9 Data Analysis with pandas

9.1 Introduction to pandas

9.1.1 What is pandas?

pandas is a widely-used, open-source Python library for data manipulation and analysis. Unlike NumPy, its basic array-like data structure, the `DataFrame` object, can contain heterogeneous data types (floats, integers, strings, dates, etc.) that may be structured in a hierarchy and indexed. It provides a large number of vectorized functions for cleaning, transforming and aggregating data efficiently using similar idioms to those used by NumPy. Its name derives from the term "panel data" (otherwise known as "longitudinal data"), which refers to data sets of several variables followed over multiple time periods for the same individual.

The pandas homepage, https://pandas.pydata.org/, contains details of the latest release and how to download and install pandas. In this chapter we follow the common convention of importing the library as the alias `pd`:

```
import pandas as pd
```

The key pandas data structures are `Series` and `DataFrame`, representing a one-dimensional sequence of values and a data table, respectively. Their basic properties and use will be described in this section, followed by more advanced features and applications to examples in subsequent sections. pandas is a large, complex library with a lot of functionality; this chapter aims to cover the basics: more detailed examples are provided on the website accompanying the book.

9.1.2 `Series`

In its simplest form, a `Series` may be created in the same way as a one-dimensional NumPy array:

```
In [x]: river_lengths = pd.Series([6300, 6650, 6275, 6400])
In [x]: river_lengths
Out[x]:
0    6300
1    6650
2    6275
3    6400
dtype: int64
```

The `Series` can be given a name and `dtype`:

```
In [x]: river_lengths = pd.Series([6300, 6650, 6275, 6400], name='Length /km',
                                  dtype=float)
Out[x]:
0    6300.0
1    6650.0
2    6275.0
3    6400.0
Name: Length /km, dtype: float64
```

Unlike a NumPy array, however, each element in a pandas `Series` is associated with an *index*. Since we did not set the index explicitly here, a default integer sequence (starting at 0) is used for the index:

```
In [x]: river_lengths.index
Out[x]: RangeIndex(start=0, stop=4, step=1)
```

`RangeIndex` is a pandas object that works in a memory-efficient way like Python's `range` built-in to provide a monotonic integer sequence. It is often useful to refer to the rows of a `Series` with some other label than an integer index. Explicit indexing of the entries can be achieved by passing a sequence as the `index` argument or by creating the `Series` from a dictionary:

```
In [x]: river_lengths = pd.Series(data=[6300, 6650, 6275, 6400],
   ...:                           index=['Yangtze', 'Nile', 'Mississippi', 'Amazon'],
   ...:                           name='Length /km')
```

or:

```
In [x]: river_lengths = pd.Series(data={'Yangtze': 6300, 'Nile': 6650,
   ...:                                 'Mississippi': 6275, 'Amazon': 6400},
   ...:                           name='Length /km')
In [x]: river_lengths
Out[x]:
Yangtze        6300
Nile           6650
Mississippi    6275
Amazon         6400
Name: Length /km, dtype: int64
```

This allows a nicely expressive way of referring to `Series` entries using the index labels instead of integers; either individually:

```
In [x]: river_lengths['Nile']
Out[x]: 6650
```

instead of `river_lengths[1]`; or from another sequence:

```
In [x]: river_lengths[['Amazon', 'Nile', 'Yangtze']]
Out[x]:
Amazon     6400
Nile       6650
Yangtze    6300
Name: Length /km, dtype: int64
```

instead of `river_lengths[ [3, 1, 0] ]`. Python-style slicing also works as expected:

```
In [x]: river_lengths[2::-1]
Out[x]:
Mississippi    6275
Nile           6650
Yangtze        6300
Name: Length /km, dtype: int64
```

It is even possible to use a slice-like notation for the index labels, but note that in this case the endpoint is *inclusive*:

```
In [x]: river_lengths['Nile':'Amazon']
Out[x]:
Nile           6650
Mississippi    6275
Amazon         6400
Name: Length /km, dtype: int64
```

Providing the index label is a valid Python identifier, one can refer to a row as an *attribute* of the `Series`:

```
In [x]: river_lengths.Mississippi
Out[x]: 6275
```

It is, of course, possible to do numerical operations on `Series` data, in a vectorized fashion, as for NumPy arrays:

```
In [x]: KM_TO_MILES = 0.621371
In [x]: river_lengths *= KM_TO_MILES
In [x]: river_lengths.name = 'Length /miles'
In [x]: river_lengths
Out[x]:
Yangtze        3914.637300
Nile           4132.117150
Mississippi    3899.103025
Amazon         3976.774400
Name: Length /miles, dtype: float64
```

In the above we have also chosen to update the `Series` object's `name` attribute. Note that the `dtype` has also changed appropriately from `int64` to `float64` to accommodate the new values.

Comparison operations and filtering a `Series` with a boolean operation creates a new `Series`:

```
In [x]: river_lengths > 4000
Out[x]:
Nile            True
Amazon         False
Yangtze        False
Mississippi    False
Name: Length /miles, dtype: bool

In [x]: river_lengths[river_lengths <= 4000]
Out[x]:
Amazon         3976.774400
Yangtze        3914.637300
Mississippi    3899.103025
Name: Length /miles, dtype: float64
```

Tests for `membership` of a `Series` examine the *index*, not the *values*:

```
In [x]: 'Yangtze' in river_lengths
Out[x]: True
```

`Series` can be *sorted*, either by their index or their values, using `Series.sort_index` and `Series.sort_values`, respectively. By default these methods return a new `Series`, but they can also be used to update the original `Series` with the argument `inplace=True`. A further argument, `ascending`, can be `True` (the default) or `False` to set the ordering:

```
In [x]: river_lengths.sort_index()
Out[x]:
Amazon          3976.774400
Mississippi     3899.103025
Nile            4132.117150
Yangtze         3914.637300
Name: Length /miles, dtype: float64

In [x]: river_lengths.sort_values(ascending=False, inplace=True)
In [x]: river_lengths
Out[x]:
Nile            4132.117150
Amazon          3976.774400
Yangtze         3914.637300
Mississippi     3899.103025
Name: Length /miles, dtype: float64
```

When two series are combined, they are aligned by index label.

```
In [x]: masses = pd.Series({'Ganymede': 1.482e23,
                            'Callisto': 1.076e23,
                            'Io': 8.932e22,
                            'Europa': 4.800e22,
                            'Moon': 7.342e22,
                            'Earth': 5.972e24}, name='mass /kg')
In [x]: radii = pd.Series({'Ganymede': 2.634e6,
                           'Io': 1.822e6,
                           'Moon': 1.737e6,
                           'Earth': 6.371e6}, name='radius /m')
In [x]: from scipy.constants import G
In [x]: surface_g = G * masses / radii**2
In [x]: surface_g.name = 'surface gravity /m.s-2'
In [x]: surface_g.index.name = 'Body'
In [x]: surface_g
Body
Callisto         NaN
Earth       9.819650
Europa           NaN
Ganymede    1.425634
Io          1.795740
Moon        1.624075
Name: surface gravity /m.s-2, dtype: float64
```

Note that where no correspondence can be made within the indexes (an index label in one `Series` that is missing from the other), the result is "Not a Number" (NaN). The methods `isnull` and `notnull` test for this:

```
In [x]: surface_g.isnull()
Out[x]:
Body
Callisto     True
Earth       False
Europa       True
Ganymede    False
Io          False
Moon        False
Name: surface gravity /m.s-2, dtype: bool
```

To return a list without any missing values, either filter with `surface_g[surface_g.notnull()]` or use the `dropna` method:

```
In [x]: surface_g.dropna()
Out[x]:
Body
Earth       9.819650
Ganymede    1.425634
Io          1.795740
Moon        1.624075
Name: surface gravity /m.s-2, dtype: float64
```

Finally, to convert a `Series` into a NumPy `ndarray` (dropping the index and other metadata), use the `values` property:

```
In [x]: surface_g.values
Out[x]: array([       nan, 9.81964974,        nan, 1.42563409, 1.79573967,
       1.62407526])
```

Example E9.1 NaN entries can be replaced in a pandas `Series` with a specified value using the `fillna` method:

```
In [x]: ser1 = pd.Series({'b': 2, 'c': -5, 'd': 6.5}, index=list('abcd'))
In [x]: ser1
Out[x]:
a    NaN
b    2.0
c   -5.0
d    6.5
dtype: float64

In [x]: ser1.fillna(1, inplace=True)
In [x]: ser1
Out[x]:
a    1.0
b    2.0
c   -5.0
d    6.5
dtype: float64
```

Infinities (represented by the floating-point `inf` value) can be replaced with the `replace` method, which takes a scalar or sequence of values and substitutes them with another, single value:

```
In [x]: ser2 = pd.Series([-3.4, 0, 0, 1], index=ser1.index)
In [x]: ser2
Out[x]:
a   -3.4
b    0.0
c    0.0
d    1.0
dtype: float64

In [x]: ser3 = ser1 / ser2
In [x]: ser3
Out[x]:
a   -0.294118
b         inf
c        -inf
d    6.500000
dtype: float64

In [x]: ser3.replace([np.inf, -np.inf], 0)
Out[x]:
a   -0.294118
b    0.000000
c    0.000000
d    6.500000
dtype: float64
```

(Assuming NumPy has been imported with `import numpy as np`.)

9.1.3 DataFrame

Creating a DataFrame

A `DataFrame` is a two-dimensional table of data that can be thought of as an ordered set of `Series` columns, which all have the same index. To create a simple `DataFrame` from a dictionary, assign value sequences[1] to column name keys:

```
In [x]: data = {'mass': [1.482e23, 1.076e23, 8.932e22, 4.800e22, 7.342e22],
                'radius': [2.634e6, None, 1.822e6, None, 1.737e6],
                'parent': ['Jupiter', 'Jupiter', 'Jupiter', 'Jupiter', 'Earth']
               }
In [x]: index = ['Ganymede', 'Callisto', 'Io', 'Europa', 'Moon']
In [x]: df = pd.DataFrame(data, index=index)
In [x]: df
Out[x]:
                  mass     radius   parent
Ganymede  1.482000e+23  2634000.0  Jupiter
Callisto  1.076000e+23        NaN  Jupiter
Io        8.932000e+22  1822000.0  Jupiter
Europa    4.800000e+22        NaN  Jupiter
Moon      7.342000e+22  1737000.0    Earth
```

Values which were `None` in the data have been assigned to NaN in the `DataFrame`. We may wish to rename a column or index row: to do this, call `rename`, declaring which

[1] The (unspecified) units here are SI units: kg and m.

axis (`'index'` [the same as `'rows'`, and the default] or `'columns'`[2]) contains the label(s) to be renamed, and passing a dictionary mapping each original label to its replacement. Remember to set `inplace=True` if you want the orginal `DataFrame` modified rather than a new copy returned. For example,

```
In [x]: df.rename({'parent': 'planet'}, axis='columns', inplace=True)
In [x]: df.rename({'Moon': 'The Moon'}) # change a row index label
Out[x]:
                 mass     radius   planet
Ganymede  1.482000e+23  2634000.0  Jupiter
Callisto  1.076000e+23        NaN  Jupiter
Io        8.932000e+22  1822000.0  Jupiter
Europa    4.800000e+22        NaN  Jupiter
The Moon  7.342000e+22  1737000.0    Earth
```

This last statement has returned a new `DataFrame` but not altered the original one, `df`.

Accessing Rows, Columns and Cells

An individual column can be obtained by indexing or by attribute (if its name is a valid Python identifier):

```
In [x]: df['mass']      # or df.mass
Out[x]:
Ganymede    1.482000e+23
Callisto    1.076000e+23
Io          8.932000e+22
Europa      4.800000e+22
Moon        7.342000e+22
Name: mass, dtype: float64
```

Since this column is just a pandas `Series`, individual values can be retrieved by position or reference to the index label:

```
In [x]: df['mass'][2]
Out[x]: 8.932e+22
In [x]: df['mass']['Io']    # or df['mass'].Io or df.mass.Io
Out[x]: 8.932e+22
```

Now, for retrieving columns and individual values this is fine, but assignment raises a warning:

```
In [x]: df['radius']['Callisto'] = 2.410e6
/Users/christian/envs/py37/bin/ipython:1: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame
...
```

In this case, it has worked:

```
In [x]: df['radius']['Callisto']
Out[x]: 2410000.0
```

but the message is a general warning that "chained indexing" (`[..][..]`) can lead to unpredictable results when used for assignment: depending on how the data are stored

[2] One can also refer to the rows and columns of a `DataFrame` with `axis=0` and `axis=1`, respectively.

in memory, it is possible for the indexing expression to yield a *copy* of the data rather than a view. Assigning to the copy rather than modifying the data in-place is what the `SettingWithCopyWarning` is warning could happen. *Chained indexing for assignment operations should be avoided.*

Two `DataFrame` methods, `loc` and `iloc`, can be used to reliably access and assign to columns, rows and cells; using them is strongly recommended. `loc` selects by row and column *labels*:

```
In [x]: df.loc['Europa']
Out[x]:
mass      4.8e+22
radius        NaN
planet    Jupiter
Name: Europa, dtype: object
```

The single row of data indexed by the label Europa is returned as a `Series`. If only a subset of the columns are required, pass their names in a sequence to the second axis:[3]

```
In [x]: df.loc['Europa', ['mass', 'planet']]
Out[x]:
mass      4.8e+22
planet    Jupiter
Name: Europa, dtype: object
```

Slicing, "fancy" indexing and boolean indexing are all supported by `loc`:

```
In [x]: df.loc[:, 'mass']  # the same as df['mass'] - returns a Series
Out[x]:
Ganymede    1.482000e+23
Callisto    1.076000e+23
Io          8.932000e+22
Europa      4.800000e+22
Moon        7.342000e+22
Name: mass, dtype: float64

In [x]: df.loc['Ganymede':'Io', ['mass', 'radius']]
Out[x]:
                  mass     radius
Ganymede  1.482000e+23  2634000.0
Callisto  1.076000e+23  2410000.0
Io        8.932000e+22  1822000.0

In [x]: df.loc[['Moon', 'Europa'], 'planet']
Out[x]:
Moon        Earth
Europa    Jupiter
Name: planet, dtype: object

In [x]: df.loc[df.planet=='Jupiter', 'radius']
Out[x]:
Ganymede    2634000.0
Callisto    2410000.0
```

[3] Note that whilst chained indexing refers to a cell in column, row order: `df[col][row]`, `loc` locates cells the other way round: `df.loc[row, col]` or `df.loc[row][col]`.

```
Io          1822000.0
Europa      NaN
Name: radius, dtype: float64
```

The value of a single cell can therefore be retrieved from the row and column labels:

```
In [x]: df.loc['Europa', 'mass']
Out[x]: 4.8e+22
```

This is the safe way to modify data in a `DataFrame`:

```
In [x]: df.loc['Europa', 'radius'] = 1.561e6     # no warning, data changed in place
In [x]: df.loc['Europa']
Out[x]:
mass         4.8e+22
radius     1.561e+06
parent       Jupiter
Name: Europa, dtype: object
```

It is common to use `loc` in combination with boolean indexing to filter rows by column values. For example, the masses of Jupiter's moons:

```
In [x]: df.loc[df.planet=='Jupiter', 'mass']
Out[x]:
Ganymede    1.482000e+23
Callisto    1.076000e+23
Io          8.932000e+22
Europa      4.800000e+22
Name: mass, dtype: float64
```

The rows corresponding to moons with radii less than 2000 km:

```
In [x]: df.loc[df.radius < 2.e6]
Out[x]:
                mass     radius   planet
Io      8.932000e+22  1822000.0  Jupiter
Europa  4.800000e+22  1561000.0  Jupiter
Moon    7.342000e+22  1737000.0    Earth
```

The second method, `iloc`, retrieves data by numerical index position:

```
In [x]: df.iloc[1]          # the second row
Out[x]:
mass      1.076e+23
radius     2.41e+06
parent       Jupiter
Name: Callisto, dtype: object

In [x]: df.iloc[:, [1, 2]]    # all rows, second and third columns
Out[x]:
             radius   planet
Ganymede  2634000.0  Jupiter
Callisto  2410000.0  Jupiter
Io        1822000.0  Jupiter
Europa    1561000.0  Jupiter
Moon      1737000.0    Earth

In [x]: df.iloc[-1, 1]      # last row, second column
Out[x]: 1737000.0
```

For single scalar values, there are also `at` and `iat`:

```
In [x]: df.at['Moon', 'mass']    # same as df.loc['Moon', 'mass']
Out[x]: 7.342e+22
In [x]: df.iat[-1, 0]             # same as df.iloc[-1, 0]
Out[x]: 7.342e+22
```

Example E9.2 There is a potential source of confusion when using `loc` for a `Series` or `DataFrame` with an integer index: it is important to remember that *`loc` always refers to the `index labels`*, whereas `iloc` takes a (zero-based) integer *location index*:

```
In [x]: df = pd.DataFrame(np.arange(12).reshape(4, 3) + 10,
                          index=[1, 2, 3, 4], columns=list('abc'))
In [x]: df
Out[x]:
    a   b   c
1  10  11  12
2  13  14  15
3  16  17  18
4  19  20  21

In [x]: df.loc[1]   # the row with index *label* 1 (the first row)
Out[x]:
a    10
b    11
c    12
Name: 1, dtype: int64

In [x]: df.iloc[1]  # the row with index *location* 1 (the row labeled 2)
a    13
b    14
c    15
Name: 2, dtype: int64
```

Note also that index labels do not have to be unique:

```
In [x]: df.index = [1, 2, 2, 3]    # change the index labels
In [x]: df
Out[x]:
    a   b   c
1  10  11  12
2  13  14  15
2  16  17  18
3  19  20  21

In [x]: df.loc[2]     # a DataFrame: all rows labeled 2
Out[x]:
    a   b   c
2  13  14  15
2  16  17  18

In [x]: df.iloc[2]    # a Series: there is only one row located at index 2
Out[x]:
a    16
b    17
```

```
c    18
Name: 2, dtype: int64
```

Combining Series and DataFrames

Another way to create a DataFrame is from a nested dictionary or from a dictionary of Series. In each case, the outer dictionary keys contain the *column* names; Series and inner dictionaries end up as rows:

```
boeing_wingspan = pd.Series({'B747-8': 68.4, 'B777-9': 64.8, 'B787-10': 60.12},
                            name='wingspan')
boeing_length = pd.Series({'B747-8': 76.3, 'B777-9': 76.7, 'B787-10': 68.28},
                            name='length')
boeing_range = pd.Series({'B777-9': 13940, 'B787-10': 11910},
                            name='range', dtype=float)

# Create a DataFrame from a dictionary of Series.
df_boeing = pd.DataFrame({'wingspan': boeing_wingspan, 'length': boeing_length,
                          'range': boeing_range})

# Create a DataFrame from a dictionary of dictionaries.
df_airbus = pd.DataFrame({'range': {'A350-1000': 16100, 'A380-800': 14800},
                          'wingspan': {'A350-1000': 64.75, 'A380-800': 79.75},
                          'length': {'A350-1000': 73.8, 'A380-800': 72.72} })

In [x]: df_boeing
Out[x]:
         wingspan  length    range
B747-8      68.40   76.30      NaN
B777-9      64.80   76.70  13940.0
B787-10     60.12   68.28  11910.0

In [x]: df_airbus
Out[x]:
           range  wingspan  length
A350-1000  16100     64.75   73.80
A380-800   14800     79.75   72.72
```

Note that missing values in the columns become NaN in the DataFrame. To concatenate two DataFrames, use pd.concat:[4]

```
In [x]: pd.concat((df_airbus, df_boeing))
Out[x]:
           length    range  wingspan
A350-1000   73.80  16100.0     64.75
A380-800    72.72  14800.0     79.75
B747-8      76.30      NaN     68.40
B777-9      76.70  13940.0     64.80
B787-10     68.28  11910.0     60.12
```

[4] Note that the concat and append functions require data to be copied into a new DataFrame and for large data sets can be slow and memory-inefficient. In this case, if at all possible, it is better to pre-allocate an empty DataFrame of the right size and to insert data directly into it.

(`df_airbus.append(df_boeing)` would give the same result.)

To add a single column to a `DataFrame`, assign a sequence of values or a `Series` object:

```
In [x]: df_airbus['speed'] = [950, 903]
In [x]: df_airbus

Out[x]:
           range  wingspan  length  speed
A350-1000  16100     64.75   73.80    950
A380-800   14800     79.75   72.72    903
```

Concatenating `DataFrames` with different columns fills the unknown values with NaN:

```
In [x]: df_aircraft = pd.concat((df_airbus, df_boeing))
In [x]: df_aircraft

Out[x]:
           length    range  speed  wingspan
A350-1000   73.80  16100.0  950.0     64.75
A380-800    72.72  14800.0  903.0     79.75
B747-8      76.30      NaN    NaN     68.40
B777-9      76.70  13940.0    NaN     64.80
B787-10     68.28  11910.0    NaN     60.12
```

Note that retrieving a `Series` as a row or column returns a *view* on the `DataFrame`, so changes to this `Series` will be reflected in it:

```
In [x]: speeds = df_aircraft['speed']
In [x]: speeds['B747-8','B787-10'] = 903, 956   # changes df_aircraft data
In [x]: jumbo = df_aircraft.loc['B747-8']
In [x]: jumbo.range = 15000                   # changes df_aircraft data
In [x]: df_aircraft
Out[x]:
           length    range  speed  wingspan
A350-1000   73.80  16100.0  950.0     64.75
A380-800    72.72  14800.0  903.0     79.75
B747-8      76.30  15000.0  903.0     68.40
B777-9      76.70  13940.0    NaN     64.80
B787-10     68.28  11910.0  956.0     60.12
```

To remove a column from a `DataFrame`, call Python's `del` keyword:

```
In [x]: del df_aircraft['speed']         # NB but not del df_aircraft.speed
In [x]: df_aircraft

Out[x]:
           length    range  wingspan
A350-1000   73.80  16100.0     64.75
A380-800    72.72  14800.0     79.75
B747-8      76.30  15000.0     68.40
B777-9      76.70  13940.0     64.80
B787-10     68.28  11910.0     60.12
```

The `drop` function can be used to selectively remove rows and columns from a `DataFrame`. A new object is returned unless `inplace=True` is specified:

```
In [x]: df_aircraft.drop(['A350-1000', 'A380-800'])    # drop rows by default
Out[x]:
          length    range  wingspan
B747-8     76.30  15000.0     68.40
B777-9     76.70  13940.0     64.80
B787-10    68.28  11910.0     60.12

In [x]: df_aircraft.drop(['length', 'wingspan'], axis='columns', inplace=True)
In [x]: df_aircraft
Out[x]:
              range
A350-1000   16100.0
A380-800    14800.0
B747-8      15000.0
B777-9      13940.0
B787-10     11910.0
```

9.1.4 Sorting, Arithmetic and Statistics

As might be expected, many of the most useful functions for data analysis are available from within pandas.

Example E9.3 The file `india-data.csv`, available at https://scipython.com/eg/bak, contains columns of demographic data on the 36 states and union territories (UTs) of India. When read in with:

```
In [x]: df = pd.read_csv('india-data.csv', index_col=0)
```

(more on this method in the next section), the `DataFrame` produced contains an `Index` of State/UT name and columns:

```
In [x]: df.index
Out[x]:
Index(['Uttar Pradesh', 'Maharashtra', 'Bihar', 'West Bengal',
       ...
       'Dadra and Nagar Haveli', 'Daman and Diu', 'Lakshadweep'],
      dtype='object', name='State/UT')

In [x]: df.columns
Out[x]:
Index(['Male Population', 'Female Population', 'Area (km2)',
       'Male Literacy (%)', 'Female Literacy (%)', 'Fertility Rate'],
      dtype='object')
```

We can quickly inspect the `DataFrame` with `df.head(n)`, which outputs the first n rows (or five rows if n is not specified):

```
In [x]: df.head()
Out[x]:
                Male Population  ...  Female Literacy (%)
State/UT                         ...
Uttar Pradesh         104480510  ...                59.26
Maharashtra            58243056  ...                75.48
Bihar                  54278157  ...                53.33
```

```
West Bengal            46809027  ...               71.16
Madhya Pradesh         37612306  ...               60.02

[5 rows x 5 columns]
```

pandas makes it straightforward to compute new columns for our `DataFrame`:

```
In [x]: df['Population'] = df['Male Population'] + df['Female Population']
In [x]: total_pop = df['Population'].sum()
In [x]: print(f'Total population: {total_pop:,d}')
Total population: 1,210,754,977

In [x]: df['Population Density (km-2)'] = df['Population'] / df['Area (km2)']
In [x]: df.loc['West Bengal', 'Population Density (km-2)']
Out[x]: 1028.440091490896       # population density of West Bengal

In [x]: total_pop / df['Area (km2)'].sum()
Out[x]: 368.3195047153525       # mean population density
```

Maximum and minimum values are obtained in the same way as in NumPy, for example:

```
In [x]: df['Male Literacy (%)'].min()
Out[x]: 73.39
```

Perhaps more usefully, `idxmin` and `idxmax` return the index *label(s)* of the minimum and maximum values, respectively:

```
In [x]: df['Area (km2)'].idxmax()   # largest state/UT by area
Out[x]: 'Rajasthan'
```

Naturally, the value returned can be passed to `df.loc` to obtain the entire row. For example, the row corresponding to the most densely populated state / UT:

```
In [x]: df.loc[df['Population Density (km-2)'].idxmax()]
Out[x]:
Male Population              8887326
Female Population            7800615
Area (km2)                   1484
Male Literacy (%)            91.03
Female Literacy (%)          80.93
Population                   16687940
Population Density (km-2)    1.124524e+04
Name: Delhi, dtype: float64
```

Correlation statistics between `DataFrames` or `Series` can be calculated with the `corr` function:

```
In [x]: df['Female Literacy (%)'].corr(df['Fertility Rate'])
Out[x]: -0.7361949271996956
```

In this case (two columns of data being compared), a single correlation coefficient is produced. More generally, the correlation *matrix* is returned as a new `DataFrame`. pandas can be used to quickly produce a variety of simple, labeled plots and charts from a `DataFrame` with a family of `df.plot` methods. By default, these use the Matplotlib backend, so the syntax is the same as presented in Chapter 7. For example,

```
In [x]: df.plot.scatter('Female Literacy (%)', 'Fertility Rate')
```

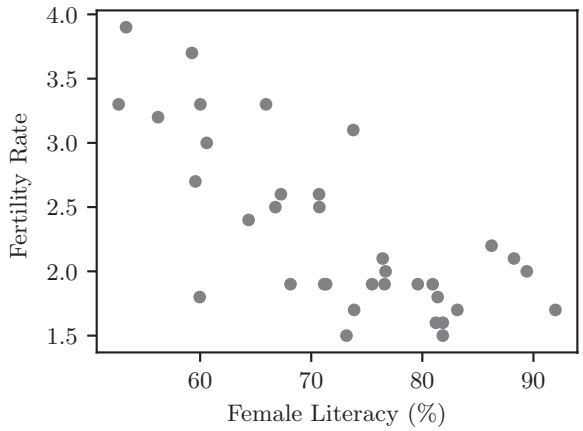

Figure 9.1 Scatter plot of fertility rate against female literacy for the 36 states and UTs of India.

Figure 9.1 shows the resulting plot.

9.2 Reading and Writing `Series` and `DataFrames`

9.2.1 Reading Text Files

Delimited Text Files

The core method for reading text files of data into a `DataFrame` is `pd.read_csv`. This works in much the same way as NumPy's `genfromtxt` method, but with additional functionality for naming columns and setting the `DataFrame` index. It takes no fewer than 49 possible arguments, but the most important are described below.[5]

- `filepath_or_buffer` (required): The path to the file to read: this can be a local file or a URL for fetching data across the internet.
- `sep`: The column delimiter; by default `','`, but use `'\s+'` for whitespace-delimited columns, `'\t'` for tab-delimiters, or `None` to force pandas to try to infer the delimiter. See also `delim_whitespace`.
- `delimiter`: An alias for `sep`.
- `header`: The row numbers (indices) to use for the column names. The default is `header=0`: use the first row for the column names. *Note*: if the file does not have a header, specify `header=None` and set the column names with the `names` argument.

[5] See the documentation at https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_csv.html for a complete description.

- names: A sequence of unique column names to use. If the file contains no header, set header=None in addition to setting names.
- index_col: The column(s) to use as the row labels in the DataFrame.
- usecols: A sequence of column indices (as for NumPy's loadtxt method) or column names identifying the columns to be read into the DataFrame.
- squeeze: If the data required consist of a single column, then squeeze=True will return a Series instead of the default, a DataFrame.
- converters: A dictionary of functions for converting the values in specified columns in the input file into data values for the DataFrame. The dictionary keys can be column indices or column names.
- skiprows: An integer giving the number of lines at the start of the file to skip over before reading the data or a sequence giving the indices of rows to skip.
- skipfooter: The number of rows at the bottom of the file to skip (by default, 0).
- nrows: The number of rows of the file to read: this is useful for reading a subset of lines from a very large file for testing or exploring its data.
- na_values: A string or sequence of strings to treat as NaN values, in addition to the default values which include 'NaN', 'NA', 'NULL' and '#N/A' (see the documentation for a full list).
- parse_dates: Set to True to parse the index column(s) as a sequence of datetime objects (see Section 9.3.2); other options are available for this argument (see the online documentation).
- comment: Specify a single character, such as '#', which, when found at the start of a line, signals that the whole line is to be ignored.
- skip_blank_lines: The default, True, skips over blank lines in the input file; set to False to interpret these as a row of NaN values instead.
- delim_whitespace: Can be set to True instead of specifying sep='\s+' to indicate that the data columns are separated by whitespace.

Example E9.4 The file ionization-energies.csv, available to download at https://scipython.com/eg/baq, contains the ionization energies (in eV) of some of the elements of the periodic table:

```
Ionization Energies (eV) of the first few elements of the periodic table
Element, IE1, IE2, IE3, IE4, IE5
H, 13.59844
He, 24.58741, 54.41778
Li, 5.39172, 75.64018, 122.45429
Be, 9.3227, 18.21116, 153.89661, 217.71865
B, 8.29803, 25.15484, 37.93064, 259.37521, 340.22580
C, 11.26030, 24.38332, 47.8878, 64.4939, 392.087
N, 14.53414, 29.6013, 47.44924, 77.4735, 97.8902
O, 13.61806, 35.11730, 54.9355, 77.41353, 113.8990
F, 17.42282, 34.97082, 62.7084, 87.1398, 114.2428
Ne, 21.5646, 40.96328, 63.45, 97.12, 126.21
Na, 5.13908, 47.2864, 71.6200, 98.91, 138.40
```

These data can be read into a `DataFrame` as follows. Here, we suppose that we are only interested in the first two periods of the periodic table and the first four ionization energies:

```
❶ In [x]: df = pd.read_csv('ionization-energies.csv', skiprows=1, index_col=0,
   ...:                    usecols=range(5), nrows=11)
❷ In [x]: df.columns = df.columns.str.strip()
In [x]: print('Second ionization energy of Li: {} eV'.format(df.loc['Li'].IE2))

Second ionization energy of Li: 75.64018 eV
```

❶ Note that the `usecols` argument includes the column we want to set to the `DataFrame` index and `nrows` includes the column headers (but not the skipped rows).

❷ The whitespace around the column names is not automatically removed. pandas provides a variety of methods for manipulating strings within the `str` "accessor" namespace, which can be applied to all the column names in one statement; this is faster than using `rename`:

```
df.rename(columns=lambda s: s.strip(), inplace=True)
```

Example E9.5 The following text file, available at https://scipython.com/eg/bao, contains data concerning 13 vitamins important for human health.

```
List of vitamins, their solubility (in fat or water) and recommended dietary
allowances for men / women.
Data from the US Food and Nutrition Board, Institute of Medicine, National
Academies

Vitamin A   Fat     900ug/700ug

Vitamin B1  Water   1.2mg/1.1mg
Vitamin B2  Water   1.3mg/1.1mg
Vitamin B3  Water   16mg/14mg
Vitamin B5  Water   5mg
Vitamin B6  Water   1.5mg/1.4mg
Vitamin B7  Water   30ug
Vitamin B9  Water   400ug
Vitamin B12 Water   2.4ug

Vitamin C   Water   90mg/75mg
Vitamin D   Fat     15ug
Vitamin E   Fat     15mg
Vitamin K   Fat     110ug/120ug
--- Data for guidance only, consult your physician ---
```

The recommended (daily) dietary allowances are listed in either of two units in the final column; sometimes these are different for men and women. If we wish to parse this column into an average value in μg, we can use a converter function as in the following code.

Listing 9.1 Reading in a text table of vitamin data

```
import pandas as pd

def average_rda_in_micrograms(col):
    def ensure_micrograms(s):
        if s.endswith('ug'):
            return float(s[:-2])
        elif s.endswith('mg'):
            return float(s[:-2]) * 1000
        raise ValueError(f'Unrecognised units in {s}')
    fields = col.split('/')
    return sum([ensure_micrograms(s) for s in fields]) / len(fields)

df = pd.read_csv('vitamins.txt', delim_whitespace=True, skiprows=4,
                 skipfooter=1, header=None, usecols=(1, 2, 3),
                 converters={'RDA': average_rda_in_micrograms},
                 names=['Vitamin', 'Solubility', 'RDA'],
                 index_col=0
                )
```

In this code, the four header rows and one footer row are skipped (blank lines are skipped automatically); the `Index` is set to the first *used* column (`index_col=0`, identifying the vitamin). The converter function averages the numerical values encountered (after conversion to µg), where multiple values are assumed to be separated by a solidus character (/).

Fixed-Width Text Files

The method `read_fwf` reads fixed-width formatted files. The field widths are passed as a list of tuples to the argument `colspecs`, giving the half-open intervals of the fields to read in from each line; i.e. `(i, j)` refers to the field from index `i` to index `j - 1`. Alternatively, if the intervals are contiguous, a list of field widths can be passed to the argument `widths`.

We return to the `np.genfromtxt` example of Section 6.2.3. The following short file, `data.txt`, consists of four columns with widths 2, 1, 9 and 3 characters (spaces are indicated with '␣'):

```
␣12␣␣100.231.03
␣11␣1201.842.04
␣11␣␣␣99.324.02
```

To read in this file with pandas, use either:

```
df = pd.read_fwf('data.txt',
                 colspecs=[(0, 2), (2, 3), (3, 12), (12, 15)],  header=None)
```

or, since the intervals are contiguous:

```
df = pd.read_fwf('data.txt', widths=(2, 1, 9, 3), header=None)
```

to give the `DataFrame`:

```
   0  1         2     3
0  1  2   100.231  0.03
1  1  1  1201.842  0.04
2  1  1    99.324  0.02
```

9.2.2 Writing Text Files

The `DataFrame` method `to_csv` outputs its data to a text file, formatted according to the arguments summarized below.[6]

- `path_or_buf`: A file path or file object to output to; if `None`, the `DataFrame` is returned as string.
- `sep`: The single-character field-delimiter (defaults to `','`).
- `na_rep`: The string to use to represent missing data (defaults to the empty string, `''`).
- `float_format`: The C-style format specifier (see Section 2.3.7) for floating-point numbers.
- `columns`: A sequence identifying the columns to output.
- `header`: By default, `True`, indicating that column names should be output; can be set to `False` or a list of column names.
- `index`: By default, `True`, indicating that row names should be output.
- `compression`: One of `'infer'`, `'gzip'`, `'bz2'`, `'zip'`, `'xz'`, `None` to specify whether and how to compress the output file. The default is `'infer'`: pandas determines the intended compression method from the filename extension.

Example E9.6 To write a comma-separated file containing data on vitamins from the `DataFrame` created in Example E9.5 `to_csv` can be used as follows:

```
df.to_csv('vitamins.csv', float_format='%.1f', columns=['Solubility', 'RDA'])
```

The file written is:

```
Vitamin,Solubility,RDA
A,Fat,800.0
B1,Water,1150.0
B2,Water,1200.0
B3,Water,15000.0
B5,Water,5000.0
B6,Water,1450.0
B7,Water,30.0
B9,Water,400.0
B12,Water,2.4
C,Water,82500.0
D,Fat,15.0
E,Fat,15000.0
K,Fat,115.0
```

---

[6] Full documentation is available at https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.to_csv.html

	A	B	C	D	E	F	G
1		Structural properties of some diatomic molecules					
2	**Molecule**	**Bond length /A**	**we /cm-1**	**wexe /cm-1**	**De /kJ.mol-1**		
3	I_2	2.666	214.5	0.614	224.1042237		
4	O_2	1.20752	1580.19	11.98	623.3408948		
5	Cl_2	1.987	559.7	2.67	350.8836826		
6	F_2	1.41193	916.64	11.236	223.640111		
7	N_2	1.09768	2358.57	14.324	1161.440719		
8	CO	1.128323	2169.81358	13.28831	1059.592595		
9	NO	1.15077	1904.20	14.075	770.4430432		
10	Data from the NIST Chemistry WebBook: Constants of Diatomic Molecules						
11	https://webbook.nist.gov						
12							
13							
14							
15							
16							
17							

Figure 9.2 An Excel sheet containing data concerning the structural properties of some diatomic molecules.

9.2.3 Microsoft Excel Files

pandas is able to read `DataFrames` from Excel files with both `.xls` and `.xlsx` extensions with the function `pd.read_excel`. You may need to install the xlrd package[7] separately from your Python package manager or using `pip` on the command line with, for example:

```
pip install xlrd
```

The file path to the Excel document is passed as the first argument to `read_excel`. Most of the additional arguments already described for `read_csv` function in the same way, except that `usecols` can be passed either a list of columnn indices or a string giving the range of Excel column labels: for example: `'B:K'`, `'A,D,G:K'`.

By default, only the first sheet of the file is used; to read in from a different sheet or more than one sheet, pass one or more indexes or sheet names to the argument `sheet_name`.

Example E9.7 The Excel file `bond-lengths.xlsx`, available online at https://scipython.com/eg/bbk, contains data on the bond lengths, vibrational constants and dissociation energies of some diatomic molecules. The single sheet is named `'Diatomics'`. Column A contains the molecular formula; the first row is a title, and the second row contains the column names. There is also a footer of two lines, as shown in Figure 9.2.

The following statement can be used to read in a `DataFrame` containing these data:

---

[7] https://pypi.org/project/xlrd/

```
df = pd.read_excel('bond-lengths.xlsx',
        index_col=0,            # the first column contains the index labels
        skipfooter=2,           # ignore the last two lines of the sheet
        header=1,               # take the column names from the second row
        usecols='A:E',          # use Excel columns labeled A-E
        sheet_name='Diatomics'  # take data from this sheet
        )

print(df)
```

```
          Bond length /A    we /cm-1  wexe /cm-1  De /kJ.mol-1
Molecule
I2              2.666000   214.50000     0.61400    224.104224
O2              1.207520  1580.19000    11.98000    623.340895
Cl2             1.987000   559.70000     2.67000    350.883683
F2              1.411930   916.64000    11.23600    223.640111
N2              1.097680  2358.57000    14.32400   1161.440719
CO              1.128323  2169.81358    13.28831   1059.592595
NO              1.150770  1904.20000    14.07500    770.443043
```

Should you be in the unfortunate position of needing to *write* to an Excel spreadsheet file, use `to_excel`, as in the following example. Again, there may be a dependency to resolve: if the `openpyxl` module[8] is not available, you can install through your package manager or using `pip`:

```
pip install openpyxl
```

Example E9.8 To create some data to write to a file, the following program generates a `DataFrame` with the height of a projectile launched at three different angles (in the columns) as a function of time (rows):

Listing 9.2 The height of a projectile as a function of time

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# Acceleration due to gravity, m.s-2.
g = 9.81

# Time grid, s.
t = np.linspace(0, 5, 500)
# Projectile launch angles, deg.
theta0 = np.array([30, 45, 80])
# Projectile launch speen, m.s-1.
v0 = 20

def z(t, v0, theta0):
    """Return the height of the projectile at time t > 0."""
    return -g/2 * t**2 + v0*t*np.sin(theta0)
```

[8] https://pypi.org/project/openpyxl/

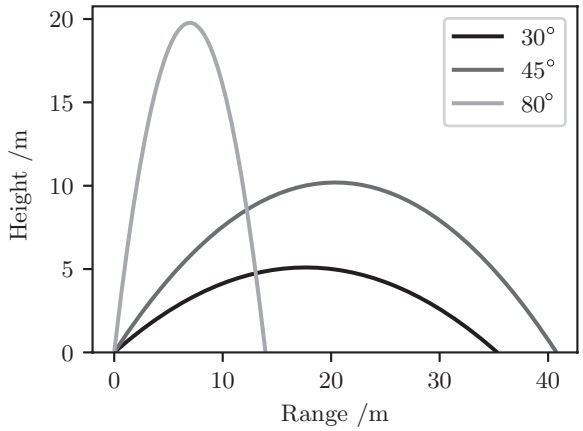

Figure 9.3 Trajectories of a projectile launched with $v_0 = 20\ \mathrm{m\,s^{-1}}$ at three different angles.

```
def x(t, v0, theta0):
    """Return the range of the projectile at time t > 0."""
    return v0 * t * np.cos(theta0)

# An empty DataFrame with columns for the different launch angles.
df = pd.DataFrame(columns=theta0, index=t)
# Populate df with the projectile heights as a function of time.
for theta in theta0:
    df[theta] = z(t, v0, np.radians(theta))
# Once the projectile has landed (z <= 0), set the height data as invalid.
df[df<=0] = np.nan

# Create a Matplotlib figure with the trajectories plotted.
fig, ax = plt.subplots()
for theta in theta0:
    ax.plot(x(t, v0, np.radians(theta)), df[theta], label=f'${theta}^\circ$')

# The maximum height obtained by the projectile for each value of theta0.
heights = df.max()
print(heights)
# Set the y-limits with a bit of padding at the top; label the axes.
ax.set_ylim(0, heights.max()*1.05)
ax.set_xlabel('Range /m')
ax.set_ylabel('Height /m')
ax.legend()
plt.show()
```

Figure 9.3 shows the plot of the trajectories that is produced by this code.

To save the `DataFrame` `df` to an Excel file in a single sheet, use `to_excel`:

```
df.to_excel('projectile.xlsx', sheet_name='Dependence on angle')
```

To write an Excel file with more than one sheet, create a `pd.ExcelWriter` object and call `to_excel` for each pandas object to output:

```
with pd.ExcelWriter('projectile2.xlsx') as writer:
    for theta in theta0:
        # Only retain the valid data for each trajectory.
        ser = df[theta].dropna()
        # Change the Series index to be the range instead of time.
        ser.index = x(ser.index, v0, np.radians(theta))
        ser.to_excel(writer, sheet_name=f'{theta} deg')
```

9.2.4 Web Scraping

The pandas function `read_html` can be used to parse web pages for data contained in HTML tables. A list of `DataFrames` is returned, and the default arguments for this function do a pretty good job on most well-formed pages. The most useful arguments are listed below.

- `io`: A URL, filepath or file object from which to parse the HTML
- `match`: An optional string to search for within the table: only tables containing this string are parsed and returned.[9]
- `header`: The row index to be used for the column headers; the default, `None`, uses the HTML `<th>` header cells, if present.
- `index_col`: The column(s) to use as the row labels in the `DataFrame`.
- `attrs`: A dictionary of HTML attributes to identify the required table; for example, `attrs={'id': 'data-table'}`.
- `thousands`: The separator used in grouping the digits of large numbers; defaults to `','`.
- `decimal`: The character used in denoting the decimal point; the default is `'.'`, as used in the United States, United Kingdom, Australia, Japan, China and South Korea; non-British European countries often use `','`.
- `na_values`: String(s) used to denote NaN data, as for `read_csv`.

Example E9.9 At the time of writing, the first table on the Wikipedia page https://en.wikipedia.org/wiki/List_of_wine-producing_regions contains columns of the rank, country name and wine production for the principal wine-producing countries in the world. To parse it with pandas:

```
In [x]: dfs = pd.read_html(
              'https://en.wikipedia.org/wiki/List_of_wine-producing_regions',
❶             index_col=1, match="Wine production by country")
❷ In [x]: dfs[0].head()
Out[x]:
                                    Rank  Production(tonnes)
Country(with link to wine article)
```

[9] `match` can be a *regular expression*.

```
Italy                1          4796900
France               2          4607850
Spain                3          4293466
United States        4          3300000
China                5          1700000
```

❶ In this case, the table is identified by a match to the the text inside the <caption> element of the first <table> on the page.
❷ dfs is a list containing a single item, the DataFrame parsed from the matching table.

9.2.5 Exercises

Problems

P9.2.1 The web page at https://scipython.com/ex/bab gives tables for the total ozone column amounts in October in "Dobson units,"[10] and the concentrations of two chlorofluorocarbon (CFC) compounds, "F11" and "F12", in parts per trillion by volume (pptv) for the years 1957 – 1984; see Farman *et al.*, *Nature* **315**, 207 (1985).

Read in and parse these data, and plot them on a suitable chart.

P9.2.2 At https://en.wikipedia.org/wiki/Abundances_of_the_elements_(data_page) Wikipedia gives a list of element abundances for the Sun and solar system in an HTML table (amongst other, similar data). Use pandas' `read_html` method to read in and parse the Kaye and Laby data (column headed "Y1") and plot a bar chart demonstrating *Oddo–Harkins rule*: that elements with even atomic numbers are more abundant than those with neighboring odd atomic numbers.

P9.2.3 The *Hertzsprung–Russell diagram* classifies stars on a scatter plot: each star is represented as a point with an x-coordinate of effective temperature and a y-coordinate of *luminosity*, a measure of the star's radiated electromagnetic power. The page at https://scipython.com/ex/bak can be used to obtain a version of the HYG-database,[11] which provides data on 119614 stars. Read in these data with pandas and plot a Hertzsprung–Russell diagram. The luminosity column is identified as `'lum'` in the header and the star temperature can be calculated from its *color index* (also referred to as $(B-V)$ and identified as the column labeled `'ci'`) using the Ballesteros formula:

$$T\ /\mathrm{K} = 4600\left(\frac{1}{0.92(B-V)+1.7}+\frac{1}{0.92(B-V)+0.62}\right).$$

[10] The Dobson unit is defined as the thickness, in units of 0.01 mm, that a layer of pure gas would form at standard conditions for temperature and pressure from its total column amount in the atmosphere above a region of the Earth's surface.

[11] https://github.com/astronexus/HYG-Database, released under a Creative Commons Attribution-ShareAlike license

Note that the luminosity is best visualized on a logarithmic scale and the temperature axis is usually plotted in reverse (decreasing temperature towards the right-hand side of the diagram).

P9.2.4 Transport for London (TfL) is the UK local government body responsible for the public transport system of Greater London; they make available an Excel document, available from the link at https://scipython.com/ex/bam, which provides statistics about the usage of the underground network (the Tube) in the form of entry and exit passenger numbers for a "typical" day at each station over the years 2007–2017.

Read in this document with pandas, and analyze it to determine: (a) the busiest station on a typical weekday in 2017; (b) the station with the greatest percentage increase in passengers over the period 2007–2017; (c) the station with the largest relative difference in passenger numbers between the working week and a typical Sunday in 2017.

P9.2.5 The HITRAN database (https://hitran.org) provides a list of molecular line intensities for modeling radiative transmission in planetary atmospheres. Its native format consists of 160-character records of fixed-width fields.

Use pandas to read in the file `CO2-transitions.par`, available from https://scipython.com/ex/ban (where a description of the fields can also be found). Plot line intensity against wavelength for these transitions in the infrared region of the spectrum (λ = 10 mm to 700 nm, corresponding to wavenumber $\tilde{\nu}$ = 1 cm^{-1} to about 14 000 cm^{-1}), where carbon dioxide (CO_2) is responsible for a significant fraction of the greenhouse effect in Earth's atmosphere.

9.3 More Advanced Indexing

9.3.1 Hierarchical Indexes with `MultiIndex`

A `DataFrame` is an intrinsically two-dimensional array of data: to represent data in higher dimensions, it is common to use hierarchical indexing to represent multiple *levels* within a single index. If the data are sparse or heterogeneous, this is much more efficient than creating a multidimensional NumPy array. For example, consider a data set concerning the mean monthly temperature and rainfall in five European cities. This could be considered three-dimensional, the dimensions being "city," "month" and "data type" (this last meaning either temperature or rainfall). For five cities (Paris, Berlin, Vienna, London, Madrid) and four months (Jan, Apr, Jul, Oct), there would therefore be 40 data points in total.

We *could* create a conventional single-level index consisting of `(city, month)` tuples, but it wouldn't be very convenient or flexible. Instead, we can create a hierarchical index with two levels from a sequence of two-item tuples using `pd.MultiIndex.from_tuples`:

```
In [x]: cities = ('Paris', 'Berlin', 'Vienna', 'London', 'Madrid')
In [x]: months = ('Jan', 'Apr', 'Jul', 'Oct')
In [x]: index = pd.MultiIndex.from_tuples(
```

```
   ...:     (city, month) for city in cities for month in months)
In [x]: index
Out[x]:
MultiIndex([( 'Paris', 'Jan'),
            ( 'Paris', 'Apr'),
            ( 'Paris', 'Jul'),
            ( 'Paris', 'Oct'),
            ('Berlin', 'Jan'),
            ...
            ('Madrid', 'Jul'),
            ('Madrid', 'Oct')],
           )
```

`MultiIndex`es of this form (the Cartesian product of two or more sequences) are so common that there is a convenience function, `from_product`, for their creation:

```
index = pd.MultiIndex.from_product((cities, months))
```

We can create a `DataFrame` with this index by assigning an array of data in the shape `(20, 2)`:

```
In [x]: index.names = ['City', 'Month']

In [x]: # Mean monthly temperature (degC) for each city in each of Jan, Apr, Jul,
Oct.
In [x]: temps = [[4.9, 11.5, 20.5, 13.0], [0.1, 9.0, 19.1, 9.4],
   ...:          [0.3, 10.7, 20.8, 10.2], [5.2, 9.9, 18.7, 12.0],
   ...:          [6.3, 12.9, 25.6, 15.1]
   ...:         ]
In [x]: # Mean monthly rainfall (mm) for each city in each of Jan, Apr, Jul,
Oct.
In [x]: rainfall = [[51.0, 51.8, 62.3, 61.5], [37.2, 33.7, 52.5, 32.2],
   ...:             [38., 45., 70., 38.], [55.2, 43.7, 44.5, 68.5],
   ...:             [33., 45., 12., 60.]
   ...:            ]

In [x]: arr = np.array((temps, rainfall)).reshape((2, 20)).T
In [x]: df = pd.DataFrame(arr, index=index, columns=['Mean temperature /degC',
   ...:                                              'Mean rainfall /mm'])

In [x]: df
Out[x]:
              Mean temperature /degC  Mean rainfall /mm
City   Month
Paris  Jan                      4.9               51.0
       Apr                     11.5               51.8
       Jul                     20.5               62.3
       Oct                     13.0               61.5
Berlin Jan                      0.1               37.2
       Apr                      9.0               33.7
       Jul                     19.1               52.5
       Oct                      9.4               32.2
Vienna Jan                      0.3               38.0
       Apr                     10.7               45.0
       Jul                     20.8               70.0
       Oct                     10.2               38.0
London Jan                      5.2               55.2
       Apr                      9.9               43.7
       Jul                     18.7               44.5
       Oct                     12.0               68.5
```

```
Madrid Jan                       6.3               33.0
       Apr                      12.9               45.0
       Jul                      25.6               12.0
       Oct                      15.1               60.0
```

The `loc` method can be used to index into the `DataFrame`'s `MultiIndex`:

```
In [x]: df.loc['Vienna']
Out[x]:
       Mean temperature /degC  Mean rainfall /mm
Month
Jan                       0.3               38.0
Apr                      10.7               45.0
Jul                      20.8               70.0
Oct                      10.2               38.0

In [x]: df.loc[('Paris', 'Jul')]
Out[x]:
Mean temperature /degC    20.5
Mean rainfall /mm         62.3
Name: (Paris, Jul), dtype: float64

In [x]: df.loc[('Paris', 'Jul'), 'Mean rainfall /mm']
Out[x]: 62.3
```

To slice a `MultiIndex`, however, it must first be sorted:

```
In [x]: df['Berlin':'London']
Out[x]: ...
UnsortedIndexError: 'Key length (1) was greater than MultiIndex lexsort depth (0)'
```

This somewhat cryptic error message is a result of the way pandas is optimized to slice only indexes which are in lexicographical order. There are several methods to sort a `MultiIndex`, but the simplest is to use `sort_index` as we did previously:

```
In [x]: df.sort_index(inplace=True)
In [x]: df['Berlin':'London']
Out[x]:
              Mean temperature /degC  Mean rainfall /mm
City   Month
Berlin Apr                      9.0               33.7
       Jan                      0.1               37.2
       Jul                     19.1               52.5
       Oct                      9.4               32.2
London Apr                      9.9               43.7
       Jan                      5.2               55.2
       Jul                     18.7               44.5
       Oct                     12.0               68.5
```

Note that this has sorted the months into alphabetical order as well. To keep them in chronological order, one approach would be to number the months instead by relabeling the index:

```
In [x]: df2 = df.rename({'Jan': 1, 'Apr': 4, 'Jul': 7, 'Oct': 10})
In [x]: df2.sort_index(inplace=True)
In [x]: df2.loc['Vienna', 'Mean temperature /degC']
Out[x]:
Month
```

```
1      0.3
4     10.7
7     20.8
10    10.2
Name: Mean temperature /degC, dtype: float64
```

The useful `xs` function makes selecting data indexed at different levels of a `MultiIndex` easier, and does not require the index to be sorted. For example, to retrieve the climate data for January in all cities:

```
In [x]: df.xs('Jan', level=1)   # look in second level of the MultiIndex for 'Jan'
Out[x]:
        Mean temperature /degC  Mean rainfall /mm
City
Berlin                     0.1               37.2
London                     5.2               55.2
Madrid                     6.3               33.0
Paris                      4.9               51.0
Vienna                     0.3               38.0
```

A `Series` or `DataFrame` with a hierarchical row index can be reshaped so as to create a `MultiIndex` on the columns instead by using the `unstack()` function:

```
In [x]: df.unstack()
Out[x]:
       Mean temperature /degC              ... Mean rainfall /mm
Month                     Apr  Jan   Jul  ...               Jan   Jul   Oct
City                                      ...
Berlin                    9.0  0.1  19.1  ...              37.2  52.5  32.2
London                    9.9  5.2  18.7  ...              55.2  44.5  68.5
Madrid                   12.9  6.3  25.6  ...              33.0  12.0  60.0
Paris                    11.5  4.9  20.5  ...              51.0  62.3  61.5
Vienna                   10.7  0.3  20.8  ...              38.0  70.0  38.0

[5 rows x 8 columns]
```

9.3.2 Timestamps and Time Series

pandas provides a `Timestamp` object, representing an instant in time to some precision. The `to_datetime` method provides a powerful and flexible way of parsing a human-readable string into a `Timestamp`. The following examples all evaluate to a timestamp representing midnight on the 12th of March, 2020: `Timestamp('2020-03-12 00:00:00')`. Note that where the date is ambiguous, by default it is resolved in favor of the US convention: MM/DD/YY: to force a string to be interpreted as DD/MM/YY set `dayfirst=True`.

```
pd.to_datetime('2020-03-12')
pd.to_datetime('12/3/20', dayfirst=True)
pd.to_datetime('3/12/20')
pd.to_datetime('12 March, 2020')
pd.to_datetime('12th of March 2020')
pd.to_datetime('Mar 12, 2020')
```

Times are also handled gracefully:

Table 9.1 Some string codes for pandas time frequencies and offsets

Code	Description
`A`	Year end
`M`	Month end
`W`	Week
`D`	Calendar day
`H`	Hour
`T`	Minute
`S`	Second
`L`	Millisecond
`U`	Microsecond

```
In [x]: pd.to_datetime('9:05 21 August 2017')
Out[x]: Timestamp('2017-08-21 09:05:00')
In [x]: pd.to_datetime('21 August 2017 09:05:23')
Out[x]: Timestamp('2017-08-21 09:05:23')
```

Indexes can be constructed as a range of regularly spaced `Timestamp`s with the `date_range` function. Ranges can be specified by passing the start and end date, or by passing the start date and the number of *periods*. The range interval is one day by default, but this can be controlled with the `freq` argument (see Table 9.1 for valid values).

```
In [x]: pd.date_range('1997-03-12', '1997-03-15')
Out[x]: DatetimeIndex(['1997-03-12', '1997-03-13', '1997-03-14', '1997-03-15'],
                      dtype='datetime64[ns]', freq='D')

In [x]: pd.date_range('1997-03-12', periods=4)
Out[x]: DatetimeIndex(['1997-03-12', '1997-03-13', '1997-03-14', '1997-03-15'],
                      dtype='datetime64[ns]', freq='D')

❶ In [x]: pd.date_range('1997-03', periods=4, freq='M')
  Out[x]: DatetimeIndex(['1997-03-31', '1997-04-30', '1997-05-31', '1997-06-30'],
                        dtype='datetime64[ns]', freq='M')

❷ In [x]: pd.date_range('1997-03', periods=4, freq='MS')
  Out[x]: DatetimeIndex(['1997-03-01', '1997-04-01', '1997-05-01', '1997-06-01'],
                        dtype='datetime64[ns]', freq='MS')
```

❶ By defaults, monthly ranges specified with `freq='M'` are marked at the end of the month. The same is true for annual ranges (`freq='A'`).

❷ To set timestamps at the start of each month use `freq='MS'` (and `freq='AS'` for annual ranges).

pandas makes a distinction between a timestamp (represented by a `Timestamp` object) and a *time period*: an interval of time between two points in time. A time period is represented by the `Period` object and its start and end points are accessed through its attributes `start_time` and `end_time`. The syntax for creating time periods is similar to date ranges:

```
In [x]: p = pd.Period('2020-04', freq='M')
In [x]: t = pd.Timestamp('2020-04-03 14:30')
In [x]: p.start_time < t < p.end_time
Out[x]: True
```

It is often necessary to resample a time series at a different (higher or lower) frequency. The `resample` method assists with this: it returns a `Resampler` object which can be used to aggregate the data in some appropriate way. For example, in *downsampling* (resampling the data to a wider time frame), it may be appropriate to take the mean, minimum, maximum or sum of the values in the resampling interval. The following example should make this clearer.

Example E9.10 The file `river-level.csv`, available at https://scipython.com/eg/bal, lists the height in meters above sea level of Chitterne Brook, a small river in Wiltshire, England. Heights are given as minimum, average, and maximum values for each day between 1 January 2014 and 31 December 2016.

The following code reads in the data and plots the daily river height along with its monthly average, minimum and maximum values.

```
import pandas as pd
import matplotlib.pyplot as plt

❶ df = pd.read_csv('river-level.csv', index_col=0, comment='#', parse_dates=True)

rs_monthly = df.resample('M')

df['avg_level'].plot(label='Daily average')
rs_monthly['avg_level'].mean().plot(label='Monthly average')
rs_monthly['min_level'].min().plot(label='Monthly minimum')
rs_monthly['max_level'].max().plot(label='Monthly maximum')

plt.xlabel('Date')
plt.ylabel('River level /m')
plt.gca().legend()
plt.show()
```

❶ Note that we need to set `parse_dates=True` to force pandas to interpret the first column as a `DatetimeIndex`.

Figure 9.4 shows the resulting plot.

9.3.3 Exercises

Problems

P9.3.1 Use pandas to read in the file, `tb-cases.txt`, available from https://scipython.com/ex/bao, which provides numbers of cases of tuberculosis in the USA, broken down by state for the years 1993–2018. Create a `DataFrame` with a hierarchical index (`MultiIndex`) consisting of the state name and year. Plot these data appropriately and determine the state with the greatest relative decrease in tuberculosis over the time period considered.

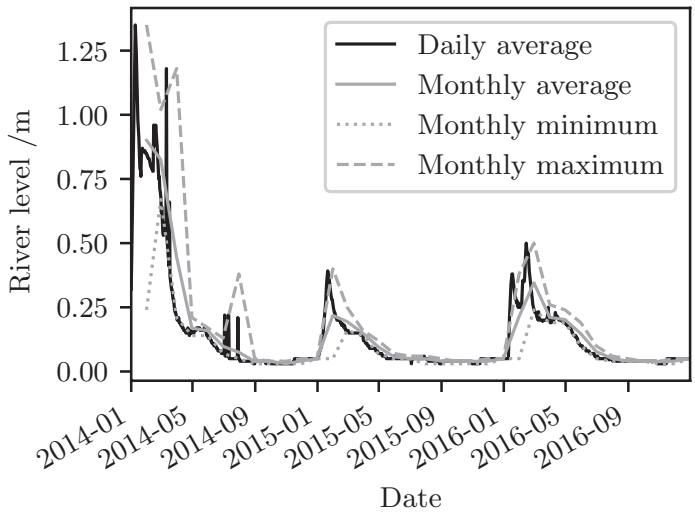

Figure 9.4 The level of Chitterne Brook in meters over the period 2014–2016.

P9.3.2 The populations of each state in the USA over the years 1993–2018 are given in the file `US-populations.txt`, available from https://scipython.com/ex/bap. Read these data into a pandas `DataFrame` with a suitable index, and analyze them for any interesting trends. Then combine these data with those of Problem P9.3.1 to determine the states with the greatest and least prevalence of tuberculosis per head of population in 2018.

9.4 Data Cleaning and Exploration

Any scientific research, particularly experimental research, will generate data sets containing invalid or missing values. Data points can be dropped or fall outside the detectable range of the measuring instrument, may get transcribed incorrectly, or are obtained incompletely from various sources. pandas provides a variety of methods for dealing with such missing values, including functions for removing them or replacing them with average or default values.

This book does not attempt to provide a guide to the scientific method, but the reader should be aware that the way in which one deals with missing or invalid data can bias the subsequent analysis towards a particular set of conclusions.

9.4.1 Missing Values

The default *sentinel value* indicating missing data is NaN. In Section 9.1.2 we have already used the methods `isnull()` and `notnull()` to test for the presence or absence of such values, and the method `dropna()`, which returns a new `DataFrame` with rows containing only non-null data:

```
In [x]: df = pd.DataFrame([[   1.1, np.nan, np.nan,   10.3],
   ...:                    [   0.8, np.nan,    3.6,    2.9],
   ...:                    [   1.2,    2.5,    1.6,    2.7],
   ...:                    [np.nan, np.nan, np.nan, np.nan],
   ...:                    [np.nan, np.nan,    3.6,    5.3]],
   ...:                   columns=list('ABCD'))

In [x]: df
     A    B    C     D
0  1.1  NaN  NaN  10.3
1  0.8  NaN  3.6   2.9
2  1.2  2.5  1.6   2.7
3  NaN  NaN  NaN   NaN
4  NaN  NaN  3.6   5.3

In [x]:  df.dropna()
Out[x]:
     A    B    C    D
2  1.2  2.5  1.6  2.7
```

You may wish to drop only rows (or columns) which consist entirely of NaN. In that case, pass the argument `how='all'` instead of using the default, `how='any'`:

```
In [x]: df.dropna(how='all')
Out[x]:
     A    B    C     D
0  1.1  NaN  NaN  10.3
1  0.8  NaN  3.6   2.9
2  1.2  2.5  1.6   2.7
4  NaN  NaN  3.6   5.3
```

It is also possible to specify a threshold number of NaN values to trigger the drop of a column or row:

```
In [x]: df.dropna(thresh=3, axis=1)    # only drop columns with three or more NaNs
Out[x]:
     A    C     D
0  1.1  NaN  10.3
1  0.8  3.6   2.9
2  1.2  1.6   2.7
3  NaN  NaN   NaN
4  NaN  3.6   5.3
```

An alternative to dropping the NaN values is to replace them with valid data according to some process. This is the purpose of the `fillna()` method. Common options are given in the following examples.

Replace all NaN values with a single value:

```
In [x]: df.fillna(3.6)
```

```
Out[x]:
     A    B    C     D
0  1.1  3.6  3.6  10.3
1  0.8  3.6  3.6   2.9
2  1.2  2.5  1.6   2.7
3  3.6  3.6  3.6   3.6
4  3.6  3.6  3.6   5.3
```

Replace NaN values with the last encountered valid value down the columns ("fill forward"):

```
In [x]: df.fillna(method='ffill')
Out[x]:
     A    B    C     D
0  1.1  NaN  NaN  10.3
1  0.8  NaN  3.6   2.9
2  1.2  2.5  1.6   2.7
3  1.2  2.5  1.6   2.7
4  1.2  2.5  3.6   5.3
```

Replace NaN values with the last encountered valid value along the rows:

```
In [x]: df.fillna(method='ffill', axis=1)
Out[x]:
     A    B    C     D
0  1.1  1.1  1.1  10.3
1  0.8  0.8  3.6   2.9
2  1.2  2.5  1.6   2.7
3  NaN  NaN  NaN   NaN
4  NaN  NaN  3.6   5.3
```

Passing a dictionary of column or index names enables close control over the filling values; chaining calls can then give a powerful and flexible way to clean data. For example, to fill in the missing data in columns `A` and `C` with their means:

```
In [x]: df.fillna({'A': df['A'].mean(), 'C': df['C'].mean()})
Out[x]:
          A    B         C     D
0  1.100000  NaN  2.933333  10.3
1  0.800000  NaN  3.600000   2.9
2  1.200000  2.5  1.600000   2.7
3  1.033333  NaN  2.933333   NaN
4  1.033333  NaN  3.600000   5.3
```

A further example:

```
In [x]: df.fillna({'A': df['A'].mean(), 'B': df['B'].mean()}).fillna(0)
Out[x]:
          A    B    C     D
0  1.100000  2.5  0.0  10.3
1  0.800000  2.5  3.6   2.9
2  1.200000  2.5  1.6   2.7
3  1.033333  2.5  0.0   0.0
4  1.033333  2.5  3.6   5.3
```

It may be that the data set being used uses a different sentinel value to indicate invalid data, for example `-1` or `-99`. The `replace` method can canonicalize such data:

```
In [x]: ser = pd.Series([1.2, 3.5, -99, -99, 4.0, -99, -0.5])
In [x]: ser.replace(-99, np.nan)
Out[x]:
0    1.2
1    3.5
2    NaN
3    NaN
4    4.0
5    NaN
6   -0.5
dtype: float64
```

`replace` can also take a dictionary mapping values to their replacements:

```
In [x]: ser.replace({-99: 0, -0.5: np.nan})
Out[x]:
0    1.2
1    3.5
2    0.0
3    0.0
4    4.0
5    0.0
6    NaN
dtype: float64
```

9.4.2 Duplicate Values

The `DataFrame` method `duplicated()` returns a `Series` of boolean values indicating whether each row is a duplicate of a previous row; `drop_duplicates()` drops such rows. By default, both methods consider all columns; to remove rows with duplicate entries in a single column or several columns, pass a column name or a sequence of column names explicitly. A further argument, `keep`, determines whether the first encountered row (`'first'`, the default) or last encountered row (`'last'`) is retained.

```
In [x]: df = pd.DataFrame([['Lithium', 'Li', 3, 6, 0.0759],
   ...:                    ['Lithium', 'Li', 3, 7, 0.9241],
   ...:                    ['Sodium', 'Na', 11, 23, 1],
   ...:                    ['Potassium', 'K', 19, 39, 0.932581],
   ...:                    ['Potassium', 'K', 19, 40, 1.17e-4],
   ...:                    ['Potassium', 'K', 19, 41, 0.067302]],
   ...:                   columns=['Element', 'Symbol', 'Z', 'A', 'Abundance'])

In [x]: df
Out[x]:
     Element Symbol   Z   A  Abundance
0    Lithium     Li   3   6   0.075900
1    Lithium     Li   3   7   0.924100
2     Sodium     Na  11  23   1.000000
3  Potassium      K  19  39   0.932581
4  Potassium      K  19  40   0.000117
5  Potassium      K  19  41   0.067302

In [x]: df.drop_duplicates(['Symbol'])
Out[x]:
     Element Symbol   Z   A  Abundance
0    Lithium     Li   3   6   0.075900
```

```
2     Sodium     Na  11  23   1.000000
3  Potassium      K  19  39   0.932581

In [x]: df.drop_duplicates(['Symbol', 'Z'], keep='last')
Out[x]:
     Element Symbol   Z   A  Abundance
1    Lithium     Li   3   7   0.924100
2     Sodium     Na  11  23   1.000000
5  Potassium      K  19  41   0.067302
```

9.4.3 Binning Data

It is often necessary to bin together large amounts of continuous data, either to reduce it to a manageable size or to categorize it based on value. The pandas function `cut` can be used to do this in a similar way to NumPy's `histogram` function (Section 6.3.3):

```
In [x]: marks = [67, 80, 34, 55, 77, 66, 59, 52, 70, 67, 58, 63, 49, 72]
In [x]: bins = [0, 40, 60, 70, 80, 100]
In [x]: dist = pd.cut(marks, bins)
In [x]: dist
[(60, 70], (70, 80], (0, 40], (40, 60], ..., (60, 70], (40, 60], (70, 80]]
Length: 14
Categories (5, interval): [(0, 40] < (40, 60] < (60, 70] < (70, 80] < (80, 100]]
```

Each mark is placed in a bin with edges defined by the sequence `bins`. The number of values in each bin is returned by `value_counts`:

```
In [x]: pd.value_counts(dist)
Out[x]:
(60, 70]     5
(40, 60]     5
(70, 80]     3
(0, 40]      1
(80, 100]    0
dtype: int64
```

By default, the right side of each interval is closed (values equal to this side are included in the bin, indicated by '`]`') and the left side is open (indicated by '`(`'); this can be swapped by setting the argument `right=False`:

```
In [x]: pd.value_counts(pd.cut(marks, bins, right=False))
Out[x]:
[40, 60)     5
[60, 70)     4
[70, 80)     3
[80, 100)    1
[0, 40)      1
dtype: int64
```

The bins can also be named by passing a sequence of strings to the `labels` argument:

```
In [x]: dist = pd.cut(marks, bins, labels=list(reversed('ABCDE')), right=False)
In [x]: dist
[C, A, E, D, B, ..., C, D, C, D, B]
Length: 14
```

```
Categories (5, object): [E < D < C < B < A]

In [x]: pd.value_counts(dist)
Out[x]:
D    5
C    4
B    3
A    1
E    1
dtype: int64
```

Note that the categories do not have any particular order. To put the counts in order of decreasing grade, we can sort the `Series` index:

```
In [x]: pd.value_counts(dist).sort_index(ascending=False)
Out[x]:
A    1
B    3
C    4
D    5
E    1
dtype: int64
```

9.4.4 Dealing with Outliers

Detecting and filtering outliers is, like dealing with missing values or invalid data, a potentially subtle process and careful thought should be given to the assumptions behind the expected underlying distribution. However, often, outlying values are expected based on detector failure (sticky pixels, cosmic rays, and the like), obvious errors, or well-understood exceptional cases. Filtering them automatically can be achieved with NumPy-like array operations.

For example, consider a simulated village in which the 200 houses have normally-distributed prices ($\mu = \$250\,000$, $\sigma = \$55\,000$), with the exception of a couple of mansions worth many times more than the average home:

```
In [x]: nhouses = 200
In [x]: mu, sigma = 250, 55       # mean, standard deviation in $1000s
❶ In [x]: prices = np.clip(np.random.randn(nhouses)*sigma + mu, 0, None).astype(int)
In [x]: prices[-2] = 1.e3
In [x]: prices[-1] = 2.e3
In [x]: df = pd.DataFrame(prices, columns=['price, $1000s'])
In [x]: df.tail()
Out[x]:
     price, $1000s
195            247
196            218
197            236
198           1000
199           2000
```

❶ `np.clip(arr, min, max)` constrains the values of `arr` to fall within `min` and `max`, here to prevent negative house prices being produced by the random sampling. This is itself a type of outlier filtering!

These outliers distort the mean and (especially) the standard deviation of the house price distribution:

```
In [x]: df.median()          # the median is a robust measure of central tendency
Out[x]:
price, $1000s    247.8
dtype: float64

In [x]: df.mean()            # the mean is affected more by the outliers
Out[x]:
price, $1000s    258.775
dtype: float64

In [x]: df.std()             # the standard deviation is greatly affected
Out[x]:
price, $1000s    145.796907
dtype: float64
```

We may be interested in analyzing the prices of "ordinary" houses in the village, ignoring the mansions. One way to do this is to identify the mansions as deviating from the mean house price by, say, three standard deviations and setting their prices to NaN:

```
In [x]: df[df > 3*df.std()+df.mean()] = np.nan
In [x]: df.tail()
     price, $1000s
195          247.0
196          218.0
197          236.0
198            NaN
199            NaN
```

Now we find values closer to the (non-mansion) house price distribution:

```
In [x]: df.mean()
Out[x]:
price, $1000s    246.237374
dtype: float64

In [x]: df.std()
Out[x]:
price, $1000s    55.995279
dtype: float64
```

Example E9.11 Robert Millikan's famous oil-drop experiments were carried out at the University of Chicago from 1909 to determine the magnitude of the charge of the electron.[12] In a single experiment, an electrically charged oil droplet was observed to fall a known distance, d, between two uncharged plates at its terminal velocity, v_g: from the time taken, t_g, the droplet's radius, a, can be deduced. Next, a voltage was applied to the plates, inducing an electric field between them. As the droplet rises under the resulting net force, the time taken, t_e, for it to move back up through the same distance, d, can be used to deduce its total charge, q, which is observed to be an integer multiple of the same base value, e, that is: $q = Ne$.

[12] Since May 2019, this quantity has been fixed by definition at $1.602176634 \times 10^{-19}$ C.

For the free-fall part of the experiment,[13] when the droplet falls at constant terminal velocity $v_g = -d/t_g$ there is no net force on it: the sum of the gravitational and drag forces is zero:

$$F_g + F_d = 0 \quad \Rightarrow -m'g - 6\pi\eta a v_g = 0,$$

where $m' = \frac{4}{3}\pi a^3 \rho' = \frac{4}{3}\pi a^3(\rho_{\text{oil}} - \rho_{\text{air}})$ is the effective mass of the droplet (after the mass of air it displaces is taken into account), $g = 9.803\ \text{m}\,\text{s}^{-1}$ is the acceleration due to gravity in Chicago, and $\eta = 1.859 \times 10^{-5}\ \text{kg}\,\text{m}^{-1}\,\text{s}^{-1}$ is the air viscosity under the experimental conditions (temperature, humidity, etc.). Rearranging, we get the following expression for the droplet radius:

$$a = \sqrt{\frac{-9\eta v_g}{2\rho' g}}.$$

When a suitable voltage is applied to the plates and the droplet moves upwards at a constant velocity $v_e = d/t_e$, the force due to the electric field is balanced by gravity and the drag force (at this new velocity):

$$F_e + F_g + F'_{\text{d}} = 0 \quad \Rightarrow qE + 6\pi\eta a v_g - 6\pi\eta a v_e = 0$$
$$\Rightarrow \quad q = \frac{6\pi\eta a(v_e - v_g)}{E} = \frac{6\pi\eta a d}{E}\left(\frac{1}{t_g} + \frac{1}{t_e}\right)$$

Each droplet (labeled A–H) was observed three times for each different charge, q, acquired by exposure to X-rays (up to seven experiments per droplet).

The data at https://scipython.com/eg/bam give the time data for a number of such experiments conducted with an oil of density $\rho_{\text{oil}} = 917.3\ \text{kg}\,\text{m}^{-3}$ on a day for which $\rho_{\text{air}} = 1.17\ \text{kg}\,\text{m}^{-3}$. The magnitude of the electric field was $E = 322.1\ \text{kN}\,\text{C}^{-1}$ and the distance the drops move, $d = 11.09$ mm. We can use these data to estimate e (assuming it is not fixed by definition) as follows.

drop	expt	tg	te	tg	te	tg	te
A	1	13.102	46.822	12.941	46.896	13.086	46.681
A	2	12.938	86.767	13.032	86.952	13.086	86.746
A	3	13.023	61.082	12.958	60.826	12.998	60.860
A	4	12.943	86.747	12.922	86.840	13.054	86.899
B	1	11.434	56.305	11.350	56.097	11.246	56.282
B	2	11.402	75.823	11.584	75.819	11.487	76.063
B	3	11.591	44.717	11.397	44.851	11.364	44.776
B	4	11.443	75.905	11.368	75.975	11.457	76.041
B	5	11.434	75.939	11.414	75.880	11.444	75.929
B	6	11.559	75.892	11.414	75.924	11.292	75.985
B	7	11.394	44.716	11.589	44.753	11.401	44.794
C	1	16.197	100.458	16.010	100.486	16.329	100.461
C	2	16.241	47.727	16.106	47.714	16.177	47.625
C	3	16.133	37.879	16.267	37.746	16.203	37.709
C	4	16.170	64.765	16.136	64.649	16.229	64.508
D	1	16.176	38.017	16.127	37.910	16.282	38.020

[13] In this example we adopt a coordinate system in which the droplet's vertical position, z, increases in the "up" direction.

```
D       2   16.275   38.280   16.092   38.208   16.133   38.092
D       3   16.422   48.327   16.073   48.284   16.212   48.184
D       4   16.134   38.202   16.258   38.270   16.105   38.229
D       5   16.164  102.562   16.217  102.673   16.194  102.696
E       1   12.275   55.020   12.116   54.962   12.307   54.978
E       2   12.157   54.772   12.183   54.967   12.046   55.219
E       3   12.146   55.004   12.118   54.938   12.346   54.869
E       4   12.319   43.635   12.243   43.552   12.073   43.582
F       1   14.172   61.946   14.174   61.970   14.069   61.959
F       2   14.145   90.718   13.955   90.707   14.075   90.866
F       3   14.070   62.147   14.074   61.961   14.247   61.892
F       4   14.017   61.968   14.101   61.921   14.106   62.174
G       1    9.723   50.375    9.527   50.482    9.502   50.508
G       2    9.463   63.755    9.670   63.853    9.509   63.827
G       3    9.448   63.804    9.407   63.899    9.563   63.768
G       4    9.327   63.855    9.518   63.967    9.533   63.824
H       1   13.192   73.375   13.167   73.338   13.316   73.449
H       2   13.042   42.642   13.387   42.428   13.334   42.459
H       3   13.389   42.379   13.244   42.373   13.055   42.610
H       4   13.114   73.161   13.226   73.384   13.207   73.257
H       5   13.030   73.295   13.022   73.419   13.438   73.512
```

First, define the necessary parameters:

```
eta = 1.859e-5                  # air viscosity, kg.m-1.s-1
rho_air = 1.17                  # air density, kg.m-3
rho_oil = 917.3                 # oil density, kg.m-3
rhop = rho_oil - rho_air
g = 9.803                       # acceleration due to gravity, m.s-2
d = 11.09e-3                    # rise/fall distance, m
E = -322.1e3                    # electric field vector (points down!)
```

Next, read in the data, assigning the first two columns to a `MultiIndex`:

```
In [x]: import pandas as pd
In [x]: df = pd.read_csv('eg10-millikan-data.txt', delim_whitespace=True,
                         index_col=[0, 1])
In [x]: df.head()
Out[x]:
               tg      te    tg.1    te.1    tg.2    te.2
drop expt
A    1     13.102  46.822  12.941  46.896  13.086  46.681
     2     12.938  86.767  13.032  86.952  13.086  86.746
     3     13.023  61.082  12.958  60.826  12.998  60.860
     4     12.943  86.747  12.922  86.840  13.054  86.899
B    1     11.434  56.305  11.350  56.097  11.246  56.282
```

Note that pandas has added a counting integer to the column names to make them distinct.

We will start with just a single droplet, taking the transpose of its data:

```
In [x]: dropA = df.loc['A'].T
In [x]: dropA
Out[x]:
expt       1       2       3       4
tg    13.102  12.938  13.023  12.943
te    46.822  86.767  61.082  86.747
tg.1  12.941  13.032  12.958  12.922
```

```
te.1  46.896  86.952  60.826  86.840
tg.2  13.086  13.086  12.998  13.054
te.2  46.681  86.746  60.860  86.899
```

We would prefer to label each row as simply `'tg'` or `'te'`:

```
In [x]: dropA.index = dropA.index.str.slice(0, 2)
In [x]: dropA
Out[x]:
expt       1       2       3       4
tg    13.102  12.938  13.023  12.943
te    46.822  86.767  61.082  86.747
tg    12.941  13.032  12.958  12.922
te    46.896  86.952  60.826  86.840
tg    13.086  13.086  12.998  13.054
te    46.681  86.746  60.860  86.899
```

We require the average of all of the values of t_g (in the absence of the electric field the droplet takes the same time to fall the distance d) and the average value of t_e for each column (each experiment may have a different droplet charge, but the fall–rise times are measured three times for each experiment):

```
In [x]: tg = dropA.loc['tg'].values.mean()
In [x]: te = dropA.loc['te'].mean()
In [x]: tg
Out[x]: 13.006916666666667
In [x]: te
Out[x]:
expt
1    46.799667
2    86.821667
3    60.922667
4    86.828667
dtype: float64
```

Now use the value of t_g to calculate the droplet's radius:

```
In [x]: a = np.sqrt(9*eta*d/tg/2/rhop/g)
In [x]: a
Out[x]: 2.8181654881967875e-06
```

or about 2.82 μm. The charge we deduce for each experiment is:

```
In [x]: q = 6 * np.pi * eta * a * d / E * (1/tg + 1/te)
In [x]: q
Out[x]:
expt
1   -3.340563e-18
2   -3.005663e-18
3   -3.172143e-18
4   -3.005631e-18
dtype: float64
```

Repeating this for all the droplets, we can add a column, `q` to the `DataFrame` `df`:

```
for drop in df.index.levels[0]:
    drop_df = df.loc[drop].T
    drop_df.index = drop_df.index.str.slice(0, 2)
```

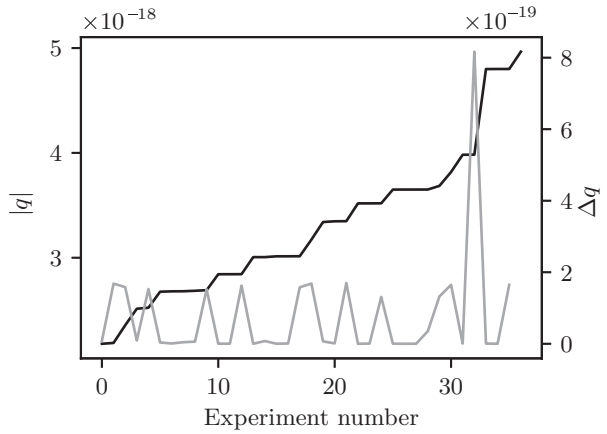

Figure 9.5 Sorted droplet charges, q, and neighbouring differences, Δq.

```
tg = drop_df.loc['tg'].values.mean()
te = drop_df.loc['te'].mean()
a = np.sqrt(9*eta*d/tg/2/rhop/g)
q = 6 * np.pi * eta * a * d / E * (1/tg + 1/te)
df.loc[drop, 'q'] = q.values
```

It is now helpful to sort the droplet charges by magnitude and to plot the sorted array and its differences (Figure 9.5):

```
In [x]: sorted_q = sorted(-df.loc[:, 'q'])
In [x]: plt.plot(sorted_q)
In [x]: plt.ylabel('$|q|$')
In [x]: plt.twinx()
In [x]: dq = np.diff(sorted_q)
In [x]: plt.plot(dq)
In [x]: plt.ylabel(r'$\Delta q$')
In [x]: plt.show()
```

It certainly seems possible that the droplet charge is always a multiple of some value between 1×10^{-19} C and 2×10^{-19} C. We can therefore estimate the value of $|e|$:

```
In [x]: e_estimate = dq[(dq>1.e-19) & (dq<2.e-19)].mean()
In [x]: e_estimate
Out[x]: 1.5697150510604604e-19
```

We can now add a column to `df` for the number of elementary charges we hypothesise for each experiment:

```
In [x]: df['N'] = (df['q'] / e_estimate).astype(int)
```

Considering all the data then gives us our estimate for the magnitude of the electron charge:

```
In [x]: (df['q']/df['N']).mean()
Out[x]: 1.5923552150386455e-19
```

within 1% of the defined value.

9.4.5 Exercises

Problems

P9.4.1 Use pandas' `cut` method to classify the stars in the data set of Problem P9.2.3 according to their temperature by placing them into the bins labeled M, K, G, F, A, B, and O with left edges (in K) at 2400, 3700, 5200, 6000, 7500, 10 000, and 30 000.

Hence modify the code in the solution to this problem to plot the stars in a color appropriate to their temperature by establishing the following mapping:

```
color_mapping = {'M': '#FFB56C', 'K': '#FFDAB5', 'G': '#FFEDE3', 'F': '#F9F5FF',
                 'A': '#D5E0FF', 'B': '#A2C0FF', 'O': '#92B5FF'}
```

Hint: pandas provides a `map` method for mapping input values from an existing column to output values in a new column using a dictionary.

P9.4.2 Reanalyze the data from Example E9.11, concerning Millikan's oil-drop experiment, to use a more accurate approximation for the effective air viscosity:

$$\eta = \frac{\eta_0}{1 + \frac{b}{ap}},$$

where $p = 100.82$ kPa is the air pressure, $\eta_0 = 1.859 \times 10^{-5}$ kg m^{-1} s^{-1}, $b = 7.88 \times 10^{-3}$ Pa m, and a is the droplet radius.

P9.4.3 The Cambridge University Digital Technology Group have been recording the weather from the roof of their department building since 1995 and make the data available to download at www.cl.cam.ac.uk/research/dtg/weather/.

Read in the entire data set and parse it with pandas to determine (a) the most common wind direction; (b) the fastest wind speed measured; (c) the year with the sunniest June; (d) the day with the highest rainfall; (e) the coldest temperature measured. Note that there are occasional missing and invalid data points in the data set.

P9.4.4 The data set at https://scipython.com/ex/baq lists the following quantities, in US dollars over time: (a) the price of gold; (b) the S&P 500 US stock market index; and (c) the price of the cryptocurrency Bitcoin. Compare the performance of these indexes over the period 2010–2020 with respect to the regular investment of $100 per month.

9.5 Data Grouping and Aggregation

9.5.1 `DataFrame` Grouping with `groupby`

The powerful pandas method `groupby` can be used to analyze data in a `Series` or `DataFrame` based on their categorization according to some key row (or column) values. The term *split–apply–combine* describes the process succinctly: first, the data is split according to its categorization; next, the analysis technique or statistical method required (for example, summing values or finding their mean) is applied to the split

groups; finally, the results of the analysis are combined into a result object. Figure 9.6 depicts a simple example of the process.

Example E9.12 Consider the following table of the yields of three compounds, A, B and C, attained in a synthesis experiment by three students, Anu, Jenny and Tom.

```
In [x]: data = [['Anu', 'A', 5.4], ['Anu', 'B', 6.7], ['Anu', 'C', 10.1],
   ...:         ['Jenny', 'A', 6.5], ['Jenny', 'B', 5.9], ['Jenny', 'C', 12.2],
   ...:         ['Tom', 'A', 4.0], ['Tom', 'B', None], ['Tom', 'C', 9.5]
   ...:        ]

In [x]: df = pd.DataFrame(data, columns=['Student', 'Compound', 'Yield /g'])

In [x]: print(df)
  Student Compound  Yield /g
0     Anu        A       5.4
1     Anu        B       6.7
2     Anu        C      10.1
3   Jenny        A       6.5
4   Jenny        B       5.9
5   Jenny        C      12.2
6     Tom        A       4.0
7     Tom        B       NaN
8     Tom        C       9.5
```

One way of analyzing these data is to group them by compound ("split" into separate data structures, each with a common value of `'Compound'`) and then apply some operation (say, finding the mean) to each group, before recombining into a single `DataFrame`, as illustrated in Figure 9.6.

```
In [x]: grouped = df.groupby('Compound')
```

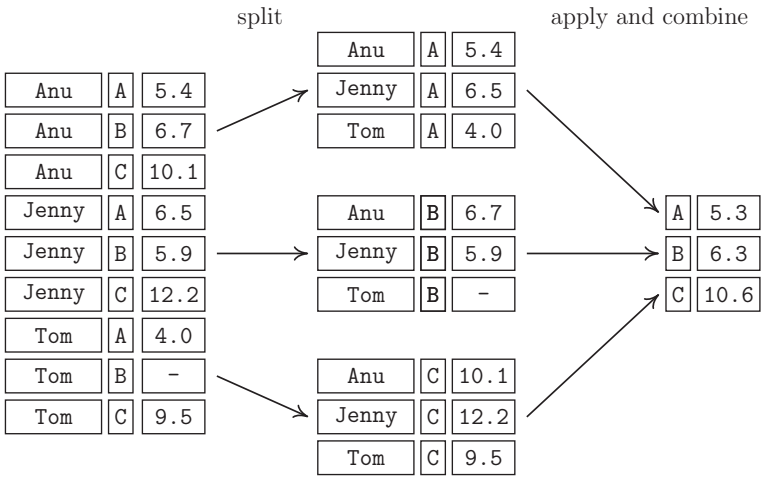

Figure 9.6 An illustration of the *split–apply–combine* paradigm for analyzing data with grouped data in pandas: the `DataFrame` is split into groups by compound (A, B and C); the `mean` function is applied to the groups; these values are combined into the returned object.

```
In [x]: grouped.mean()
Out[x]:
          Yield /g
Compound
A              5.3
B              6.3
C             10.6
```

Here, the `'Student'` column has been ignored as a so-called "nuisance" column: there is no helpful way to take the mean of a string. The `max()` and `min()` functions, however, consider the strings' lexigraphical ordering:

```
In [x]: grouped.max()
Out[x]:
         Student  Yield /g
Compound
A            Tom       6.5
B            Tom       6.7
C            Tom      12.2
```

Note that `max()` has returned `'Tom'` for every row, since this name is lexigraphically last ('greatest') in the `'Student'` column. The column `'Yield /g'` consists of the maximum yields for each compound, across all students. To apply the function to a subset of the columns only (which may be necessary for very large `DataFrames`), select them before the function call, for example:

```
In [x]: grouped['Yield /g'].min()
Out[x]:
Compound
A    4.0
B    5.9
C    9.5
Name: Yield /g, dtype: float64
```

The object returned by `groupby()` can be iterated over:

```
In [x]: for compound, group in grouped:
   ...:     print('Compound:', compound)
   ...:     print(group)
   ...:
Compound: A
  Student Compound  Yield /g
0     Anu        A       5.4
3   Jenny        A       6.5
6     Tom        A       4.0
Compound: B
  Student Compound  Yield /g
1     Anu        B       6.7
4   Jenny        B       5.9
7     Tom        B       NaN
Compound: C
  Student Compound  Yield /g
2     Anu        C      10.1
5   Jenny        C      12.2
8     Tom        C       9.5
```

We can also group by the `'Student'` column:

```
In [x]: grouped = df.groupby('Student')

In [x]: grouped.mean()
Out[x]:
         Yield /g
Student
Anu          7.40
Jenny        8.20
Tom          6.75
```

Another powerful feature is the ability to group on the basis of a specified mapping, provided, for example, by a dictionary. Suppose each student is undertaking a different degree programme:

```
In [x]: degree_programmes = {'Anu': 'Chemistry',
                             'Jenny': 'Chemistry',
                             'Tom': 'Pharmacology'}
```

First, turn the `'Student'` column into an `Index` and then group, not by the `Index` itself but using the provided mapping:

```
In [x]: df.set_index('Student', inplace=True)
In [x]: df.groupby(degree_programmes).mean()
Out[x]:
             Yield /g
Chemistry         7.80
Pharmacology      6.75
```

That is, the average yield for students of chemistry was 7.8 g, whereas for pharmacology it was only 6.75 g.

9.5.2 Exercises

Problems

P9.5.1 The Organisation for Economic Co-operation and Development (OECD), within its Programme for International Student Assessment (PISA), publishes an evaluation of the educational systems around the world by measuring the performance of 15-year-old school pupils on mathematics, science, and reading. The evaluation is carried out every three years.

Historical PISA data can be downloaded from https://scipython.com/ex/bza. Read these data in to a pandas `DataFrame` and use its grouping functionality to determine and visualize (a) the overall performance of all studied countries over time; (b) the gender disparity (if any) in each of reading, mathematics and science; and (c) the correlation between the performances in each of these areas across all countries.

P9.5.2 Read in the data at https://scipython.com/ex/bar concerning recent Formula One Grands Prix seasons, and rank (a) the drivers by their number of wins; (b) the constructors by their number of wins; and (c) the circuits by their average fastest lap per race.

9.6 Examples

The following examples demonstrate the practical use of pandas in two case studies involving the analysis and visualization of real data.

Example E9.13 The file `nuclear-explosion-data.csv`, available to download at https://scipython.com/eg/ban, contains data on all nuclear explosions between 1945 and 1998.[14] We will use pandas to analyze it in various ways.

Inspection of the file in a text editor shows that it contains a header line naming the columns, so we can load it straight away with `pd.read_csv` and inspect its key features:

```
In [x]: import pandas as pd
In [x]: df = pd.read_csv('nuclear-explosion-data.csv')
In [x]: df.head()

Out[x]:
       date      time     id country  ... yield_upper purpose       name     type
0  19450716  123000.0  45001     USA  ...         21.0      WR    TRINITY    TOWER
1  19450805  231500.0  45002     USA  ...         15.0  COMBAT  LITTLEBOY  AIRDROP
2  19450809   15800.0  45003     USA  ...         21.0  COMBAT     FATMAN  AIRDROP
3  19460630  220100.0  46001     USA  ...         21.0      WE       ABLE  AIRDROP
4  19460724  213500.0  46002     USA  ...         21.0      WE      BAKER       UW

[5 rows x 16 columns]

In [x]: df.index
Out[x]: RangeIndex(start=0, stop=2051, step=1)

In [x]: df.columns
Out[x]:
Index(['date', 'time', 'id', 'country', 'region', 'source', 'lat', 'long',
       'mb', 'Ms', 'depth', 'yield_lower', 'yield_upper', 'purpose', 'name',
       'type'],
      dtype='object')
```

There are 16 columns; here we will be concerned with those described in Table 9.2.

It is natural to assign the date and time of the explosion to the `DataFrame` index. Some helper functions facilitate this:

```
from datetime import datetime
def parse_time(t):
    hour, t = divmod(t, 10000)
    minute, t = divmod(t, 100)
    return int(hour), int(minute), int(t)

def parse_datetime(date, time):
    date_and_time = datetime.strptime(str(date), '%Y%m%d')
    hour, minute, second = parse_time(time)
    return date_and_time.replace(hour=hour, minute=minute, second=second)

df.index = pd.DatetimeIndex([parse_datetime(date, time) for date, time in
                    zip(df['date'], df['time'])])
```

[14] from N.-O. Bergkvist and R. Ferm, *Nuclear Explosions 1945–1998*, Swedish Defence Research Establishment/SIPRI, Stockholm, July 2000.

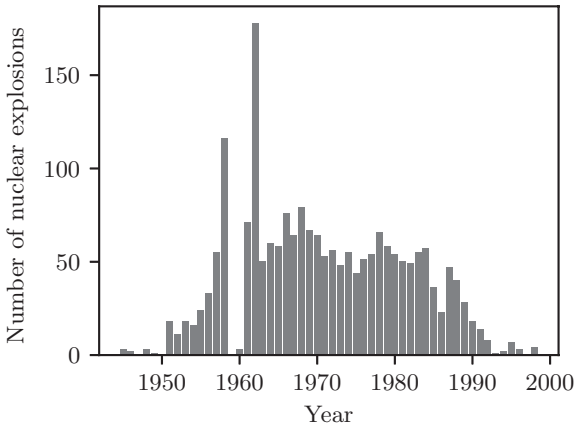

Figure 9.7 Bar chart of the number of nuclear explosions by year between 1945 and 1998.

We can plot the number of explosions in each year by grouping on `index.year` and finding the size of each group; a regular Matplotlib bar chart can then be produced:

```
explosion_number = df.groupby(df.index.year).size()

import matplotlib.pyplot as plt
fig, ax = plt.subplots()
ax.bar(explosion_number.index, explosion_number.values)
ax.set_xlabel('Year')
ax.set_ylabel('Number of nuclear explosions')
plt.show()
```

Figure 9.7 shows the resulting plot.

Table 9.2 Important columns of nuclear explosion data in file `nuclear-explosion-data.csv`

Column	Description
`date`	Date of explosion in format `YYYYMMDD`
`time`	Time of explosion in format `HHMMSS.Z`, where `Z` represents tenths of seconds
`country`	The state that carried out the explosion
`lat`	The latitude of the explosion in degrees, relative to the equator
`long`	The longitude of the explosion in degrees, relative to the prime meridian
`yield_lower`	Lower estimate of the yield in kilotons (kt) of TNT
`yield_upper`	Upper estimate of the yield in kilotons (kt) of TNT
`type`	The method of deployment of the nuclear device

A stacked bar chart can break down the annual count of explosions by country. First, group by both year and country and get the explosion counts for this grouping with `size()`:

```
df2 = df.groupby([df.index.year, df.country])
explosions_by_country = df2.size()
print(explosions_by_country.head(7))
```

```
      country
1945  USA          3
1946  USA          2
1948  USA          3
1949  USSR         1
1951  USA         16
      USSR         2
1952  UK           1
dtype: int64
```

Next, `unstack` the second index into columns, filling the empty entries with zeros:

```
explosions_by_country = explosions_by_country.unstack().fillna(0)
print(explosions_by_country.head(7))
```

```
country  CHINA  FRANCE  INDIA  PAKISTAN   UK   USA  USSR
1945       0.0     0.0    0.0       0.0  0.0   3.0   0.0
1946       0.0     0.0    0.0       0.0  0.0   2.0   0.0
1948       0.0     0.0    0.0       0.0  0.0   3.0   0.0
1949       0.0     0.0    0.0       0.0  0.0   0.0   1.0
1951       0.0     0.0    0.0       0.0  0.0  16.0   2.0
1952       0.0     0.0    0.0       0.0  1.0  10.0   0.0
1953       0.0     0.0    0.0       0.0  2.0  11.0   5.0
```

Each row in this `DataFrame` can then be plotted as stacked bars on a Matplotlib chart:

```
countries = ['USA', 'USSR', 'UK', 'FRANCE', 'CHINA', 'INDIA', 'PAKISTAN']
bottom = np.zeros(len(explosions_by_country))
fig, ax = plt.subplots()
for country in countries:
    ax.bar(explosions_by_country.index, explosions_by_country[country],
            bottom=bottom, label=country)
    bottom += explosions_by_country[country].values

ax.set_xlabel('Year')
ax.set_ylabel('Number of nuclear explosions')
ax.legend()
plt.show()
```

Figure 9.8 shows the resulting stacked bar chart.

The `geopandas` package provides a convenient way to plot the yield data on a world map. A full description of geographic information systems (GIS) is beyond the scope of this book, but `geopandas` is relatively self-contained and easy to use. First, read in the `DataFrame` for a low-resolution earth map (included with `geopandas`), and plot it on a Matplotlib Axes object. We'll accept the default equirectangular projection but customize the borders and fill the land areas in gray:

```
import geopandas
```

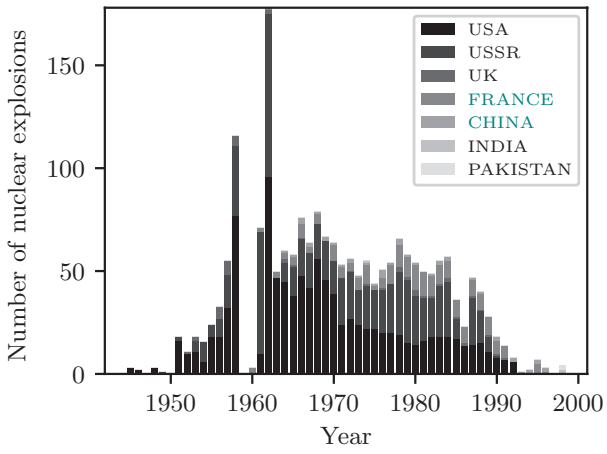

Figure 9.8 Stacked bar chart of the number of nuclear explosions by year caused by different countries between 1945 and 1998.

```
world = geopandas.read_file(geopandas.datasets.get_path('naturalearth_lowres'))
fig, ax = plt.subplots()
world.plot(ax=ax, color="0.8", edgecolor='black', linewidth=0.5)
```

The data provide lower and upper estimates of the explosion yield, so take the average and add circles as a scatter plot at the explosions' latitudes and longitudes. There is quite a large dynamic range from a few kilotons of TNT up to 50 million tons for the Tsar Bomba hydrogen bomb test of 1961, so clip the lower circle size to ensure that the smaller explosions are visible on the map:

```
df['yield_estimate'] = df[['yield_lower','yield_upper']].mean(axis=1)
sizes = (df['yield_estimate'] / 120).clip(10)
ax.scatter(df['long'], df['lat'], s=sizes, fc='r', ec='none', alpha=0.5)
ax.set_ylim(-60, 90)
plt.axis('off')
plt.show()
```

The result is Figure 9.9.

Example E9.14 The file `volcanic-eruptions.csv`, available to download at https://scipython.com/eg/bap, contains data concerning 822 significant volcanic events on Earth between 1750 BCE and 2020 CE from the US National Centres for Environmental Information (NCEI).[15] The information on each event is given in comma-separated fields and includes date, volcano name, location, type, estimated number of human deaths and "Volcanic Explosivity Index" (VEI).

[15] https://www.ngdc.noaa.gov/hazard/volcano.shtml.

Figure 9.9 A map of nuclear explosions, showing the blast yield, between 1945 and 1998.

The data are readily parsed into a `DataFrame` with:

```
In [x]: df = pd.read_csv('volcanic-eruptions.csv', index_col=0)
```

The most deadly volcanic eruption in the database is that of Ilopango, around the middle of the fifth century CE:

```
In [x]: df.loc[df['Deaths'].idxmax()]
Out[x]:
Year                 450
Month                NaN
Day                  NaN
Name             Ilopango
Location     El Salvador
Country      El Salvador
Latitude          13.672
Longitude        -89.053
Elevation            450
Type              Caldera
VEI                    6
Deaths             30000
Name: 25, dtype: object
```

It would be helpful to have a column with the day, month and year of the explosion parsed into a string. Define a helper function, `get_date`:

```
def get_date(year, month, day):
    if year < 0:
        s_year = f'{-year} BCE'
    else:
        s_year = str(year)
    if pd.isnull(month):
        return s_year
    s_date = f'{int(month)}/{s_year}'
    if pd.isnull(day):
        return s_date
    return f'{int(day)}/{s_date}'
```

and apply it to the `DataFrame`:

```
In [x]: df['date'] = [get_date(year, month, day) for year, month, day in
                                    zip(df['Year'], df['Month'], df['Day'])]
```

Simple filtering can give us a list of eruptions with a VEI of at least 6 since the start of the nineteenth century:

```
In [x]: df[(df['VEI'] >= 6) & (df['Year'] >= 1800)]
Out[x]:
     Year  Month   Day         Name  ...           Type  VEI   Deaths        date
218  1815    4.0  10.0      Tambora  ...  Stratovolcano  7.0  11000.0   10/4/1815
322  1883    8.0  27.0     Krakatau  ...        Caldera  6.0   2000.0   27/8/1883
365  1902   10.0  25.0  Santa Maria  ...  Stratovolcano  6.0   2500.0  25/10/1902
386  1912    9.0   6.0    Novarupta  ...        Caldera  6.0      2.0    6/9/1912
650  1991    6.0  15.0     Pinatubo  ...  Stratovolcano  6.0    350.0   15/6/1991
```

To find the 10 most explosive eruptions, we could filter out those with unknown VEI values before sorting:

```
In [x]: df[pd.notnull(df['VEI'])].sort_values('VEI').tail(10)[
   ...:              ['date', 'Name', 'Type', 'Country', 'VEI']]
Out[x]:
          date            Name            Type           Country  VEI
29         653        Dakataua         Caldera  Papua New Guinea  6.0
25         450        Ilopango         Caldera       El Salvador  6.0
22         240         Ksudach   Stratovolcano            Russia  6.0
21         230           Taupo         Caldera       New Zealand  6.0
18          60  Bona-Churchill   Stratovolcano     United States  6.0
99   19/2/1600    Huaynaputina   Stratovolcano              Peru  6.0
1     1750 BCE      Veniaminof   Stratovolcano     United States  6.0
40        1000    Changbaishan   Stratovolcano       North Korea  7.0
218  10/4/1815         Tambora   Stratovolcano         Indonesia  7.0
3     1610 BCE       Santorini  Shield volcano            Greece  7.0
```

However, there are many entries with a VEI of 6 and their ordering here is not clear. A better approach might be to sort first by VEI and next by deaths, setting `na_position='first'` to ensure that the null values are placed before numerical values (and therefore effectively rank lowest):

```
In [x]: df.sort_values(['VEI', 'Deaths'], na_position='first').tail(10)[
   ...:             ['date', 'Name', 'Type', 'Country', 'VEI', 'Deaths']]
Out[x]:
           date          Name             Type           Country  VEI   Deaths
386     6/9/1912      Novarupta         Caldera     United States  6.0      2.0
650    15/6/1991       Pinatubo   Stratovolcano       Philippines  6.0    350.0
99     19/2/1600  Huaynaputina   Stratovolcano              Peru  6.0   1500.0
120         1660   Long Island  Complex volcano  Papua New Guinea  6.0   2000.0
322    27/8/1883       Krakatau         Caldera         Indonesia  6.0   2000.0
365   25/10/1902   Santa Maria   Stratovolcano         Guatemala  6.0   2500.0
25           450       Ilopango         Caldera       El Salvador  6.0  30000.0
3       1610 BCE      Santorini  Shield volcano            Greece  7.0      NaN
40          1000  Changbaishan   Stratovolcano       North Korea  7.0      NaN
218    10/4/1815        Tambora   Stratovolcano         Indonesia  7.0  11000.0
```

We can also plot some histograms summarizing the data (Figure 9.10):

```
fig, axes = plt.subplots(nrows=2, ncols=2)
df['Day'].hist(bins=31, ax=axes[0][0], grid=False)
```

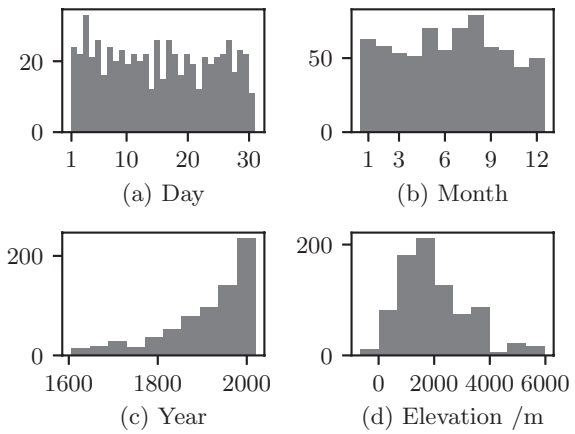

Figure 9.10 Histograms summarizing some of the columns of volcanic event data: (a) day of month; (b) month of year; (c) frequency by year since 1600 – hopefully, volcanic events have been progressively better recorded since 1600 and have not actually increased in frequency; (d) volcano elevation.

```
axes[0][0].set_xlabel('(a) Day')
df['Month'].hist(bins=np.arange(1, 14) - 0.5, ax=axes[0][1], grid=False)
axes[0][1].set_xticks(range(1, 13))
axes[0][1].set_xlabel('(b) Month')
df[df['Year']>1600]['Year'].hist(ax=axes[1][0], grid=False)
axes[1][0].set_xlabel('(c) Year')
df['Elevation'].hist(ax=axes[1][1], grid=False)
axes[1][1].set_xlabel('(d) Elevation /m')
plt.tight_layout()
plt.show()
```