SEE1002
Introduction to Computing for Energy and Environment

Part 4: Python for Science and Engineering
Section 1: Reading and writing files

Course Outline

Part 1: Introduction to computing

Part 2: Elements of Python programming

Section 1: Fundamentals

Section 2: Branching or Decision Making

Section 3: Loops

Section 4: Functions

Part 3: Basic Python programming

Section 1: Modules

Section 2: Structure of a Python program

Section 3: Good programming practices

Part 4: Python for science and engineering

Section 1: File input and output

Section 2: Vectors, matrices and arrays

Section 3: Other topics

Outline

1. Preliminaries
2. Basic operations
3. More advanced operations

1. Preliminaries

Motivation

- We have covered how to write Python programs.
- But for most real-world applications we need to be able to write to and read from files.

Recall that there are two basic types of files:

1. Text file (readable by humans)
2. Binary (readable only by computers)

Text files are convenient but they take up more disk space. Thus some applications will save their output in raw binary.

Example 1: mylib.py resisted

```python
def perimeterareaRectangle(L,W):
    return (L*W,2*(L+W))

def perimeterRectangle(L,W):
    return (2*L + 2*W)

def areaRectangle(L,W):
    area=L*W
    return (area)
```

mylib.py

mylib.pyc

For efficiency, Python creates a binary version of library files. This binary version is used by Python; it's not meant for humans.

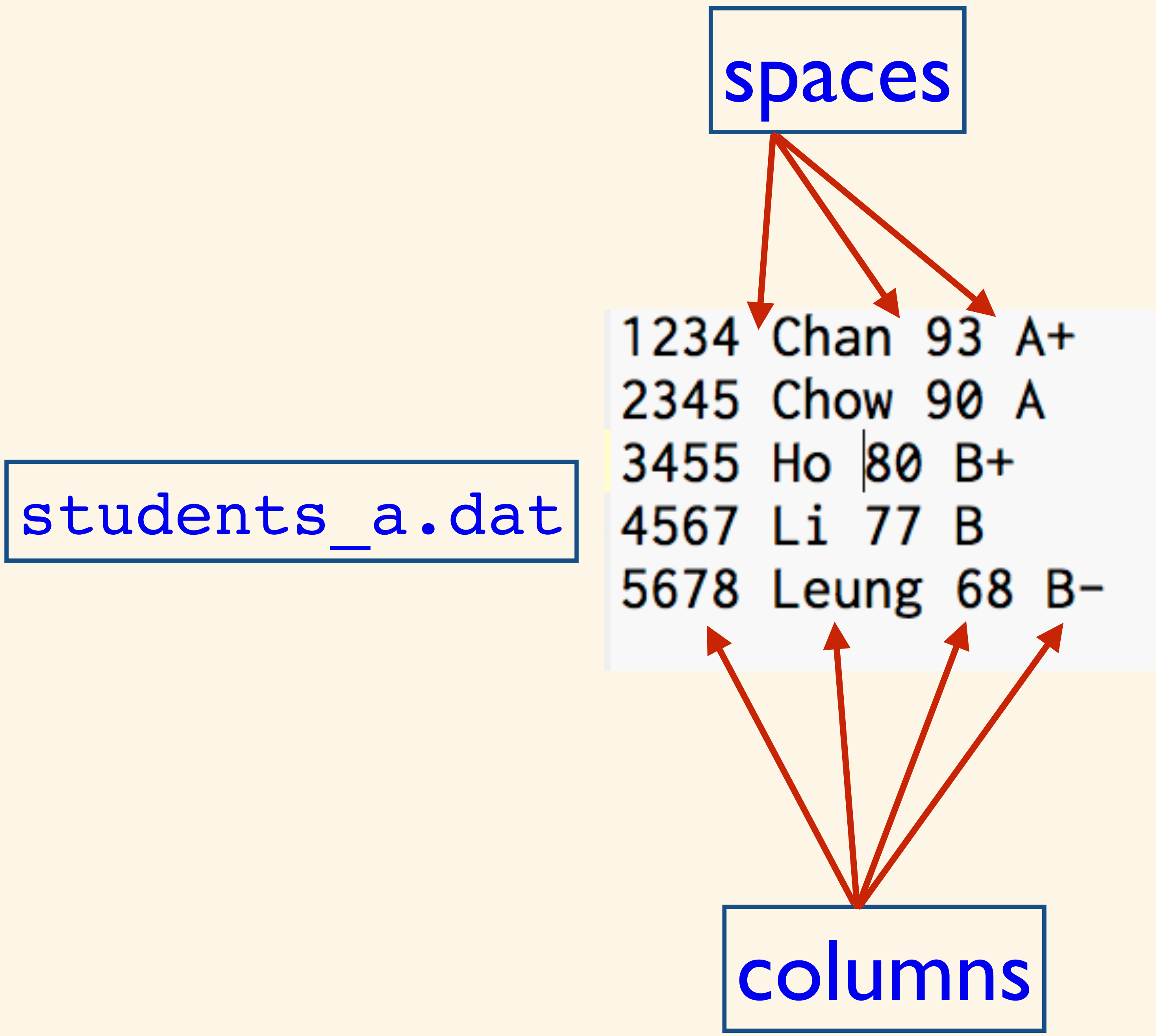
i) Structure of text files

Text files are almost always structured, i.e., the data are stored in a way that reflects the organisation of the original data.

The simplest approach is to store data as a table, i.e. row-column format with each line of data defining a row.

- Typically each column is separated with spaces.
- Alternatively one can separate columns with another symbol or delimiter (e.g. a comma or space).
- CSV format refers to comma-separated values. It usually (though not always) implies a comma for a delimiter and single quotes around the entries,

Example 2a: text data with spaces as delimiters



Columns aren't perfectly aligned

Example 2b: text data with tabs as delimiters

tabs

students_b.dat

```
1234	Chan	93	A+
2345	Chow	90	A
3455	Ho	80	B+
4567	Li	77	B
5678	Leung	68	B-
```

Columns are perfectly aligned!

columns

Example 2c: text data with commas as delimiters

commas

students_c.dat

```
1234, Chan, 93, A+
2345, Chow, 90, A
3455, Ho, 80, B+
4567, Li, 77, B
5678, Leung, 68, B-
```

Columns aren't perfectly aligned

columns

Example 2d: standard CSV format

commas

students_d.dat

```
'1234', 'Chan', '93', 'A+'
'2345', 'Chow', '90', 'A'
'3455', 'Ho', '80', 'B+'
'4567', 'Li', '77', 'B'
'5678', 'Leung', '68', 'B-'
```

Columns aren't perfectly aligned

data between quotes

Operations on text files

File operations or input/ouput on text files are analogous to those for input from the keyboard and output to the screen:

- Read one line (row) of data at a time
- Write one (row) of data at a time

ii) Binary files

- We will not deal with binary files in this course, but they're sometimes needed in real-world computing. Binary files require significantly less disk space than text files.

- Raw binary files are usually unstructured, i.e., just a stream of 0s and 1s.

- However, some special binary file formats contain metadata or data about data.

  - Metadata for MP3 files contain information such as song title, artist, etc.

  - Metadata for scientific file formats contain information such as the total number of variables, names of variables, etc.

Advantages of binary files

- Why do we want to bother with binary files?

  1. They require significantly less disk space than text files.

  2. They allow for random access of data, i.e., it's possible to jump to a specific location in a file.

- Text files only work with sequential access. This means that the data must be accessed line by line.

- We will see examples of this shortly…

- *Note*: disk benchmarks usually distinguish between random and sequential access. Generally sequential access is faster (when it's appropriate!).

File operations from an operating-system perspective

As far as the operating system is concerned, **operations involving files aren't fundamentally different from those with the keyboard or screen.**

Effectively all we need to do is create a "device" associated with a file.

Crudely speaking, this device is a variable. In some languages it's referred to as a file handle. In Python, it's an object.

- We shall refer to it as `f` or `f2`. File operations will be invoked by calling functions, e.g. `f.func()`
- Note similarity to qualified references for modules,

3. Basic operations

Creating a file object

To read or write to a file, a file object must be created using `open`.

```
f = open('testfile',mode)
```

`mode` refers to what will be done with the file:

- `'r'`: read-only mode
- `'w'`: write-only mode
- `'a'`:append (*writes to end of existing file if needed*)
- `'r+'`: read and write (*assumes file exists*)
- `'a+'`: read and write (file need not exist)

Not very useful in practice

Closing a file object

After we've finished with an object `f`, we need to close it using `close`.

```
f.close()
```

Closing a file serves two purposes:

1. Avoids accidental writes to the file.
2. Ensures that all the data are written.

Closing a file that has been used for input isn't as important (but it's still a good idea).

Example 3a: opening files in different modes

```python
f1 = open('testfile', 'r')
f2 = open('testfile2', 'w')
f3 = open('testfile3', 'r+')
```

write

read

read+write

```
ipdb> f1
<open file 'testfile', mode 'r' at 0x1179e3390>
```

```
ipdb> f2
<open file 'testfile2', mode 'w' at 0x1179e3420>
```

```
ipdb> f3
<open file 'testfile3', mode 'r+' at 0x1179e34b0>
```

Example 3b: closing files

```
f1.close()
f2.close()
f3.close()
```

```
In [219]: f1
Out[219]: <closed file 'testfile', mode 'r' at 0x1179e3390>

In [220]: f2
Out[220]: <closed file 'testfile2', mode 'w' at 0x1179e3420>

In [221]: f3
Out[221]: <closed file 'testfile3', mode 'r+' at 0x1179e34b0>
```

Each file that has been opened must be closed.

i) Writing to a file

A string `s` can be be written to a file using

```
f.write(s)
```

By default Python will attempt to write everything on the same line. To advance to the next line, you need to use the newline character `\n`

```
f.write('this is a string!\n')
```

Example 4: writing a string to a file

```
f2 = open('testfile2', 'w')

f2.write('this is the first string') # all on same line
f2.write('this is the second string')

f2.write('\n') # advance to new line
f2.write('\n') # blank line
f2.write('This is the first string\n') # separate lines
f2.write('This is the second string\n')
f2.close()
```

```
this is the first stringthis is the second string

This is the first string
This is the second string
```

testfile2

Where is the output file?

By default, Anaconda writes (and reads files) from the same directory as your program file. Jupyter Notebook and Google Colab work the same way.

In case there are problems, select Preferences/Settings and under Working directory settings select `Keep The directory of the file being executed.`

Writing numbers to a file

- The easiest way to write number to a file is to use formatted output.
- The syntax is identical to that for `print`, e.g.

```
f.write('integer={:d}, float={:f} \n'.format(i,x))
```

Example 5: writing numbers to a file

```python
from math import *

f2 = open('example5.dat', 'w')

f2.write( 'Pi = {:f} \n'.format(pi) )
f2.write( 'e = {:f} \n'.format(e) )
f2.write( 'cos(0) = {:f} \n'.format(cos(0)) )

f2.close()
```

```
Pi = 3.141593
e = 2.718282
cos(0) = 1.000000
```

example5.dat

Aligned output

- As usual, we can separate columns using spaces, e.g.

```
f.write('{:f} {:f}'.format(var1,var2))
```

- To obtain neatly aligned output, use tabs or `\t` as a separator:

```
f.write('{:f}\t{:f}'.format(var1,var2))
```

Comparison of spaces and tabs

```
8 2.828427
9 3.000000
10 3.162278
11 3.316625
```

```
97 9.848858
98 9.899495
99 9.949874
100 10.000000
```

spaces

```
8	2.828427
9	3.000000
10	3.162278
11	3.316625
```

```
97	9.848858
98	9.899495
99	9.949874
100	10.000000
```

tabs

Overwriting a file

- What happens if we attempt to write to a file that exists?
- If the file is opened in the default `'w'` mode, the old file will be overwritten, i.e., the previous contents will be lost.
- If the file is opened in `'a'` mode, then the data will be appended to the old data.

Example 6: appending to a file

```
from math import *

f2 = open('example5.dat', 'a') # append output to example5.dat

f2.write('sqrt(Pi) = {:f} \n'.format(sqrt(pi)) )
f2.write('log(e) = {:f} \n'.format(log(e)) )
f2.write('cos(pi) = %f \n'.format( cos(pi)) )

f2.close()
```

```
Pi = 3.141593
e = 2.718282
cos(0) = 1.000000
sqrt(Pi) = 1.772454
log(e) = 1.000000
cos(pi) = -1.000000
```

appended output

example5.dat

ii) Reading from a file

One can read an entire file using

```
f.read()
```

This yields a string variable.

Example 7: reading an entire file

```
f1 = open('testfile2', 'r') # output from Example 4
entirefile=f1.read() # contents of file stored in string variable
f1.close()

print( entirefile )
```

```
this is the first stringthis is the second string

This is the first string
This is the second string
```

output is identical to before!

Comments

Reading the entire file all at once usually isn't a good idea.

- For big files, this can require a lot of memory.
- Usually we want to analyse or process the data in a file.

4. More advanced operations

Reading data line-by-line (1)

- It's usually preferable to read the data line-by-line.
- The easiest way is to loop over the file object:

```
for line in f:
    print(line, end='')
```

Needed to avoid extra blank lines

Example 8: reading an entire file using a `for` loop

```
f1 = open('testfile2', 'r') # output from Example 4

for line in f1:
    print(line, end='')  # don't add extra newline
    # print(line)  #  add extra newline

f1.close()
```

```
this is the first stringthis is the second string

This is the first string
This is the second string
```

output is still identical!

Reading data line-by-line (2)

- An equivalent approach is to use

```
f.readline()
```

  to read a single line. At the end of the file, this returns an empty string, ‘’.

- Note that `readline` also captures `'\n'`, the special code for new lines. Thus one needs to be careful when using `readline` with `print`.

Example 9: simple application of `readline()`.

Assume that the length of a file is known.

```
f1 = open('testfile2', 'r') # output from Example 4

N=10 # we're given that input file has 4 lines

for n in range(N):
    line=f1.readline()
    print(line, end='')
    #print(line)

f1.close()
```

```
this is the first stringthis is the second string

This is the first string
This is the second string
```

output is still identical!

How do we read data from different columns?

- Most of the time, you'll be dealing with files containing numbers in different columns. However, by default Python reads strings.
- To read the data into variables, we need to split the string using a specified delimiter.

Splitting a string

- We can split a string using `string.split()`

$$\boxed{\texttt{string.split(sep)}}$$

where `sep` is the delimiter.

- This returns a **list** with the elements corresponding to the different words separated by the delimiter.

Example 10: splitting a string.

```
string='this is a test'
split_string= string.split(' ')

print( 'original string:',string )
print( 'split string (list):',split_string )
```

```
original string: this is a test
split string (list): ['this', 'is', 'a', 'test']
```

Example 10b: splitting a string using another delimiter

```
string='green, eggs, and, ham, are, very, delicious, indeed'
string_split=string.split(',')
print( 'string=',string )
print('string_split=',string_split )
```

```
string= green, eggs, and, ham, are, very, delicious, indeed
string_split= ['green', ' eggs', ' and', ' ham', ' are', ' very', ' delicious', ' indeed']
```

Comments

1. This procedure can be easily modified to handle other kinds of delimiters or file formats.

2. Specialised Python modules exist for reading and writing CSV files; however, they're a bit complicated for our needs.

3. There are easier ways to read data into vectors and matrices. We will cover them in the next section.

Summary

1. File input and output is handled with a file object, `f`.
2. File objects can be opened for read or write.
3. A string can be written to a file using `f.write()`.
4. There are various ways of reading strings from a file. Usually `f.readline()` is simplest.
5. To read data from multiple columns, the input string needs to be split.