1
1       1002945              5.0                   4.0                      4
2       1015425              3.0                   1.0                      1
3       1016277              6.0                   8.0                      8
4       1017023              4.0                   1.0                      1
..          …                …                    …                      …
```

```
694      776715                3.0                     1.0                        1
695      841769                2.0                     1.0                        1
696      888820                5.0                    10.0                       10
697      897471                4.0                     8.0                        6
698      897471                4.0                     8.0                        8

     marginal_adhesion  single_ep_cell_size bare_nuclei  bland_chromatin  \
0                    1                    2           1              3.0
1                    5                    7          10              3.0
2                    1                    2           2              3.0
3                    1                    3           4              3.0
4                    3                    2           1              3.0
..                 …                    …           …              …
694                  1                    3           2              1.0
695                  1                    2           1              1.0
696                  3                    7           3              8.0
697                  4                    3           4             10.0
698                  5                    4           5             10.0

     normal_nucleoli  mitoses      class doctor_name
0                1.0        1     benign     Dr. Doe
1                2.0        1     benign   Dr. Smith
2                1.0        1     benign     Dr. Lee
3                7.0        1     benign   Dr. Smith
4                1.0        1     benign    Dr. Wong
..               …        …        …         …
694              1.0        1     benign     Dr. Lee
695              1.0        1     benign   Dr. Smith
696             10.0        2  malignant     Dr. Lee
697              6.0        1  malignant     Dr. Lee
698              4.0        1  malignant    Dr. Wong

[699 rows x 12 columns]
```

Before we do any actual wrangling, let's get familiar with the data frame in its current form.

1.2 Exploring the Data Frame

We can explore the data frame by looking at it's attributes, such as its shape, column names, and data types:

[3]:
```
bcd.columns
```

[3]:
```
Index(['patient_id', 'clump_thickness', 'cell_size_uniformity',
       'cell_shape_uniformity', 'marginal_adhesion', 'single_ep_cell_size',
       'bare_nuclei', 'bland_chromatin', 'normal_nucleoli', 'mitoses', 'class',
       'doctor_name'],
      dtype='object')
```

Use the cells below to get the shape and data types (`dtypes`) of our data frame.

[4]:
```
bcd.shape
```

[4]:
```
(699, 12)
```

[5]:
```
bcd.dtypes
```

[5]:
```
patient_id                  int64
clump_thickness           float64
cell_size_uniformity      float64
cell_shape_uniformity       int64
marginal_adhesion           int64
single_ep_cell_size         int64
bare_nuclei                object
bland_chromatin           float64
normal_nucleoli           float64
mitoses                     int64
class                      object
doctor_name                object
dtype: object
```

In the cell below, use the `describe()` method to get a summary of the numerical columns.

[6]:
```
bcd.describe()
```

[6]:
```
         patient_id  clump_thickness  cell_size_uniformity  \
count  6.990000e+02       698.000000            698.000000
mean   1.071704e+06         4.416905              3.137536
std    6.170957e+05         2.817673              3.052575
min    6.163400e+04         1.000000              1.000000
25%    8.706885e+05         2.000000              1.000000
50%    1.171710e+06         4.000000              1.000000
75%    1.238298e+06         6.000000              5.000000
max    1.345435e+07        10.000000             10.000000

       cell_shape_uniformity  marginal_adhesion  single_ep_cell_size  \
count             699.000000         699.000000           699.000000
mean                3.207439           2.793991             3.216023
std                 2.971913           2.843163             2.214300
min                 1.000000           1.000000             1.000000
25%                 1.000000           1.000000             2.000000
50%                 1.000000           1.000000             2.000000
75%                 5.000000           3.500000             4.000000
max                10.000000          10.000000            10.000000
```

```
       bland_chromatin  normal_nucleoli     mitoses
count       695.000000       698.000000  699.000000
mean          3.447482         2.868195    1.589413
std           2.441191         3.055647    1.715078
min           1.000000         1.000000    1.000000
25%           2.000000         1.000000    1.000000
50%           3.000000         1.000000    1.000000
75%           5.000000         4.000000    1.000000
max          10.000000        10.000000   10.000000
```

1.3 Modifying a text column

We'll often want to "tune up" columns that contain text. We might encounter, for example, a column containing full names that we need to break up into separate columns for the first and last names.

Let's look at the column for the doctors' names. Use the cell below to take a peek.

[7]:
```
bcd['doctor_name']
```

[7]:
```
0         Dr. Doe
1       Dr. Smith
2         Dr. Lee
3       Dr. Smith
4        Dr. Wong
          …
694       Dr. Lee
695     Dr. Smith
696       Dr. Lee
697       Dr. Lee
698      Dr. Wong
Name: doctor_name, Length: 699, dtype: object
```

The doctors' name data are redundant; each one has a "Dr. " in front of the actual name, but we already know these are doctors by the column name. Further, the entries have white space in them, which can cause us problems down the road. So let's modify this column so it only contains the surnames of the doctors.

One great thing about pandas is that it has versions of many of Python's string methods that operate *element-wise on an entire column of strings*. Here, we want to separate the "Dr. " from the actual name, which is exactly what Python's `str.split()` function does. So chances are, pandas has a version of this function that operates element-wise on data frames.

String Splitting Review: Let's briefly remind ourselves of splitting up Python strings and extracting bits of them.

[8]:
```
# Here's a string of the form: surname, first initial.
myStr = 'SirString, A.'
print(myStr)
```

```
SirString, A.
```

Let's say we wanted to get the surname. We could split this string into a Python list at the white space like this:

[9]:
```
spltStr = myStr.split()    # split() defaults to splitting at white space
print(spltStr)
```

```
['SirString,', 'A.']
```

We now have a list in which the items contain the text on either side of the split. This is close to what we want: the first entry in the list has the surname, but it also has an unwanted comma.

Let's split the string at the comma instead:

[10]:
```
spltStr = myStr.split(',')   # tell Python to split at commas
print(spltStr)
```

```
['SirString', ' A.']
```

Now we have isolated the last name, and we can fetch it by indexing:

[11]:
```
surname = spltStr[0]
print(surname)
```

```
SirString
```

In the cell below, see if you can extract the surname from `myStr` in one line of code:

[12]:
```
myStr
```

[12]:
```
'SirString, A.'
```

[13]:
```
test_spr = myStr.split(',')[0]
test_spr
```

[13]:
```
'SirString'
```

Alright, time to replace the `bcd['doctor_name']` column values with just the doctors' last names.

We could do this in one step, but let's break it out for clarity. First, let's copy the name column out into a new series.

[14]:
```
dr_names = bcd['doctor_name']
dr_names
```

[14]:
```
0         Dr. Doe
1       Dr. Smith
2         Dr. Lee
3       Dr. Smith
4        Dr. Wong
          …
694       Dr. Lee
695     Dr. Smith
696       Dr. Lee
697       Dr. Lee
698      Dr. Wong
Name: doctor_name, Length: 699, dtype: object
```

Note: pandas objects behave like ordinary Python objects. So, strictly speaking, we have not created a new object (pandas Series), rather, *we have created a new label that refers to the "doctor_name" column of `bcd`.*

In the cell below, use the `id()` function to compare the object IDs of `dr_names` and the corresponding column of `bcd`.

[15]:
```
id(dr_names)
```

[15]:
```
140289344521120
```

[16]:
```
id(bcd['doctor_name'])
```

[16]:
```
140289344521120
```

Now let's split all the names in the `doctor_name` column at the whitespace by using pandas `DataFrame.str.split()` function.

[17]:
```
split_dr_names = dr_names.str.split()
split_dr_names
```

[17]:
```
0         [Dr., Doe]
1       [Dr., Smith]
2         [Dr., Lee]
3       [Dr., Smith]
4        [Dr., Wong]
             …
694       [Dr., Lee]
695     [Dr., Smith]
```

```
696       [Dr., Lee]
697       [Dr., Lee]
698      [Dr., Wong]
Name: doctor_name, Length: 699, dtype: object
```

`DataFrame.str.split()`, however, *does* create a new object.

Use the cell below to confirm that the `split()` spawed a new object.

[20]:
```
split_dr_names = dr_names.str.split('.')
split_dr_names
```

[20]:
```
0         [Dr,  Doe]
1       [Dr,  Smith]
2         [Dr,  Lee]
3       [Dr,  Smith]
4        [Dr,  Wong]
            …
694       [Dr,  Lee]
695     [Dr,  Smith]
696       [Dr,  Lee]
697       [Dr,  Lee]
698      [Dr,  Wong]
Name: doctor_name, Length: 699, dtype: object
```

Now we have a column of lists, each with two elements. The first element of each list is the "Dr. " bit, and the second consists of the surnames we want.

We can get these by using pandas string indexing, `Series.str[index]`.

[23]:
```
surnames = split_dr_names.str[1]
surnames
```

[23]:
```
0          Doe
1        Smith
2          Lee
3        Smith
4         Wong
        …
694        Lee
695      Smith
696        Lee
697        Lee
698       Wong
Name: doctor_name, Length: 699, dtype: object
```

Note that, like the splitting, the string indexing worked on the entire `Series` automatically.

Now we can change the column in our main data frame, `bcd`.

[24]:
```
bcd['doctor_name'] = surnames
```

[25]:
```
bcd['doctor_name']
```

[25]:
```
0          Doe
1        Smith
2          Lee
3        Smith
4         Wong
        …
694        Lee
695      Smith
696        Lee
697        Lee
698       Wong
Name: doctor_name, Length: 699, dtype: object
```

[26]:
```
bcd.head()
```

[26]:
```
   patient_id  clump_thickness  cell_size_uniformity  cell_shape_uniformity  \
0     1000025              5.0                   1.0                      1
1     1002945              5.0                   4.0                      4
2     1015425              3.0                   1.0                      1
3     1016277              6.0                   8.0                      8
4     1017023              4.0                   1.0                      1

   marginal_adhesion  single_ep_cell_size bare_nuclei  bland_chromatin  \
0                  1                    2           1              3.0
1                  5                    7          10              3.0
2                  1                    2           2              3.0
3                  1                    3           4              3.0
4                  3                    2           1              3.0

   normal_nucleoli  mitoses   class doctor_name
0              1.0        1  benign         Doe
1              2.0        1  benign       Smith
2              1.0        1  benign         Lee
3              7.0        1  benign       Smith
4              1.0        1  benign        Wong
```

Success!

1.4 Converting a column type (and other aggravations)

Let's look at those data types again.

[27]:
```
bcd.dtypes
```

[27]:
```
patient_id                int64
clump_thickness         float64
cell_size_uniformity    float64
cell_shape_uniformity     int64
marginal_adhesion         int64
single_ep_cell_size       int64
bare_nuclei              object
bland_chromatin         float64
normal_nucleoli         float64
mitoses                   int64
class                    object
doctor_name              object
dtype: object
```

Notice that "class" and "doctor_name" are of dtype "object", which refers to a general purpose column type, and is how pandas imports text columns by default. Most of the others are numeric (integers or floats), except for "bare_nuclei".

In the cell below, take a quick glance at 'bcd' again, and see if the "bare_nuclei" column should be a different data type that, say "marginal_adhesion".

[37]:
```
bcd.head()
```

[37]:
```
   patient_id  clump_thickness  cell_size_uniformity  cell_shape_uniformity  \
0     1000025              5.0                   1.0                      1
1     1002945              5.0                   4.0                      4
2     1015425              3.0                   1.0                      1
3     1016277              6.0                   8.0                      8
4     1017023              4.0                   1.0                      1

   marginal_adhesion  single_ep_cell_size bare_nuclei  bland_chromatin  \
0                  1                    2           1              3.0
1                  5                    7          10              3.0
2                  1                    2           2              3.0
3                  1                    3           4              3.0
4                  3                    2           1              3.0

   normal_nucleoli  mitoses   class doctor_name
0              1.0        1  benign         Doe
1              2.0        1  benign       Smith
2              1.0        1  benign         Lee
3              7.0        1  benign       Smith
4              1.0        1  benign        Wong
```

It looks like "bare_nuclei" was intended to be a numeric column, so let's try and convert it using the `DataFrame.astype()` converter method.

```
[29]: bcd['bare_nuclei'] = bcd['bare_nuclei'].astype('int64')
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Input In [29], in <cell line: 1>()
----> 1 bcd['bare_nuclei'] = bcd['bare_nuclei'].astype('int64')

File ~/opt/anaconda3/lib/python3.9/site-packages/pandas/core/generic.py:5912, i␣
↪NDFrame.astype(self, dtype, copy, errors)
   5905     results = [
   5906         self.iloc[:, i].astype(dtype, copy=copy)
   5907         for i in range(len(self.columns))
   5908     ]
   5910 else:
   5911     # else, only a single dtype is given
-> 5912     new_data = self._mgr.astype(dtype=dtype, copy=copy, errors=errors)
   5913     return self._constructor(new_data).__finalize__(self,␣
↪method="astype")
   5915 # GH 33113: handle empty frame or series

File ~/opt/anaconda3/lib/python3.9/site-packages/pandas/core/internals/managers
↪py:419, in BaseBlockManager.astype(self, dtype, copy, errors)
    418 def astype(self: T, dtype, copy: bool = False, errors: str = "raise") -␣
↪T:
--> 419     return self.apply("astype", dtype=dtype, copy=copy, errors=errors)

File ~/opt/anaconda3/lib/python3.9/site-packages/pandas/core/internals/managers
↪py:304, in BaseBlockManager.apply(self, f, align_keys, ignore_failures,␣
↪**kwargs)
    302         applied = b.apply(f, **kwargs)
    303     else:
--> 304         applied = getattr(b, f)(**kwargs)
    305 except (TypeError, NotImplementedError):
    306     if not ignore_failures:

File ~/opt/anaconda3/lib/python3.9/site-packages/pandas/core/internals/blocks.p␣:
↪580, in Block.astype(self, dtype, copy, errors)
    562 """
    563 Coerce to the new dtype.
    564
   (…)
    576 Block
    577 """
    578 values = self.values
--> 580 new_values = astype_array_safe(values, dtype, copy=copy, errors=errors)
```

```
    582 new_values = maybe_coerce_values(new_values)
    583 newb = self.make_block(new_values)

File ~/opt/anaconda3/lib/python3.9/site-packages/pandas/core/dtypes/cast.py:
 ↪1292, in astype_array_safe(values, dtype, copy, errors)
   1289     dtype = dtype.numpy_dtype
   1291 try:
-> 1292     new_values = astype_array(values, dtype, copy=copy)
   1293 except (ValueError, TypeError):
   1294     # e.g. astype_nansafe can fail on object-dtype of strings
   1295     #  trying to convert to float
   1296     if errors == "ignore":

File ~/opt/anaconda3/lib/python3.9/site-packages/pandas/core/dtypes/cast.py:
 ↪1237, in astype_array(values, dtype, copy)
   1234     values = values.astype(dtype, copy=copy)
   1236 else:
-> 1237     values = astype_nansafe(values, dtype, copy=copy)
   1239 # in pandas we don't store numpy str dtypes, so convert to object
   1240 if isinstance(dtype, np.dtype) and issubclass(values.dtype.type, str):

File ~/opt/anaconda3/lib/python3.9/site-packages/pandas/core/dtypes/cast.py:
 ↪1154, in astype_nansafe(arr, dtype, copy, skipna)
   1150 elif is_object_dtype(arr.dtype):
   1151
   1152     # work around NumPy brokenness, #1987
   1153     if np.issubdtype(dtype.type, np.integer):
-> 1154         return lib.astype_intsafe(arr, dtype)
   1156     # if we have a datetime/timedelta array of objects
   1157     # then coerce to a proper dtype and recall astype_nansafe
   1159     elif is_datetime64_dtype(dtype):

File ~/opt/anaconda3/lib/python3.9/site-packages/pandas/_libs/lib.pyx:668, in␣
 ↪pandas._libs.lib.astype_intsafe()

ValueError: invalid literal for int() with base 10: '?'
```

And, argh, we get an error! If we look at the bottom of the error message, it seems that the error involves question marks ("?") in the data, which would also explain why this column imported as text rather than numbers in the first place.

Let's check.

In the cell below, use logical indexing to show the rows of `bcd` in which `bcd[bare_nuclei]` contains a question mark.

```
[41]: bcd['bare_nuclei'].str.find('?')
```

```
[41]: 0     -1.0
      1     -1.0
      2     -1.0
      3     -1.0
      4     -1.0
            …
      694   -1.0
      695   -1.0
      696   -1.0
      697   -1.0
      698   -1.0
      Name: bare_nuclei, Length: 699, dtype: float64
```

Sure enough. Rather than leaving the cells of missing values empty, somebody has made the poor decision to enter question marks instead.

When you are dealing with other peoples' data, you'll find that this sort of the happens a LOT. It can be very aggravating, so we need to learn to treat these things as challenging puzzles instead of hassles!

Let's replace the question marks with nothing, so that this column becomes consistent with the rest. Fortunately, `DataFrame` (and `Series`) objects have a `replace()` function built in, so let's use that.

```
[34]: bcd['bare_nuclei'] = bcd['bare_nuclei'].replace('?', '')
```

In the cell below, confirm that we no longer have question marks in our "bare_nuclei" column.

```
[42]: bcd['bare_nuclei'].str.find('?')
```

```
[42]: 0     -1.0
      1     -1.0
      2     -1.0
      3     -1.0
      4     -1.0
            …
      694   -1.0
      695   -1.0
      696   -1.0
      697   -1.0
      698   -1.0
      Name: bare_nuclei, Length: 699, dtype: float64
```

Note: As mentioned above, extracting columns or other subsets of data from a pandas `DataFrame` or `Series` does not create a new object but rather a new label to the existing object.

So, for example, `the_IDs = bcd['patient_id']` does not make a new object, but rather creates a second label referring to the original object (consistent with the behavior of base Python).

In general, however, pandas methods (functions) *do* create new objects. Thus, the step of assigning the output of `.replace()` back to the original data frame column is necessary.

In the cell below, confirm that the output of `.replace()` and `bcd['bare_nuclei']` have different IDs.

```
[43]: id(bcd['bare_nuclei'] )
```

```
[43]: 140289448713760
```

```
[45]: id(bcd['bare_nuclei'].replace('?', ''))
```

```
[45]: 140288943200192
```

And now we can convert the column to numeric values.

```
[46]: bcd['bare_nuclei'] = pd.to_numeric(bcd['bare_nuclei'])
```

In the cell below, check the data types of columns in `bcd`.

```
[49]: bcd['bare_nuclei'].dtypes
```

```
[49]: dtype('float64')
```

Okay! We have now have gotten our data somewhat into shape, meaning:

- missing data are actually missing
- columns of numeric data are numeric in type
- the column of doctor names contains only last names

So now we can explore some ways to deal with missing values.

1.5 Dealing with missing data

1.5.1 Finding missing values

Even though this dataset isn't all that large:

```
[50]: bcd.shape
```

```
[50]: (699, 12)
```

699 rows is lot to look through "by hand" in order to find missing values.

We can test for missing values using the `DataFrame.isna()` method.

```
[51]: bcd.isna()
```

```
[51]:      patient_id  clump_thickness  cell_size_uniformity  cell_shape_uniformity  \
      0         False            False                 False                  False
      1         False            False                 False                  False
      2         False            False                 False                  False
      3         False            False                 False                  False
      4         False            False                 False                  False
      ..          …              …                     …                      …
      694       False            False                 False                  False
      695       False            False                 False                  False
      696       False            False                 False                  False
      697       False            False                 False                  False
      698       False            False                 False                  False

           marginal_adhesion  single_ep_cell_size  bare_nuclei  bland_chromatin  \
      0                False                False        False            False
      1                False                False        False            False
      2                False                False        False            False
      3                False                False        False            False
      4                False                False        False            False
      ..                 …                    …          …              …
      694              False                False        False            False
      695              False                False        False            False
      696              False                False        False            False
      697              False                False        False            False
      698              False                False        False            False

           normal_nucleoli  mitoses  class  doctor_name
      0              False    False  False        False
      1              False    False  False        False
      2              False    False  False        False
      3              False    False  False        False
      4              False    False  False        False
      ..               …        …     …         …
      694            False    False  False        False
      695            False    False  False        False
      696            False    False  False        False
      697            False    False  False        False
      698            False    False  False        False

      [699 rows x 12 columns]
```

By itself, that doesn't help us much. But if we combine it with summation (remember that `True` values count as 1 and `False` counts as zero):

```
[52]: bcd.isna().sum()
```

```
[52]: patient_id              0
      clump_thickness         1
      cell_size_uniformity    1
      cell_shape_uniformity   0
      marginal_adhesion       0
      single_ep_cell_size     0
      bare_nuclei            18
      bland_chromatin         4
      normal_nucleoli         1
      mitoses                 0
      class                   0
      doctor_name             0
      dtype: int64
```

Now we have the counts by variable, and can easly see that there are missing values for a few of the variables.

The "bare_nuclei" variable we dealt with earlier has the most missing values, with "bland_chromatin" coming in a distant second.

Let's check some of the rows with missing values and make sure everything else looks normal in those rows. Notice above that the output of `.isna()` is Boolean, so we can use it to do logical indexing.

```
[53]: bcd[bcd['bland_chromatin'].isna()]
```

```
[53]:      patient_id  clump_thickness  cell_size_uniformity  cell_shape_uniformity  \
      342      814265              2.0                   1.0                      1
      343      814911              1.0                   1.0                      1
      359      873549             10.0                   3.0                      5
      365      897172              2.0                   1.0                      1

           marginal_adhesion  single_ep_cell_size  bare_nuclei  bland_chromatin  \
      342                  1                    2          1.0              NaN
      343                  1                    2          1.0              NaN
      359                  4                    3          7.0              NaN
      365                  1                    2          1.0              NaN

           normal_nucleoli  mitoses      class doctor_name
      342              1.0        1     benign         Lee
      343              1.0        1     benign         Doe
      359              5.0        3  malignant         Doe
      365              1.0        1     benign         Lee
```

In the cell below, check the rows that have missing values for either clump thickness or cell size uniformity. Do this in one go rather than separately (remember about the element-wise or operator, "|".

[55]:
```
bcd[(bcd['clump_thickness'].isna()) | (bcd['cell_size_uniformity'].isna())]
```

[55]:
```
    patient_id  clump_thickness  cell_size_uniformity  cell_shape_uniformity  \
6      1018099              1.0                   NaN                      1
12     1041801              NaN                   3.0                      3

    marginal_adhesion  single_ep_cell_size  bare_nuclei  bland_chromatin  \
6                   1                    2         10.0              3.0
12                  3                    2          3.0              4.0

    normal_nucleoli  mitoses      class doctor_name
6               1.0        1     benign         Doe
12              4.0        1  malignant       Smith
```

So far so good. It looks like the rows that have missing values just have one missing value, and everything else seems fine. But let's do check that no rows have more than one missing value.

To do this, we can sum the number of missing values across the columns (i.e. within each row), and then see what the maximum number of missing values within a row is.

[56]:
```
row_na_totals = bcd.isna().sum(axis = 1)
row_na_totals.max()
```

[56]:
```
1
```

So we see that no row has more than one missing value.

In the cell below, do the above calculation in one line.

[57]:
```
bcd.isna().sum(axis = 1).max()
```

[57]:
```
1
```

1.5.2 Dealing with missing values

Now that we have determined that there are missing values, we have to determine how to deal with them.

Ignoring missing values elementwise One way to handle missing values is just to ignore them. Most of the standard math and statisitical functions will do that by default.

So this:

```
[58]: bcd['clump_thickness'].mean()
```

```
[58]: 4.416905444126074
```

Computes the mean clump thickness ignoring the one missing value.

We can compute the mean (again ignoring missing values) for all the numeric columns like this:

```
[59]: bcd.mean(numeric_only = True)  # the numeric_only refers to columns, not␣
        ↪missing values
```

```
[59]: patient_id                1.071704e+06
      clump_thickness           4.416905e+00
      cell_size_uniformity      3.137536e+00
      cell_shape_uniformity     3.207439e+00
      marginal_adhesion         2.793991e+00
      single_ep_cell_size       3.216023e+00
      bare_nuclei               3.538913e+00
      bland_chromatin           3.447482e+00
      normal_nucleoli           2.868195e+00
      mitoses                   1.589413e+00
      dtype: float64
```

That worked, but the output is a little awkward because the patient ID is being treated as a numeric variable. We can fix that by converting the patient ID variable to a string variable.

```
[61]: bcd['patient_id'] = bcd['patient_id'].astype('string')
      bcd['patient_id']
```

```
[61]: 0      1000025
      1      1002945
      2      1015425
      3      1016277
      4      1017023
               …
      694     776715
      695     841769
      696     888820
      697     897471
      698     897471
      Name: patient_id, Length: 699, dtype: string
```

And now the means should look a little better because we won't have the mean for the ID column in the millions>

Recompute the mean for the numeric columns in the cell below.

```
[62]: bcd.mean(numeric_only = True)   # the numeric_only refers to columns, not␣
       ↪missing values
```

```
[62]: clump_thickness           4.416905
      cell_size_uniformity      3.137536
      cell_shape_uniformity     3.207439
      marginal_adhesion         2.793991
      single_ep_cell_size       3.216023
      bare_nuclei               3.538913
      bland_chromatin           3.447482
      normal_nucleoli           2.868195
      mitoses                   1.589413
      dtype: float64
```

Removing missing values We are about to start learning how to remove missing values from our data frame, *however…*

Before we start messing around too much with the values in our data frame, let's make sure we can easily "hit the reset button" and get back to a nice starting point. To do this, we'll want to

- reload the data
- modify the column of Dr. names
- set the patient ID to type str
- remove the question marks from the bare nuclei column
- set the bare nuclei column to numeric

This is a perfect job for a function!

In the cell below, finish writing the function to reset our data frame to the desired starting point.

```
[101]: def hit_reset():
           bcd = pd.read_csv('./data/breast_cancer_data.csv')       # Reload the data

           split_dr_names = dr_names.str.split()                    # Modify the␣
         ↪column of Dr. Names
           surnames = split_dr_names.str[1]
           bcd['doctor_name'] = surnames

           bcd['patient_id'] = bcd['patient_id'].astype('string')   # Set the patient␣
         ↪ID to type str

           bcd['bare_nuclei'] = bcd['bare_nuclei'].replace('?', '') # Remove the␣
         ↪question marks from the bare nuclei column
```

```
    bcd['bare_nuclei'] = pd.to_numeric(bcd['bare_nuclei'])    # Set the bare␣
↪nuclei column to numeric

    return bcd
```

[102]:
```
hit_reset()
```

[102]:
```
     patient_id  clump_thickness  cell_size_uniformity  cell_shape_uniformity  \
0       1000025              5.0                   1.0                      1
1       1002945              5.0                   4.0                      4
2       1015425              3.0                   1.0                      1
3       1016277              6.0                   8.0                      8
4       1017023              4.0                   1.0                      1
..          …              …                   …                    …
694      776715              3.0                   1.0                      1
695      841769              2.0                   1.0                      1
696      888820              5.0                  10.0                     10
697      897471              4.0                   8.0                      6
698      897471              4.0                   8.0                      8

     marginal_adhesion  single_ep_cell_size  bare_nuclei  bland_chromatin  \
0                    1                    2          1.0              3.0
1                    5                    7         10.0              3.0
2                    1                    2          2.0              3.0
3                    1                    3          4.0              3.0
4                    3                    2          1.0              3.0
..                 …                  …          …              …
694                  1                    3          2.0              1.0
695                  1                    2          1.0              1.0
696                  3                    7          3.0              8.0
697                  4                    3          4.0             10.0
698                  5                    4          5.0             10.0

     normal_nucleoli  mitoses      class doctor_name
0                1.0        1     benign         Doe
1                2.0        1     benign       Smith
2                1.0        1     benign         Lee
3                7.0        1     benign       Smith
4                1.0        1     benign        Wong
..               …      …        …         …
694              1.0        1     benign         Lee
695              1.0        1     benign       Smith
696             10.0        2  malignant         Lee
697              6.0        1  malignant         Lee
698              4.0        1  malignant        Wong
```

```
[699 rows x 12 columns]
```

[80]:
```
bcd.dtypes
```

[80]:
```
patient_id                 string
clump_thickness           float64
cell_size_uniformity      float64
cell_shape_uniformity       int64
marginal_adhesion           int64
single_ep_cell_size         int64
bare_nuclei               float64
bland_chromatin           float64
normal_nucleoli           float64
mitoses                     int64
class                      object
doctor_name                object
dtype: object
```

Removing rows with missing values Obviously, rows in which all values are missing won't do us any good, so we can drop them with:

[81]:
```
bcd = bcd.dropna(how = 'all')
```

This drops rows in which *all* of the values are missing. This code ran without error, but we know it also didn't do anything in this case because we don't have any rows in which all the values are missing!

Sometimes a case can be made for throwing out all observations (rows) that are incomplete, that is, if they contain *any* missing values.

[82]:
```
bcd = bcd.dropna(how = 'any')
```

In the cell below, check the (new) shape of `bcd`.

[84]:
```
bcd.shape
```

[84]:
```
(674, 12)
```

It should have fewer rows now.

And now is a perfect time to test our function! In the cell below, hit the reset button on bcd.

[85]:
```
hit_reset()
```

```
[85]:      patient_id  clump_thickness  cell_size_uniformity  cell_shape_uniformity  \
      0       1000025              5.0                   1.0                      1
      1       1002945              5.0                   4.0                      4
      2       1015425              3.0                   1.0                      1
      3       1016277              6.0                   8.0                      8
      4       1017023              4.0                   1.0                      1
      ..          …                 …                     …                      …
      694      776715              3.0                   1.0                      1
      695      841769              2.0                   1.0                      1
      696      888820              5.0                  10.0                     10
      697      897471              4.0                   8.0                      6
      698      897471              4.0                   8.0                      8

           marginal_adhesion  single_ep_cell_size  bare_nuclei  bland_chromatin  \
      0                    1                    2          1.0              3.0
      1                    5                    7         10.0              3.0
      2                    1                    2          2.0              3.0
      3                    1                    3          4.0              3.0
      4                    3                    2          1.0              3.0
      ..                 …                    …            …                …
      694                  1                    3          2.0              1.0
      695                  1                    2          1.0              1.0
      696                  3                    7          3.0              8.0
      697                  4                    3          4.0             10.0
      698                  5                    4          5.0             10.0

           normal_nucleoli  mitoses      class doctor_name
      0                1.0        1     benign         Doe
      1                2.0        1     benign       Smith
      2                1.0        1     benign         Lee
      3                7.0        1     benign       Smith
      4                1.0        1     benign        Wong
      ..               …        …         …          …
      694              1.0        1     benign         Lee
      695              1.0        1     benign       Smith
      696             10.0        2  malignant         Lee
      697              6.0        1  malignant         Lee
      698              4.0        1  malignant        Wong

      [699 rows x 12 columns]
```

Check the shape.

```
[86]: bcd.shape
```

```
[86]: (674, 12)
```

Check the data types of the columns.

```
[87]: bcd.dtypes
```

```
[87]: patient_id                 string
      clump_thickness           float64
      cell_size_uniformity      float64
      cell_shape_uniformity       int64
      marginal_adhesion           int64
      single_ep_cell_size         int64
      bare_nuclei               float64
      bland_chromatin           float64
      normal_nucleoli           float64
      mitoses                     int64
      class                      object
      doctor_name                object
      dtype: object
```

Check the doctor name column.

```
[88]: bcd['doctor_name']
```

```
[88]: 0        Doe
      1      Smith
      2        Lee
      3      Smith
      4       Wong
             …
      694      Lee
      695    Smith
      696      Lee
      697      Lee
      698     Wong
      Name: doctor_name, Length: 674, dtype: object
```

Removing columns with missing values And we could do the same for columns if we wished, though this is less frequently done. We just need to change the axis (direction) over which `DataFrame.dropna()` works.

```
[89]: bcd = bcd.dropna(axis = 1, how = 'any') # drop columns rather than rows
```

This leaves us with only the complete columns.

```
[90]: bcd.shape
```

```
[90]: (674, 12)
```

Let's see which they are.

```
[91]: bcd.columns
```

```
[91]: Index(['patient_id', 'clump_thickness', 'cell_size_uniformity',
              'cell_shape_uniformity', 'marginal_adhesion', 'single_ep_cell_size',
              'bare_nuclei', 'bland_chromatin', 'normal_nucleoli', 'mitoses', 'class',
              'doctor_name'],
             dtype='object')
```

Filling in missing values Occasionally, we may want to fill in missing values. This isn't very common, but might be useful if some other function you are using doesn't handle missing values gracefully.

Before filling in missing values, we need to restore our data frame so it actually has missing values. Good thing we wrote that function!

```
[92]: bcd = hit_reset()
```

We can fill in missing values with any single value we want, such as a zero.

```
[93]: bcd = bcd.fillna(0)
```

In the cell below, check to see that we no longer have missing values.

```
[104]: bcd.isna()
       bcd.isna().sum()
```

```
[104]: patient_id               0
       clump_thickness          0
       cell_size_uniformity     0
       cell_shape_uniformity    0
       marginal_adhesion        0
       single_ep_cell_size      0
       bare_nuclei              0
       bland_chromatin          0
       normal_nucleoli          0
       mitoses                  0
       class                    0
       doctor_name              0
       dtype: int64
```

In the cell below, reset the data and verify that the missing data are back.

```
[109]: bcd = hit_reset()
```

---

[108]:
```
bcd.isna()
bcd.isna().sum()
```

[108]:
```
patient_id              0
clump_thickness         1
cell_size_uniformity    1
cell_shape_uniformity   0
marginal_adhesion       0
single_ep_cell_size     0
bare_nuclei            18
bland_chromatin         4
normal_nucleoli         1
mitoses                 0
class                   0
doctor_name             0
dtype: int64
```

In the cell below, fill the missing values in each column with the column mean. (Hint: this is pandas, so this is actually easy!)

[111]:
```
bcd.describe()
```

[111]:
```
       clump_thickness  cell_size_uniformity  cell_shape_uniformity  \
count       698.000000            698.000000             699.000000
mean          4.416905              3.137536               3.207439
std           2.817673              3.052575               2.971913
min           1.000000              1.000000               1.000000
25%           2.000000              1.000000               1.000000
50%           4.000000              1.000000               1.000000
75%           6.000000              5.000000               5.000000
max          10.000000             10.000000              10.000000

       marginal_adhesion  single_ep_cell_size  bare_nuclei  bland_chromatin  \
count         699.000000           699.000000   681.000000       695.000000
mean            2.793991             3.216023     3.538913         3.447482
std             2.843163             2.214300     3.639493         2.441191
min             1.000000             1.000000     1.000000         1.000000
25%             1.000000             2.000000     1.000000         2.000000
50%             1.000000             2.000000     1.000000         3.000000
75%             3.500000             4.000000     6.000000         5.000000
max            10.000000            10.000000    10.000000        10.000000

       normal_nucleoli     mitoses
count       698.000000  699.000000
mean          2.868195    1.589413
std           3.055647    1.715078
min           1.000000    1.000000
```

```
25%           1.000000    1.000000
50%           1.000000    1.000000
75%           4.000000    1.000000
max          10.000000   10.000000
```

[115]:
```
bcd = bcd.fillna('mean')
```

And now verify that there are no more missing values.

[116]:
```
bcd.isna().sum()
```

[116]:
```
patient_id               0
clump_thickness          0
cell_size_uniformity     0
cell_shape_uniformity    0
marginal_adhesion        0
single_ep_cell_size      0
bare_nuclei              0
bland_chromatin          0
normal_nucleoli          0
mitoses                  0
class                    0
doctor_name              0
dtype: int64
```

1.6 Summary

In this tutorial, we learned or remembered how to do some of the foundational data wrangling tasks. These are:

- importing data into pandas from a data file
- cleaning up the data in the columns
- converting columns to the appropriate type
- removing or filling in missing values