Lecture 03. Data Loading and Cleaning

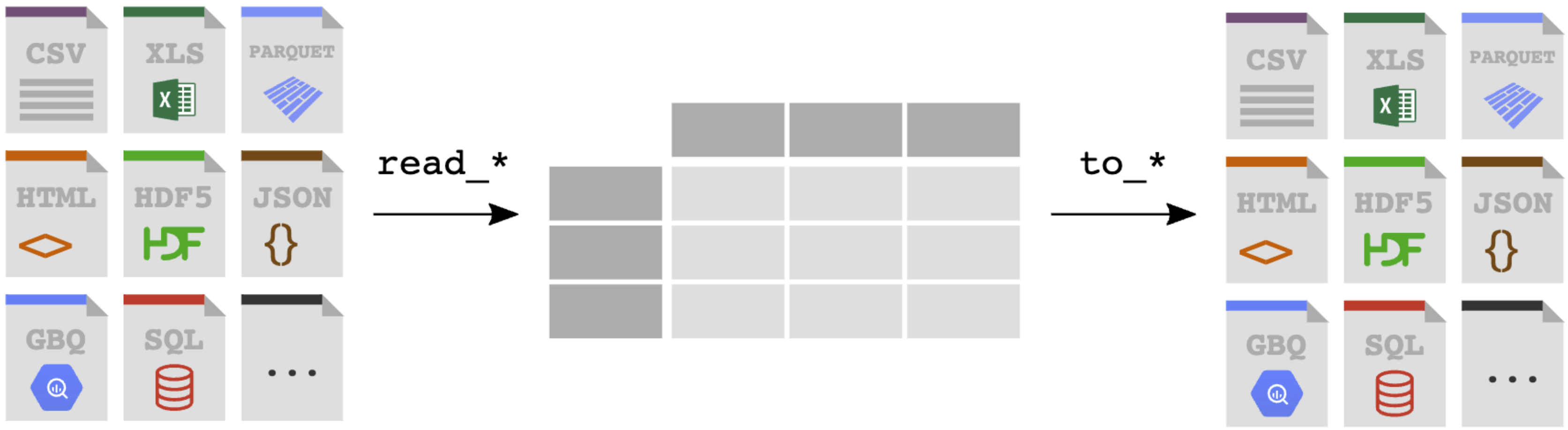
Instructor: Luping Yu

Mar 14, 2023

Accessing data is a necessary first step for using most of the tools in this course. I'm going to be focused on data input and output using pandas.

Reading and writing data in text format

`pandas` features a number of functions for reading **tabular data** as a `DataFrame` object.



The following table summarizes some of them, though `read_csv` is likely the ones you'll use the most.

Function	Description
read_csv	Load delimited data from a file, URL, or file-like object; use comma as default delimiter
read_excel	Read tabular data from an Excel XLS or XLSX file
read_stata	Read a dataset from Stata file format
read_sas	Read a SAS dataset stored in one of the SAS system's custom storage formats
read_html	Read all tables found in the given HTML document
read_json	Read data from a JSON (JavaScript Object Notation) string representation
read_pickle	Read an arbitrary object stored in Python pickle format
read_sql	Read the results of a SQL query (using SQLAlchemy) as a pandas DataFrame

---

Reading and Writing .csv (comma-separated values)

`.csv` is a delimited text file that uses a **comma** to separate values. A `.csv` file typically stores **tabular data** (numbers and text) in **plain text**.

Let's start with a small `.csv` text file: ex1.csv

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

pd.read_csv('examples/ex1.csv') # relative path
```

Out[1]:

	a	b	c	d	message
0	1	2	3	4	hello
1	5	6	7	8	world
2	9	10	11	12	foo

In [2]:
```
# absolute path  (absolute path differs between Windows and Mac)
pd.read_csv('/Users/luping/desktop/teaching/examples/ex1.csv')
```

Out[2]:

	a	b	c	d	message
0	1	2	3	4	hello
1	5	6	7	8	world
2	9	10	11	12	foo

`pandas.read_csv()` perform type inference. That means you don't necessarily have to specify which columns are numeric, integer, boolean, or string:

In [3]:
```
df = pd.read_csv('/Users/luping/desktop/teaching/examples/ex1.csv')

df.dtypes
```

Out[3]:
```
a            int64
b            int64
c            int64
d            int64
message     object
dtype: object
```

A file will not always have a **header row**. Consider this file: ex2.csv

In [4]:
```
cat examples/ex2.csv
```

```
1,2,3,4,hello
5,6,7,8,world
9,10,11,12,foo
```

Here I used the Linux/macOS `cat` shell command to print the raw contents of the file to the screen. If you're on Windows, you can use `type` instead of cat to achieve the same effect.

To read this file, you have a couple of options. You can allow pandas to assign default column names, or you can specify names yourself:

In [5]:
```
pd.read_csv('examples/ex2.csv', header=None)
```

Out[5]:

	0	1	2	3	4
0	1	2	3	4	hello
1	5	6	7	8	world
2	9	10	11	12	foo

In [6]:
```
pd.read_csv('examples/ex2.csv', names=['a', 'b', 'c', 'd', 'message'])
```

Out[6]:

	a	b	c	d	message
0	1	2	3	4	hello
1	5	6	7	8	world
2	9	10	11	12	foo

Suppose you wanted the message column to be the index of the returned `DataFrame`. You can use the `index_col` argument:

In [7]:
```
pd.read_csv('examples/ex2.csv', names=['a', 'b', 'c', 'd', 'message'], index_col='message')
```

Out[7]:

	a	b	c	d
message				
hello	1	2	3	4
world	5	6	7	8
foo	9	10	11	12

Handling missing values is an important and frequently nuanced part of the file parsing process. Consider this file: ex3.csv

In [8]:
```
cat examples/ex3.csv
```

```
something,a,b,c,d,message
one,1,2,3,4,NA
two,5,6,,8,world
three,9,10,11,12,foo
```

Missing data is usually either not present (empty string) or marked by some **sentinel** value, such as **NA** and **NULL**.

In [9]:
```
pd.read_csv('examples/ex3.csv')
```

Out[9]:

	something	a	b	c	d	message
0	one	1	2	3.0	4	NaN
1	two	5	6	NaN	8	world
2	three	9	10	11.0	12	foo

In [10]:
```
df = pd.read_csv('examples/ex3.csv')

pd.notnull(df)
```

Out[10]:

	something	a	b	c	d	message
0	True	True	True	True	True	False
1	True	True	True	False	True	True
2	True	True	True	True	True	True

Data can also be exported to a delimited format. Let's consider one of the `.csv` files read before:

In [11]:
```
df
```

Out[11]:

	something	a	b	c	d	message
0	one	1	2	3.0	4	NaN
1	two	5	6	NaN	8	world
2	three	9	10	11.0	12	foo

Using `to_csv()` method, we can write the data out to a comma-separated file:

In [12]:
```
df.to_csv('examples/out1.csv')
```

In [13]:
```
cat examples/out1.csv
```

```
,something,a,b,c,d,message
0,one,1,2,3.0,4,
1,two,5,6,,8,world
2,three,9,10,11.0,12,foo
```

With no other options specified, both the row and column labels are written. Both of these can be disabled:

In [14]:
```
df.to_csv('examples/out2.csv', index=False, header=False)
```

In [15]:
```
cat examples/out2.csv
```

```
one,1,2,3.0,4,
two,5,6,,8,world
three,9,10,11.0,12,foo
```

Parameters of data loading functions

Because of how messy data in the real world can be, data loading functions (especially `read_csv()`) have grown very complex in their options over time. The **online pandas documentation** has many examples about how each of them works.

API reference (pandas documentation) of `read_csv()`: https://pandas.pydata.org/docs/reference/api/pandas.read_csv.html

---

Reading Microsoft excel files

`pandas` also supports reading tabular data stored in Excel 2003 (and higher) files using `pandas.read_excel()` function:

Internally these tools use the add-on packages **xlrd** and **openpyxl** to read XLS and XLSX files, respectively. You may need to install these manually with pip.

In [16]:
```
df = pd.read_excel('examples/ex1.xlsx', 'Sheet1')

df
```

Out[16]:

	a	b	c	d	message
0	1	2	3	4	hello
1	5	6	7	8	world
2	9	10	11	12	foo

To write pandas data to Excel format, you can pass a file path to `to_excel()` :

In [17]:
```
df.to_excel('examples/out1.xlsx')
```

Data cleaning and preparation

During the course of doing data analysis and modeling, a significant amount of time is spent on data preparation: **loading**, **cleaning**, **transforming**, and **rearranging**. Such tasks are often reported to take up 80% or more of an analyst's time.

Sometimes the way that data is stored in files or databases is not in the right format for a particular task. Fortunately, `pandas` provides you with a high-level, flexible, and fast set of tools to enable you to manipulate data into the right form.

Handling Missing Data

Missing data (**NA**, which stands for **not available**) occurs commonly in many data analysis applications. For numeric data, pandas uses the floating-point value **NaN** (not a number) to represent missing data.

With `DataFrame` objects, you may want to drop rows or columns that are all NA or only those containing any NAs.

`dropna()` by default drops any row containing a missing value:

In [18]:
```
df = pd.DataFrame([[1., 6.5, 3.],
                   [1., None, None],
                   [None, None, None],
                   [None, 6.5, 3.]])

df
```

Out[18]:

	0	1	2
0	1.0	6.5	3.0
1	1.0	NaN	NaN
2	NaN	NaN	NaN
3	NaN	6.5	3.0

In [19]:
```
df.dropna()
```

Out[19]:

	0	1	2
0	1.0	6.5	3.0

Passing `how='all'` will only drop rows that are all NA:

In [20]:
```
df.dropna(how='all')
```

Out[20]:

	0	1	2
0	1.0	6.5	3.0
1	1.0	NaN	NaN
3	NaN	6.5	3.0

To drop columns in the same way, pass `axis=1`:

In [21]:
```
df[3] = None

df
```

Out[21]:

	0	1	2	3
0	1.0	6.5	3.0	None
1	1.0	NaN	NaN	None
2	NaN	NaN	NaN	None
3	NaN	6.5	3.0	None

In [22]:
```
df.dropna(axis=1, how='all')
```

Out[22]:

	0	1	2
0	1.0	6.5	3.0
1	1.0	NaN	NaN
2	NaN	NaN	NaN
3	NaN	6.5	3.0

Filling In Missing Data

Rather than filtering out missing data, you may want to fill in the "holes" in any number of ways. The `fillna` method is the function to use.

Calling `fillna` with a constant replaces missing values with that value:

In [23]:
```
df = pd.DataFrame([[10, 30, 20, 40],
                   [8, 25, 15, 35],
                   [6, 20, 10, None],
                   [None, None, None, None],
                   [None, None, 10, 30]],
                  columns=['class participation', 'homework', 'midterm', 'final'])

df
```

Out[23]:

	class participation	homework	midterm	final
0	10.0	30.0	20.0	40.0
1	8.0	25.0	15.0	35.0
2	6.0	20.0	10.0	NaN
3	NaN	NaN	NaN	NaN
4	NaN	NaN	10.0	30.0

In [24]:
```
df.fillna(5)
```

Out[24]:

	class participation	homework	midterm	final
0	10.0	30.0	20.0	40.0
1	8.0	25.0	15.0	35.0
2	6.0	20.0	10.0	5.0
3	5.0	5.0	5.0	5.0
4	5.0	5.0	10.0	30.0

Calling `fillna()` with a dict, you can use a different fill value for each column:

In [25]:
```
df.fillna({'class participation': 5, 'final': 30})
```

Out[25]:

	class participation	homework	midterm	final
0	10.0	30.0	20.0	40.0
1	8.0	25.0	15.0	35.0
2	6.0	20.0	10.0	30.0
3	5.0	NaN	NaN	30.0
4	5.0	NaN	10.0	30.0

The **interpolation methods** can be used with fillna:

In [26]:
```
df.fillna(method='ffill')
```

Out[26]:

	class participation	homework	midterm	final
0	10.0	30.0	20.0	40.0
1	8.0	25.0	15.0	35.0
2	6.0	20.0	10.0	35.0
3	6.0	20.0	10.0	35.0
4	6.0	20.0	10.0	30.0

With fillna you can do lots of other things with a little creativity. For example, you might pass the mean or median value of a Series:

In [27]:

```
df.describe() #summary statistics
```

Out[27]:

	class participation	homework	midterm	final
count	3.0	3.0	4.000000	3.0
mean	8.0	25.0	13.750000	35.0
std	2.0	5.0	4.787136	5.0
min	6.0	20.0	10.000000	30.0
25%	7.0	22.5	10.000000	32.5
50%	8.0	25.0	12.500000	35.0
75%	9.0	27.5	16.250000	37.5
max	10.0	30.0	20.000000	40.0

In [28]:

```
df.fillna(df.mean())
```

Out[28]:

	class participation	homework	midterm	final
0	10.0	30.0	20.00	40.0
1	8.0	25.0	15.00	35.0
2	6.0	20.0	10.00	35.0
3	8.0	25.0	13.75	35.0
4	8.0	25.0	10.00	30.0

Removing Duplicates

Duplicate rows may be found in a `DataFrame` for any number of reasons. Here is an example:

In [29]:
```
df = pd.DataFrame({'k1': ['one', 'two'] * 3 + ['two'],
                   'k2': [1, 1, 2, 3, 3, 4, 4]})

df
```

Out[29]:

	k1	k2
0	one	1
1	two	1
2	one	2
3	two	3
4	one	3
5	two	4
6	two	4

The `DataFrame` method `duplicated()` returns a boolean `Series` indicating whether each row is a duplicate (has been observed in a previous row) or not:

In [30]:
```
df.duplicated()
```

Out[30]:
```
0    False
1    False
2    False
3    False
4    False
5    False
6     True
dtype: bool
```

Relatedly, `drop_duplicates()` returns a `DataFrame` where the duplicated array is False:

In [31]:
```
df.drop_duplicates()
```

Out[31]:

	k1	k2
0	one	1
1	two	1
2	one	2
3	two	3
4	one	3
5	two	4

`drop_duplicates()` considers all of the columns; alternatively, you can specify any **subset** of them to detect duplicates.

Suppose we had an additional column of values and wanted to filter duplicates only based on the 'k1' column:

In [32]:
```
df['k3'] = range(7)

df
```

Out[32]:

	k1	k2	k3
0	one	1	0
1	two	1	1
2	one	2	2
3	two	3	3
4	one	3	4
5	two	4	5
6	two	4	6

In [33]:

```
df.drop_duplicates(['k1'])
```

Out[33]:

	k1	k2	k3
0	one	1	0
1	two	1	1

`duplicated()` and `drop_duplicates()` by default keep the **first** observed value combination. Passing `keep='last'` will return the last one:

In [34]:

```
df.drop_duplicates(['k1', 'k2'], keep='last')
```

Out[34]:

	k1	k2	k3
0	one	1	0
1	two	1	1
2	one	2	2
3	two	3	3
4	one	3	4
6	two	4	6

Vectorized string functions in pandas

Cleaning up a messy dataset for analysis often requires a lot of string munging and **regularization**. To complicate matters, a column containing strings will sometimes have missing data:

In [35]:
```
df = pd.Series({'Dave': 'dave@google.com',
                'Jack': 'jack@xmu.edu.cn',
                'Steve': 'steve@gmail.com',
                'Rose': 'rose@xmu.edu.cn',
                'Tony': None})

df
```

Out[35]:
```
Dave     dave@google.com
Jack     jack@xmu.edu.cn
Steve    steve@gmail.com
Rose     rose@xmu.edu.cn
Tony                None
dtype: object
```

`Series` has array-oriented methods for string operations that skip NA values. These are accessed through Series's `str` attribute.

For example, we could check whether each email address has 'xmu.edu' in it with `str.contains`:

In [36]:
```
df.str.contains('xmu.edu')
```

Out[36]:
```
Dave     False
Jack      True
Steve    False
Rose      True
Tony      None
dtype: object
```

You can similarly **slice** strings using this syntax:

In [37]:
```
df.str[:5]
```

Out[37]:
```
Dave      dave@
Jack      jack@
Steve     steve
Rose      rose@
Tony       None
dtype: object
```

In [38]:
```
df.str.split('@')
```

Out[38]:
```
Dave      [dave, google.com]
Jack      [jack, xmu.edu.cn]
Steve     [steve, gmail.com]
Rose      [rose, xmu.edu.cn]
Tony                    None
dtype: object
```

In [39]:
```
df.str.split('@').str.get(0)
```

Out[39]:
```
Dave       dave
Jack       jack
Steve     steve
Rose       rose
Tony       None
dtype: object
```

Partial listing of vectorized string methods.

Method	Description
cat	Concatenate strings element-wise with optional delimiter
contains	Return boolean array if each string contains pattern/regex
count	Count occurrences of pattern
extract	Use a regular expression with groups to extract one or more strings from a Series of strings
endswith	Equivalent to x.endswith(pattern) for each element
startswith	Equivalent to x.startswith(pattern) for each element
findall	Compute list of all occurrences of pattern/regex for each string
get	Index into each element (retrieve i-th element)
join	Join strings in each element of the Series with passed separator

Method	Description
len	Compute length of each string
lower,upper	Convert cases;equivalent to x.lower() or x.upper() for each element
match	Use re.match with the passed regular expression on each element
replace	Replace occurrences of pattern/regex with some other string
slice	Slice each string in the Series
split	Split strings on delimiter or regular expression
strip	Trim whitespace from both sides, including newlines