Pandas

Dr. Noman Islam

Introduction

- It contains data structures and data manipulation tools designed to make data cleaning and analysis fast and easy in Python.
- Pandas is designed for working with tabular or heterogeneous data.
- NumPy, by contrast, is best suited for working with homogeneous numerical array data.

Series

```
In [11]: obj = pd.Series([4, 7, -5, 3])


In [13]: obj.values
Out[13]: array([ 4,  7, -5,  3])

In [14]: obj.index  # like range(4)
Out[14]: RangeIndex(start=0, stop=4, step=1)
In [15]: obj2 = pd.Series([4, 7, -5, 3], index=['d', 'b', 'a', 'c'])

In [16]: obj2
Out[16]:
d    4
b    7
a   -5
c    3
dtype: int64

In [17]: obj2.index
Out[17]: Index(['d', 'b', 'a', 'c'], dtype='object')
```

Using indexes to access data

```
In [18]: obj2['a']
Out[18]: -5

In [19]: obj2['d'] = 6

In [20]: obj2[['c', 'a', 'd']]
Out[20]:
c    3
a   -5
d    6
dtype: int64
```

Filtering

```
In [21]: obj2[obj2 > 0]
Out[21]:
d    6
b    7
c    3
dtype: int64

In [22]: obj2 * 2
Out[22]:
d    12
b    14
a   -10
c     6
dtype: int64
```

```
In [24]: 'b' in obj2
Out[24]: True
```

Creating series from dictionary

```
In [26]: sdata = {'Ohio': 35000, 'Texas': 71000, 'Oregon': 16000, 'Utah': 5000}

In [27]: obj3 = pd.Series(sdata)

In [28]: obj3
Out[28]:
Ohio      35000
Oregon    16000
Texas     71000
Utah       5000
dtype: int64
```

```
In [29]: states = ['California', 'Ohio', 'Oregon', 'Texas']

In [30]: obj4 = pd.Series(sdata, index=states)

In [31]: obj4
Out[31]:
California        NaN
Ohio          35000.0
Oregon        16000.0
Texas         71000.0
dtype: float64
```

Checking for null values

```
In [32]: pd.isnull(obj4)
Out[32]:
California     True
Ohio          False
Oregon        False
Texas         False
dtype: bool

In [33]: pd.notnull(obj4)
Out[33]:
California    False
Ohio           True
Oregon         True
Texas          True
dtype: bool
```

Adding two series

```
In [35]: obj3
Out[35]:
Ohio      35000
Oregon    16000
Texas     71000
Utah       5000
dtype: int64

In [36]: obj4
Out[36]:
California        NaN
Ohio          35000.0
Oregon        16000.0
Texas         71000.0
dtype: float64
```

```
In [37]: obj3 + obj4
Out[37]:
California         NaN
Ohio           70000.0
Oregon         32000.0
Texas         142000.0
Utah               NaN
```

Dataframe

```
data = {'state': ['Ohio', 'Ohio', 'Ohio', 'Nevada', 'Nevada', 'Nevada'],
        'year': [2000, 2001, 2002, 2001, 2002, 2003],
        'pop': [1.5, 1.7, 3.6, 2.4, 2.9, 3.2]}
frame = pd.DataFrame(data)
```

```
In [51]: frame2['state']
Out[51]:
one        Ohio
two        Ohio
three      Ohio
four     Nevada
five     Nevada
six      Nevada
Name: state, dtype: object
```

```
In [46]: frame.head()
Out[46]:
   pop   state  year
0  1.5    Ohio  2000
1  1.7    Ohio  2001
2  3.6    Ohio  2002
3  2.4  Nevada  2001
4  2.9  Nevada  2002
```

```
In [48]: frame2 = pd.DataFrame(data, columns=['year', 'state', 'pop', 'debt'],
   ....:                       index=['one', 'two', 'three', 'four',
   ....:                              'five', 'six'])
```

```
In [51]: frame2['state']
Out[51]:
one        Ohio
two        Ohio
three      Ohio
four     Nevada
five     Nevada
six      Nevada
Name: state, dtype: object
```

```
In [52]: frame2.year
Out[52]:
one      2000
two      2001
three    2002
four     2001
five     2002
six      2003
```

```
In [53]: frame2.loc['three']
Out[53]:
year     2002
state    Ohio
pop       3.6
debt      NaN
Name: three, dtype: object
```

Modifying column values

```
In [54]: frame2['debt'] = 16.5

In [55]: frame2
Out[55]:
       year   state  pop  debt
one    2000    Ohio  1.5  16.5
two    2001    Ohio  1.7  16.5
three  2002    Ohio  3.6  16.5
four   2001  Nevada  2.4  16.5
five   2002  Nevada  2.9  16.5
six    2003  Nevada  3.2  16.5

In [56]: frame2['debt'] = np.arange(6.)
```

Adding / deleting a new column

```
In [61]: frame2['eastern'] = frame2.state == 'Ohio'

In [62]: frame2
Out[62]:
       year   state  pop  debt  eastern
one    2000    Ohio  1.5   NaN     True
two    2001    Ohio  1.7  -1.2     True
three  2002    Ohio  3.6   NaN     True
four   2001  Nevada  2.4  -1.5    False
five   2002  Nevada  2.9  -1.7    False
six    2003  Nevada  3.2   NaN    False
```

```
In [63]: del frame2['eastern']

In [64]: frame2.columns
Out[64]: Index(['year', 'state', 'pop', 'debt'], dtype='object')
```

Transpose a dataframe

```
In [68]: frame3.T
Out[68]:
        2000  2001  2002
Nevada   NaN   2.4   2.9
Ohio     1.5   1.7   3.6
```

Slicing in dataframe

```
In [70]: pdata = {'Ohio': frame3['Ohio'][:-1],
   ....:          'Nevada': frame3['Nevada'][:2]}

In [71]: pd.DataFrame(pdata)
Out[71]:
      Nevada  Ohio
2000     NaN   1.5
2001     2.4   1.7
```

Values attribute

As with Series, the `values` attribute returns the data contained in the DataFrame as a two-dimensional ndarray:

```
In [74]: frame3.values
```

Table 5-1. Possible data inputs to DataFrame constructor

Type	Notes
2D ndarray	A matrix of data, passing optional row and column labels
dict of arrays, lists, or tuples	Each sequence becomes a column in the DataFrame; all sequences must be the same length
NumPy structured/record array	Treated as the "dict of arrays" case
dict of Series	Each value becomes a column; indexes from each Series are unioned together to form the result's row index if no explicit index is passed
dict of dicts	Each inner dict becomes a column; keys are unioned to form the row index as in the "dict of Series" case
List of dicts or Series	Each item becomes a row in the DataFrame; union of dict keys or Series indexes become the DataFrame's column labels
List of lists or tuples	Treated as the "2D ndarray" case
Another DataFrame	The DataFrame's indexes are used unless different ones are passed
NumPy MaskedArray	Like the "2D ndarray" case except masked values become NA/missing in the DataFrame result

Indexing, selection and filtering

```
In [117]: obj = pd.Series(np.arange(4.), index=['a', 'b', 'c', 'd'])
```

```
In [121]: obj[2:4]
Out[121]:
c    2.0
d    3.0
dtype: float64
```

```
In [122]: obj[['b', 'a', 'd']]
Out[122]:
b    1.0
a    0.0
d    3.0
dtype: float64
```

```
In [124]: obj[obj < 2]
Out[124]:
a    0.0
b    1.0
dtype: float64
```

Dropping entries from axis

```
In [105]: obj = pd.Series(np.arange(5.), index=['a', 'b', 'c', 'd', 'e'])

In [106]: obj
Out[106]:
a    0.0
b    1.0
c    2.0
d    3.0
e    4.0
dtype: float64

In [107]: new_obj = obj.drop('c')

In [108]: new_obj
Out[108]:
a    0.0
b    1.0
d    3.0
e    4.0
dtype: float64
```

Axis parameter

```
In [113]: data.drop('two', axis=1)
Out[113]:
          one  three  four
Ohio        0      2     3
Colorado    4      6     7
Utah        8     10    11
New York   12     14    15

In [114]: data.drop(['two', 'four'], axis='columns')
Out[114]:
          one  three
Ohio        0      2
Colorado    4      6
Utah        8     10
New York   12     14
```

Indexing and slicing

```
In [119]: obj['b']
Out[119]: 1.0
```

```
In [120]: obj[1]
Out[120]: 1.0
```

```
In [121]: obj[2:4]
Out[121]:
c    2.0
d    3.0
dtype: float64
```

```
In [122]: obj[['b', 'a', 'd']]
Out[122]:
b    1.0
a    0.0
d    3.0
dtype: float64
```

```
In [123]: obj[[1, 3]]
Out[123]:
b    1.0
d    3.0
dtype: float64
```

Selection with loc and iloc

```
In [137]: data.loc['Colorado', ['two', 'three']]
Out[137]:
two      5
three    6
Name: Colorado, dtype: int64
```

```
In [138]: data.iloc[2, [3, 0, 1]]
Out[138]:
four    11
one      8
two      9
Name: Utah, dtype: int64
```

```
In [139]: data.iloc[2]
Out[139]:
one       8
two       9
three    10
```

Apply function

```
In [193]: f = lambda x: x.max() - x.min()

In [194]: frame.apply(f)
Out[194]:
b    1.802165
d    1.684034
e    2.689627
dtype: float64
```

Function application and mapping

```
In [192]: np.abs(frame)
Out[192]:
               b         d         e
Utah    0.204708  0.478943  0.519439
Ohio    0.555730  1.965781  1.393406
Texas   0.092908  0.281746  0.769023
Oregon  1.246435  1.007189  1.296221
```

Descriptive statistics

```
In [233]: df.sum(axis='columns')
Out[233]:
a    1.40
b    2.60
c     NaN
d   -0.55
dtype: float64
```

```
In [234]: df.mean(axis='columns', skipna=False)
Out[234]:
a       NaN
b     1.300
c       NaN
```

```
In [237]: df.describe()
Out[237]:
            one       two
count  3.000000  2.000000
mean   3.083333 -2.900000
std    3.493685  2.262742
min    0.750000 -4.500000
25%    1.075000 -3.700000
50%    1.400000 -2.900000
75%    4.250000 -2.100000
max    7.100000 -1.300000
```

```
In [236]: df.cumsum()
Out[236]:
    one  two
a  1.40  NaN
b  8.50 -4.5
c   NaN  NaN
d  9.25 -5.8
```

```
In [235]: df.idxmax()
Out[235]:
one    b
two    d
dtype: object
```

Method	Description
count	Number of non-NA values
describe	Compute set of summary statistics for Series or each DataFrame column
min, max	Compute minimum and maximum values
argmin, argmax	Compute index locations (integers) at which minimum or maximum value obtained, respectively
idxmin, idxmax	Compute index labels at which minimum or maximum value obtained, respectively
quantile	Compute sample quantile ranging from 0 to 1
sum	Sum of values
mean	Mean of values
median	Arithmetic median (50% quantile) of values
mad	Mean absolute deviation from mean value
prod	Product of all values
var	Sample variance of values
std	Sample standard deviation of values
skew	Sample skewness (third moment) of values
kurt	Sample kurtosis (fourth moment) of values
cumsum	Cumulative sum of values
cummin, cummax	Cumulative minimum or maximum of values, respectively
cumprod	Cumulative product of values
diff	Compute first arithmetic difference (useful for time series)
pct_change	Compute percent changes

Reading and writing data

```
In [9]: df = pd.read_csv('examples/ex1.csv')

In [10]: df
Out[10]:
   a   b   c   d message
0  1   2   3   4   hello
1  5   6   7   8   world
2  9  10  11  12     foo
```

```
In [13]: pd.read_csv('examples/ex2.csv', header=None)
Out[13]:
   0   1   2   3      4
0  1   2   3   4  hello
1  5   6   7   8  world
2  9  10  11  12    foo
```

Filtering out missing values

```
In [19]: data = pd.DataFrame([[1., 6.5, 3.], [1., NA, NA],
   ....:                      [NA, NA, NA], [NA, 6.5, 3.]])

In [20]: cleaned = data.dropna()

In [21]: data
Out[21]:
     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 [22]: cleaned
Out[22]:
     0    1    2
0  1.0  6.5  3.0
```

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

```
In [23]: data.dropna(how='all')
Out[23]:
     0    1    2
```

Filling missing data

```
In [33]: df.fillna(0)
Out[33]:
```

Reading data in pandas

Table 6-1. Parsing functions in pandas

Function	Description
`read_csv`	Load delimited data from a file, URL, or file-like object; use comma as default delimiter
`read_table`	Load delimited data from a file, URL, or file-like object; use tab (`'\t'`) as default delimiter
`read_fwf`	Read data in fixed-width column format (i.e., no delimiters)
`read_clipboard`	Version of `read_table` that reads data from the clipboard; useful for converting tables from web pages
`read_excel`	Read tabular data from an Excel XLS or XLSX file
`read_hdf`	Read HDF5 files written by pandas
`read_html`	Read all tables found in the given HTML document
`read_json`	Read data from a JSON (JavaScript Object Notation) string representation
`read_msgpack`	Read pandas data encoded using the MessagePack binary format
`read_pickle`	Read an arbitrary object stored in Python pickle format
`read_sas`	Read a SAS dataset stored in one of the SAS system's custom storage formats
`read_sql`	Read the results of a SQL query (using SQLAlchemy) as a pandas DataFrame
`read_stata`	Read a dataset from Stata file format
`read_feather`	Read the Feather binary file format

Optional arguments

- *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.

Examples

```
df = pd.read_csv('examples/ex1.csv')

pd.read_table('examples/ex1.csv', sep=',')

pd.read_csv('examples/ex2.csv', header=None)

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

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

- Passing regular expression as separator
  - result = pd.read_table('examples/ex3.txt', sep='\s+')
- Skipping rows
  - pd.read_csv('examples/ex4.csv', skiprows=[0, 2, 3])

Handling null values

```
result = pd.read_csv('examples/ex5.csv', na_values=['NULL'])
```

```
In [31]: sentinels = {'message': ['foo', 'NA'], 'something': ['two']}

In [32]: pd.read_csv('examples/ex5.csv', na_values=sentinels)
```

Table 6-2. Some read_csv/read_table function arguments

Argument	Description
`path`	String indicating filesystem location, URL, or file-like object
`sep` or `delimiter`	Character sequence or regular expression to use to split fields in each row
`header`	Row number to use as column names; defaults to 0 (first row), but should be `None` if there is no header row
`index_col`	Column numbers or names to use as the row index in the result; can be a single name/number or a list of them for a hierarchical index
`names`	List of column names for result, combine with `header=None`

Argument	Description
`skiprows`	Number of rows at beginning of file to ignore or list of row numbers (starting from 0) to skip.
`na_values`	Sequence of values to replace with NA.
`comment`	Character(s) to split comments off the end of lines.
`parse_dates`	Attempt to parse data to `datetime`; `False` by default. If `True`, will attempt to parse all columns. Otherwise can specify a list of column numbers or name to parse. If element of list is tuple or list, will combine multiple columns together and parse to date (e.g., if date/time split across two columns).
`keep_date_col`	If joining columns to parse date, keep the joined columns; `False` by default.
`converters`	Dict containing column number of name mapping to functions (e.g., `{'foo': f}` would apply the function `f` to all values in the `'foo'` column).
`dayfirst`	When parsing potentially ambiguous dates, treat as international format (e.g., 7/6/2012 -> June 7, 2012); `False` by default.
`date_parser`	Function to use to parse dates.
`nrows`	Number of rows to read from beginning of file.
`iterator`	Return a `TextParser` object for reading file piecemeal.
`chunksize`	For iteration, size of file chunks.
`skip_footer`	Number of lines to ignore at end of file.
`verbose`	Print various parser output information, like the number of missing values placed in non-numeric columns.
`encoding`	Text encoding for Unicode (e.g., `'utf-8'` for UTF-8 encoded text).
`squeeze`	If the parsed data only contains one column, return a Series.
`thousands`	Separator for thousands (e.g., `','` or `'.'`).

Reading text file in pieces

```
In [33]: pd.options.display.max_rows = 10
```

```
In [36]: pd.read_csv('examples/ex6.csv', nrows=5)
```

```
chunker = pd.read_csv('examples/ex6.csv', chunksize=1000)
```

Iterating over chunk

```
chunker = pd.read_csv('examples/ex6.csv', chunksize=1000)

tot = pd.Series([])
for piece in chunker:
    tot = tot.add(piece['key'].value_counts(), fill_value=0)

tot = tot.sort_values(ascending=False)
```

Writing file

```
data.to_csv('examples/out.csv')
```

```
import sys

data.to_csv(sys.stdout, sep='|')
```

```
data.to_csv(sys.stdout, na_rep='NULL')
```

```python
data.to_csv(sys.stdout, index=False, header=False)
```

```python
data.to_csv(sys.stdout, index=False, columns=['a', 'b', 'c'])
```

Working with delimited format

```
import csv
f = open('examples/ex7.csv')

reader = csv.reader(f)
```

```
with open('examples/ex7.csv') as f:
    lines = list(csv.reader(f))
```

```
header, values = lines[0], lines[1:]
```

Writing csv data

```
with open('mydata.csv', 'w') as f:
    writer = csv.writer(f, dialect=my_dialect)
    writer.writerow(('one', 'two', 'three'))
    writer.writerow(('1', '2', '3'))
    writer.writerow(('4', '5', '6'))
    writer.writerow(('7', '8', '9'))
```

JSON format

```
obj = """
{"name": "Wes",
 "places_lived": ["United States", "Spain", "Germany"],
 "pet": null,
 "siblings": [{"name": "Scott", "age": 30, "pets": ["Zeus", "Zuko"]},
              {"name": "Katie", "age": 38,
               "pets": ["Sixes", "Stache", "Cisco"]}]
}
"""
```

- The pandas.read_json can automatically convert JSON datasets in specific arrange- ments into a Series or DataFrame
- The default options for pandas.read_json assume that each object in the JSON array is a row in the table

```
data = pd.read_json('examples/example.json')
```

```
print(data.to_json())
```

```
print(data.to_json(orient='records'))
```

Web scraping

```
conda install lxml
pip install beautifulsoup4 html5lib
```

- The pandas.read_html function has a number of options, but by default it searches for and attempts to parse all tabular data contained within tags.

```
In [73]: tables = pd.read_html('examples/fdic_failed_bank_list.html')

In [74]: len(tables)
Out[74]: 1

In [75]: failures = tables[0]

In [76]: failures.head()
Out[76]:
                     Bank Name             City  ST   CERT  \
0                  Allied Bank         Mulberry  AR     91
1  The Woodbury Banking Company        Woodbury  GA  11297
2        First CornerStone Bank  King of Prussia  PA  35312
```

Saving in binary format

```
frame.to_pickle('examples/frame_pickle')
```

```
pd.read_pickle('examples/frame_pickle')
```

HD5 format

```
In [92]: frame = pd.DataFrame({'a': np.random.randn(100)})

In [93]: store = pd.HDFStore('mydata.h5')

In [94]: store['obj1'] = frame

In [95]: store['obj1_col'] = frame['a']

In [96]: store
```

Excel format

```
xlsx = pd.ExcelFile('examples/ex1.xlsx')

pd.read_excel(xlsx, 'Sheet1')
```

```
writer = pd.ExcelWriter('examples/ex2.xlsx')

frame.to_excel(writer, 'Sheet1')

writer.save()
```

Interacting with Web API

```
import requests

url = 'https://api.github.com/repos/pandas-dev/pandas/issues'

resp = requests.get(url)

resp
<Response [200]>
```

```
data = resp.json()

data[0]['title']
```

```python
issues = pd.DataFrame(data, columns=['number', 'title',
                                     'labels', 'state'])
```

Interacting with database

```
In [121]: import sqlite3

In [122]: query = """
   ......: CREATE TABLE test
   ......: (a VARCHAR(20), b VARCHAR(20),
   ......:  c REAL,        d INTEGER
   ......: );"""

In [123]: con = sqlite3.connect('mydata.sqlite')

In [124]: con.execute(query)
Out[124]: <sqlite3.Cursor at 0x7f6b12a50f10>

In [125]: con.commit()
```

```python
data = [('Atlanta', 'Georgia', 1.25, 6),
        ('Tallahassee', 'Florida', 2.6, 3),
        ('Sacramento', 'California', 1.7, 5)]

stmt = "INSERT INTO test VALUES(?, ?, ?, ?)"

con.executemany(stmt, data)
<sqlite3.Cursor at 0x7f6b15c66ce0>

con.commit()
```

```python
cursor = con.execute('select * from test')

rows = cursor.fetchall()
```

```
In [133]: cursor.description
Out[133]:
(('a', None, None, None, None, None, None),
 ('b', None, None, None, None, None, None),
 ('c', None, None, None, None, None, None),
 ('d', None, None, None, None, None, None))

In [134]: pd.DataFrame(rows, columns=[x[0] for x in cursor.description])
Out[134]:
```

Data Cleaning and Preparation

Handling missing data

```
In [10]: string_data = pd.Series(['aardvark', 'artichoke', np.nan, 'avocado'])

In [11]: string_data
Out[11]:
0     aardvark
1    artichoke
2          NaN
3      avocado
dtype: object

In [12]: string_data.isnull()
Out[12]:
0    False
1    False
2     True
3    False
dtype: bool
```

Table 7-1. NA handling methods

Argument	Description
`dropna`	Filter axis labels based on whether values for each label have missing data, with varying thresholds for how much missing data to tolerate.
`fillna`	Fill in missing data with some value or using an interpolation method such as `'ffill'` or `'bfill'`.
`isnull`	Return boolean values indicating which values are missing/NA.
`notnull`	Negation of `isnull`.

```
In [15]: from numpy import nan as NA

In [16]: data = pd.Series([1, NA, 3.5, NA, 7])

In [17]: data.dropna()
Out[17]:
0    1.0
2    3.5
4    7.0
dtype: float64
```

```
In [18]: data[data.notnull()]
Out[18]:
0    1.0
2    3.5
4    7.0
dtype: float64
```

dropna options

- With DataFrame objects, things are a bit more complex. 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
- Passing how='all' will only drop rows that are all NA
- To drop columns in the same way, pass axis=1
- df.dropna(thresh=2)

Filling In Missing Data

- Calling fillna with a constant replaces missing values with that value
- Calling fillna with a dict, you can use a different fill value for each column
- df.fillna({1: 0.5, 2: 0})
- fillna returns a new object, but you can modify the existing object in-place:
  - _ = df.fillna(0, inplace=True)

```
In [37]: df = pd.DataFrame(np.random.randn(6, 3))

In [38]: df.iloc[2:, 1] = NA

In [39]: df.iloc[4:, 2] = NA

In [40]: df
Out[40]:
          0         1         2
0  0.476985  3.248944 -1.021228
1 -0.577087  0.124121  0.302614
2  0.523772       NaN  1.343810
3 -0.713544       NaN -2.370232
4 -1.860761       NaN       NaN
5 -1.265934       NaN       NaN
```

```
In [41]: df.fillna(method='ffill')
Out[41]:
          0         1         2
0  0.476985  3.248944 -1.021228
1 -0.577087  0.124121  0.302614
2  0.523772  0.124121  1.343810
3 -0.713544  0.124121 -2.370232
4 -1.860761  0.124121 -2.370232
5 -1.265934  0.124121 -2.370232
```

```
In [42]: df.fillna(method='ffill', limit=2)
Out[42]:
          0         1         2
0  0.476985  3.248944 -1.021228
1 -0.577087  0.124121  0.302614
2  0.523772  0.124121  1.343810
3 -0.713544  0.124121 -2.370232
4 -1.860761       NaN -2.370232
5 -1.265934       NaN -2.370232
```

Table 7-2. fillna function arguments

Argument	Description
`value`	Scalar value or dict-like object to use to fill missing values
`method`	Interpolation; by default `'ffill'` if function called with no other arguments
`axis`	Axis to fill on; default `axis=0`
`inplace`	Modify the calling object without producing a copy
`limit`	For forward and backward filling, maximum number of consecutive periods to fill

Removing Duplicates

- data.duplicated()
- Relatedly, drop_duplicates returns a DataFrame where the duplicated array is False
- You can specify any subset of them to detect duplicates
- data.drop_duplicates(['k1'])
- duplicated and drop_duplicates by default keep the first observed value combina- tion. Passing keep='last' will return the last one

Map

```
In [52]: data = pd.DataFrame({'food': ['bacon', 'pulled pork', 'bacon',
   ....:                              'Pastrami', 'corned beef', 'Bacon',
   ....:                              'pastrami', 'honey ham', 'nova lox'],
   ....:                     'ounces': [4, 3, 12, 6, 7.5, 8, 3, 5, 6]})
```

```
meat_to_animal = {
  'bacon': 'pig',
  'pulled pork': 'pig',
  'pastrami': 'cow',
  'corned beef': 'cow',
  'honey ham': 'pig',
  'nova lox': 'salmon'
}
```

```
lowercased = data['food'].str.lower()

In [57]: data['animal'] = lowercased.map(meat_to_animal)

In [59]: data['food'].map(lambda x: meat_to_animal[x.lower()])
```

Replace

- data.replace(-999, np.nan)
- data.replace([-999, -1000], np.nan)
- data.replace([-999, -1000], [np.nan, 0])
- data.replace({-999: np.nan, -1000: 0})

Map with index

```
In [66]: data = pd.DataFrame(np.arange(12).reshape((3, 4)),
   ....:                     index=['Ohio', 'Colorado', 'New York'],
   ....:                     columns=['one', 'two', 'three', 'four'])

In [67]: transform = lambda x: x[:4].upper()

In [68]: data.index.map(transform)
Out[68]: Index(['OHIO', 'COLO', 'NEW '], dtype='object')


In [69]: data.index = data.index.map(transform)
```

Other index methods

- data.rename(index=str.title, columns=str.upper)
- data.rename(index={'OHIO': 'INDIANA'}, columns={'three': 'peekaboo'})
- data.rename(index={'OHIO': 'INDIANA'}, inplace=True)

Discretization and binning

- Let’s divide data into bins
  - ages = [20, 22, 25, 27, 21, 23, 37, 31, 61, 45, 41, 32]
  - bins = [18, 25, 35, 60, 100]
  - cats = pd.cut(ages, bins)
- Lets see the code for each data and categories:
  - cats.codes
  - cats.categories
- What are the counts for each category?
  - pd.value_counts(cats)

- Making the left side a close interval
  - pd.cut(ages, [18, 26, 36, 61, 100], right=False)
- Providing your own group names:
  - group_names = ['Youth', 'YoungAdult', 'MiddleAged', 'Senior']
  - pd.cut(ages, bins, labels=group_names)

Detecting and filtering outliers

- data.describe()
- Suppose you wanted to find values in one of the columns exceeding 3 in absolute value:
  - col = data[2]
  - col[np.abs(col) > 3]
- To select all rows having a value exceeding 3 or –3, you can use the any method on a boolean DataFrame:
  - data[(np.abs(data) > 3).any(1)]
- Here is code to cap values outside the inter- val –3 to 3
  - data[np.abs(data) > 3] = np.sign(data) * 3

Permutation and random sampling

- Calling permutation with the length of the axis you want to permute produces an array of integers indicating the new ordering:
- df = pd.DataFrame(np.arange(5 * 4).reshape((5, 4))
- sampler = np.random.permutation(5)
- df.take(sampler)
- To select a random subset without replacement, you can use the sample method on Series and DataFrame:
  - df.sample(n=3)

- To generate a sample with replacement (to allow repeat choices), pass replace=True to sample:
  - choices = pd.Series([5, 7, -1, 6, 4])
  - draws = choices.sample(n=10, replace=True)