Financial Data Analysis with Python

Instructor: Luping Yu

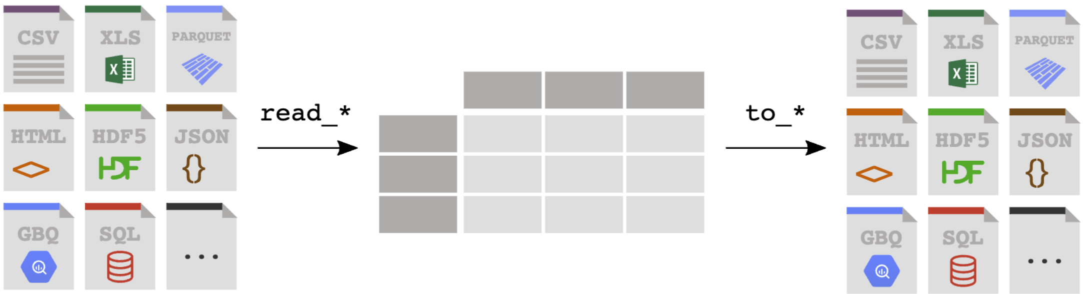
Mar 8, 2022

Lecture 03. Data Loading and Cleaning

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** and **read_table** are 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 comma-separated (CSV) text file: ex1.csv

In [244…
```
cat examples/ex1.csv
```

```
a,b,c,d,message
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.

In [245…
```
import pandas as pd

pd.read_csv('examples/ex1.csv') # 相对路径
```

Out[245]:

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

In [246…
```
# 绝对路径 (注意windows和mac绝对路径命名方式不同)
pd.read_csv('/Users/luping/desktop/teaching/examples/ex1.csv')
```

Out[246]:

	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 [247…
```
df = pd.read_csv('/Users/luping/desktop/teaching/examples/ex1.csv')

df.dtypes
```

Out[247]:
```
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 [248…
```
cat examples/ex2.csv
```

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

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 [249…
```
pd.read_csv('examples/ex2.csv', header=None)
```

Out[249]:

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

In [250…
```
pd.read_csv('examples/ex2.csv', names=['a', 'b', 'c', 'd', 'message'])
```

Out[250]:

	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 either indicate you want the column at index 4 or named 'message' using the index_col argument:

In [251…
```
names = ['a', 'b', 'c', 'd', 'message']

pd.read_csv('examples/ex2.csv', names=names, index_col='message')
```

Out[251]:

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

The parser functions have many additional arguments to help you handle the wide variety of exception file formats that occur. Consider this file: ex3.csv

In [252…
```
cat examples/ex3.csv
```

```
# Hey!
a,b,c,d,message
# Author: Luping Yu
# 厦门大学管理学院
1,2,3,4,hello
5,6,7,8,world
9,10,11,12,foo
```

For example, you can skip the first, third, and fourth rows of a file with **skiprows**:

In [253…
```
pd.read_csv('examples/ex3.csv', skiprows=[0, 2, 3])
```

Out[253]:

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

Handling missing values is an important and frequently nuanced part of the file parsing process. Missing data is usually either not present (empty string) or marked by some sentinel value. Consider this file: ex4.csv

In [254…
```
cat examples/ex4.csv
```

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

By default, pandas uses a set of commonly occurring sentinels, such as **NA** and **NULL**:

In [255…
```
pd.read_csv('examples/ex4.csv')
```

Out[255]:

	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 [256…
```
df = pd.read_csv('examples/ex4.csv')

pd.isnull(df)
```

Out[256]:

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

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

In [257…
```
df
```

Out[257]:

	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 DataFrame's to_csv method, we can write the data out to a comma-separated file:

In [258…
```
df.to_csv('examples/out1.csv')
```

In [259…
```
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 [260…
```
df.to_csv('examples/out2.csv', index=False, header=False)
```

In [261…
```
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

The optional arguments for these functions may fall into a few categories:

- Indexing

  ```
  Can treat one or more columns as the returned DataFrame, and
  whether to get column names from the file, the user, or not at
  all.
  ```
- Type inference and data conversion

  ```
  This includes the user-defined value conversions and custom list
  of missing value markers.
  ```
- Datetime parsing

  ```
  Includes combining capability, including combining date and time
  information spread over multiple columns into a single column in
  the result.
  ```
- Iterating

  ```
  Support for iterating over chunks of very large files.
  ```
- Unclean data issues

  ```
  Skipping rows or a footer, comments, or other minor things like
  numeric data with thousands separated by commas.
  ```

Because of how messy data in the real world can be, some of the data loading functions (especially read_csv) have grown very complex in their options over time. It's normal to feel overwhelmed by the number of different parameters (read_csv has over 50 as of this writing). The **online pandas documentation** has many examples about how each of them works, so if you're struggling to read a particular file, there might be a similar enough example to help you find the right parameters.

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 either the ExcelFile class or 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 or conda.

To use ExcelFile, pass the filename to pandas.read_excel:

In [262…
```
df = pd.read_excel('examples/ex1.xlsx', 'Sheet1')

df
```

Out[262]:

	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 [263…
```
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, along with the built-in Python language features, provides you with a high-level, flexible, and fast set of tools to enable you to manipulate data into the right form.

Much of the design and implementation of pandas has been driven by the needs of real-world applications.

Handling Missing Data

Missing data occurs commonly in many data analysis applications.

For numeric data, pandas uses the floating-point value **NaN** (Not a Number) to represent missing data.

In pandas, we've adopted a convention used in the R programming language by referring to missing data as **NA**, which stands for **not available**. When cleaning up data for analysis, it is often important to do analysis on the missing data itself to identify data collection problems or potential biases in the data caused by 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 [264…
```
df = pd.DataFrame([[1., 6.5, 3.],
                   [1., None, None],
                   [None, None, None],
                   [None, 6.5, 3.]])

df
```

Out[264]:

	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 [265…
```
df.dropna()
```

Out[265]:

	0	1	2
0	1.0	6.5	3.0

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

In [266…
```
df.dropna(how='all')
```

Out[266]:

	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 [267…
```
df[3] = None

df
```

Out[267]:

	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 [268…
```
df.dropna(axis=1, how='all')
```

Out[268]:

	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 (and potentially discarding other data along with it), you may want to fill in the “holes” in any number of ways. For most purposes, the fillna method is the workhorse function to use.

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

In [269…
```
df = pd.DataFrame([[10, 10, 20, 60],
                   [8, 5, 12, 48],
                   [6, 10, 14, None],
                   [None, None, None, None],
                   [None, None, 10, 30]],
                  columns=['平时分', '小作业', '大作业', '期末'])

df
```

Out[269]:

	平时分	小作业	大作业	期末
0	10.0	10.0	20.0	60.0
1	8.0	5.0	12.0	48.0
2	6.0	10.0	14.0	NaN
3	NaN	NaN	NaN	NaN
4	NaN	NaN	10.0	30.0

In [270…
```
df.fillna(0)
```

Out[270]:

	平时分	小作业	大作业	期末
0	10.0	10.0	20.0	60.0
1	8.0	5.0	12.0	48.0
2	6.0	10.0	14.0	0.0
3	0.0	0.0	0.0	0.0
4	0.0	0.0	10.0	30.0

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

In [271…
```
df
```

Out[271]:

	平时分	小作业	大作业	期末
0	10.0	10.0	20.0	60.0
1	8.0	5.0	12.0	48.0
2	6.0	10.0	14.0	NaN
3	NaN	NaN	NaN	NaN
4	NaN	NaN	10.0	30.0

In [272…

```
df.fillna({'平时分': 5, '期末': 30})
```

Out[272]:

	平时分	小作业	大作业	期末
0	10.0	10.0	20.0	60.0
1	8.0	5.0	12.0	48.0
2	6.0	10.0	14.0	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 [273…

```
df
```

Out[273]:

	平时分	小作业	大作业	期末
0	10.0	10.0	20.0	60.0
1	8.0	5.0	12.0	48.0
2	6.0	10.0	14.0	NaN
3	NaN	NaN	NaN	NaN
4	NaN	NaN	10.0	30.0

In [274…

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

Out[274]:

	平时分	小作业	大作业	期末
0	10.0	10.0	20.0	60.0
1	8.0	5.0	12.0	48.0
2	6.0	10.0	14.0	48.0
3	6.0	10.0	14.0	48.0
4	6.0	10.0	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 [275…

```
df
```

Out[275]:

	平时分	小作业	大作业	期末
0	10.0	10.0	20.0	60.0
1	8.0	5.0	12.0	48.0
2	6.0	10.0	14.0	NaN
3	NaN	NaN	NaN	NaN
4	NaN	NaN	10.0	30.0

In [276…

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

Out[276]:

	平时分	小作业	大作业	期末
0	10.0	10.000000	20.0	60.0
1	8.0	5.000000	12.0	48.0
2	6.0	10.000000	14.0	46.0
3	8.0	8.333333	14.0	46.0
4	8.0	8.333333	10.0	30.0

Removing Duplicates

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

In [277…

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

df
```

Out[277]:

	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 [278…

```
df.duplicated()
```

Out[278]:
```
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 [279…
```
df.drop_duplicates()
```

Out[279]:

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

Both of these methods by default consider 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 [280…
```
df['v1'] = range(7)

df
```

Out[280]:

	k1	k2	v1
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 [281…
```
df.drop_duplicates(['k1'])
```

Out[281]:

	k1	k2	v1
0	one	1	0
1	two	1	1

duplicated and drop_duplicates by default keep the first observed value combina- tion. Passing keep='last' will return the last one:

In [282…
```
df
```

Out[282]:

	k1	k2	v1
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 [283…

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

Out[283]:

	k1	k2	v1
0	one	1	0
1	two	1	1
2	one	2	2
3	two	3	3
4	one	3	4
6	two	4	6

Replacing Values

Filling in missing data with the fillna method is a special case of more general value replacement. replace provides a simpler and more flexible way to do so. Let's consider this Series:

In [284…

```
df = pd.Series([3321, 1742.5, -999, 381.2, 13.42])

df
```

Out[284]:

```
0     3321.00
1     1742.50
2     -999.00
3      381.20
4       13.42
dtype: float64
```

The -999 values might be sentinel values for missing data. To replace these with NA values that pandas understands, we can use replace, producing a new Series:

In [285…

```
df.replace(-999, None)
```

Out[285]:

```
0     3321.0
1     1742.5
2       None
3      381.2
4      13.42
dtype: object
```

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 [286…
```
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[286]:
```
Dave      dave@google.com
Jack      jack@xmu.edu.cn
Steve     steve@gmail.com
Rose      rose@xmu.edu.cn
Tony                 None
dtype: object
```

To cope with this, 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 [287…
```
df.str.contains('xmu.edu')
```

Out[287]:
```
Dave     False
Jack      True
Steve    False
Rose      True
Tony      None
dtype: object
```

You can similarly slice strings using this syntax:

In [288…
```
df.str[:5]
```

Out[288]:
```
Dave     dave@
Jack     jack@
Steve    steve
Rose     rose@
Tony      None
dtype: object
```

In [289…
```
df.str.split('@')
```

Out[289]:
```
Dave     [dave, google.com]
Jack     [jack, xmu.edu.cn]
Steve    [steve, gmail.com]
Rose     [rose, xmu.edu.cn]
Tony                   None
dtype: object
```

In [290…
```
df.str.split('@').str.get(0)
```

Out[290]:
```
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
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