Johnny Wei-Bing Lin

A Hands-On Introduction to Using Python in the Atmospheric and Oceanic Sciences

http://www.johnny-lin.com/pyintro

2012

 Printed version: ISBN 978-1-300-07616-2. PDF versions: No ISBNs are assigned.

Who would *not* want to pay money for this book?: if you do not need a black-and-white paper copy of the book, a color PDF copy with functional hyperlinks, have limited funds, or are interested in such a small portion of the book that it makes no sense to buy the whole thing. The book's web site (http://www.johnny-lin.com/pyintro) has available, for free, PDFs of every chapter as separate files.

Who would want to pay money for this book?: if you want a black-and-white paper copy of the book, a color PDF copy with functional hyperlinks, or you want to help support the author financially. You can buy a black-and-white paper copy of the book at http://www.johnny-lin.com/pyintro/buypaper.shtml and a hyperlink-enabled color PDF copy of the book at http://www.johnny-lin.com/pyintro/buypdf.shtml.

A special appeal to instructors: Instruction at for-profit institutions, as a commercial use, is not covered under the terms of the CC BY-NC-SA, and so instructors at those institutions should not make copies of the book for students beyond copying permitted under Fair Use. Instruction at not-for-profit institutions is not a commercial use, so instructors may legally make copies of this book for the students in their classes, under the terms of the CC BY-NC-SA, so long as no profit is made through the copy and sale (or Fair Use is not exceeded). However, most instruction at not-for-profit institutions still involves payment of tuition: lots of people are getting paid for their contributions. Please consider also paying the author of this book something for his contribution.

Regardless of whether or not you paid money for your copy of the book, you are free to use any and all parts of the book under the terms of the CC BY-NC-SA.

Chapter 5

File Input and Output

The atmospheric and oceanic sciences (AOS) are "data" intensive fields, whether data refers to observations or model output. Most of the analysis we do involve datasets, and so facilities for file input/output (i/o) are critical. Fortunately, Python has very robust facilities for file i/o. In this chapter, we will sort-of touch on those facilities.

Why do I say "sort-of?": because I am in somewhat of a quandary when it comes to talking about file input/output. On the one hand, I want to show you how to use routines that you will want to use for your own research and analysis. On the other hand, because this book is an introduction to Python, I want to show you the fundamentals of how to do things in the language. In Python, the most robust file i/o packages, while not difficult to use, are still rather conceptually advanced.[1] The introduction of these packages might distract from the larger goal of showing you the fundamentals. As a compromise, this chapter will describe ways of handling file i/o that, while not the most efficient or optimized, nonetheless will work for many (if not most) AOS uses and which illustrate basic methods in using Python (this is particularly true when I discuss handling strings). In the summary in Section 5.4, I will briefly describe additional packages for file i/o that you may want to look into.

In this chapter we will look at input/output to and from text files and netCDF files. But before we do so, I want to make some comments about file objects, which are foundational to how Python interfaces with a file, whether text, netCDF, or another format.

[1] An example is the PyTables package, a really great package for large datasets that utilizes some very advanced optimization methods.

5.1 File objects

File objects represent the file to the interpreter.

A file object is a "variable" that represents the file to Python. This is a subtle but real difference with procedural languages. For instance, in Fortran, you use functions to operate on a file and unit numbers to specify the file you're operating on (e.g., `read(3,*)`, where `3` is the unit number that represents the file to Fortran). In Python, you use methods attached to file objects to operate on the file. (More on objects is in Ch. 7.)

Creating file objects using the `open` statement.

File objects are created like any other object in Python, that is, by assignment. For text files, you **instantiate** a file object with the built-in `open` statement:

```
fileobj = open('foo.txt', 'r')
```

The first argument in the `open` statement gives the filename. The second argument sets the mode for the file: `'r'` for reading-only from the file, `'w'` for writing a file, and `'a'` for appending to the file.

Creating netCDF file objects.

Python has a number of modules available for handling netCDF files; for the netCDF package we'll be using in this chapter, there is a different command to create file objects that correspond to the netCDF file, but the syntax is similar:

```
fileobj = S.NetCDFFile('air.mon.mean.nc',
                       mode='r')
```

As with `open`, the string `'r'` means read, etc. In Section 5.3, when we discuss netCDF input/output in more detail, I'll explain the rest of the syntax of the above file object creation statement. For now, I just want to point out that the file object `fileobj` is created by assignment to the return of the `S.NetCDFFile` command.

The `close` method of file objects.

One method common to both the text and netCDF file objects we'll be looking at is the `close` method, which, as you might expect, closes the file object. Thus, to close a file object `fileobj`, execute:

```
fileobj.close()
```

5.2 Text input/output

Once you've created the text file object, you can use various methods attached to the file object to interact with the file.

5.2.1 Text input/output: Reading a file

To read one line from the file, use the `readline` method:

```
aline = fileobj.readline()
```

Because the file object is connected to a text file, `aline` will be a string. Note that `aline` contains the newline character, because each line in a file is terminated by the newline character.

To read the rest of a file that you already started reading, or to read an entire file you haven't started reading, and then put the read contents into a list, use the `readlines` method:

The `readlines` method.

```
contents = fileobj.readlines()
```

Here, `contents` is a list of strings, and each element in `contents` is a line in the `fileobj` file. Each element also contains the newline character, from the end of each line in the file.

Note that the variable names `aline` and `contents` are not special; use whatever variable name you would like to hold the strings you are reading in from the text file.

5.2.2 Text input/output: Writing a file

To write a string to the file that is defined by the file object `fileobj`, use the `write` method attached to the file object:

```
fileobj.write(astr)
```

Here, `astr` is the string you want to write to the file. Note that a newline character is *not* automatically written to the file after the string is written. If you want a newline character to be added, you have to append it to the string prior to writing (e.g., `astr+'\n'`).

To write a list of strings to the file, use the `writelines` method:

The `writelines` method; `write` and `writelines` do not write newline by default.

```
fileobj.writelines(contents)
```

Here, `contents` is a list of strings, and, again, a newline character is *not* automatically written after the string (so you have to explicitly add it if you want it written to the file).

5.2.3 Text input/output: Processing file contents

Let's say you've read-in the contents of a file from the file and now have the file contents as a list of strings. How do you do things with them? In particular, how do you turn them into numbers (or arrays of numbers) that you can analyze? Python has a host of string manipulation methods, built-in to string variables (a.k.a., objects), which are ideal for dealing with contents from text files. We will mention only a few of these methods.

The string method `split`.

The `split` method of a string object takes a string and breaks it into a list using a separator. For instance:

```
a = '3.4 2.1 -2.6'
print a.split(' ')
['3.4', '2.1', '-2.6']
```

will take the string `a`, look for a blank space (which is passed in as the argument to `split`, and use that blank space as the delimiter or separator with which one can split up the string.

The string method `join`.

The `join` method takes a separator string and puts it between items of a list (or an array) of strings. For instance:

```
a = ['hello', 'there', 'everyone']
'\t'.join(a)
'hello\tthere\teveryone'
```

will take the list of strings `a` and concatenate these elements together, using the tab string (`'\t'`) to separate two elements from each other. (For a short list of some special strings, see p. 19.)

Converting strings to numerical types.

Finally, once we have the strings we desire, we can convert them to numerical types in order to make calculations. Here are two ways of doing so:

- If you loop through a list of strings, you can use the `float` and `int` functions on the string to get a number. For instance:

```
import numpy as N
anum = N.zeros(len(a), 'd')
for i in xrange(len(a)):
    anum[i] = float(a[i])
```

takes a list of strings `a` and turns it into a NumPy array of double-precision floating point numbers `anum`.[2]

- If you make the list of strings a NumPy array of strings, you can use the `astype` method for type conversion to floating point or integer. For instance:

  ```
  anum = N.array(a).astype('d')
  ```

  takes a list of strings `a`, converts it from a list to an array of strings using the `array` function, and turns that array of strings into an array of double-precision floating point numbers `anum` using the `astype` method of the array of strings.

A gotcha: Different operating systems may set the end-of-line character to something besides `'\n'`. Make sure you know what your text file uses. (For instance, MS-DOS uses `'\r\n'`, which is a carriage return followed by a line feed.) By the way, Python has a platform independent way of referring to the end-of-line character: the attribute `linesep` in the module os. If you write your program using that variable, instead of hard-coding in `'\n'`, your program will write out the specific end-of-line character for the system you're running on.

Different OSes have different end-of-line characters.

Example 41 (Writing and reading a single column file):
Take the following list of temperatures `T`:

```
T = [273.4, 265.5, 277.7, 285.5]
```

write it to a file *one-col_temp.txt*, then read the file back in.

Solution and discussion: This code will do the trick (note I use comment lines to help guide the reader):

[2]Note that you can specify the array `dtype` without actually writing the `dtype` keyword; NumPy array constructors like `zeros` will understand a typecode given as the second positional input parameter.

```
import numpy as N

outputstr = ['\n']*len(T)         #- Convert to string
for i in xrange(len(T)):          #  and add newlines
    outputstr[i] = \
        str(T[i]) + outputstr[i]

fileout = open('one-col_temp.txt', 'w')  #- Write out
fileout.writelines(outputstr)            #  to the
fileout.close()                          #  file

filein = open('one-col_temp.txt', 'r')   #- Read in
inputstr = filein.readlines()            #  from the
filein.close()                           #  file

Tnew = N.zeros(len(inputstr), 'f')   #- Convert
for i in xrange(len(inputstr)):      #  string to
    Tnew[i] = float(inputstr[i])     #  numbers
```

Note you don't have to strip off the newline character before converting the number to floating point using `float`.

A caveat about reading text files: In the beginning of this chapter, I said I would talk about file reading in a way that teaches the fundamentals of Python, not in a way that gives you the most efficient solution to file i/o for AOS applications. This is particularly true for what I've just told you about reading text files. String methods, while powerful, are probably too low-level to bother with every time you want to read a text file; you'd expect someone somewhere has already written a function that automatically processes text formats typically found in AOS data files. Indeed, this is the case: see the `asciiread` function in PyNGL,[3] the `readAscii` function in the Climate Data Analysis Tools (CDAT),[4] and the SciPy Cookbook i/o page[5] for examples.

[3] http://www.pyngl.ucar.edu/Functions/Ngl.asciiread.shtml (accessed August 16, 2012).

[4] http://www2-pcmdi.llnl.gov/cdat/tips_and_tricks/file_IO/reading_ASCII.html (accessed August 16, 2012).

[5] http://www.scipy.org/Cookbook/InputOutput (accessed August 16, 2012).

5.2.4 Exercise to read a multi-column text file

▷ **Exercise 16 (Reading in a multi-column text file):**

You will find the file *two-col_rad_sine.txt* in the *datasets* sub-directory of *course_files*. Write code to read the two columns of data in that file into two arrays, one for angle in radians (column 1) and the other for the sine of the angle (column 2). (The *course_files* directory of files is available online at the book's website. See p. viii for details on obtaining the files. Alternately, feel free to use a text data file of your own.)

The two columns of *two-col_rad_sine.txt* are separated by tabs. The file's newline character is just `'\n'` (though this isn't something you'll need to know to do this exercise). The file has no headers.

Solution and discussion: Here's my solution:

```
import numpy as N
fileobj = open('two-col_rad_sine.txt', 'r')
data_str = fileobj.readlines()
fileobj.close()

radians = N.zeros(len(data_str), 'f')
sines = N.zeros(len(data_str), 'f')
for i in xrange(len(data_str)):
    split_istr = data_str[i].split('\t')
    radians[i] = float(split_istr[0])
    sines[i] = float(split_istr[1])
```

The array `radians` holds the angles (in radians) and the array `sines` holds the sine of those angles. Note that the above code does not need to know ahead of time how many lines are in the file; all the lines will be read in by the `readlines` method call.

5.3 NetCDF input/output

NetCDF is a platform-independent binary file format that facilitates the storage and sharing of data along with its metadata. Versions of the tools needed to read and write the format are available on practically every operating system and in every major language used in the atmospheric and oceanic sciences.

The structure of netCDF files.

Before discussing how to do netCDF i/o in Python, let's briefly review the structure of netCDF. There are four general types of parameters in a netCDF file: global attributes, variables, variable attributes, and dimensions. Global attributes are usually strings that describe the file as a whole, e.g., a title, who created it, what standards it follows, etc.[6] Variables are the entities that hold data. These include both the data-proper (e.g., temperature, meridional wind, etc.), the domain the data is defined on (delineated by the dimensions), and metadata about the data (e.g., units). Variable attributes store the metadata associated with a variable. Dimensions define a domain for the data-proper, but they also have values of their own (e.g., latitude values, longitude values, etc.), and thus you usually create variables for each dimension that are the same name as the dimension.[7]

As an example of a set of variables and dimensions for a netCDF file, consider the case where you have a timeseries of surface temperature for a latitude-longitude grid. For such a dataset, you would define "lat", "lon", and "time" dimensions and corresponding variables for each of those dimensions. The variable "lat" would be 1-D with the number of elements given for the "lat" dimension and likewise for the variables "lon" and "time", respectively. Finally, you would define the variable "Ts" as 3-D, dimensioned "lat", "lon", and "time".

Several Python packages can read netCDF files, including: the Ultrascale Visualization-Climate Data Analysis Tools (UV-CDAT), CDAT, PyNIO, pysclint, PyTables, and ScientificPython. We'll be discussing ScientificPython in this section, not because it's the best package of this bunch but because it was one of the earliest Python netCDF interfaces, and many subsequent packages have emulated its user-interface.

5.3.1 NetCDF input/output: Reading a file

Importing the ScientificPython netCDF submodule.

ScientificPython is another one of those packages whose "human-readable" name is different from its "imported" name. In addition, the netCDF utilities are in a subpackage of ScientificPython. Thus, the import name for the netCDF utilities is long and you will almost always want to assign the imported package to an alias:

```
import Scientific.IO.NetCDF as S
```

[6] It is unfortunate that "global attributes" and "variable attributes" are called attributes, since the term attributes has a very specific meaning in object-oriented languages. When I talk about object attributes in close proximity to netCDF attributes, in this section, I'll try to make the object attributes occurrence glossary-linked.

[7] For more on netCDF, see http://www.unidata.ucar.edu/software/netcdf/docs (accessed August 16, 2012).

The command to create a file object, as we mentioned earlier, is very similar to the `open` command used for text files, except that the constructor is in the subpackage NetCDF and is named `NetCDFFile` (the NetCDF subpackage is itself in the IO subpackage of the Scientific package). The filename is the first argument and you specify the mode in which you wish to open the file by the `mode` keyword input parameter (set to `'r'` for read, `'w'` for write, and `'a'` for append; if you forget to write `mode=`, it will still all work fine). Thus, to open the file *file.nc* in read-only mode, type:

```
fileobj = S.NetCDFFile('file.nc', mode='r')
```

With netCDF files, the conceptualization of a file as a file object has a cognitive benefit. If we think of a file object as being the file itself (as Python sees it), we might expect that the netCDF global attributes should be actual attributes of the file object. Indeed, that is the case, and so, in the case of our above example, if the netCDF file has a global attribute named "title", the file object `fileobj` will have an attribute named `title` (referred to as `fileobj.title`, following the standard Python objects syntax) that is set to the value of the global attribute.

The `variables` attribute is a dictionary of variable objects.

NetCDF file objects have an attribute `variables` which is a dictionary. The keys are strings that are the names of the variables, and the values are variable objects (which is a kind of object specially defined for netCDF handling) that contain the variable's value(s) as well as the variable's metadata (the variable's variable attributes). NetCDF file objects also have an attribute `dimensions` which is a dictionary. The keys are strings that are the names of the dimensions, and the values are the lengths of the dimensions. Let's look at an example of reading a variable named `'Ts'` from a netCDF file:

Example 42 (Reading a variable named `'Ts'`):

Get variable object data values with the `getValue` method.

In this example, we'll read in the data associated with the name `'Ts'` (which is probably surface temperature) and one piece of metadata. Note that the "name" of the variable is a string; I'm not assuming that the "name" is the actual variable itself (i.e., a variable `Ts`). To do the first task, we will access the data in the variable and put it in a NumPy array. This code would do the trick:

```
data = fileobj.variables['Ts'].getValue()
```

The variable is found in the `variables` attribute, which is a dictionary, so we use the variable name as the key (`'Ts'`). What is returned from that dictionary is a special kind of object called a variable object. This object has a method called `getValue` which returns the values of the data in the object,

so we call that method (which takes no arguments, so we pass it an empty argument list). Finally, we use assignment to put the values into the NumPy array `data`.

Our second task is to obtain metadata about `'Ts'`, in particular the units. To do so, we'll read the variable attribute `units` (which gives the units of `'Ts'`) that is attached to the `'Ts'` variable object and save it to a scalar Python variable `unit_str`. Here's the code that would do this:

```
units_str = fileobj.variables['Ts'].units
```

Again, `variables` is an attribute of `fileobj` and is a dictionary. Thus, the `'Ts'` key applied to that dictionary will extract the variable object that contains the data and metadata of `'Ts'`. Variable attributes are attributes of the variable object, so to obtain the units you specify the `units` attribute. Remember, `fileobj.variables['Ts']` gives you a variable object. The `units` attribute is a string, which gets set to the variable `units_str`, and we're done.

Let's put all this together and look at a more complex example of reading a netCDF dataset, in this case, the NCEP/NCAR Reanalysis 1:

Example 43 (Reading a netCDF dataset):

The code below reads the monthly mean surface/near-surface air temperature from the NCEP/NCAR Reanalysis 1 netCDF dataset found in the subdirectory *datasets* of the *course_files* directory. The netCDF file is named *air.mon.mean.nc*. Without running it, what do you expect would be output? Try to explain what each line of the code below does before you read the solution:

```
1  import numpy as N
2  import Scientific.IO.NetCDF as S
3  fileobj = S.NetCDFFile('air.mon.mean.nc', mode='r')
4  print fileobj.title
5  print fileobj.dimensions
6  print fileobj.variables
7  data = fileobj.variables['air'].getValue()
8  print N.shape(data)
9  print data[0:10,30,40]
10 print fileobj.variables['air'].long_name
```

Solution and discussion: The following is output to screen by the code above (note though, because the `print` command does not, in general, word-wrap properly, I put in line breaks after each item in the dictionary and every four items in the `data` listing to make them more readable on this page):

```
Monthly mean air temperature NCEP Reanalysis
{'lat': 73, 'lon': 144, 'time': None}
{'lat': <NetCDFVariable object at 0x2194270>,
'air': <NetCDFVariable object at 0x2194738>,
'lon': <NetCDFVariable object at 0x21946a8>,
'time': <NetCDFVariable object at 0x21946f0>}
(755, 73, 144)
[ 24.64419365  28.36103058  29.27451515  28.94766617
  25.15870857  24.2053318   24.1325798   23.70580482
  23.58633614  23.20644951]
Monthly Mean Air Temperature
```

(Note, in the discussion below, the line numbers refer to the code, not the screen output.) The first line after the NumPy import statement imports the `NetCDF` subpackage of `Scientific.IO` and aliases it to `S`. The next line creates the file object representation of the netCDF file and sets the mode to read-only. The global attribute `title`, which is the title of the entire file, is printed out in line 4.

In lines 5 and 6, the `dimensions` and `variables` attributes are printed out. As those attributes are dictionaries, key:value pairs are printed out. This shows there are three dimensions (latitude, longitude, and time) and four variables (the dimensions plus the air temperature). Note that the dimension of `'time'` is set to `None` because that dimension is this netCDF file's unlimited dimension (the dimension along which one can append new latitude-longitude slices).

In line 7, the NumPy array `data` is created from the value of the variable named `'air'`, and in the next line, the shape of `data` is printed out. (The array is dimensioned [time, latitude, longitude]; remember that the rightmost dimension is the fastest varying dimension.) In line 9, a subarray of `data` is printed out, the data from the first ten time points at a single physical location. The last line prints out the long name of the variable named `'air'`.

(You can type the code in to run it. Alternately, this code can be found in the *code_files* subdirectory of *course_files*, in the file *example-netcdf.py*.)

The National Center for Atmospheric Research Receives a New Cluster

5.3.2 NetCDF input/output: Writing a file

In order to write out a netCDF file, you first have to create a file object that is set for writing, for instance:

```
fileobj = S.NetCDFFile('file.nc', mode='w')
```

Once the file object exists, you use methods of the file object to create the dimensions and variable objects that will be in the file. You have to create the dimensions before the variable objects (since the latter depends on the former), and you have to create the variable objects first before you fill them with values and metadata.

The `createDimension` method creates a dimension. This method both creates the name of the dimension and sets the value (length) of the dimension. The `createVariable` method creates a variable object; note that it only creates the infrastructure for the variable (e.g., the array shape) and does not fill the values of the variable, set variable attributes, etc.

Filling netCDF array and scalar variables.

To fill array variables, use the slicing syntax (i.e., the colon) with the variable object in an assignment operation. (This will make more sense once we see it in the example below.) The values of scalar variables are assigned to the variable object through the `assignValue` method of the *variable object* (not of the file object). Finally, variable attributes are set using Python's regular object assignment syntax, as applied to the variable object.

To illustrate the writing process, let's walk through an example (the example's code can be found in *course_files/code_files* in the file *example-netcdf.py*):

Example 44 (Writing a netCDF file):

What does the following code do?:

```
1  fileobj = S.NetCDFFile('new.nc', mode='w')
2  lat = N.arange(10, dtype='f')
3  lon = N.arange(20, dtype='f')
4  data1 = N.reshape(N.sin(N.arange(200, dtype='f')*0.1),
5                                    (10,20))
6  data2 = 42.0
7  fileobj.createDimension('lat', len(lat))
8  fileobj.createDimension('lon', len(lon))
9  lat_var = fileobj.createVariable('lat', 'f', ('lat',))
10 lon_var = fileobj.createVariable('lon', 'f', ('lon',))
11 data1_var = fileobj.createVariable('data1', 'f',
12                                      ('lat','lon'))
13 data2_var = fileobj.createVariable('data2', 'f', ())
14 lat_var[:] = lat[:]
15 lon_var[:] = lon[:]
16 data1_var[:,:] = data1[:,:]
17 data1_var.units = 'kg'
18 data2_var.assignValue(data2)
19 fileobj.title = "New netCDF file"
20 fileobj.close()
```

Solution and discussion: The first line creates the file object connected to the netCDF file we'll be writing to. The lines 2–6, we create the data variables we'll be writing: two vectors, one 2-D array, and one scalar. After that, in line 7, we create the latitude and longitude dimensions (named `'lat'` and `'lon'`, respectively) based on the lengths of the two vectors.

Lines 9–13 create variable objects using the `createVariable` method of the file object. Note how `lat_var` will be the variable in the file named `'lat'` and is a 1-D variable dimensioned by the dimension named `'lat'`. That is to say, the `'lat'` in the first argument of `createVariable` refers to the variable's name while `'lat'` in the third argument of `createVariable` (which is part of a 1-element tuple) refers to the dimension created two code lines above. Variable `lon_var` is structured in a similar way. Finally, note how because `data2_var` is a scalar, the dimensioning tuple is empty.

Lines 14–16 fill the three non-scalar variables. These arrays are filled using slicing colons to select both the source values and their destination elements. In the case of line 14, as a specific example, such use of the slicing colon is interpreted as meaning "put the values of the array `lat` into the values of the variable object `lat_var`."

Line 17 attaches a units attribute to the variable object `data1_var`, and line 18 assigns a scalar value to `data2_var`. Line 19 assigns the global attribute `title`, and the final line closes the file attached to the file object.

5.3.3 Exercise to read and write a netCDF file

▷ **Exercise 17 (Read and write a netCDF reanalysis dataset):**

Open the netCDF NCEP/NCAR Reanalysis 1 netCDF dataset of monthly mean surface/near-surface air temperature and read in the values of the `time` variable. (The example data is in the *datasets* subdirectory of *course_files* in the file *air.mon.mean.nc*.)

Alter the time values so that the first time value is 0.0 (i.e., subtract out the minimum of the values). Change the units string to just say `'hours'`, i.e., eliminate the datum. (The original units string from the netCDF file gave a datum.)

Write out the new time data and units as a variable in a new netCDF file.

Solution and discussion: The solution is found in *course_files/code_files* in the file *exercise-netcdf.py* and is reproduced below (with some line continuations added to fit it on the page):

```
1  import numpy as N
2  import Scientific.IO.NetCDF as S
3
4  fileobj = S.NetCDFFile('air.mon.mean.nc', mode='r')
5  time_data = fileobj.variables['time'].getValue()
6  time_units = fileobj.variables['time'].units
7  fileobj.close()
8
9  time_data = time_data - N.min(time_data)
10 time_units = 'hours'
11
12 fileobj = S.NetCDFFile('newtime.nc', mode='w')
13 fileobj.createDimension('time', N.size(time_data))
14 time_var = fileobj.createVariable('time',
15                                    'd', ('time',))
16 time_var.units = time_units
17 time_var[:] = time_data[:]
18 fileobj.title = \
19     "New netCDF file for the time dimension"
20 fileobj.close()
```

Note again how array syntax makes the calculation to eliminate the time datum (line 9) a one-liner ☺.

5.4 Summary

In this chapter we saw that Python conceptualizes files as objects, with attributes and methods attached to them (as opposed to merely unit number addresses). To manipulate and access those files, you use the file object's methods. For the contents of text files, we found string methods to be useful, and for netCDF files, there are a variety of methods that give you the numerical data.

While many of the methods we discussed in this chapter can work for much daily work, you probably will find any one of a number of Python packages to be easier to use when it comes to doing file input/output. These include: UV-CDAT, PyNIO, pysclint, PyTables, etc. Some of these packages include text input/output functions that do line splitting and conversion for you. Some of these packages can also handle other formats such as HDF, GRIB, etc. For a list of more file input/output resources, please see Ch. 10.