Chapter 19
Reading and Writing Text Files

Abstract GIS data often comes in formats that can't be imported into ArcGIS or read with `arcpy` cursors without modification. Automating these modifications can be a significant time saver if you have numerous files that require the same changes. The `arcpy` cursors discussed in Chapter 17 are specialized for reading and writing attribute tables that conform to a prescribed format, such as dBASE files, shapefiles, and GRID rasters. This chapter discusses Python file handling functions that can read and write text files, regardless of format. The chapter begins with Python file handling syntax and then uses GIS examples to demonstrate common tabular text file modifications.

Chapter Objectives
After reading this chapter, you'll be able to do the following:

- Write and read text files.
- Use the Python `with` structure for reading and writing.
- Separate the parts in a line of a tabular data file.
- Modify a tabular data file.
- Differentiate between the Python working directory and the `arcpy` workspace.
- Handle `IOError` exceptions.
- Avoid file locking.
- Write a file that preserves Python data structures.

19.1 Working with `file` Objects

The first step to reading or writing a file is to call the built-in `open` function which returns a `file` object. The `'file'` data type has a number of methods for reading and writing a file. The following line of code creates a `file` object named `'f'` that allows you to read a file:

```
>>> f = open('C:/gispy/data/ch19/poem.txt', 'r')
```

The `open` function requires one argument, the name of the file. The `file` object can be created for reading, writing, or both. The second argument is an optional

L. Tateosian, *Python For ArcGIS*, DOI 10.1007/978-3-319-18398-5_19

Table 19.1 Approaches to reading a text file.

`with open(<filename>, 'r') as f:`	
f open in `'r'` mode	**What it does**
`f.read()`	Read the entire contents of the file and returns a string
`f.read(8)`	Read the next 8 bytes of the file and return a string
`f.readline()`	Read the next line and return a string
`f.readlines()`	Read every line and return a list of strings (one string per line)
`for line in f:`	Iterates through each remaining line in the file

argument for setting the mode. The mode describes how the file will be used: `'r'` for reading text, `'w'` for writing text, `'r+'` for both reading and writing ASCII text files. Other modes can be used to handle binary (non-text) file formats, but the examples in this chapter focus on reading and writing text files (`'r'` and `'w'`), as these are the most common usage scenarios. Depending on the mode, the `file` object returned by the `open` function has either read or write methods. In the code above, 'poem.txt' was opened in read mode. There are several methods for reading all or part of a file (See Table 19.1). The `read` method reads the entire file and returns it as a string:

```
>>> f = open('C:/gispy/data/ch19/poem.txt', 'r')
>>> f.read()
'\nScripterwocky\n`Twas brillig, and the Python lists\nDid join
and pop-le in the loop:\nAll-splitsy were the string literals,\nAnd
the boolean values were true.\t \n'
```

Run the code above, then browse to 'poem.txt' and open it to view the contents in a text editor, such as 'Wordpad'. The file contains five lines of text, a Pythonic verse inspired by Lewis Carroll's poem, 'Jabberwocky'. The string printed by Python contains a carriage return escape sequence (`'\n'`) at the end of each line of text.

Add another line to the poem and try to save the file. If you have run the file opening code, you will receive a message that the file is in use by another application and it cannot be saved. Like `arcpy` cursors, Python `file` objects lock the file. To release the file, you must use the `close` method. There should be one `close` statement for each `open` statement to prevent locking issues. The `close` method does not require any arguments, but parentheses must be used, since it is a method. Use the following statement to close the `file` object and release the lock:

```
>>> f.close()
```

Similar to the process of reading a book (open->read->close), when reading or writing a file, the code should open the file, read or write, and then close the file. If a book is left open, it can be splattered with soup. If a file is left open, it will be locked and can cause errors or data corruption.

19.1.1 The WITH Statement

The WITH statement is an alternative to closing the file explicitly. The usage of WITH statements for file objects is similar to the usage described in Chapter 17 for arcpy cursors. Place the open function call between the with and as keywords; Place the file object variable name after the as keyword; Complete the statement with a colon. Indent the code that deals with the file object. The file object is only usable inside this indented code block. Dedented code signals the end of the block. The following two lines of code,

```
>>> with open('C:/gispy/data/ch19/poem.txt', 'r') as f:
...     print f.read()
```

are equivalent to the following three lines of code,

```
>>> f = open('C:/gispy/data/ch19/poem.txt', 'r')
>>> f.read()
>>> f.close()
```

except that the with block ensures that locks are released even if an exception is thrown while attempting to read the file. When a with statement is used, the file close method does not need to be called. Upon exiting the with block, the file is automatically closed. Attempting to call the read method outside of this block returns a value error, since the file is automatically closed when the code exits the with code block:

```
>>> with open('C:/gispy/data/ch19/poem.txt', 'r') as f:
...     print f.read()
...
>>> f.read()
Traceback (most recent call last):
File "<interactive input>", line 1, in <module>
ValueError: I/O operation on closed file
```

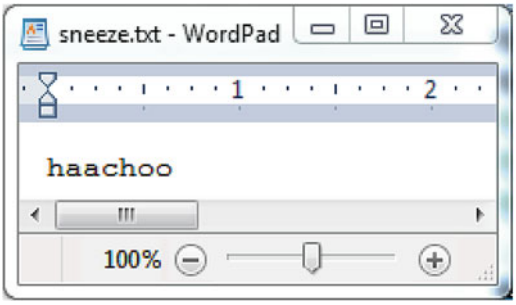
The open->read->close approach is handy for Interactive Window examples since the read feedback is immediate and the explicit close statements make the scope of the file object obvious. Otherwise, the with statement is a good technique. This chapter uses both approaches.

19.1.2 Writing Text Files

There are other methods for reading files which we'll return to momentarily. Writing to file is less nuanced, so we'll address that first. To write a text file, call the open function in 'w' mode. Then the file object has a write method which requires a string argument, the text to be written to the file:

```
>>> f = open('C:/gispy/data/ch19/sneeze.txt', 'w')
>>> f.write('haa')
>>> f.write('choo')
>>> f.close()
```

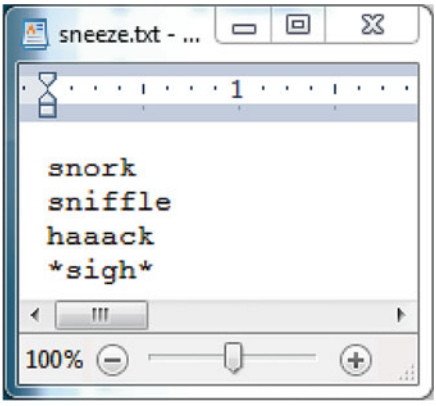
Separate calls to the `write` method do not automatically place the strings on separate lines. Though two separate write statements are used, the output is all on the same line in the file:



Carriage returns (`'\n'`) must be placed in the strings where the new lines are desired. Notice the new line can be placed anywhere in a string (e.g., the middle as well as the end):

```
>>> f = open('C:/gispy/data/ch19/sneeze.txt', 'w')
>>> f.write('snork\nsniffle\n')
>>> f.write('haaack\n')
>>> f.write('*sigh*')
>>> f.close()
```

The output from the following code contains four lines:

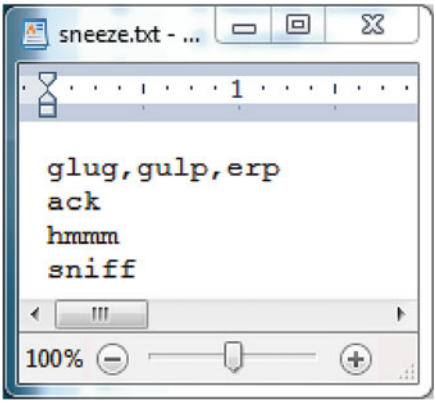


Notice that the original text in 'sneeze.txt.' has been overwritten. When a file is opened for writing, any existing text is erased, unless an append mode (`'a'`) is specified. The following code creates an empty file:

```
>>> f = open('C:/gispy/data/ch19/sneeze.txt', 'w')
>>> f.close()
```

The string `join` function can be used to write a list of strings as comma separated values on the same line or to write a list of strings on separate lines.

```
>>> f = open('C:/gispy/data/ch19/sneeze.txt', 'w')
>>> csv = ','.join(['glug', 'gulp', 'erp'])
>>> f.write(csv)
>>> f.write('\n')
>>> lines = '\n'.join(['ack', 'hmmm', 'sniff'])
>>> f.write(lines)
>>> f.close()
```



The write method only takes string arguments. Attempting to write a number results in a `TypeError`:

```
>>> f = open('C:/gispy/data/ch19/sneeze.txt', 'w')
>>> f.write(5000)
Traceback (most recent call last):
File "<interactive input>", line 1, in <module>
TypeError: expected a character buffer object
```

Casting or string formatting can be used to write numeric values to a file:

```
>>> f.write(str(5000))
>>> lament = 'I sneezed {0} times today.'.format(5000)
>>> f.write(lament)
>>> f.close()
```

19.1.3 Safe File Reading

When a file is opened in write mode, if the file already exists, the contents will be overwritten. If the file doesn't exist yet, it will be created. For a file to be opened in read mode, however, it must exist. If you attempt to open a nonexistent or corrupted file in read mode, an `IOError` is thrown:

```
>>> f = open('C:/gispy/data/ch19/bogus.txt', 'r')
Traceback (most recent call last):
File "<interactive input>", line 1, in ?
IOError: [Errno 2] No such file or directory: 'bogus.txt'
```

When the file might not exist, you can use code like the following to catch an `IOError` exception and handle the possibility of failure gracefully:

```
# safeFileRead.py
import os, sys

infile = sys.argv[1]

try:
    f = open(infile, 'r'
    print f.read()
    f.close()
except IOError:
    print "{0} doesn't exist or can't be opened.".format(infile)
```

19.1.4 The `os` Working Directory vs. the `arcpy` Workspace

When you're writing geoprocessing scripts and setting the `arcpy` workspace, it's easy to make the mistake of assuming this will work for Python `file` objects as well. The built-in `open` function is not part of the `arcpy` package, so the full path file name may need to be used even if the `arcpy` workspace is set. For example, the

following code sets the arcpy workspace and then attempts to open 'poem.txt' which resides in the workspace:

```
>>> import arcpy
>>> arcpy.env.workspace = 'C:/gispy/data/ch19'
>>> f = open('poem.txt', 'r')
Traceback (most recent call last):
File "<interactive input>", line 1, in <module>
IOError: [Errno 2] No such file or directory: 'poem.txt'
```

An IOError exception is thrown even though the os.path method isfile proves that this file does exist in that workspace:

```
>>> import os
>>> os.path.isfile('C:/gispy/data/ch19/poem.txt')
True
```

In fact, Python has its own native workspace which is stored by the os module. If the full path file name is not specified, Python searches for the file in the os module working directory. The getcwd method returns the current working directory (the CWD). The CWD depends on a number of factors, including the IDE. If PythonWin is opened by browsing to a script, right-clicking, and selecting 'Edit with PythonWin', the CWD is the path to the script. If, instead, PythonWin is opened first (e.g., by clicking on a shortcut to PythonWin), the CWD is the path to the PythonWin executable, as in the following example:

```
>>> os.getcwd()
'C:\\Python27\\ArcGIS10.3\\Lib\\site-packages\\pythonwin'
```

The chdir method changes the current working directory. The following code changes the os current working directory and then the file is opened for reading:

```
>>> os.chdir('C:/gispy/data/ch19')
>>> os.getcwd()
'C:\\gispy\\data'
>>> f = open('poem.txt', 'r')
>>> f.readline()
'Scripterwocky\n'
>>> f.close()
```

The arcpy workspace and the os current working directory are mutually independent. Changing the os module working directory does not modify the arcpy workspace.

19.1.5 Reading Text Files

The `read` method returns the entire contents of the file as a string with lines separated only by carriage return characters. For many applications, you may want to read each line, one at a time, using the `readline` method or a `FOR`-loop. The `readline` method reads the next line in the file. When the file is first opened, the "next" line is the first line of the file:

```
>>> f = open('C:/gispy/data/ch19/poem.txt', 'r')
>>> f.readline()
'Scripterwocky\n'
```

The `file` object keeps track of where it is in the file. You can think of the `file` object as maintaining an internal position cursor. When `readline` is called again, it reads the next line:

```
>>> f.readline()
'Twas brillig, and the Python lists\n'
```

The `readline` method returns each line with a trailing carriage return sequence. This sequence is interpreted and printed as a blank line when a `print` function is used.

```
>>> f.readline()
Did join and pop-le in the loop:

>>>
>>> f.readline()
'All-splitsy were the string literals,\n'
>>>
```

The `readline` method returns an empty string when the end of the file is reached:

```
>>> f.readline()
'And the boolean values were true.\t \n'
>>> f.readline()
''
>>> f.close()
```

The `file` object can also be used as an iterator in a `FOR`-loop. This approach is frequently used.

```
>>> for line in f:
...     print line
...
```

```
Scripterwocky

`Twas brillig, and the Python lists

Did join and pop-le in the loop:

All-splitsy were the string literals,

And the boolean values were true.
```

The loop iterates over each line until the end of the file is reached.

```
>>> f.readline()
''
>>> f.close()
```

Looping over a `file` object can be used together with the `readline` method. The iteration in the `FOR`-loop will start wherever the `readline` stopped. The following code reads the first line of the file, the poem title, with the `readline` method, then reads the rest of the file by looping over the `file` object.

```
>>> f = open('C:/gispy/data/ch19/poem.txt', 'r')
>>> f.readline()
'Scripterwocky\n'
>>> for line in f:
...     print line
...
Twas brillig, and the Python lists

Did join and pop-le in the loop:

All-splitsy were the string literals,

And the boolean values were true.

>>> f.close()
```

Calling `readline` followed by a `FOR`-loop is often used to handle header lines separately from table records. This approach is used again in upcoming examples on handling file content. This section used `read`, `readline`, and `FOR`-loops to read a file. Though other approaches exist, these are sufficient for the most common applications. Table 19.1 summarizes these methods and variations.

19.2 Parsing Line Contents

Once you read a line of a text file with a script, you need to *parse* the contents, in other words, break them down into usable parts. Each line is returned as a single string with a trailing carriage return. Depending on the text file contents, parsing the text may involve stripping trailing whitespace, splitting the string into a list of items,

casting the string to a numeric value and so forth. Several common parsing operations are demonstrated here.

Suppose, for example, you want to find the sum of a line in 'report.txt'. Though the file contains tab delimited numbers, the `readline` function always returns a single string. In the following example, readline returns a string containing numbers and escape sequences:

```
>>> f = open('C:/gispy/data/ch19/report.txt', 'r')
>>> f.readline()
'1\t2.07\t5.21\t4.05\t3.64\t2.03\t3.74\n'\'
```

To use the contents, you need to store them in a variable. The code above read line one, but did not store it. For example's sake, we'll move on to line two. The following code stores and prints line two:

```
>>> line = f.readline()
>>> line
'2\t3.51\t7.29\t4.2\t4.44\t3.67\t4.46\n'
>>> print line
2    4.51    7.29    4.2    4.44    3.67    4.46
```

The tabs appear as whitespace when the print statement is used because it interprets them, but the `line` variable is a string data type containing all the numbers and tabs within one string. Now that the line contents are stored in a variable, you can use the string `split` method to separate the numbers. By default, the `split` method separates the string contents at the white spaces, so no argument is needed. The following code splits the values into a list:

```
>>> lineList = line.split()
>>> lineList
['2', '3.51', '7.29', '4.2', '4.44', '3.67', '4.46']
```

The `lineList` variable is not quite ready for numeric calculations, since the list elements are strings. You can use a list comprehension to convert each one to a number:

```
>>> nums = [float(i) for i in lineList]
>>> nums
[2.0, 3.51, 7.29, 4.2, 4.44, 3.67, 4.46]
```

Suppose the first column in 'report.txt' is an ID number, which you don't want to include in the sum. The following code uses slicing to exclude the first column and then recomputes the sum:

```
>>> data = nums[1:]
>>> data
```

```
[3.51, 7.29, 4.2, 4.44, 3.67, 4.46]
>>> sum(data)
27.57
```

If other operations need to be performed on individual elements of a line, list comprehension or FOR-loops can be used. Suppose, for example, the values in the report need to be capped at 5. Example 19.1 uses a nested FOR-loop to cap the data values of 'report.txt'. This script also finds the sum and count of data elements for each line. The third line is missing an element, as the output shows:

```
ID: 1.0 Sum: 20.53 Count 6
Data: [2.07, 5, 4.05, 3.64, 2.03, 3.74]
ID: 2.0 Sum: 25.28 Count 6
Data: [3.51, 5, 4.2, 4.44, 3.67, 4.46]
ID: 3.0 Sum: 19.72 Count 5
Data: [3.9, 4.24, 4.05, 4.04, 3.49]
ID: 4.0 Sum: 22.64 Count 6
Data: [3.18, 3.5, 4.73, 4.39, 3.28, 3.56]
```

Example 19.1

```
# parseTable.py
# Purpose: Parse numeric values in a tabular text file.
cap = 5
infile = 'C:/gispy/data/ch19/report.txt'
try:
    with open(infile, 'r') as f:
    for line in f:
        # String to list of strings.
        lineList = line.split()
        # String items to float items.
        nums = [float(i) for i in lineList]
        # First col is ID, rest are data values.
        ID = nums[0]
        data = nums[1:]
        # Cap the data values at 5.
        for index, val in enumerate(data):
            if val > cap:
                data[index] = cap
        # Count and sum the values and report the results.
        count = len(data)
        total = sum(data)
        print 'ID: {0} Sum: {1} Count {2}'.format(ID, total, count)
        print 'Data: {0}'.format(data)
except IOError:
    print "{0} doesn't exist or can't be opened.".format(infile)
```

The basics steps outlined above are used frequently for reading tabular data: Store the line in a variable, split the contents, cast each element to a number, and slice the columns.

The split method is needed whenever you're reading tabular data. The argument for the split method varies depending on the delimiter (comma, space, and so forth). There is a specialized Python module `csv` for handling comma separated values, but these can also be handled using the generic file handling described here. Example 19.2 reads 'cfactors.txt', which contains a set of numerical erosion factors and their labels, delimited by an equals sign (=). The factors and labels are placed in a dictionary:

```
{1: 'stable', 2: 'low deposition', 3: 'moderate deposition', 4:
'high deposition', 5: 'severe deposition'}
```

The script uses both a `readline` call and a `FOR`-loop. The first line of the file, a header, is not needed. The code reads past the first line, (not saving it), then reads the rest into the dictionary. The erosion factor numbers are cast to integer and the labels are right-stripped of whitespace using the following statements:

```
factor = int(line[0])
label = line[1].rstrip()
```

If these statements were replaced by simpler statements:

```
factor = line[0]
label = line[1]
```

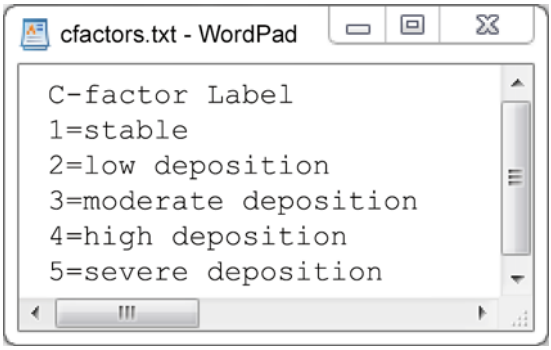
the resulting dictionary would have string keys and the values would have trailing carriage returns:

```
{'1': 'stable\n', '3': 'moderate deposition\n', '2': 'low deposition\n',
'5': 'severe deposition\n', '4': 'high deposition\n'}
```

Example 19.2

```
# cfactor.py
# Purpose: Read a text file contents into a dictionary.
factorDict = {}
infile = 'C:/gispy/data/ch19/cfactors.txt'
try:
    with open(infile, 'r') as f:
        f.readline()
        for line in f:
            line = line.split('=')
            factor = int(line[0])
```

```
            label = line[1].rstrip()
            factorDict[factor] = label
    print factorDict
except IOError:
    print "{0} doesn't exist.".format(infile)
```



19.2.1 Parsing Field Names

Example 19.2 hard-codes the index of the fields (0 and 1) in the split line containing those fields. If the field name is known, but the position of the column is not, you can use the list `index` method, which returns the index of the first occurrence of a value in the list:

```
>>> mylist = ['a','b','c','d']
>>> mylist.index('c')
2
```

The 'fieldIndex.py' script in Example 19.3 calls a `getIndex` function that strips the trailing whitespace from the field names string, splits the string into a list, and then finds the index of the field based on its name 'Label'. Since 'Label' is the second field in the 'cfactors.txt' data file, the script finds that the field named 'Label' has index 1.

Example 19.3

```
# fieldIndex.py
# Purpose: Find the index of a field name in a text
#          file with space separated fields in the first row.
infile = 'C:/gispy/data/ch19/cfactors.txt'
fieldName = 'Label'
```

```
def getIndex(delimString, delimiter, name):
    '''Get position of item in a delimited string'''
    delimString = delimString.strip()
    lineList = delimString.split(delimiter)
    index = lineList.index(name)
    return index

with open(infile, 'r') as f:
    line = f.readline()
    ind = getIndex(line, ' ', fieldName)
    print '{0} has index {1}'.format(fieldName, ind)
```

19.3 Modifying Text Files

For many applications you will need to modify files. Reading and writing to the same file using `'r+'` mode is not usually the best solution for this, because managing existing content and modifications simultaneously can be complicated. A simpler approach is to read the original file, make modifications to the content within the script, then write the modified content to another file. The script can then replace the original file with the new one, if desired. This flow would be as follows:

open->read->parse->modify->write->close->replace

As an example, suppose you want to use the c-factor table from 'cfactor.txt' in ArcGIS. Because of the spacing in the file, ArcGIS parses the field values incorrectly:

C_factor	Label
1=stable	
2=low	deposition
3=moderate	deposition
4=high	deposition
5=severe	deposition

To correct this, 'cfactorModify.py' in Example 19.4 places a comma between the field names and replaces the equals signs with commas. When there is a space on the first line of a text file, ArcGIS interprets the space separated items on this line as field names and then it interprets spaces on additional lines as the delimiter.

In Example 19.4, the script names the output file based on the input file name, opens the input file for reading and the output file for writing. Each line is read and

modified and then written to the output file. In this case, the first line is handled differently than the rest, so a `readline` call is used for the first line and a `FOR`-loop is used for the rest. The input and output files are then closed implicitly when the code exits the WITH statement. The output table displays correctly in ArcGIS:

C_factor	Label
1	stable
2	low deposition
3	moderate deposition
4	high deposition
5	severe deposition

The examples and exercises in this book forgo the step of replacing the original file with the new one so that the input and output can be easily compared. You can add this functionality using the built-in Python shell utilities module named `shutil`. The `move` method takes two file names as arguments, a source and a destination. It moves the first file to the location of the second file. Moving the modified version to the location of the original version overwrites the original with the modified file. Add the following code to the end of Example 19.3 and 'cfactorsv2.txt' will be moved to 'cfactors.txt', replacing the original and deleting version 2 at the same time:

```
import shutil
shutil.move(outfile, infile)
```

Example 19.4

```
# cfactorModify.py
# Purpose: Demonstrate reading and writing files.

import os

infile = 'C:/gispy/data/ch19/cfactors.txt'
baseN = os.path.basename(infile)
outfile = 'C:/gispy/scratch/' + os.path.splitext(baseN)[0] + 'v2.txt'
try:
    #OPEN the input and output files.
    with open(infile, 'r') as fin:
        with open(outfile, 'w') as fout:
            #READ/MODIFY/WRITE the first line.
            line = fin.readline()
            line = line.replace(' ',',')
            fout.write(line)
            #FOR the remaining lines.
            for line in fin:
```

```
                #MODIFY the line.
                line = line.replace('=', ',')
                #WRITE to output.
                fout.write(line)
            print '{0} created.'.format(outfile)
 except IOError:
     print "{0} doesn't exist.".format(infile)
```

19.3.1 Pseudocode for Modifying Text Files

The flow for modifying text files can usually be described by the following pseudocode:

```
SET the input and output file name
IF the input file exists:
  OPEN the input and output files
  FOR EACH line in the input file
    MODIFY the line.
    WRITE the modified line to the output file.
  END FOR
  CLOSE the input and output files (automatic if WITH block is used).
  REPLACE original
ENDIF
```

Placing this pseudocode in a script and then filling in the details can be a good place to start any file modification task.

19.3.2 Working with Tabular Text

Some common modification tasks, such as removing lines or columns, take a slightly different approach. To demonstrate these functions, we'll use the raw output from a device that tracks eye movement and records the points on a map where the eye fixates. The raw eye tracking data is not designed for use in ArcGIS, so it needs to be modified for import into ArcGIS and visualized as a shapefile. Some entries need to be deleted. The `arcpy` update cursor function has a `deleteRow` method for this purpose, but when the data is not ArcGIS compatible this method can't be used. The Python native `file` objects do not have a delete function. However, deletion can be achieved by omission. The usual flow is to read a line of the input file, modify the line, and then write the line to the output file. For lines that are to be deleted, you can read the line from the input, but not write it to the output. The

internal position cursor needs to read each line in the input file to step through it methodically, but not all the lines that are read need to be written to the output!

ArcGIS expects the first line of a table to be a set of field names; However, tables are often preceded by metadata, so removing header lines is a common task. The first two lines of the 'eyeTrack.csv' file contain metadata about the conditions under which eye movements were tracked. These lines must be deleted so that the `MakeXYEventLayer` management tool can be used to bring the points into ArcGIS. To strip these lines from the modified file, 'removeHeader.py', in Example 19.5 simply reads these lines, but does not write them to output.

Example 19.5

```
# removeHeader.py
# Purpose: Remove header rows.

import os
headers = 2
infile = 'C:/gispy/data/ch19/eyeTrack.csv'
baseN = os.path.basename(infile)
outfile = 'C:/gispy/scratch/' + os.path.splitext(baseN)[0] \
                + 'v2.txt'
try:
    with open(infile, 'r') as fin:
        with open(outfile, 'w') as fout:
            #READ header lines, but don't write them.
            for i in range(headers):
                fin.readline()
            #READ and WRITE the remaining lines.
            for line in fin:
                fout.write(line)
            print '{0} created.'.format(outfile)
except IOError:
    print "{0} doesn't exist.".format(infile)
```

Deleting header lines boils down to reading but not writing the first few lines to the output file. Sometimes other lines need to be deleted. For example, we also need to remove lines in 'eyeTrack.csv' where the fixation point-of-gaze `x` or `y` values (`FPOGX` and `FPOGY` fields) are not positive; These represent invalid readings. The pseudocode for removing delimited records based on a condition is as follows:

```
FOR each line in the input file:
    SPLIT the line (string to list)
    IF condition is TRUE THEN
        WRITE the string line to the output file.
    ENDIF
ENDFOR
```

In short, you check the condition and only write the line when the condition is true. Example 19.6 shows a code snippet from sample script 'removeRecords.py'. This code only writes table lines to the output file that have values greater than zero for the fields `FPOGX` and `FPOGY`. This has the effect of deleting the lines with non-positive values for either of these fields. This script calls the `getIndex` function defined in Example 19.3 and then when it encounters each line, it splits the line, and gets the float values of the columns of interest using the field indices. It then checks the required condition (positive for both columns) and writes the line only if the condition is met.

Example 19.6

```
# Excerpt from sample script 'removeRecords.py'
# REMOVE selected lines (only write acceptable lines)
# READ and WRITE field names
fieldNameLine = fin.readline()
fout.write(fieldNameLine)
# FIND field indices
sep = ','
findex1 = getIndex(fieldNameLine, sep, 'FPOGX')
findex2 = getIndex(fieldNameLine, sep, 'FPOGY')

# FOR the remaining lines:
for line in fin:
    lineList = line.split(sep)
    v1 = float(lineList[findex1])
    v2 = float(lineList[findex2])
    # IF condition is TRUE, write this record.
    if v1 > 0 and v2 > 0:
        fout.write(line)
```

Several columns in the eye tracking data contain non-numeric values. These columns need to be removed entirely. Since the `file` object reads each line individually and the tabular line contents can be split into lists, dealing with columns involves list operations. To remove columns, you need to remove the corresponding item in each line's list. The `getIndex` function defined in Example 19.3 can be used again to determine the index of a column. Then the string `pop` method can be used to remove that index:

```
>>> fields = ['FireId', 'Org', 'State', 'FireType', 'Protection']
>>> index = 2
>>> fields.pop(index)
'State'
>>> fields
['FireId', 'Org', 'FireType', 'Protection']
```

The pop removes the item with the specified index. It returns the value of the item it removed, but we're not using that value in this application, so we can just discard it. Instead, we are using the resulting list which has one less item. Care must be taken to remove the intended items. When pop is used, the indices of the rest of the items coming after that one in the list are decremented by one. Trying to pop indices 2 and 4 in that order results in an index error:

```
>>> fields = ['FireId', 'Org', 'State', 'FireType', 'Protection']
>>> indexA = 2
>>> indexB = 4
>>> fields.pop(indexA)
'State'
>>> fields.pop(indexB)
Traceback (most recent call last):
File "<interactive input>", line 1, in <module>
IndexError: pop index out of range
```

To avoid this problem, pop the indices in descending order:

```
>>> fields = ['FireId', 'Org', 'State', 'FireType', 'Protection']
>>> fields.pop(indexB)
'Protection'
>>> fields
['FireId', 'Org', 'State', 'FireType']
>>> fields.pop(indexA)
'State'
>>> fields
['FireId', 'Org', 'FireType']
```

Eight columns in the 'eyeTrack.csv' file need to be removed. Example 19.7 lists these fields in the removeFields variable. This code snippet from sample script 'removeColumns.py' reads the line containing the field names, determines the indices of these unwanted columns, then calls a removeItems function. This function sorts the indices in reverse order and then pops the unwanted values off a list that is derived from a delimited string. The removeItems function is called once for the field names line and then it is called again for each record in the table. The function splits the line string into a list, removes the unwanted items in the list and then rejoins the modified list into a delimited string. This shortened string is returned to the caller and the script writes the modified line to the file.

Example 19.7

```
# Excerpt from sample script 'removeColumns.py'
# REMOVE named columns (only write valid columns)

removeFields = ['LPCX','LPCY', 'RPCX', 'RPCY',
                'LGX', 'LGY', 'RGX', 'RGY']
```

```
def removeItems(indexList, delimiter, delimString):
    '''Remove items at given indices in a delimited string'''
    lineList = delimString.split(delimiter)
    indexList.sort(reverse = True)
    for i in indexList:
        lineList.pop(i)
    lineString = delimiter.join(lineList)
    return lineString

 ###code to OPEN files and READ past header omitted###
        # READ field names.
        fieldNamesLine = fin.readline()
        # DETERMINE field indices.
        rfIndex = []
        for field in removeFields:
            index = getIndex(fieldNamesLine, sep, field)
            rfIndex.append(index)
        # REMOVE items with these indices and WRITE the line.
        line = removeItems(rfIndex, sep, fieldNamesLine)
        fout.write(line)
        # READ/MODIFY/WRITE the remaining lines.
        for line in fin:
            line = removeItems(rfIndex, sep, line)
            fout.write(line)
```

You can combine functionality such as removing lines and removing columns of a tabular text file within a single script by working from top to bottom in a text file, one line at a time, parsing and modifying data along the way. The 'eyeTrackSHP.py' sample script in the exercises deletes rows and columns and performs a transformation from an eye tracking reference system to a geographic reference system. In Figure 19.1, the eye fixation point shapefile is overlaid on the world map that the viewer was viewing as the eye movements were recorded.

19.4 Pickling

The standard `file` object `write` function requires a string argument. Suppose your scripts generate a number of dictionaries and lists that you want to save in a file for further analysis. This means converting other data types before writing them to file. For example, to write a number in a file, you need to first cast it to a string; To write a list to a file, you need to represent it as a string by joining the elements. If instead, you want to preserve the original data types within a file, you can use a built-in Python module named `pickle`.

Pickling allows you to write and read any Python data type. The file itself contains information that encodes the data type. When opened in Wordpad, the encoding

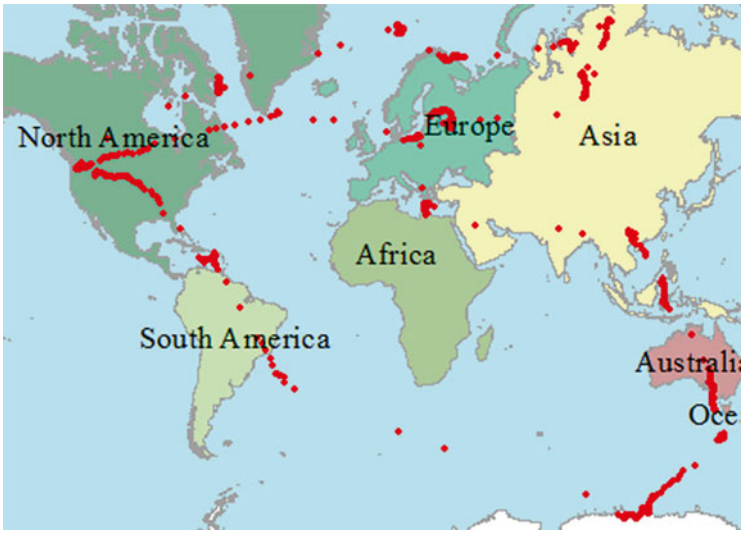


Figure 19.1 Eye fixation points drawn on the stimulus map.

doesn't look the same as the Python syntax for these types, but Python can read them using the `pickle` module.

The pickle methods for writing and reading are named `dump` and `load`. The `dump` method can be used when a file is opened for writing. It takes two arguments, the object to be written, which can be any data type, and a `file` object created in write mode. The following code dumps a float and then a list into the 'gherkin.txt' file:

```
>>> import pickle
>>> f = open('C:/gispy/data/ch19/gherkin.txt', 'w')
>>> pickle.dump(2.71828,f)
>>> pickle.dump(['FireId', 'Org', 'FireType'],f)
>>> f.close()
```

The resulting 'gherkin.txt' file is shown in Figure 19.2. The following code opens the pickle file in read mode and then uses the `load` method to read the first item:

```
>>> f2 = open('C:/gispy/data/ch19/gherkin.txt', 'r')
>>> thing1 = pickle.load(f2)
>>> thing1
2.71828
```

As expected, `thing1` has a `'float'` data type:

```
>>> type(thing1)
<type 'float'>
```

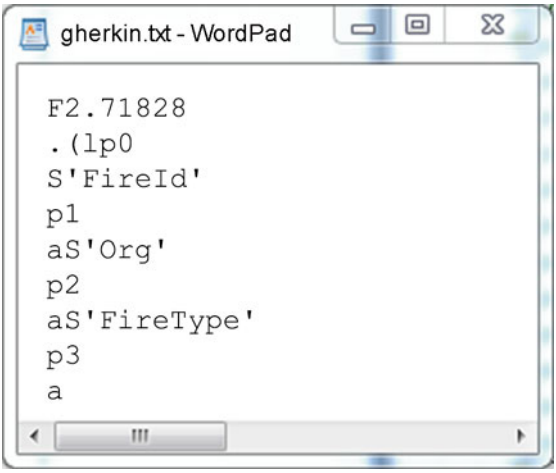


Figure 19.2 A file written with the `pickle` module.

and the second item loaded from the file into `thing2` has a `'list'` data type.

```
>>> thing2 = pickle.load(f2)
>>> thing2
['FireId', 'Org', 'FireType']
>>> type(thing2)
<type 'list'>
>>> f2.close()
```

Files written with pickling are only meant to be read with pickling. Standard `file` object reading will not preserve the data types. The following code reads the first line of the pickled file and instead of a float, it returns a string that starts with `'F'`:

```
f3 = open("C:/gispy/data/ch19/gherkin.txt", "r")
f3.readline()
'F2.71828\n'
f3.close()
```

19.5 Discussion

This chapter discussed `file` object methods and techniques for parsing and modifying files. `file` objects have additional methods, such as `seek` and `writelines`, but most file handling can be accomplished using only the `read`, `write`, `readline`, and `FOR`-loop methods discussed here. Using the WITH statement to open

files provides a convenient protection against locking. If a WITH statement is not used to open the file, the `close` method must be called before the file is released. If an exception occurs before the `close` method is called, the file will remain locked and you may need to type a call to the `close` method in the Interactive Window. The `close` method can be used along with the WITH structure, but it is not necessary.

The built-in `file` object `read` functions always read file content as strings and file content must be written as strings. This means that most of the parsing and modification work in file handling involves string operations—casting when numbers are involved and converting to lists when the lines are part of a table. Pickling is a useful alternative for writing files that are only meant to be read by Python; Whereas, generic Python `file` objects provide techniques for manipulating the data into a format that can be imported into GIS software.

19.6 Key Terms

The `'file'` data type
The built-in `open` function
The file `read`, `readline`, `write`, and `close` methods
The WITH statement for opening files.
The **`for`** `line` **`in`** `f` reading approach
The `IOError` exception
Current working directory vs. `arcpy` workspace
Parse
The open/read/parse/modify/write/close/replace workflow
The `shutil` module
The list `pop` method
The `pickle` module

19.7 Exercises

1. **writeBio.py** Create a Python script which writes a file in 'C:/gispy/scratch' called 'mybio.txt' containing your name on the first line, your favorite GIS tool on the next line, your home town on the third line, and a short autobiography on the last line.
2. **fileIO.py** Write a script to use common file handling operations and methods to perform the following tasks in the order given:
   (a) Open a file named 'C:/gispy/scratch/test.txt' for writing.
   (b) Write the number 365 on the first line of the file.
   (c) Write tab separated week days on the next line of the file.
   (d) Write the numbers 1, 2, and 3 separated by commas on the third line.
   (e) Close the file.
   (f) Open the same file for reading.
   (g) Read the first line and print the results.

(h) Read the second line, split the line on the tabs and print the results one day per line.
(i) Read the third line, split the line on the commas, sum the numbers, and print the results.
(j) Close the file.

The printed output should look like this:

```
>>> 365

Monday
Tuesday
Wednesday
Thursday
Friday

The sum is 6.
The output text file should look line this:
365
Monday  Tuesday  Wednesday  Thursday  Friday  1,2,3
```

3. **countLines.py** Write a script that counts the number of lines of an input text file and prints the number of lines. Take the full path of the input file name as an argument. Make sure to catch the `IOError` if the input file does not exist.

   Example input1: C:/gispy/data/ch19/RDUforest.txt

   Example output1:
   `>>> C:/gispy/data/ch19//RDUforest.txt has 1271 lines.`

   Example input2: C:/gispy/data/ch19/noSuchFile.txt

   Example output2:
   `>>> C:/gispy/data/ch19/noSuchFile.txt doesn't exist or can't be opened.`

4. **eyeTrackSHP.py** Sample script 'eyeTrackSHP.py' is missing ten lines of code which need to be filled in. Practice file reading and writing methods by replacing the `###` comments as instructed in the script. When complete, the script reads an eye tracking data file, creates a modified version of the text file, and creates a point shapefile of eye fixations. The fixations indicate where the viewer was looking on the screen. The script takes two arguments, the full path file name of an eye tracking data file and an output directory. The shapefile points are shown in Figure 19.1 for the following script input:

   Example input: C:/gispy/data/ch19/eyeTrack.csv C:/gispy/scratch/

   Example output:
   `>>> C:/gispy/scratch/eyeTrackv2.csv complete.`
   `Shapefile complete. View C:/gispy/scratch/eyeTrack.shp in ArcCatalog.`

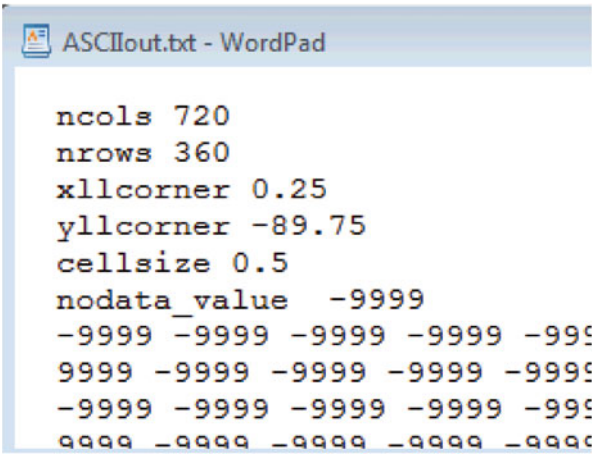
5. **ipcc2ESRI.py** Sample script 'ipcc2ESRI.py' is missing eight lines of code which need to be filled in. Practice file reading and writing methods by replacing the `###` comments as instructed in the script. When the code is complete, the script will modify a weather data file downloaded from the International Panel on Climate Change (IPCC) Web site. These IPCC files contain monthly weather conditions averaged over 10 or 30 year periods. However, the raw format downloaded from the site can not be directly imported into ArcGIS. Once you've filled in the missing lines of code, this script will read the raw IPCC data and convert it to ESRI ASCII format, which will then be imported into an ESRI GRID raster with the `arcpy` ASCIIToRaster (Conversion) tool. The script takes two arguments, a full path input file name and an output directory.

   Example input: C:/gispy/data/ch19/precipitation6190.dat C:/gispy/scratch/

   Example output:

```
>>> Converting IPCC format file
C:/gispy/data/ch19/precipitation6190.dat to ESRI ascii format.
Processing header:
Processing lines...
ASCII file created. View ASCIIout.txt in Windows Explorer.
Converting ASCII to Raster
Conversion complete. View precGRID in ArcCatalog.
```

   The beginning of the output ASCII file appears as follows:



The output raster appears as follows:

6. **loblolly.py** Write a script that reads the tab separated sample data table, 'C:/gispy/data/ch19/RDUforest.txt', and writes a modified version to 'C:/gispy/scratch/loblollyPine.txt'. The modified version should contain the field names (the first line in 'RDUforest.txt') and the 696 records (one per line) with the species `'LP'` (loblolly pine). Use the `getIndex` function to find the index of the `SPECIES` column. No arguments are required (hard-code the input full path file name, the output full path file name, the field name, the delimiter, and the selected species). Print feedback to the user as the following:

   Example input: (No arguments needed.)

   Example printed output:

```
>>> Reading file C:/gispy/data/ch19/RDUforest.txt...
696 records with SPECIES 'LP' written to
C:/gispy/scratch/loblollyPines.txt
```

7. **wrDirInfo.py** Write a Python script which lists all the files in an input directory and writes the names, sizes, and `arcpy Describe` object `dataType` of the files in an output text file. Write one file per line in the output file and separate the name, size, and type by a semicolon. Then add code to the script that reads the text file you just created and prints each line to the Interactive Window. The script should take two arguments, the directory with files to list and the full path output text file name.

   Example input: C:/gispy/data/ch19/smallDir C:/gispy/scratch/dirContents.txt

   Example printed output:

```
>>> a.dbf;2540;ShapeFile
a.kml;4684;File
a.prj;145;File
```

   . . . (and so forth)

8. **avgNumbers.py** Write a script that reads a file with lines of tab separated numeric values, finds the average of each line, and prints these averages to an output file named 'out.txt'. Use two script arguments, the full path filename of the input file and an output directory. Use try and except to handle `IOError` exceptions as shown in the first example.

SiteNo.	6/2/2000	6/24/2000	7/14/2000	6/13/2001	6/28/2001	6/2/2002	6/27/2002	7/2/2002
1	4.07	4.21	4.15	[illegible]	4.03	3.74	4.56	4.5
2	4.51	4.29	4.4	-4.69	3.77	4.46	4.76	3.76
3	3.9	[illegible]	4.05	4.04	3.49	3.91	4.52	4.52
4	3.63	[illegible]	4.92	4.64	3.75	4.1	4.4	4.17
5	3.15	3.33	4.08	4.73	3.61	3.66	4.39	4.84

Figure 19.3 The first six lines of wheatYield.txt.

Example input1: C:/gispy/data/ch19/cop_yield.txt C:/gispy/scratch

Example output1:

```
>>> Warning: C:/gispy/data/ch19/cop_yield.txt could not be opened.
```

Example input2: C:/gispy/data/ch19/cop_yield.txt C:/gispy/scratch

Example output2:

```
>>> C:/gispy/scratch/out.txt created.
```

The first three lines of the 'out.txt' will appear as follows:
4.13777777778
6.27111111111
3.82333333333
. . . (and so forth)

9. **removeLines.py** Write a script to read the 'wheatYield.txt' sample dataset, remove invalid records, and write the results to an output text file. The output file should be named 'wheatYield_edited.txt' and created as described here. Figure 19.3 shows the first six rows of the dataset which contains wheat yield samples collected at various sites between June 2000 and July 2002. The first line of the file contains the dates. The values in each record are separated by spaces. Some records contain errors: all values should be positive and each record should contain eight sample wheat yields, but some entries are missing or negative. None of the site numbers are missing, so you can check for missing values by checking the length of the list of items in each record against the number of fields. Any sites with invalid data entries should be removed. For example, the errors shown in Figure 19.3 mean that sites 2 and 4 should be removed, as in Figure 19.4. Use two script arguments, the full path filename of the input file and an output directory.

   Example input: C:/gispy/data/ch19/wheatYield.txt C:/gispy/scratch/

   Example output:

```
>>> Number of records with errors is: 10
Corrected file is: C:/gispy/scratch/wheatYield_edited.txt
```

10. **remove2000.py** Write a script that removes the 'wheatYields.txt' samples that were taken in the year 2000. Use the `removeItems` function found in Example 19.7 to remove columns 2–4. Write the results to 'wheatYieldv2.txt'.

SiteNo.	6/2/2000	6/24/2000	7/14/2000	6/13/2001	6/28/2001	6/2/2002	6/27/2002	7/2/2002
1	4.07	4.21	4.15	4.64	4.03	3.74	4.56	4.5
3	3.9	4.64	4.05	4.04	3.49	3.91	4.52	4.52
5	3.15	3.55	4.08	4.73	3.61	3.66	4.39	4.84
7	3.42	3.35	4.07	4.66	3.72	3.84	4.44	3.4
8	3.97	3.61	4.67	4.49	3.75	4.11	4.64	2.99

Figure 19.4 The first six lines of wheatYield_edited.txt.

Use two script arguments, the full path filename of the input file and an output directory.

Example input: C:/gispy/data/ch19/wheatYield.txt C:/gispy/scratch/

Example output:

```
>>> Number of columns removed: 3
Corrected file is: C:/gispy/scratch/wheatYieldv2.txt
```

The first three lines of the modified file will appear as follows:

SiteNo. 6/13/2001 6/28/2001 6/2/2002 6/27/2002 7/2/2002
1 4.64 4.03 3.74 4.56 4.5
2 -4.69 3.77 4.46 4.76 3.76
. . . (and so forth)

11. **avgDBH.py** Write a script that reads a text file and collects information into a dictionary. The 'RDUforest.txt' sample data file has a table with four fields: Block, Plot, Species, and DBH. The last field, DBH (Diameter Breast Height) measures the outside bark diameter of a tree at breast height, an indicator of maturity. This script will determine the average DBH for each species in the RDU Forest by reading the file and populating a dictionary. Each dictionary item will consist of a species as the key and a list of DBH measurements for that species as a value. A portion of the dictionary appears as follows:

```
{ 'LOB': [24.0, 18.0, 25.0, 17.0, ...
'BE': [17.0],
'WD': [17.0, 15.0, 15.0, 11.0, 16.0, ...
...}
```

As the last step, the script should loop through the items in the completed dictionary and use the Python built-in `sum` and `len` functions to calculate the average DBH for each species. The script should take a full path file name as input and print the average DBH as follows:

Example input: C:/gispy/data/ch19/RDUforest.txt

Example output:

```
>>> Average DBH for SPECIES BE = 17.0
Average DBH for SPECIES WD = 12.2
Average DBH for SPECIES WO = 12.329787234
Average DBH for SPECIES HK = 11.4210526316
```

. . . (and so forth)