Python #13
Reading and writing files

Location: https://people.sc.fsu.edu/~jburkardt/classes/python_2022/python13/python13.pdf

Freely adapted from the Python lessons at https://software-carpentry.org/

Files

- *Computer files are stored in a variety of formats;*
- *The filename extension usually indicates the file format;*
- *Text files (extension* `.txt`*) are the simplest format;*
- *Writing or reading a single array to a file is easy using* `np.loadtxt()` *or* `np.savetxt()`*;*
- *Python provides functions to write data to files, or read it back;*
- *Complications: data to be written must be a string, or converted to a string.*
- *An output line of multiple data items will require multiple write statements.*
- *An output line of multiple data will require spaces between items, and a final new line.*

1 Text files

The simplest kind of computer file is known as a text file. The default extension for such a file is `.txt`. However, there are many other file extensions which are usually text files. For instance, when you save a Python program as a `.py` file, you are saving a text file. Generally, you can think of a text file as anything that you could have patiently typed in from your keyboard, using letters, numbers, punctuation, common symbols, carriage returns ("new lines") and tab characters.

Because we think of it as similar to a document written on a typewriter, we can regard a text file as consisting of a number of lines, each terminated by a new line character.

2 Writing a text file containing a single array

Sometimes your have a simple need to save one array as a text file. In that case, the `numpy` function `savetxt()` will do the job with a single command line.

The Hilbert matrix is defined by $H_{i,j} = \frac{1}{i+j+1}$. Suppose we've created a 5×5 version of this matrix:

```
H = np.zeros ( [ 5, 5 ] )
for i in range ( 0, 5 ):
  for j in range ( 0, 5 ):
    H[i,j] = 1 / ( i + j + 1 )
```

Now we want to save it to the file *hilbert_matrix.txt*:

```
np.savetxt ( 'hilbert_matrix.txt', H )
```

3 Reading a text file containing a single array

Suppose we have a text file containing some kind of $m \times n$ table of values, which are all of the same type. Then we can use a single command line to call `np.loadtxt()`, which will return an array of those values. As an example, the Hilbert matrix data that we just wrote can be handled in this way.

```
  filename = 'hilbert_matrix.txt'
  data = np.loadtxt ( filename )
  m, n = data.shape
  print ( 'The table in "' + filename + '" has ' + str ( m ) + ' rows and ' + str ( n ) + '
      columns.' )
  print ( 'The maximum entries in each column:' )
  print ( np.max ( data, axis = 0 ) )\\
```

4 Write a text file line by line

If the things we want to write to a file aren't packed up into a single data object, then we have to look at other options. A simple program to create a text file from a general set of data would then use some kind of print statement for each line to be written. We would also need to prepare the file before using it, and afterwards issue a command to indicate that the file has been completed.

For a start, let us consider using a Python program to create a text file containing the following information:

```
Yesterday, upon the stair,
I met a man who wasn't there
He wasn't there again today
I wish, I wish he'd go away.
```

We begin by choosing a name for the file, say *mystery.txt*. Now we issue a command to Python which prepares a blank file, with the correct name, and provides us with a file pointer we can use in order to reference the file:

```
output = open ( 'mystery.txt', 'w' )
```

Here `output` is the name we have chosen for the file pointer. The second argument, `'w'`, to `open()` indicates that we will be writing information into this file. Therefore, whether a file of that name already exists or not, an empty file will be created, ready for our data.

Now we can use the `.write()` method to write lines of text into our file. Note that, unlike when we use the `print()` statement, we need to explicitly request new lines using the
`\n` character:

```
output.write ( 'Yesterday, upon the stair,\n' )
output.write ( 'I met a man who wasn\'t there.\n' )
output.write ( 'He wasn\'t there again today\n' )
output.write ( 'I wish, I wish he\'d go away.\n' )
```

and once we have completed the transfer of information, we need to close the file:

```
output.close()
```

You should be able to see a new file on your computer, in the directory where your program is running. If you can type or print or edit the file, you should see that our little poem has been saved.

5 Read a text file line by line

A simple program to extract information from such a file would "read" it one line at a time. Once we have extracted a line of text, we have to decide what to do with it, before proceeding to the next line. Here, we will simply want to print each line.

When we open the file, we add the second argument `'r'`, which indicates that this file already exists, and we plan to read it. If there is no file of this name, then the call to `open()` will fail, causing an error.

While the commands to do this are simple, each line that we read will end with a "new line" character. The `print()` function would print this line and automatically add its own new line character as well. To avoid double spacing our output, we can use the `end=''` argument.

```
input = open ( filename, 'r' )
for line in input:
  print ( line, end = '' )
input.close ( )
```

6 Adding more lines to a text file

We discover four more lines of the poem.

```
When I came home last night at three
The man was waiting there for me
But when I looked around the hall
I couldn't see him there at all!
```

We can add them to the existing file, without disturbing what we already have written, by using the "append" argument in our call to the `open()` function. This automatically opens the file and moves to the last line, ready to add new lines.

```
output = open ( 'mystery.txt', 'a' )
output.write ( '\n' )
output.write ( 'When I came home last night at three\n' )
output.write ( 'The man was waiting there for me\n' )
output.write ( 'But when I looked around the hall\n' )
output.write ( 'I couldn\'t see him there at all!\n' )
output.close()
```

7 Using `.split()` to see the words in a line

If our text file contains information that we want to process, we may need to see the individual words or items on each line. The Python method `.split()` will take a string and return a list of its words, that is, the pieces of the string that are separated by blanks.

For instance, we can get the words in the first line of our poem:

```
  s = 'Yesterday, upon the stair'
  w = s.split()
  print ( w )
  ['Yesterday,', 'upon', 'the', 'stair']
```

This means that we can figure out how many words there are in this line, and we can print out word number 2, for instance.

A typical case might involve a list of names, heights and weights to be used for our BMI application. A short version of such a file might be called *patients.txt*, and look like this:

```
Alice 60 120.3
Bob 65 200.5
Carol 70 160.1
```

A program to read this data, compute the BMI and print everything might be:

```
filename = 'patient.txt'
bio = open ( filename, 'r' )
for line in bio:
  w = line.split()
  name = w[0]
  height = int ( w[1] )
  weight = float ( w[2] )
  bmi = ( weight / 2.204 ) * ( 39.370 / height )**2
  print ( name, height, weight, bmi )
bio.close ( )
```

8 Texting while deriving

We can make a table of the divisors of the integers from 1 to 20. We don't start out with this information; instead, we derive it as we go along. The lines of this table will vary in length. Moreover, the items to write are integers, not a string. On a given line for the divisors of the number n, we have to write each divisor with a separate `.write()` statement. Since `.write()` can only handle strings, we have to convert each divisor into a string using the `int()` function. Once we have completed the line, we must terminate it with a new line character.

```
  filename = 'divisors.txt'
  output = open ( filename, 'w' )
  for n in range ( 1, 21 ):
    for i in range ( 1, n + 1 ):
      if ( ( n % i ) == 0 ):
        output.write ( ' ' + str ( i ) )
    output.write ( '\n' )
  output.close ( )
```

9 Reading variable length data

Reading our divisor data is a little easier. We want to save the information we read. It can't be saved as an array, because the rows are of different lengths. Instead, we set up an empty list. We read a line at a time, and use `.split()` to break it into a list of strings. We convert these strings to integers. Then we add the new list of divisors as another "row" of our data.

```
  divisors = []
  filename = 'divisors.txt'
```

```
  output = open ( filename, 'r' )
  for line in output:
    w = line.split()
    for i in range ( 0, len ( w ) ):
      w[i] = int ( w[i] )
    divisors.append ( w )
  output.close ( )
```

10 Writing several items of different length and type

Some files are more irregular in their structure than the examples we have seen. In that case, it may be necessary to be more careful in how we process the lines. In linear algebra, it is useful to create linear systems $A * x = b$, in which all three objects are known. Such cases can be used to test an algorithm for speed or accuracy. It makes sense to store all three items in one file. The Sparse Linear Algebra Package (SLAP) uses example matrices in which most of the matrix entries are zero. Thus, rather than storing $A_{i,j}$ for every index, it will store a data line $i, j, A_{i,j}$ only for nonzero entries.

Some associated variables are:

- **nz**: The number of nonzero entries in A;
- **sym**: 1 if the matrix A is symmetric;
- **rhs_given**: 1 if a right hand side is included in the file;
- **x_given**: 1 if a solution is included in the file.

The special SLAP file format is then:

- **first line**: n nz sym rhs_given x_given
- **nz lines**: i, j, Aij
- **n lines**, if `rhs_given==1`: b[i]
- **n lines**, if `x_given==1`: x[i]

Consider the linear algebra system $A * x = b$ of the form

$$\begin{pmatrix} 11 & 12 & 0 & 0 & 15 \\ 21 & 22 & 0 & 0 & 0 \\ 0 & 0 & 33 & 0 & 35 \\ 0 & 0 & 0 & 44 & 0 \\ 51 & 0 & 53 & 0 & 55 \end{pmatrix} * \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{pmatrix} = \begin{pmatrix} 110 \\ 65 \\ 274 \\ 176 \\ 485 \end{pmatrix}$$

We can write this information to a file, using the SLAP format:

```
n = 5
nz = 11
sym = 0
rhs_given = 1
x_given = 1
output = open ( 'slap_matrix.txt', 'w' )
output.write ( str ( n ) + ' ' )
output.write ( str ( nz ) + ' ' )
output.write ( str ( sym ) + ' ' )
output.write ( str ( rhs_given ) + ' ' )
output.write ( str ( x_given ) + '\n' )
for i in range ( 0, n ):
  for j in range ( 0, n ):
    if ( A[i,j] != 0.0 ):
```

```
      output.write ( str ( i ) + ' ' + str ( j ) + ' ' + str ( A[i,j] ) + '\n' )
for i in range ( 0, n ):
  output.write ( str ( b[i] ) + '\n' )
for i in range ( 0, n ):
  output.write ( str ( x[i] ) + '\n' )
output.close()
```

11 Reading several items of different length and type

Now let's try to read the file we just created. Although it's easy to read a line, the meaning and length of the line changes as we process different parts of the file. That means that, as we read, we need to keep in mind what kind of information we are expecting. A logical outline of how to read the file would consider four stages:

```
read_size:
  read one line of 5 values: N  NZ SYM  RHS_GIVEN  X_GIVEN
  then allocate A
read_matrix:
  read NZ lines, each of 3 values, I, J, AIJ.
  then allocate B
read_rhs:
  read N lines, the I-th line is B(I)
  then allocate X
read_x:
  read N lines, the I-th line is X(I)
```

We know how many lines to read in each stage, and so we know when we are about to move from one stage to the next.

```
filename = 'slap_matrix.txt'
input = open ( filename, 'r' )
stage = 'read_size'
for line in input:
  w = line.split()
  if ( stage == 'read_size' )
    n = int ( w[0] )
    nz = int ( w[1] )
    sym = int ( w[2] )
    rhs_given = int ( w[3] )
    x_given = int ( w[4] )
    stage = 'read_matrix'
    iz = 0
    A = np.zeros ( [ n, n ] )
  elif ( stage == 'read_matrix' ):
    i = int(w[0]) - 1
    j = int(w[1]) - 1
    aij = float ( w[2] )
    A[i,j] = aij
    iz = iz + 1
    if ( nz <= iz ):
      if ( rhs_given == 0 ):
        break
      else:
        stage = 'read_rhs'
        b = np.zeros ( n )
        i = 0
  elif ( stage == 'read_rhs' ):
    b[i] = float ( w[0] )
```

```
    i = i + 1
    if ( n <= i ):
      if ( x_given == 0 ):
        break
      else:
        stage = 'read_x'
        x = np.zeros ( n )
        i = 0
  elif ( stage == 'read_x' ):
    x[i] = float ( w[0] )
    i = i + 1
    if ( n <= i ):
      break

input.close ()
```

Admittedly, this program is longer and more complicated than our previous examples. However, it is typical that we have to deal with files whose internal structure is complicated. In such a case, we may have to know what stage of the process we are in, so that we can properly read and interpret the next line of data.

Once the program has run, you can check that $A * x = b$ by computing `Ax=np.dot ( A, x )` and comparing this to the vector `b`.