CHAPTER 18

Data Input and Output

In nearly all scientific computing and data analysis applications there is a need for data input and output, for example, to load datasets or to persistently store results. Getting data in and out of programs is consequently a key step in the computational workflow. There are many standardized formats for storing structured and unstructured data. The benefits of using standardized formats are obvious: you can use existing libraries for reading and writing data, saving yourself both time and effort. In the course of working with scientific and technical computing, it is likely that you will face a variety of data formats through interaction with colleagues and peers, or when acquiring data from sources such as equipment and databases. As a computational practitioner, it is important to be able to handle data efficiently and seamlessly, regardless of which format it comes in. This motivates why this entire chapter is devoted to this topic.

Python has good support for many file formats. In fact, multiple options exist for dealing with most common formats. In this chapter we survey data storage formats with applications in computing, and discuss typical situations where each format is suitable. We also introduce Python libraries and tools for handling a selection of data formats that are common in computing.

Data can be classified into several categories and types. Important categories are structured and unstructured data, and values can be categorical (finite set of values), ordinal (values with meaningful ordering), or numerical (continuous or discrete). Values also have types, such as string, integer, floating-point number, etc. A data format for storing or transmitting data should ideally account for these concepts in order to avoid loss of data or metadata, and we frequently need to have fine-grained control of how data is represented.

In computing applications, most of the time we deal with structured data, for example, arrays and tabular data. Examples of unstructured datasets include free-form texts, or nested list with nonhomogeneous types. In this chapter we focus on the CSV family of formats and the HDF5 format for structured data, and toward the end of the chapter we discuss the JSON format as a lightweight and flexible format that can be used to store both simple and complex data sets, with a bias toward storing lists and dictionaries. This format is well suited for storing unstructured data. We also briefly discuss methods of serializing objects into storable data using the msgpack format and Python's built-in pickle format.

Because of the importance of data input and output in many data-centric computational applications, several Python libraries have emerged with the objective to simplify and assist in handling data in different formats, and for moving and converting data. For example, the Blaze library (`http://blaze.pydata.org/en/latest`) provides a high-level interface for accessing data of different formats and from different types of sources. Here we focus mainly on lower-level libraries for reading specific types of file formats that are useful for storing numerical data and unstructured datasets. However, the interested reader is encouraged to also explore higher-level libraries such as Blaze.

R. Johansson, *Numerical Python*, DOI 10.1007/978-1-4842-0553-2_18

Importing Modules

In this chapter we use a number of different libraries for handling different types of data. In particular, we require NumPy and pandas, which as usual we import as `np` and `pd`, respectively:

```
In [1]: import numpy as np
In [2]: import pandas as pd
```

We also use the `csv` and `json` modules from the Python standard library:

```
In [3]: import csv
In [4]: import json
```

For working with the HDF5 format for numerical data, we use the `h5py` and the `pytables` libraries:

```
In [5]: import h5py
In [6]: import tables
```

Finally, in the context of serializing objects to storable data, we explore the `pickle` and `msgpack` libraries:

```
In [7]: import pickle  # or alternatively: import cPickle as pickle
In [8]: import msgpack
```

Comma-Separated Values

Comma-separated values (CSV) is an intuitive and loosely defined[1] plain-text file format that is simple yet effective, and very prevalent for storing tabular data. In this format each record is stored as a line, and each field of the record is separated with a delimiter character (for example, a comma). Optionally, each field can be enclosed in quotation marks, to allow for string-valued fields that contain the delimiter character. Also, the first line is sometimes used to store column names, and comment lines are also common. An example of a CSV file is shown in Listing 18-1.

Listing 18-1. Example of a CSV file with a comment line, a header line, and mixed numerical and string-valued data fields. Data source: http://www.nhl.com

```
# 2013-2014 / Regular Season / All Skaters / Summary / Points
Rank,Player,Team,Pos,GP,G,A,P,+/-,PIM,PPG,PPP,SHG,SHP,GW,OT,S,S%,TOI/GP,Shift/GP,FO%
1,Sidney Crosby,PIT,C,80,36,68,104,+18,46,11,38,0,0,5,1,259,13.9,21:58,24.0,52.5
2,Ryan Getzlaf,ANA,C,77,31,56,87,+28,31,5,23,0,0,7,1,204,15.2,21:17,25.2,49.0
3,Claude Giroux,PHI,C,82,28,58,86,+7,46,7,37,0,0,7,1,223,12.6,20:26,25.1,52.9
4,Tyler Seguin,DAL,C,80,37,47,84,+16,18,11,25,0,0,8,0,294,12.6,19:20,23.4,41.5
5,Corey Perry,ANA,R,81,43,39,82,+32,65,8,18,0,0,9,1,280,15.4,19:28,23.2,36.0
```

CSV is occasionally also taken to be an acronym for character-separated value, reflecting the fact that the CSV format commonly refers to a family of formats using different delimiters between the fields. For example, instead of comma the TAB character is often used, in which case the format is sometimes call TSV instead of CSV. The term delimiter-separated values (DSV) is also occasionally used to refer to these types of formats.

[1]Although RFC 4180, http://tools.ietf.org/html/rfc4180, is sometimes taken as an unofficial specification, in practice there exist many varieties and dialects of CSV.

In Python there are several ways to read and write data in the CSV format, each with different use-cases and advantages. To begin with, the standard Python library contains a module called `csv` for reading CSV data. To use this module we can call the `csv.reader` function with a file handle given as argument. It returns a class instance that can be used as an iterator that parses lines from the given CSV file into Python lists of strings. For example, to read the file `playerstats-2013-2014.csv` (shown in Listing 18-1) into a nested list of strings, we can use:

```
In [9]: rows = []
In [10]: with open("playerstats-2013-2014.csv") as f:
    ...:     csvreader = csv.reader(f)
    ...:     for fields in csvreader:
    ...:         rows.append(fields)
In [11]: rows[1][1:6]
Out[11]: ['Player', 'Team', 'Pos', 'GP', 'G']
In [12]: rows[2][1:6]
Out[12]: ['Sidney Crosby', 'PIT', 'C', '80', '36']
```

Note that by default each field in the parsed rows is string-valued, even if the field represents a numerical value, such as 80 (games played) or 36 (goals) in the example above. While the `csv` module provides a flexible way of defining custom CSV reader classes, this module is most convenient for reading CSV files with string-valued fields.

In computational work it is common to store and load arrays with numerical values, such as vectors and matrices. The NumPy library provides the `np.loadtxt` and `np.savetxt` for this purpose. These functions take several arguments to fine tune the type of CSV format to read or write: For example, with the `delimiter` argument we can select which character to use to separate fields, and the `header` and `comments` arguments can be used to specify a header row and comment rows that are prepended to the header, respectively.

As an example, consider saving an array with random numbers and of shape (100, 3) to a file `data.csv` using `np.savetxt`. To give the data some context we add a header and a comment line to the file as well, and we explicitly request using the comma character as field delimiter with the argument `delimiter=","` (the default delimiter is the space character):

```
In [13]: data = np.random.randn(100, 3)
In [14]: np.savetxt("data.csv", data, delimiter=",", header="x,y,z",
    ...:            comments="# Random x, y, z coordinates\n")
In [15]: !head -n 5 data.csv
# Random x, y, z coordinates
x,y,z
1.6522766342545047 72e-01,9.522165919962696234e-01,4.659850998659530452e-01
8.699729536125471174e-01,1.187589118344758443e+00,1.788104702180680405e+00
-8.106725710122602013e-01,2.765616277935758482e-01,4.456864674903074919e-01
```

To read data on this format back into a NumPy array we can use the `np.loadtxt` function. It takes arguments that are similar to those of `np.savetxt`: In particular, we again set the `delimiter` argument to `","`, to indicate the fields are separated by a comma character. We also need to use the `skiprows` argument to skip over the first two lines in the file (the comment and header line), since they do not contain numerical data:

```
In [16]: data_load = np.loadtxt("data.csv", skiprows=2, delimiter=",")
```

The result is a new NumPy array that is equivalent to the original one written to the `data.csv` file using `np.savetxt`:

```
In [17]: (data == data_load).all()
Out[17]: True
```

Note that in contrast to the CSV reader in the `csv` module in the Python standard library, by default the `loadtxt` function in NumPy converts all fields into numerical values, and the result is a NumPy with numerical `dtype` (`float64`):

```
In [18]: data_load[1,:]
Out[18]: array([ 0.86997295,  1.18758912,  1.7881047 ])
In [19]: data_load.dtype
Out[19]: dtype('float64')
```

To read CSV files that contain non-numerical data using `np.loadtxt` – such as the `playerstats-2013-2014.csv` file that we read using the Python standard library above – we must explicitly set the data type of the resulting array using the `dtype` argument. If we attempt to read a CSV file with non-numerical values without setting `dtype` we get an error:

```
In [20]: np.loadtxt("playerstats-2013-2014.csv", skiprows=2, delimiter=",")
---------------------------------------------------------------------------
ValueError: could not convert string to float: b'Sidney Crosby'
```

Using `dtype=bytes` (or `str` or `object`), we get a NumPy array with unparsed values:

```
In [21]: data = np.loadtxt("playerstats-2013-2014.csv", skiprows=2, delimiter=",",
                           dtype=bytes)
In [22]: data[0][1:6]
Out[22]: array([b'Sidney Crosby', b'PIT', b'C', b'80', b'36'], dtype='|S13')
```

Alternatively, if we want to read only columns with numerical types, we can select to read a subset of columns using the `usecols` argument:

```
In [23]: np.loadtxt("playerstats-2013-2014.csv", skiprows=2, delimiter=",", usecols=[6,7,8])
Out[23]: array([[  68.,  104.,   18.],
                [  56.,   87.,   28.],
                [  58.,   86.,    7.],
                [  47.,   84.,   16.],
                [  39.,   82.,   32.]])
```

While the NumPy `savetxt` and `loadtxt` functions are configurable and flexible CSV writers and readers, they are most convenient for all numerical data. The Python standard library module `csv`, on the other hand, is most convenient for CSV files with string-valued data. A third method to read CSV files in the Python is to use the pandas `read_csv` function. We have already seen examples of this function in Chapter 12, where we used it to create pandas data frames from TSV formatted data files. The `read_csv` function in Pandas is very handy when reading CSV files with both numerical and string-valued fields, and in most cases it will automatically determine which type a field has and converts it accordingly. For example, when reading

the playerstats-2013-2014.csv file using read_csv, we obtain a pandas data frame with all the fields parsed into columns with suitable type:

```
In [24]: df = pd.read_csv("playerstats-2013-2014.csv", skiprows=1)
In [25]: df = df.set_index("Rank")
In [26]: df[["Player", "GP", "G", "A", "P"]]
Out[26]:
```

Rank	Player	GP	G	A	P
1	Sidney Crosby	80	36	68	104
2	Ryan Getzlaf	77	31	56	87
3	Claude Giroux	82	28	58	86
4	Tyler Seguin	80	37	47	84
5	Corey Perry	81	43	39	82

Using the info method of the DataFrame instance df we can see explicitly which type each column has been converted to (here the output is truncated for brevity):

```
In [27]: df.info()
<class 'pandas.core.frame.DataFrame'>
Int64Index: 5 entries, 1 to 5
Data columns (total 20 columns):
Player      5 non-null object
Team        5 non-null object
Pos         5 non-null object
GP          5 non-null int64
G           5 non-null int64
...
S           5 non-null int64
S%          5 non-null float64
TOI/GP      5 non-null object
Shift/GP    5 non-null float64
FO%         5 non-null float64
dtypes: float64(3), int64(13), object(4)
memory usage: 840.0+ bytes
```

Data frames can also be written to CSV files using the to_csv method of the DataFrame object:

```
In [28]: df[["Player", "GP", "G", "A", "P"]].to_csv("playerstats-2013-2014-subset.csv")
In [29]: !head -n 5 playerstats-2013-2014-subset.csv
Rank,Player,GP,G,A,P
1,Sidney Crosby,80,36,68,104
2,Ryan Getzlaf,77,31,56,87
3,Claude Giroux,82,28,58,86
4,Tyler Seguin,80,37,47,84
```

The combination of the Python standard library, NumPy, and Pandas provides a powerful toolbox for both reading and writing CSV files of various flavors. However, although CSV files are convenient and effective for tabular data, there are obvious shortcomings with the format. For starters, it can only be used to store one- or two-dimensional arrays, and it does not contain metadata that can help interpret the data. Also, it is not very efficient in terms of either storage or reading and writing, and it cannot be used to store more than one array per file, requiring multiple files for multiple arrays even if they are closely related. The use of CSV should therefore be used be limited to simple datasets. In the following section we will look the HDF5 file format, which was designed to store numerical data efficiently and to overcome all the shortcomings of simple data formats such as CSV and related formats.

HDF5

The *Hierarchical Data Format 5* (HDF5) is a format for storing numerical data. It is developed by The HDF Group,[2] a nonprofit organization, and it is available under the BSD open source license. The HDF5 format, which was released in 1998, is designed and implemented to efficiently handle large datasets, including support for high-performance parallel I/O. The HDF5 format is therefore suitable for use on distributed high-performance supercomputers, and can be used to store and operate on datasets of terabyte scale, or even larger. However, the beauty of HDF5 is that it is equally suitable for small datasets. As such it is a truly versatile format, and an invaluable tool for a computational practitioner.

The hierarchical aspect of the format allows organizing datasets within a file, using a hierarchical structure that resembles a filesystem. The terminology used for entities in a HDF5 file is *groups* and *datasets*, which correspond to directories and files in the filesystem analogy. Groups in an HDF5 file can be nested to create a tree structure, and hence *hierarchical* in the name of the format. A dataset in an HDF5 file is a homogenous array of certain dimensions and elements of a certain type. The HDF5 type system supports all standard basic data types and also allows defining custom compound data types. Both groups and datasets in an HDF5 file can also have *attributes*, which can be used to store metadata about groups and datasets. Attributes can themselves have different types, such as numeric or string valued.

In addition to the file format itself, The HDF Group also provides a library and a reference implementation of the format. The main library is written in C, and wrappers to its C API are available for many programming languages. The HDF5 library for accessing data from an HDF5 file have sophisticated support for partial read and write operations, which can be used to access a small segment of the entire dataset. This is a powerful feature that enables computations on datasets that are larger than what can be fit a computer's memory.[3] The HDF5 format is a mature file format with widespread support on different platforms and computational environments. This also makes HDF5 a suitable choice for long-term storage of data. As a data storage platform HDF5 provides a solution to a number of problems: cross-platform storage, efficient I/O and storage that scales up to very large data files, and a metadata system (attributes) that can be used to annotate and describe the groups and datasets in a file to make the data self-describing. Altogether, these features make HDF5 a great tool for computational work.

For Python there are two libraries for using HDF5 files: h5py and PyTables. These two libraries take different approaches to using HDF5, and it is well worth being familiar with both of these libraries. The h5py library provides an API that is relatively close to the basic HDF5 concepts, with a focus on groups and datasets. It provides a NumPy-inspired API for accessing datasets, which makes it very intuitive for someone that is familiar with NumPy.

[2] http://www.hdfgroup.org.

[3] This is also known as out-of-core computing. For another recent project that also provides out-of-core computing capabilities in Python, see the dask library (http://dask.pydata.org/en/latest).

■ **h5py** The h5py library provides a Pythonic interface to the HDF5 file format, and a NumPy-like interface to its datasets. For more information about the project, including its official documentation, see its web page at http://www.h5py.org. At the time of writing the most recent version of the library is 2.5.0.

The PyTables library provides a higher-level data abstraction based on the HDF5 format, providing database-like features, such as tables with easily customizable data types. It also allows querying datasets as a database and the use of advanced indexing features.

■ **PyTables** The PyTables library provides a database-like data model on top of HDF5. For more information about the project and its documentation, see the web page http://pytables.github.io. At the time of writing the latest version of PyTables is 3.2.0.

In the following two sections we explore in more detail how the h5py and PyTables libraries can be used to read and write numerical data with HDF5 files.

h5py

We begin with a tour of the h5py library. The API for h5py is surprisingly simple and pleasant to work with, yet at the same time full featured. This is accomplished through thoughtful use of Pythonic idiom such as dictionary and NumPy's array semantics. A summary of basic objects and methods in the h5py library is shown in Table 18-1. In the following we explore how to use these methods through a series of examples.

Table 18-1. *Summary of the main objects and methods in the h5py API*

Object	Method/Attribute	Description
`h5py.File`	`__init__(name, mode, ...)`	Open an existing HDF5, or create a new one, with filename `name`. Depending on the value of the mode argument, the file can be opened in read-only or read-write mode (see main text).
	`flush()`	Write buffers to file.
	`close()`	Close an open HDF5 file.
`h5py.File`, `h5py.Group`	`create_group(name)`	Create a new group with name `name` (can be a path) within the current group.
	`create_dataset(name, data=..., shape=..., dtype=..., ...)`	Create a new dataset.
	`[] dictionary syntax`	Access items (groups and datasets) within a group.

(*continued*)

Table 18-1. (*continued*)

Object	Method/Attribute	Description
`h5py.Dataset`	`dtype`	Data type.
	`shape`	Shape (dimensions) of the dataset.
	`value`	The full array of the underlying data of the dataset.
	`[] array syntax`	Access elements or subsets of the data in a dataset.
`h5py.File`, `h5py.Group`, `h5py.Dataset`	`name`	Name (path) of the object in the HDF5 file hierarchy.
	`attrs`	Dictionary-like attribute access.

Files

We begin by looking at how to open existing and create new HDF5 files using the `h5py.File` object. The initializer for this object only takes a file name as a required argument, but we will typically also need to specify the `mode` argument, with which we can choose to open a file in read-only or read-write mode, and if a file should be truncated or not when opened. The mode argument takes string values similar to the built-in Python function `open`: `"r"` is used for read-only (file must exist), `"r+"` for read-write (file must exist), `"w"` for creating a new file (truncate if exists), `"w-"` for creating a new file (error if exists), and `"a"` for read-write (if exist, otherwise create). To create a new file in read-write mode, we can therefore use:

```
In [30]: f = h5py.File("data.h5", mode="w")
```

The result is a file handle, here assigned to the variable `f`, which we can use to access and add content to the file. Given a file handle we can see which mode it is opened in using the `mode` attribute:

```
In [31]: f.mode
Out[31]: 'r+'
```

Note that even though we opened the file in mode `"w"`, once the file has been opened it is either read-only (`"r"`) or read-write (`"r+"`). Other file-level operations that can be performed using the HDF5 file object are flushing buffers containing data that has not yet been written to the file using the `flush` method, and closing the file using the `close` method:

```
In [32]: f.flush()
In [33]: f.close()
```

Groups

At the same time as representing an HDF5 file handle, the `File` object also represents the HDF5 group object known as the *root group*. The name of a group is accessible through the `name` attribute of the group object. The name takes the form of a path, similar to a path in a filesystem, which specifies where in the hierarchical structure of the file the group is stored. The name of the root group is `"/"`:

```
In [34]: f = h5py.File("data.h5", "w")
In [35]: f.name
Out[35]: '/'
```

A group object has the method `create_group` for creating a new group within an existing group. A new group created with this method becomes a subgroup of the group instance for which the `create_group` method is invoked:

```
In [36]: grp1 = f.create_group("experiment1")
In [37]: grp1.name
Out[37]: '/experiment1'
```

Here the group `experiment1` is a subgroup of root group, and its name and path in the hierarchical structure is therefore `/experiment1`. When creating a new group, its immediate parent group does not necessarily have to exist beforehand. For example, to create a new group `/experiment2/measurement`, we can directly use the `create_group` method of the root group *without* first creating the `experiment2` group explicitly. Intermediate groups are created automatically.

```
In [38]: grp2_meas = f.create_group("experiment2/measurement")
In [39]: grp2_meas.name
Out[39]: '/experiment2/measurement'
In [40]: grp2_sim = f.create_group("experiment2/simulation")
In [41]: grp2_sim.name
Out[41]: '/experiment2/simulation'
```

The group hierarchy of an HDF5 file can be explored using a dictionary-style interface. To retrieve a group with a given path name we can perform a dictionary-like lookup from one of its ancestor groups (typically the root node):

```
In [42]: f["/experiment1"]
Out[42]: <HDF5 group "/experiment1" (0 members)>
In [43]: f["/experiment2/simulation"]
Out[43]: <HDF5 group "/experiment2/simulation" (0 members)>
```

The same type of dictionary lookup works for subgroups, too (not only the root node):

```
In [44]: grp_experiment2 = f["/experiment2"]
In [45]: grp_experiment2['simulation']
Out[45]: <HDF5 group "/experiment2/simulation" (0 members)>
```

The `keys` method returns an iterator over the names of subgroups and datasets within a group, and the `items` method returns an iterator over `(name, value)` tuples for each entity in the group. These can be used to traverse the hierarchy of groups programmatically.

```
In [46]: list(f.keys())
Out[46]: ['experiment1', 'experiment2']
In [47]: list(f.items())
Out[47]: [('experiment1', <HDF5 group "/experiment1" (0 members)>),
          ('experiment2', <HDF5 group "/experiment2" (2 members)>)]
```

To traverse the hierarchy of groups in an HDF5 file we can also use the method `visit`, which takes a function as argument and calls that function with the name for each entity in the file hierarchy:

```
In [48]: f.visit(lambda x: print(x))
experiment1
experiment2
experiment2/measurement
experiment2/simulation
```

or the `visititems` method which does the same thing except that it calls the a function with both the item name and the item itself as argument:

```
In [49]: f.visititems(lambda name, item: print(name, item))
experiment1 <HDF5 group "/experiment1" (0 members)>
experiment2 <HDF5 group "/experiment2" (2 members)>
experiment2/experiment <HDF5 group "/experiment2/measurement" (0 members)>
experiment2/simulation <HDF5 group "/experiment2/simulation" (0 members)>
```

In keeping with the semantics of Python dictionaries we can also operate on `Group` objects using the set membership testing with the `in` Python keyword:

```
In [50]: "experiment1" in f
Out[50]: True
In [51]: "simulation" in f["experiment2"]
Out[51]: True
In [52]: "experiment3" in f
Out[52]: False
```

Using the `visit` and `visititems` methods, together with the dictionary-style methods `keys` and `items`, we can easily explore the structure and content of an HDF5 file, even if we have no prior information on what it contains and how the data is organized within it. The ability to conveniently explore HDF5 is an important aspect of the usability of the format. There are also external non-Python tools for exploring the content of an HDF5 file that frequently are useful when working with this type of file. In particular, the `h5ls` command-line tool is handy for quickly inspecting the content of an HDF5 file:

```
In [53]: f.flush()
In [54]: !h5ls -r data.h5
/                        Group
/experiment1             Group
/experiment2             Group
/experiment2/measurement Group
/experiment2/simulation  Group
```

Here we used the `-r` flag to the `h5ls` program to recursively show all items in the file. The `h5ls` program is part of a series of HDF5 utility programs provided by a package called `hdf5-tools` (see also `h5stat`, `h5copy`, `h5diff`, etc.). Even though these are not Python tools, they are very useful when working with HDF5 files in general, also from within Python.

Datasets

Now that we have explored how to create and access groups within an HDF5 file, it is time to look at how to store datasets. Storing numerical data is after all the main purpose of the HDF5 format. There are two main methods to create a datasets in an HDF5 file using h5py. The easiest way to create a dataset is to simply assign a NumPy array to an item within a HDF5 group, using the dictionary index syntax. The second method is to create an empty dataset using the `create_dataset` method, as we will see examples of later in this section.

For example, to store two NumPy arrays, `array1` and `meas1`, into the root group and the `experiment2/measurement` groups, respectively, we can use:

```
In [55]: array1 = np.arange(10)
In [56]: meas1 = np.random.randn(100, 100)
In [57]: f["array1"] = array1
In [58]: f["/experiment2/measurement/meas1"] = meas1
```

To verify that the datasets for the assigned NumPy arrays where added to the file, let's traverse through the file hierarchy using the `visititems` method:

```
In [59]: f.visititems(lambda name, value: print(name, value))
array1 <HDF5 dataset "array1": shape (10,), type "<i8">
experiment1 <HDF5 group "/experiment1" (0 members)>
experiment2 <HDF5 group "/experiment2" (2 members)>
experiment2/measurement <HDF5 group "/experiment2/measurement" (1 members)>
experiment2/measurement/meas1 <HDF5 dataset "meas1": shape (100, 100), type "<f8">
experiment2/simulation <HDF5 group "/experiment2/simulation" (0 members)>
```

We see that indeed, the `array1` and `meas1` datasets are now added to the file. Note that the paths used as dictionary keys in the assignments determine the locations of the datasets within the file. To retrieve a dataset we can use the same dictionary-like syntax as we used to retrieve a group. For example, to retrieve the `array1` dataset, which is stored in the root group, we can use `f["array1"]`:

```
In [60]: ds = f["array1"]
In [61]: ds
Out[61]: <HDF5 dataset "array1": shape (10,), type "<i8">
```

The result is a `Dataset` object, not a NumPy array like the one that we assigned to the `array1` item. The `Dataset` object is a proxy for the underlying data within the HDF5. Like a NumPy array, a `Dataset` object has several attributes that describe the dataset, including `name`, `dtype`, and `shape`. It also has the method `len` that returns the length of the dataset:

```
In [62]: ds.name
Out[62]: '/array1'
In [63]: ds.dtype
Out[63]: dtype('int64')
In [64]: ds.shape
Out[64]: (10,)
In [65]: ds.len()
Out[65]: 10
```

The actual data for the dataset can be accessed using the value attribute. This returns the entire dataset as a NumPy array, which here is equivalent to the array that we assigned to the `array1` dataset.

```
In [66]: ds.value
Out[66]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

To access a dataset deeper down the group hierarchy we can use a filesystem-like path name. For example, to retrieve the `meas1` dataset in the group `experiment2/measurement`, we can use:

```
In [67]: ds = f["experiment2/measurement/meas1"]
In [68]: ds
Out[68]: <HDF5 dataset "meas1": shape (100, 100), type "<f8">
```

Again we get a `Dataset` object, whose basic properties can be inspected using the object attributes we introduced earlier:

```
In [69]: ds.dtype
Out[69]: dtype('float64')
In [70]: ds.shape
Out[70]: (100, 100)
```

Note that the data type of this dataset is `float64`, while for the dataset `array1` the data type is `int64`. This type of information was derived from the NumPy arrays that were assigned to the two datasets. Here again we could use the value attribute to retrieve the array as a NumPy array. An alternative syntax for the same operation is to use bracket indexing with the ellipsis notation: `ds[...]`.

```
In [71]: data_full = ds[...]
In [72]: type(data_full)
Out[72]: numpy.ndarray
In [73]: data_full.shape
Out[73]: (100, 100)
```

This is an example of NumPy-like array indexing. The `Dataset` object supports most of the indexing and slicing types used in NumPy, and this provides a powerful and flexible method for partially reading data from a file. For example, to retrieve only the first column from the `meas1` dataset, we can use:

```
In [74]: data_col = ds[:, 0]
In [75]: data_col.shape
Out[75]: (100,)
```

The result is a 100-element array corresponding to the first column in the dataset. Note that this slicing is performed within the HDF5 library, and not in NumPy, so in this example only 100 elements were read from the file and stored in the resulting NumPy array, without every fully loading the dataset into memory. This is an important feature when working with large datasets that do not fit in memory.

For example, the `Dataset` object also supports strided indexing:

```
In [76]: ds[10:20:3, 10:20:3] # 3 stride
Out[76]: array([[-0.22321057, -0.61989199,  0.78215645,  0.73774187],
                [-1.03331515,  2.54190817, -0.24812478, -2.49677693],
                [ 0.17010011,  1.88589248,  1.91401249, -0.63430569],
                [ 0.4600099 , -1.3242449 ,  0.41821078,  1.47514922]])
```

as well as "fancy indexing," where a list of indices are given for one of the dimensions of the array (does not work for more than one index):

```
In [77]: ds[[1,2,3], :].shape
Out[77]: (3, 100)
```

We can also use Boolean indexing, where a Boolean-valued NumPy array is used to index a `Dataset`. For example, to single out the first five columns (index `:5` on the second axis) for each row whose value in the first column (`ds[:, 0]`) is larger than 2, we can index the dataset with the Boolean mask `ds[:, 0] > 2`:

```
In [78]: mask = ds[:, 0] > 2
In [79]: mask.shape, mask.dtype
Out[79]: ((100,), dtype('bool'))
In [80]: ds[mask, :5]
Out[80]: array([[ 2.1224865 ,  0.70447132, -1.71659513,  1.43759445, -0.61080907],
                [ 2.11780508, -0.2100993 ,  1.06262836, -0.46637199,  0.02769476],
                [ 2.41192679, -0.30818179, -0.31518842, -1.78274309, -0.80931757],
                [ 2.10030227,  0.14629889,  0.78511191, -0.19338282,  0.28372485]])
```

Since the `Dataset` object uses the NumPy's indexing and slicing syntax to select subsets of the underlying data, working with large HDF5 datasets in Python using h5py comes very naturally to someone who is familiar with NumPy. Also remember that for large files, there is a big difference in index slicing on the `Dataset` object rather than on the NumPy array that can be access through the `value` attribute, since the former avoids loading the entire dataset into memory.

So far we have seen how to create datasets in an HDF5 file by explicitly assigning data into an item in a group object. We can also create datasets explicitly using the `create_dataset` method. It takes the name of the new dataset as the first argument, and we can either set the data for the new dataset using the `data` argument, or create an empty array by setting the `shape` argument. For example, instead of the assignment `f["array2"] = np.random.randint(10, size=10)`, we can also use the `create_dataset` method:

```
In [81]: ds = f.create_dataset("array2", data=np.random.randint(10, size=10))
In [82]: ds
Out[82]: <HDF5 dataset "array2": shape (10,), type "<i8">
In [83]: ds.value
Out[83]: array([2, 2, 3, 3, 6, 6, 4, 8, 0, 0])
```

When explicitly calling the `create_dataset` method, we have a finer level of control of the properties of the resulting data set. For example, we can explicitly set the data type for the dataset using the `dtype` argument, and we can choose a compression method using the `compress` argument, setting the chunk size using the `chunks` argument, and setting the maximum allowed array size for resizable datasets using the `maxsize` argument. There are also many other advanced features related to the `Dataset` object. See the docstring for `create_dataset` for details.

When creating an empty array by specifying the `shape` argument instead of providing an array for initializing a dataset, we can also use the `fillvalue` argument to set the default value for the dataset. For example, to create an empty dataset of shape (5, 5) and default value -1, we can use:

```
In [84]: ds = f.create_dataset("/experiment2/simulation/data1", shape=(5, 5), fillvalue=-1)
In [85]: ds
Out[85]: <HDF5 dataset "data1": shape (5, 5), type "<f4">
In [86]: ds.value
```

```
Out[86]: array([[-1., -1., -1., -1., -1.],
                [-1., -1., -1., -1., -1.],
                [-1., -1., -1., -1., -1.],
                [-1., -1., -1., -1., -1.],
                [-1., -1., -1., -1., -1.]], dtype=float32)
```

HDF5 is clever about disk usage for empty datasets and will not store more data than necessary, in particular if we select a compression method using the `compression` argument. There are several compression methods available, for example, `'gzip'`. Using dataset compression we can create very large datasets and gradually fill them with data, for example, when measurement results or results of computations become available, without initially wasting a lot of storage space. For example, let's create a large dataset with shape (5000, 5000, 5000) with the `data1` in the group `experiment1/simulation`:

```
In [87]: ds = f.create_dataset("/experiment1/simulation/data1", shape=(5000, 5000, 5000),
                               fillvalue=0, compression='gzip')
In [88]: ds
Out[88]: <HDF5 dataset "data1": shape (5000, 5000, 5000), type "<f4">
```

To begin with this dataset uses neither memory nor disk space, until we start filling it with data. To assign values to the dataset we can again use the NumPy-like indexing syntax and assign values to specific elements in the dataset, or to subsets selected using slicing syntax:

```
In [89]: ds[:, 0, 0] = np.random.rand(5000)
In [90]: ds[1, :, 0] += np.random.rand(5000)
In [91]: ds[:2, :5, 0]
Out[91]: array([[ 0.67240328, 0.        , 0.        , 0.        , 0.        ],
                [ 0.99613971, 0.48227152, 0.48904559, 0.78807044, 0.62100351]],
                dtype=float32)
```

Note that the elements that have not been assign values are set to the value of `fillvalue` that was specified when the array was created. If we do not know what fill value a dataset has, we can find out by looking at the `fillvalue` attribute of the `Dataset` object:

```
In [92]: ds.fillvalue
Out[92]: 0.0
```

To see that the newly created dataset is indeed stored in the group where we intended to assign it we can again use the `visititems` method to list the content of the `experiment1` group:

```
In [93]: f["experiment1"].visititems(lambda name, value: print(name, value))
simulation <HDF5 group "/experiment1/simulation" (1 members)>
simulation/data1 <HDF5 dataset "data1": shape (5000, 5000, 5000), type "<f4">
```

Although the dataset `experiment1/simulation/data1` is very large (4×5000^3 bytes ~ 465 Gb), since we have not yet filled it with much data the HDF5 file still does not take a lot of disk space (only about 357 Kb):

```
In [94]: f.flush()
In [95]: f.filename
Out[95]: 'data.h5'
In [96]: !ls -lh data.h5
-rw-r--r--@ 1 rob  staff   357K Apr  5 18:48 data.h5
```

So far we have seen how to create groups and datasets within an HDF5 file. It is of course sometimes also necessary to delete items from a file. With h5py we can delete items from a group using the Python `del` keyword, again complying with the semantics of Python dictionaries:

```
In [97]: del f["/experiment1/simulation/data1"]
In [98]: f["experiment1"].visititems(lambda name, value: print(name, value))
simulation <HDF5 group "/experiment1/simulation" (0 members)>
```

Attributes

Attributes are a component of the HDF5 format that makes it a great format for annotating data and providing self-describing data through the use of metadata. For example, when storing experimental data, there are often external parameters and conditions that should be recorded together with the observed data. Likewise, in a computer simulation, it is usually necessary to store additional model or simulation parameters together with the generated simulation results. In all these cases, the best solution is to make sure that the required additional parameters are stored as metadata together with the main datasets.

The HDF5 format supports this type of meta data through the use of attributes. An arbitrary number of attributes can be attached to each group and dataset within an HDF5 file. With the h5py library, attributes are accessed using a dictionary-like interface, just like groups are. The Python attribute `attrs` of `Group` and `Dataset` objects is used to access a dictionary-like object with HDF5 attributes:

```
In [99]: f.attrs
Out[99]: <Attributes of HDF5 object at 4462179384>
```

To create an attribute we simply assign to the `attrs` dictionary for the target object. For example, to create an attribute `description` for the root group, we can use:

```
In [100]: f.attrs["description"] = "Result sets for experiments and simulations"
```

Similarly, to add `date` attributes to the `experiment1` and `experiment2` groups:

```
In [101]: f["experiment1"].attrs["date"] = "2015-1-1"
In [102]: f["experiment2"].attrs["date"] = "2015-1-2"
```

We can also add attributes directly to datasets (not only groups):

```
In [103]: f["experiment2/simulation/data1"].attrs["k"] = 1.5
In [104]: f["experiment2/simulation/data1"].attrs["T"] = 1000
```

Like for groups, we can use the `keys` and `items` method of the `Attribute` object to retrieve iterators over the attributes it contains:

```
In [105]: list(f["experiment1"].attrs.keys())
Out[105]: ['date']
In [106]: list(f["experiment2/simulation/data1"].attrs.items())
Out[106]: [('k', 1.5), ('T', 1000)]
```

The existence of an attribute can be tested with the Python `in` operator, in keeping with the Python dictionary semantics:

```
In [107]: "T" in f["experiment2/simulation/data1"].attrs
Out[107]: True
```

To delete existing attributes we can use the `del` keyword:

```
In [108]: del f["experiment2/simulation/data1"].attrs["T"]
In [109]: "T" in f["experiment2/simulation"].attrs
Out[109]: False
```

The attributes of HDF5 groups and datasets are suitable for storing metadata together with the actual datasets. Using attributes generously can help providing context to the data, which often must be available for the data to be useful.

PyTables

The PyTables library offers an alternative interface to HDF5 for Python. The focus on this library is higher-level table-based data model implemented using the HDF5 format, although PyTables can also be used to create and read generic HDF5 groups and datasets, like the h5py library. Here we focus on the table data model, as it complements the h5py library that we discussed in the previous section. We demonstrate the use of PyTables table objects using the NHL player statistics dataset that we used earlier in this chapter, and where we construct a PyTables table from a Pandas data frame for that dataset. We therefore begin with reading in the dataset into a `DataFrame` object using the `read_csv` function:

```
In [110]: df = pd.read_csv("playerstats-2013-2014.csv", skiprows=1)
     ...: df = df.set_index("Rank")
```

Next we proceed to create a new PyTables HDF5 file handle by using the `tables.open_file` function.[4] This function takes a file name as first argument and the file mode as optional second argument. The result is a PyTables HDF5 file handle (here assigned to the variable `f`):

```
In [111]: f = tables.open_file("playerstats-2013-2014.h5", mode="w")
```

Like with the h5py library, we can create HDF5 groups with the method `create_group` of the file handle object. It takes the path to the parent group as the first argument, the group name as the second argument, and optionally also the argument `title`, with which a descriptive HDF5 attribute can be set on the group.

```
In [112]: grp = f.create_group("/", "season_2013_2014",
     ...:                      title="NHL player statistics for the 2013/2014 season")
In [113]: grp
Out[113]: /season_2013_2014 (Group) 'NHL player statistics for the 2013/2014 season'
            children := []
```

[4]Note that the Python module provided by the PyTables library is named `tables`. Therefore, `tables.open_file` refers to `open_file` function in the `tables` module provided by the PyTables library.

Unlike the h5py library, the file handle object in PyTables does not represent the root group in the HDF5 file. To access the root node, we must use the `root` attribute of the file handle object:

```
In [114]: f.root
Out[114]: / (RootGroup) ''
            children := ['season_2013_2014' (Group)]
```

A nice feature of the PyTables library is that it is easy to create tables with mixed column types, using the struct-like compound data type of HDF5. The simplest way to define such a table data structure with PyTables is to create a class that inherits from the `tables.IsDescription` class. It should contain fields composed of data-type representations from the `tables` library. For example, to create a specification of the table structure for the player statistics dataset we can use:

```
In [115]: class PlayerStat(tables.IsDescription):
     ...:     player = tables.StringCol(20, dflt="")
     ...:     position = tables.StringCol(1, dflt="C")
     ...:     games_played = tables.UInt8Col(dflt=0)
     ...:     points = tables.UInt16Col(dflt=0)
     ...:     goals = tables.UInt16Col(dflt=0)
     ...:     assists = tables.UInt16Col(dflt=0)
     ...:     shooting_percentage = tables.Float64Col(dflt=0.0)
     ...:     shifts_per_game_played = tables.Float64Col(dflt=0.0)
```

Here the class `PlayerStat` represents the table structure of a table with eight columns, where the first two columns are fixed-length strings (`tables.StringCol`), the following four columns are unsigned integers (`tables.UInt8Col` and `tables.UInt16Col`, of 8- and 16-bit size), and where the last two columns have floating-point types (`tables.Float64Col`). The optional `dflt` argument to data-type objects specifies the fields default value. Once the table structure is defined using a class on this form, we can create the actual table in the HDF5 file using the `create_table` method. It takes a group object or the path to the parent node as first argument, the table name as second argument, the table specification class as third argument, and optionally a table title as fourth argument (stored as an HDF5 attribute for the corresponding dataset):

```
In [116]: top30_table = f.create_table(grp, 'top30', PlayerStat, "Top 30 point leaders")
```

To insert data into the table we can use the `row` attribute of the table object to retrieve a `Row` accessor class that can be used as a dictionary to populate the row with values. When the row object is fully initialized, we can use the `append` method to actually insert the row into the table:

```
In [117]: playerstat = top30_table.row
In [118]: for index, row_series in df.iterrows():
     ...:     playerstat["player"] = row_series["Player"]
     ...:     playerstat["position"] = row_series["Pos"]
     ...:     playerstat["games_played"] = row_series["GP"]
     ...:     playerstat["points"] = row_series["P"]
     ...:     playerstat["goals"] = row_series["G"]
     ...:     playerstat["assists"] = row_series["A"]
     ...:     playerstat["shooting_percentage"] = row_series["S%"]
     ...:     playerstat["shifts_per_game_played"] = row_series["Shift/GP"]
     ...:     playerstat.append()
```

The `flush` method force a write of the table data to the file:

```
In [119]: top30_table.flush()
```

To access data from the table we can use the `cols` attribute to retrieve columns as NumPy arrays:

```
In [120]: top30_table.cols.player[:5]
Out[120]: array([b'Sidney Crosby', b'Ryan Getzlaf', b'Claude Giroux',
                 b'Tyler Seguin', b'Corey Perry'],        dtype='|S20')
In [121]: top30_table.cols.points[:5]
Out[121]: array([104,  87,  86,  84,  82], dtype=uint16)
```

To access data in a row-wise fashion we can use the `iterrows` method to create an iterator over all the rows in the table. Here we use this approach to loop through all the rows and print them to the standard output (here the output is truncated for brevity):

```
In [122]: def print_playerstat(row):
     ...:     print("%20s\t%s\t%s\t%s" %
     ...:           (row["player"].decode('UTF-8'), row["points"],
     ...:            row["goals"], row["assists"]))
In [123]: for row in top30_table.iterrows():
     ...:     print_playerstat(row)
Sidney Crosby   104     36      68
Ryan Getzlaf    87      31      56
Claude Giroux   86      28      58
Tyler Seguin    84      37      47
...
Jaromir Jagr    67      24      43
John Tavares    66      24      42
Jason Spezza    66      23      43
Jordan Eberle   65      28      37
```

One of the most powerful features of the PyTables table interface is the ability to selectively extract rows from the underlying HDF5 using queries. For example, the `where` method allows us to pass an expression in terms of the table columns as a string that is used by PyTables to filter rows:

```
In [124]: for row in top30_table.where("(points > 75) & (points <= 80)"):
     ...:     print_playerstat(row)
Phil Kessel             80      37      43
Taylor Hall             80      27      53
Alex Ovechkin           79      51      28
Joe Pavelski            79      41      38
Jamie Benn              79      34      45
Nicklas Backstrom       79      18      61
Patrick Sharp           78      34      44
Joe Thornton            76      11      65
```

With the where method we can also define conditions in terms of multiple column variables:

```
In [125]: for row in top30_table.where("(goals > 40) & (points < 80)"):
     ...:     print_playerstat(row)
Alex Ovechkin   79      51      28
Joe Pavelski    79      41      38
```

What this feature allows us to do is to query a table in a database-like fashion. Although for a small dataset like the current one, we could just as well perform these kind of operations directly in memory using a pandas data frame, but remember that HDF5 files are stored on disk, and the efficient use of I/O in the PyTables library enables us to work with very large datasets that do not fit in memory, which would preventing us from using for example NumPy or pandas on the entire dataset.

Before we conclude this section, let us inspect the structure of the resulting HDF5 file that contains the PyTables table that we have just created:

```
In [126]: f
Out[126]: File(filename=playerstats-2013-2014.h5, title='', mode='w', root_uep='/',
               filters=Filters(complevel=0, shuffle=False,
                               fletcher32=False, least_significant_digit=None))
          / (RootGroup) '' /season_2013_2014 (Group) 'NHL player stats for the 2013/2014 season'
          /season_2013_2014/top30 (Table(30,)) 'Top 30 point leaders'
              description := {
              "assists": UInt16Col(shape=(), dflt=0, pos=0),
              "games_played": UInt8Col(shape=(), dflt=0, pos=1),
              "goals": UInt16Col(shape=(), dflt=0, pos=2),
              "player": StringCol(itemsize=20, shape=(), dflt=b'', pos=3),
              "points": UInt16Col(shape=(), dflt=0, pos=4),
              "position": StringCol(itemsize=1, shape=(), dflt=b'C', pos=5),
              "shifts_per_game_played": Float64Col(shape=(), dflt=0.0, pos=6),
              "shooting_percentage": Float64Col(shape=(), dflt=0.0, pos=7)}
            byteorder := 'little'
            chunkshape := (1489,)
```

From the string representation of the PyTables file handle, and the HDF5 file hierarchy that it contains, we can see that the PyTables library has created a dataset /season_2013_2014/top30 that uses an involved compound data type that was created according to the specification in the PlayerStat object that we created earlier. Finally, when we are finished modifying a dataset in a file we can flush its buffers and force a write to the file using the flush method, and when we are finished working with a file we can close it using the close method:

```
In [127]: f.flush()
In [128]: f.close()
```

Although we do not cover other types of datasets here, such as regular homogenous arrays, it is worth mentioning that the PyTables library supports these types of data structures as well (similar to what h5py provides). For example, we can use the create_array, create_carray, and create_earray to construct fixed-sized arrays, chunked arrays, and enlargeable arrays, respectively. For more information on how to use these data structures, see the corresponding docstring.

Pandas HDFStore

A third way to store data in HDF5 files using Python is to use the `HDFStore` object in pandas. It can be used to persistently store data frames and other pandas objects in an HDF5 file. To use this feature in pandas, the PyTables library must be installed. We can create an `HDFStore` object by passing a file name to its initializer. The result is an `HDFStore` object that can be used as a dictionary to which we can assign pandas `DataFrame` instances to have them stored into the HDF5 file:

```
In [129]: store = pd.HDFStore('store.h5')
In [130]: df = pd.DataFrame(np.random.rand(5,5))
In [131]: store["df1"] = df
In [132]: df = pd.read_csv("playerstats-2013-2014-top30.csv", skiprows=1)
In [133]: store["df2"] = df
```

The HDFStore object behaves as a regular Python dictionary, and we can see what objects it contains by calling the keys method:

```
In [134]: store.keys()
Out[134]: ['/df1', '/df2']
```

and we can test for the existence of an object with a given key using the Python `in` keyword:

```
In [135]: 'df2' in store
Out[135]: True
```

To retrieve an object form the store we again use the dictionary-like semantic and index the object with its corresponding key:

```
In [136]: df = store["df1"]
```

From the `HDFStore` object we can also access the underlying HDF5 handle using the `root` attribute. This is actually nothing but a PyTables root group:

```
In [137]: store.root
Out[137]: / (RootGroup) ''   children := ['df1' (Group), 'df2' (Group)]
```

Once we are finished with an `HDFStore` object we should close it using the close method, to ensure that all data associated with it is written to the file.

```
In [138]: store.close()
```

Since HDF5 is a standard file format, there is of course nothing that prevents us from opening and HDF5 file created with pandas `HDFStore` or PyTables with any other HDF5 compatible software, such as for example the h5py library. If we open the file produced with `HDFStore` with h5py we can easily inspect its content and see how the `HDFStore` object arranges the data of the `DataFrame` objects that we assigned to it:

```
In [139]: f = h5py.File("store.h5")
In [140]: f.visititems(lambda x, y: print(x, "\t" * int(3 - len(str(x))//8), y))
df1                     <HDF5 group "/df1" (4 members)>
df1/axis0               <HDF5 dataset "axis0": shape (5,), type "<i8">
df1/axis1               <HDF5 dataset "axis1": shape (5,), type "<i8">
```

```
df1/block0_items         <HDF5 dataset "block0_items": shape (5,), type "<i8">
df1/block0_values        <HDF5 dataset "block0_values": shape (5, 5), type "<f8">
df2                      <HDF5 group "/df2" (8 members)>
df2/axis0                <HDF5 dataset "axis0": shape (21,), type "|S8">
df2/axis1                <HDF5 dataset "axis1": shape (30,), type "<i8">
df2/block0_items         <HDF5 dataset "block0_items": shape (3,), type "|S8">
df2/block0_values        <HDF5 dataset "block0_values": shape (30, 3), type "<f8">
df2/block1_items         <HDF5 dataset "block1_items": shape (14,), type "|S4">
df2/block1_values        <HDF5 dataset "block1_values": shape (30, 14), type "<i8">
df2/block2_items         <HDF5 dataset "block2_items": shape (4,), type "|S6">
df2/block2_values        <HDF5 dataset "block2_values": shape (1,), type "|O8">
```

We can see that the `HDFStore` object stores each `DataFrame` object in a group of its own, and that it has split each data frame into several heterogeneous HDF5 datasets (blocks) where the columns are grouped by the their data type. Furthermore, the column names and values are stored in separate HDF5 datasets.

```
In [141]: f["/df2/block0_items"].value
Out[141]: array([b'S%', b'Shift/GP', b'FO%'], dtype='|S8')
In [142]: f["/df2/block0_values"][:3]
Out[142]: array([[ 13.9,  24. ,  52.5],
                 [ 15.2,  25.2,  49. ],
                 [ 12.6,  25.1,  52.9]])
In [143]: f["/df2/block1_values"][:3, :5]
Out[143]: array([[  1,  80,  36,  68, 104],
                 [  2,  77,  31,  56,  87],
                 [  3,  82,  28,  58,  86]])
```

JSON

The JSON[5] (JavaScript Object Notation) is a human-readable, lightweight plain-text format that is suitable for storing datasets made up from lists and dictionaries. The values of such lists and dictionaries can themselves be lists or dictionaries, or must be of the following basic data types: string, integer, float and Boolean, or the value `null` (like the `None` value in Python). This data model allows storing complex and versatile datasets, without structural limitations such as the tabular form required by formats such as CSV. A JSON document can be used as a key-value store, where the values for different keys can have different structure and data types.

The JSON format was primarily designed to be used as a data interchange format for passing information between web services and JavaScript applications. In fact, JSON is a subset of JavaScript language and, as such, valid JavaScript code. However, the JSON format itself is a language-independent data format that can be readily parsed and generated from essentially every language and environment, including Python. The JSON syntax is also almost valid Python code, making it familiar and intuitive to work with from Python as well.

We have already seen an example of a JSON dataset in Chapter 10, where we looked at the graph of the Tokyo Metro network. Before we revisit that dataset, we begin with a brief overview of JSON basics and how to read and write JSON in Python. The Python standard library provides the module `json` for working with JSON formatted data. Specifically, this module contains functions for generating JSON data from a Python data structure (list or dictionary): `json.dump` and `json.dumps`, and for parsing JSON data into a Python data structure: `json.load` and `json.loads`. The functions `loads` and `dumps` take Python strings as input and output, while the `load` and `dump` operate on a file handle and read and write data to a file.

[5]For more information about JSON, see http://json.org.

For example, we can generate the JSON string of a Python list by calling the `json.dumps` function. The return value is a JSON string representation of the given Python list that closely resembles the Python code that could be used to create the list. However, a notable exception is the Python value `None`, which is represented as the value `null` in JSON:

```
In [144]: data = ["string", 1.0, 2, None]
In [145]: data_json = json.dumps(data)
In [146]: data_json
Out[146]: '["string", 1.0, 2, null]'
```

To convert the JSON string back into a Python object, we can use `json.loads`:

```
In [147]: data = json.loads(data_json)
In [148]: data
Out[148]: ['string', 1.0, 2, None]
In [149]: data[0]
Out[149]: 'string'
```

We can use exactly the same method to store Python dictionaries as JSON strings. Again, the resulting JSON string is essentially identical to the Python code for defining the dictionary:

```
In [150]: data = {"one": 1, "two": 2.0, "three": "three"}
In [151]: data_json = json.dumps(data)
In [152]: data_json
Out[152]: '{"two": 2.0, "three": "three", "one": 1}'
```

To parse the JSON string and convert it back into a Python object we again use `json.loads`:

```
In [153]: data = json.loads(data_json)
In [154]: data["two"]
Out[154]: 2.0
In [155]: data["three"]
Out[155]: 'three'
```

The combination of lists and dictionaries makes a versatile data structure. For example, we can store lists or dictionaries of lists with variable number of elements. This type of data would be difficult to store directly as a tabular array, and further level of nested list and dictionaries would make it very impractical. When generating JSON data with the `json.dump` and `json.dumps` function we can optionally give the argument `indent=True`, to obtain indented JSON code that can be easier to read:

```
In [156]: data = {"one": [1],
     ...:         "two": [1, 2],
     ...:         "three": [1, 2, 3]}
In [157]: data_json = json.dumps(data, indent=True)
In [158]: data_json
```

```
Out[158]: {
            "two": [
             1,
             2
             ],
            "three": [
             1,
             2,
             3
            ],
            "one": [
             1
            ]
          }
```

As an example of a more complex data structure, consider a dictionary containing a list, a dictionary, a list of tuples, and a text string. We could use the same method as above to generate a JSON representation of the data structure using `json.dumps`, but instead here we write the content to a file using the `json.dump` function. Compared to `json.dumps`, it additionally takes a file handle as a second argument, which we need to create beforehand:

```
In [159]: data = {"one": [1],
     ...:         "two": {"one": 1, "two": 2},
     ...:         "three": [(1,), (1, 2), (1, 2, 3)],
     ...:         "four": "a text string"}
In [160]: with open("data.json", "w") as f:
     ...:     json.dump(data, f)
```

The result is that the JSON representation of the Python data structure is written to the file `data.json`:

```
In [161]: !cat data.json
{"four": "a text string", "two": {"two": 2, "one": 1}, "three": [[1], [1, 2], [1, 2, 3]],
 "one": [1]}
```

To read and parse a JSON formatted file into a Python data structure we can use `json.load`, to which we need to a pass a handle to an open file:

```
In [162]: with open("data.json", "r") as f:
     ...:     data_from_file = json.load(f)
In [163]: data_from_file["two"]
Out[163]: [1, 2]
In [164]: data_from_file["three"]
Out[164]: [[1], [1, 2], [1, 2, 3]]
```

The data structure returned by `json.load` is not always identical to the one stored with `json.dump`. In particular, JSON is stored as Unicode, so strings in the data structure returned by `json.load` are always Unicode strings. Also, as we can see from the example above, JSON does not distinguish between tuples and lists, and the `json.load` always produce lists rather than tuples, and the order in which keys for a dictionary are displayed is not guaranteed, unless using the `sorted_keys=True` argument to the `dumps` and `dump` functions.

Now that we have seen how Python lists and dictionaries can be converted to and from JSON representation using the `json` module, it is worthwhile to revisit the Tokyo Metro dataset from Chapter 10. This is a more realistic dataset, and an example of a data structure that mixes dictionaries, lists of variable lengths, and string values. The first 20 lines of the JSON file is shown here:

```
In [165]: !head -n 20 tokyo-metro.json
{
     "C": {
         "color": "#149848",
          "transfers": [
             [
                 "C3",
                  "F15"
             ],
              [
                 "C4",
                  "Z2"
             ],
             [
                 "C4",
                  "G2"
             ],
              [
                 "C7",
                  "M14"
             ],
```

To load the JSON data into a Python data structure we use `json.load` in the same way as before:

```
In [166]: with open("tokyo-metro.json", "r") as f:
     ...:     data = json.load(f)
```

The result is a dictionary with a key for each metro line:

```
In [167]: data.keys()
Out[167]: dict_keys(['N', 'M', 'Z', 'T', 'H', 'C', 'G', 'F', 'Y'])
```

The dictionary value for each metro line is again a dictionary that contains line color, lists of transfer points, and the travel times between stations on the line:

```
In [168]: data["C"].keys()
Out[168]: dict_keys(['color', 'transfers', 'travel_times'])
In [169]: data["C"]["color"]
Out[169]: '#149848'
In [170]: data["C"]["transfers"]
Out[170]: [['C3', 'F15'],  ['C4', 'Z2'],  ['C4', 'G2'],  ['C7', 'M14'],  ['C7', 'N6'],
           ['C7', 'G6'],  ['C8', 'M15'],  ['C8', 'H6'],  ['C9', 'H7'],  ['C9', 'Y18'],
           ['C11', 'T9'],  ['C11', 'M18'],  ['C11', 'Z8'],  ['C12', 'M19'],  ['C18', 'H21']]
```

With the dataset loaded as a nested structure of Python dictionaries and lists, we can iterate over and filter items from the data structure with ease, for example using Pythons list comprehension syntax. The following example demonstrates how to select the set of connected nodes in the graph on the C line, which has a travel time of one minute:

```
In [175]: [(s, e, tt) for s, e, tt in data["C"]["travel_times"] if tt == 1]
Out[175]: [('C3', 'C4', 1), ('C7', 'C8', 1), ('C9', 'C10', 1)]
```

The hierarchy of dictionaries and the variable length of the lists stored in the dictionaries make this a good example of a dataset that does not have a strict structure, and which therefore is suitable to store in a versatile format such as JSON.

Serialization

In the previous section we used the JSON format to generate representation of in-memory Python objects, such as lists and dictionaries. This process is called serialization, which in this case resulted in a JSON plain-text representation of the objects. An advantage of the JSON format is that it is language independent, and can easily be read by other software. Its disadvantages are that JSON files are not space efficient, and they can only be used to serialize a limited type of objects (list, dictionaries, basic types, as discussed in the previous section). There are many alternative serialization techniques that address these issues. Here we briefly will look at two alternatives that address the space efficiency issue and the types of objects that can be serialized, respectively: the msgpack library and the Python `pickle` module.

We begin with msgpack, which is a binary protocol for storing JSON like data efficiently. The msgpack software is available for many languages and environments. For more information about the library and its Python bindings, the projects web page is http://msgpack.org. In analogy to the JSON module, the `msgpack` library provides two sets of functions that operate on byte lists (`msgpack.packb` and `msgpack.unpackb`) and file handles (`msgpack.pack` and `msgpack.unpack`), respectively. The `pack` and `packb` function converts a Python data structure into a binary representation, and the `unpack` and `unpackb` functions perform the reverse operation. For example, the JSON file for the Tokyo Metro dataset is relatively large and takes about 27 Kb on disk:

```
In [171]: !ls -lh tokyo-metro.json
-rw-r--r--@ 1 rob  staff    27K Apr  7 23:18 tokyo-metro.json
```

Packing the data structure with msgpack rather than JSON results in a considerably smaller file, at around 3 Kb:

```
In [172]: data_pack = msgpack.packb(data)
In [173]: type(data_pack)
Out[173]: bytes
In [174]: len(data_pack)
Out[174]: 3021
In [175]: with open("tokyo-metro.msgpack", "wb") as f:
     ...:     f.write(data_pack)
In [176]: !ls -lh tokyo-metro.msgpack
-rw-r--r--@ 1 rob  staff   3.0K Apr  8 00:40 tokyo-metro.msgpack
```

More precisely, the byte list representation of the dataset uses only 3021 bytes. In applications where storage space or bandwidth is essential, this is can be a significant improvement. However, the price we have paid for this improvement is that we must use the msgpack library to unpack the data, and it uses a binary format and therefore is not human readable. Whether this is an acceptable trade off or not will depend on the application at hand. To unpack a binary `msgpack` byte list we can use the `msgpack.unpackb` function, which recovers the original data structure:

```
In [177]: del data
In [178]: with open("tokyo-metro.msgpack", "rb") as f:
     ...:     data_msgpack = f.read()
     ...:     data = msgpack.unpackb(data_msgpack)
In [179]: list(data.keys())
Out[180]: ['T', 'M', 'Z', 'H', 'F', 'C', 'G', 'N', 'Y']
```

The other issue with JSON serialization is that only certain types of Python objects can be stored as JSON. The Python `pickle` module[6] can create a binary representation of nearly any kind of Python object, including class instances and function. Using the `pickle` module follows the exact same use pattern as the `json` module: we have `dump` and `dumps` functions for serializing an object to a byte array and a file handle, respectively, and the `load` and `loads` for deserializing a pickled object.

```
In [181]: with open("tokyo-metro.pickle", "wb") as f:
     ...:     pickle.dump(data, f)
In [182]: del data
In [183]: !ls -lh tokyo-metro.pickle
-rw-r--r--@ 1 rob  staff   8.5K Apr  8 00:40 tokyo-metro.pickle
```

The size of the pickled object is considerably smaller than the JSON serialization, but larger than the serialization produced by msgpack. We can recover a pickled object using the `pickle.load` function, which expects a file handle as an argument:

```
In [184]: with open("tokyo-metro.pickle", "rb") as f:
     ...:     data = pickle.load(f)
In [185]: data.keys()
Out[185]: dict_keys(['T', 'M', 'Z', 'H', 'F', 'C', 'G', 'N', 'Y'])
```

The main advantage with pickle is that almost any type of Python object can be serialized. However, Python pickles cannot be read by software not written in Python, and it is also not a recommended format for long-term storage, because compatibility between Python versions and with different version of libraries that define the objects that are pickled cannot always be guaranteed. If possible, using JSON for serializing list- and dictionary-based data structures is generally a better approach, and if file size is an issue the `msgpack` library provides a popular and easily accessible alternative to JSON.

[6] An alternative to the `pickle` module is the `cPickle` module, which is a more efficient reimplementation that is also available in the Python standard library. See also the `dill` library at http://trac.mystic.cacr.caltech.edu/project/pathos/wiki/dill.

Summary

In this chapter we have reviewed common data formats for reading and writing numerical data to files on disk, and we introduced a selection of Python libraries that are available for working with these formats. We first looked at the ubiquitous CSV file format, which is a simple and transparent format that is suitable for small and simple datasets. The main advantage of this format is that it is human-readable plain text, which makes it intuitively understandable. However, it lacks many features that are desirable when working with numerical data, such as metadata describing the data and support for multiple datasets. The HDF5 format naturally takes over as the go-to format for numerical data when the size and complexity of the data grows beyond what is easily handled using a CSV format. HDF5 is a binary file format, so it is not a human-readable format like CSV, but there are good tools for exploring the content in an HDF5 file, both programmatically and using command line and GUI-based user interfaces. In fact, due to the possibility of storing meta data in attributes, HDF5 is a great format for self-describing data. It is also a very efficient file format for numerical data, both in terms of I/O and storage, and it can even be used as a data model for computing with very large datasets that do not fit in the memory of the computer. Overall, HDF5 is a fantastic tool for numerical computing that anyone working with computing should benefit greatly from being familiar with. Toward the end of the chapter we also briefly reviewed JSON, msgpack, and Python pickles for serializing data into text and binary format.

Further Reading

An informal specification of the CSV file is given in RFC 4180, `http://tools.ietf.org/html/rfc4180`. It outlines many of the commonly used features of the CSV format, although not all CSV readers and writers comply with every aspect of this document. An accessible and informative introduction to the HDF5 format and the h5py library is given by the creator of h5py in a book by Collette. It is also worth reading about the NetCDF (Network Common Data Format), `http://www.unidata.ucar.edu/software/netcdf`, which is another widely used format form numerical data. The pandas library also provides I/O functions beyond what we have discussed here, such as the ability to read Excel files (`pandas.io.excel.read_excel`) and the fixed-width format (`read_fwf`). Regarding the JSON format, a concise but complete specification of the format is available at the web site `http://json.org`. With the increasingly important role of data in computing, there has been a rapid diversification of formats and data storage technologies in recent years. As a computational practitioner, reading data from databases, such as SQL and NoSQL databases is now also an important task. Python provides a common database API for standardizing database access from Python applications, as described by PEP 249 (`https://www.python.org/dev/peps/pep-0249`). Another notable project for reading databases from Python is SQLAlchemy (`http://www.sqlalchemy.org`).

References

Collette, A. (2013). *Python and HDF5*. Sebastopol: O'Reilly.