Unit 9: Input and output

Richard Foltyn

November 27, 2020

Contents

1 Input and output

In this unit we discuss input and output, or I/O for short. In doing so, we focus exclusively on I/O routines used to load and store data from files that are relevant for numerical computation and data analysis.

1.1 I/O with NumPy

We have already encountered the most basic, and probably most frequently used NumPy I/O routine, `np.loadtxt()`. We often use files that store data as text files containing character-separated values (CSV) since virtually any application supports this data format. The most important I/O functions to process text data are:

- `np.loadtxt()`: load data from a text file.
- `np.genfromtxt()`: load data from a text file and handle missing data.
- `np.savetxt()`: save a NumPy array to a text file.

There are a few other I/O functions in NumPy, for example to write arrays as raw binary data. We won't cover them here, but you can find them in the official documentation.

Imagine we have the following tabular data from FRED which we already used in the first unit, where the first two rows look as follows:

Year	GDP	CPI	UNRATE
1948	2118.5	24.0	3.8
1949	2106.6	23.8	6.0

To load this CSV file as a NumPy array, we use `loadtxt()`:

[1]:
```
import numpy as np

# relative path to CSV file
file = '../data/FRED.csv'

# load CSV
data = np.loadtxt(file, skiprows=1, delimiter=',')
data[:2]         # Display first two rows
```

[1]:
```
array([[1948. , 2118.5,   24. ,    3.8],
       [1949. , 2106.6,   23.8,    6. ]])
```

The default settings will in many cases be appropriate to load whatever CSV file we might have. However, we'll occasionally want to specify the following arguments to override the defaults:

- `delimiter`: Character used to separate individual fields (default: space)
- `skiprows=n`: Skip the first `n` rows. For example, if the CSV file contains a header with variable names, `skiprows=1` needs to be specified as NumPy by default cannot process these names.
- `dtype`: Enforce a particular data type for the resulting array.
- `encoding`: Set the character encoding of the input data. This is usually not needed, but can be required to import data with non-latin characters that are not encoded using Unicode.

While `loadtxt()` is simple to use, it quickly reaches its limits with more complex data sets. For example, when we try to load our sample of universities with `loadtxt()`, we get the following error:

[2]:
```python
import numpy as np

file = '../data/universities.csv'

# Try to load CSV data that contains strings
# This will result in an error!
data = np.loadtxt(file, delimiter=';', skiprows=1)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-1-b1ec39e83237> in <module>
      5 # Try to load CSV data that contains strings
      6 # This will result in an error!
----> 7 data = np.loadtxt(file, delimiter=';', skiprows=1)

~/.conda/envs/py3-default/lib/python3.7/site-packages/numpy/lib/npyio.py in␣
↪loadtxt(fname, dtype, comments, delimiter, converters, skiprows, usecols,␣
↪unpack, ndmin, encoding, max_rows)
   1137         # converting the data
   1138         X = None
-> 1139         for x in read_data(_loadtxt_chunksize):
   1140             if X is None:
   1141                 X = np.array(x, dtype)

~/.conda/envs/py3-default/lib/python3.7/site-packages/numpy/lib/npyio.py in␣
↪read_data(chunk_size)
   1065
   1066             # Convert each value according to its column and store
-> 1067             items = [conv(val) for (conv, val) in zip(converters, vals)]
   1068
   1069             # Then pack it according to the dtype's nesting

~/.conda/envs/py3-default/lib/python3.7/site-packages/numpy/lib/npyio.py in␣
↪<listcomp>(.0)
   1065
   1066             # Convert each value according to its column and store
-> 1067             items = [conv(val) for (conv, val) in zip(converters, vals)]
   1068
   1069             # Then pack it according to the dtype's nesting

~/.conda/envs/py3-default/lib/python3.7/site-packages/numpy/lib/npyio.py in␣
↪floatconv(x)
    761         if '0x' in x:
    762             return float.fromhex(x)
--> 763         return float(x)
    764
    765     typ = dtype.type
```

```
ValueError: could not convert string to float: '"University of Glasgow"'
```

This code fails for two reasons:

1. The file contains strings and floats, and `loadtxt()` by default cannot load mixed data.
2. There are missing values (empty fields), which `loadtxt()` cannot handle either.

We can address the first issue by creating a so-called structured array, ie. an array that contains fields with mixed data. This is accomplished by constructing a special `dtype` object that specifies the field names and their data types:

[3]:
```python
# Define names and data types for fields in CSV file
#   Data types are defined using two tokens:
#       1.  The main data type (U: unicode string, f: float, i: integer)
#       2.  The precision or field width
dtypes = np.dtype([('Institution', 'U30'),      # unicode string of length 30
                   ('Country', 'U20'),          # unicode string of length 20
                   ('Founded', 'i4'),           # integer, 4 bytes
                   ('Students', 'i4'),
                   ('Budget', 'f8'),            # float, 8 bytes
                   ('Ranking', 'i4')])

data = np.loadtxt(file, delimiter=';', skiprows=1, dtype=dtypes)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-1-e1c8dcd4434d> in <module>
     10                     ('Ranking', 'i4')])
     11
---> 12 data = np.loadtxt(file, delimiter=';', skiprows=1, dtype=dtypes)

~/.conda/envs/py3-default/lib/python3.7/site-packages/numpy/lib/npyio.py in loadtxt(fname, dtype, comments, delimiter, converters, skiprows, usecols, unpack, ndmin, encoding, max_rows)
   1137         # converting the data
   1138         X = None
-> 1139         for x in read_data(_loadtxt_chunksize):
   1140             if X is None:
   1141                 X = np.array(x, dtype)

~/.conda/envs/py3-default/lib/python3.7/site-packages/numpy/lib/npyio.py in read_data(chunk_size)
   1065
   1066             # Convert each value according to its column and store
-> 1067             items = [conv(val) for (conv, val) in zip(converters, vals)]
   1068
   1069             # Then pack it according to the dtype's nesting

~/.conda/envs/py3-default/lib/python3.7/site-packages/numpy/lib/npyio.py in <listcomp>(.0)
   1065
   1066             # Convert each value according to its column and store
-> 1067             items = [conv(val) for (conv, val) in zip(converters, vals)]
   1068
   1069             # Then pack it according to the dtype's nesting

~/.conda/envs/py3-default/lib/python3.7/site-packages/numpy/lib/npyio.py in floatconv(x)
    761         if '0x' in x:
    762             return float.fromhex(x)
--> 763         return float(x)
    764
```

```
    765     typ = dtype.type

ValueError: could not convert string to float:
```

However, this still fails because the budget for Swansea University is missing.

We can get around this by using `genfromtxt()`, which is more flexible and can also deal with missing data. Missing values are assigned the value `np.nan`:

[4]:
```
# load data using genfromtxt()
# We still need to specify the dtype defined above!
data = np.genfromtxt(file, delimiter=';', dtype=dtypes, encoding='utf8')

data[-1]          # print last observation
```

[4]:
```
('"Swansea University"', '"Wales"', 1920, 20620, nan, 251)
```

While the CSV file can now be processed without errors, you see that NumPy does not remove the double quotes around strings such as the university name.

Instead of trying to fix this, it is advisable to just use pandas to load this kind of data which handles all these problems automatically. We examine this alternative below.

Finally, to save a NumPy array to a CSV file, there is a logical counterpart to `np.loadtxt()` which is called `np.savetxt()`.

[5]:
```
import numpy as np
import os.path
import tempfile

# Generate some random data on [0,1)
data = np.random.default_rng(123).random(size=(10, 5))

# create temporary directory
d = tempfile.TemporaryDirectory()

# path to CSV file
file = os.path.join(d.name, 'data.csv')

# Print destination file - this will be different each time
print(f'Saving CSV file to {file}')

# Write NumPy array to CSV file
np.savetxt(file, data, delimiter=';', fmt='%8.5f')
```

```
Saving CSV file to /tmp/tmpujo7sdo_/data.csv
```

The above code creates a 10×5 matrix of random floats and stores these in the file `data.csv` using 5 significant digits. The destination file is located in a temporary directory which will be different every time this code is run. We use the `tempfile` module to create this writeable temporary directory.

1.2 I/O with pandas

Pandas's I/O routines are more powerful than those implemented in NumPy:

- They support reading and writing numerous file formats.
- They support heterogeneous data without having to specify the data type in advance.
- They gracefully handle missing values.

For these reasons, it is often preferrable to directly use pandas to process data instead of NumPy.

The most important routines are:

- `read_csv()`, `to_csv()`: Read or write CSV text files
- `read_fwf()`: Read data with fixed field widths, ie. text data that does not use delimiters to separate fields.
- `read_excel()`, `to_excel()`: Read or write Excel spreadsheets
- `read_stata()`, `to_stata()`: Read or write Stata's `.dta` files.

For a complete list of I/O routines, see the official documentation.

To illustrate, we repeat the above examples using pandas's `read_csv()`. Since the FRED data contains only floating-point data, the result is very similar to reading in a NumPy array.

[6]:
```
import pandas as pd

# relative path to CSV file
file = '../data/FRED.csv'

df = pd.read_csv(file, sep=',')
df.head(2)            # Display the first 2 rows of data
```

[6]:
```
   Year     GDP   CPI  UNRATE
0  1948  2118.5  24.0     3.8
1  1949  2106.6  23.8     6.0
```

The difference between NumPy and pandas become obvious when we try to load our university data: this works out of the box, without the need to specify any data types or to handle missing values:

[7]:
```
import pandas as pd

# relative path to CSV file
file = '../data/universities.csv'

df = pd.read_csv(file, sep=';')
df.tail(3)        # show last 3 rows
```

[7]:
```
                 Institution           Country  Founded  Students  Budget  \
20      University of Stirling          Scotland     1967      9548   113.3
21  Queen's University Belfast  Northern Ireland     1810     18438   369.2
22          Swansea University             Wales     1920     20620     NaN

    Rank
20   301
21   200
22   251
```

Note that missing values are correctly converted to `np.nan` and the double quotes surrounding strings are automatically removed!

Unlike NumPy, pandas can also process other popular data formats such as MS Excel files (or OpenDocument spreadsheets):

[8]:
```
import pandas as pd

# Excel file containing university data
file = '../data/../data/universities.xlsx'

df = pd.read_excel(file, sheet_name='universities')
df.head(3)
```

[8]:
```
                Institution   Country  Founded  Students  Budget  Rank
0     University of Glasgow  Scotland     1451     30805   626.5    92
1   University of Edinburgh  Scotland     1583     34275  1102.0    30
2  University of St Andrews  Scotland     1413      8984   251.2   201
```

The routine `read_excel()` takes the argument `sheet_name` to specify the sheet that should be read.

- Note that the Python package `xlrd` needs to be installed in order to read files from Excel 2003 and above.

Finally, we often encounter text files with fixed field widths, since this is a commonly used format in older applications (for example, fixed-width files are easy to create in Fortran). For example, the fixed-width variant of our FRED data looks like this:

```
Year GDP     CPI  UNRATE
 1948 2118.5   24      3.8
 1949 2106.6 23.8        6
 1950 2289.5 24.1      5.2
 1951 2473.8   26      3.3
 1952 2574.9 26.6        3
```

You see that the column `Year` occupies the first 5 characters, the `GDP` column the next 7 characters, and so on. To read such files, the width (ie. the number of characters) has to be explicitly specified:

[9]:
```python
import pandas as pd

# File name of FRED data, stored as fixed-width text
file = '../data/FRED-fixed.csv'

# field widths are passed as list to read_fwf()
df = pd.read_fwf(file, widths=[5, 7, 5, 8])
df.head(3)
```

[9]:

	Year	GDP	CPI	UNRATE
0	1948	2118.5	24.0	3.8
1	1949	2106.6	23.8	6.0
2	1950	2289.5	24.1	5.2

Here the `widths` argument accepts a list that contains the number of characters to be used for each field.

1.3 Pickling

A wholly different approach to data I/O is taken by Python's built-in `pickle` module (see official documentation). Almost any Python object can be dumped into a binary file and read back using `pickle.dump()` and `pickle.load()`.

The big advantage over other methods is that hierarchies of objects are automatically supported. For example, we can pickle a list containing a `tuple`, a string and a NumPy array:

[10]:
```python
import numpy as np
import pickle
import tempfile
import os.path

# Generate 2d array of integers
arr = np.arange(10).reshape((2, -1))
tpl = (1, 2, 3)
text = 'Pickle is very powerful!'

# data: several nested containers and strings
data = [tpl, text, arr]

# create temporary directory
d = tempfile.TemporaryDirectory()
# Binary destination file
file = os.path.join(d.name, 'data.bin')
```

```python
# print destination file path
print(f'Pickled data written to {file}')

with open(file, 'wb') as f:
    pickle.dump(data, f)
```

```
Pickled data written to /tmp/tmp7t67n1ub/data.bin
```

We can then read back the data as follows:

[11]:
```python
# load pickle data from above
with open(file, 'rb') as f:
    data = pickle.load(f)

# expand data into its components
tpl, text, arr = data
arr          # prints previously generated 2d array
```

[11]:
```
array([[0, 1, 2, 3, 4],
       [5, 6, 7, 8, 9]])
```

The above example introduces a few concepts we have not countered so far:

1. The built-in function `open()` is used to open files for reading or writing.
   - The second argument indicates whether a file should be read-only, `r`, or writeable, `w`.
   - The `b` sets the file mode to *binary*, ie. its contents are *not* human-readable text.
2. We usually access files using a so-called *context manager*. A context manager is created via the `with` statement.

   A big advantage of using a context manager is that the file resource made available as `f` in the block following `with` is automatically cleaned up as soon as the block exits. This is particularly important when writing data.

So why not always use `pickle` to load and store data?

1. Pickling is Python-specific and no other application can process pickled data.
2. The pickle protocol can change in a newer version of Python, and you might not be able to read back your old pickled objects.
3. Even worse, because projects such as NumPy and pandas implement their own pickling routines, you might not even be able to unpickle old DataFrames when you upgrade to a newer pandas version!
4. `pickle` is not secure: It is possible to construct binary data that will execute arbitrary code when unpickling, so you don't want to unpickle data from untrusted sources.
5. Some object cannot be pickled automatically. For example, this applies to any classes defined with Numba or Cython, unless special care is taken to implement the pickle protocol.

`pickle` is great for internal use when you do not need to exchange data with others and have complete control over your computing environment (ie. you can enforce a specific version of Python and the libraries you are using). For anything else, you should avoid it.