Files and I/O in Python

Prof. Pai H. Chou
National Tsing Hua University

Outline

- Steps in accessing a file
- Python features
  - File routines: open, read, readlines, write, writelines, close
  - `with`-statement
- System features
  - Standard I/O, standard error
  - file redirection and pipes
- Text vs. Binary files
- os package and directories

Steps in accessing a file

- open a file
  - give a file name, reading/writing mode
  - get a file-handle (fh) data structure
- Use the file handle data structure
  - read or write (by # of bytes, by line, etc)
  - move the "file head" within the file (forward, backward, to a given position, relative to a position)
- Close a file

Opening a file

- built-in function `open`(*path*, *mode*)
  - *path*: file name, possibly with directory
  - *mode*: read, write (truncate, append)
  - return value: a "file handle" (*fh*) to reference a file
- Typical usage:
  - `fh = open('fileName', 'r')  # for reading`
  - `fh = open('fileName', 'w')  # for writing`
  - `fh = open('fileName')   # defaults to reading`

file path and mode

- path examples, following Unix convention

`hello.py`	file name in current directory
`./hello.py`	file name in current directory
`../hello.py`	file name in parent directory
`hw4/hello.py`	file name in subdirectory
`/usr/local/hello.py`	absolute file path

- mode

`'r'`	**r**eading (file must already exist)
`'w'`	**w**riting (wipe out ('truncate') if exists; else create)
`'a'`	**a**ppend if exists; else create

methods for reading file

- methods:
  - fh`.readline()` (singular)
    - reads a single line at a time until end of file
  - fh`.readlines()` (plural)
    - reads all the lines and put them in a list
  - fh`.read(`*n*`)`
    - reads *n* characters (text) or bytes (binary); returns `''` (empty string) on end of file.
- Concept: file head position moves as you read data

readline(): read one line at a time

```
>>> fh = open('arg.py', 'r')
>>> fh.readline()
'#!/usr/bin/python\n'
>>> fh.readline()
'\n'
>>> fh.readline()
'import sys\n'
>>> fh.readline()
'L = len(sys.argv)\n'
>>> fh.readline()
''
```

- delimiter: newline
- how do you know when you reach the end of file?
  - readline() returns empty string ''

readlines(): read lines as list of strings

```
>>> fh = open('arg.py', 'r')
>>> fh.readlines()
['!#/usr/bin/python\n', '\n', 'import sys\n', 'L = len(sys.argv)\n']
```

- Similar to readline() multiple times
  - delimiter: newline
- Purpose: allow use with for loop
  - `for` *line* `in` fh`.readlines():`

    … compute using line as string

read(): read a number of characters (text) or bytes (binary)

```
>>> fh = open('arg.py', 'r')     # open text file
>>> fh.read(10)  # read 10 characters
'!#/usr/bin'
>>> fh.read(9)   # read the next 9 characters
'/python\n\n'
>>> fh.read()    # read till end of file
'import sys\nL = len(sys.argv)\n'
>>> fh.read(100) # try reading 100 more characters, but no more
''
```

- parameter: # of characters (text) or bytes (binary) you try to read
- actual # units (characters or bytes) read, can be less if not enough
- omitted => read the rest of the file

methods for writing to file

- methods:
  - fh`.write(`s`)` # this is used for most purposes
    - text file: writes the string *s* to file in the default encoding;
    - binary file: write the raw bytes sequence s to file
  - fh`.writelines(`*L*`)` (plural)
    - writes all the lines and put them in a list
    - does not write newline at the end of each line!!
- Concept:
  - file head position moves as you write bytes
  - there is no fh`.writeline(`*s*`)` (singular)!!

methods for writing to file

- writes string `s` to file
  - unlike `print()`, `write()` doesn't add newline

```
$ python3
>>> fh = open('e.txt', 'w')
>>> fh.write('ABCDE\nWXYZ\n')
>>> fh.close()
>>> ^D
$ more e.txt
ABCDE
WXYZ
$
```

write() to text file must take a str

- get an error if you try to write a non-string (e.g., a list)

```
>>> fh = open('out.txt', 'w')
>>> L = [1,2,3]
>>> fh.write(L)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: write() argument must be str, not list
>>>
```

with-as statement

- a way to scope an open file, close on exit
- syntax

  `with` `open(...)` **`as`** *fh*: # like fh = open(..)

      statements to use fh
- file fh will be closed upon exiting the suite

```
fh = open('arg.py', 'r')
L = fh.readlines()
print(L)
fh.close()
```

⇔

```
with open('arg.py', 'r') as fh:
    L = fh.readlines()
    print(L)
# automatically closes the file!
```

Random access: **`seek()`, `tell()`**

- fh.`tell()`: get the file head's position
- `fh.seek(`*o*`,` *r*`)`: go to file positions
  - *o*: byte offset
  - *r*: relative to (current position, beginning, end)

seek(): set position of file head

```
>>> fh = open('arg.py', 'r')
>>> fh.readline()
'#!/usr/bin/python\n'
>>> fh.readline()
'\n'
>>> fh.readline()
'import sys\n'
>>> fh.readline()
'L = len(sys.argv)\n'
>>> fh.readline()
''
-----------------------------------------------
>>> fh.seek(0) # goto beginning
>>> fh.readline()
'#!/usr/bin/python\n'
>>> fh.readline()
'\n'
```

- seek(0) sets the file position to the beginning
  - => can read again!!
- can seek to any other position

tell(): get position of file head

```
>>> fh = open('arg.py', 'r')
>>> fh.readline()
'#!/usr/bin/python\n'
>>> fh.readline()
'\n'
-----------------------------------------
>>> savedPos = fh.tell()
>>> savedPos
19
-----------------------------------------
>>> fh.readline()
'import sys\n'
>>> fh.readline()
'L = len(sys.argv)\n'
-----------------------------------------
>>> fh.seek(savedPos)
>>> fh.readline()
'import sys\n'
```

- fh.tell() gets the file position
- can save the position, then later seek() to it

Standard I/O and files

- standard input: keyboard (by default)
  - read using `input()` function, one line at a time
- standard output: text display (by default)
  - written by `print()` to the text terminal
- They are file-like devices
  - already open by default; don't open/close them
  - `import sys`
    `sys.stdin` and `sys.stdout` for their "file handles"

Standard I/O example

- Echo keyboard to output
  - read by `sys.stdin.readline()`
  - write by `sys.stdout.write(`*s*`)`

```
#!/usr/bin/env python3
"Demonstrates stdio by echoing"
import sys
while True:
    line = sys.stdin.readline()
    if line == '':
        break
    sys.stdout.write(line)
```

I/O redirection

- a (unix) shell feature, not a Python feature
- Redirects `stdout` to a file using > *file*

```
$ grep return *.py > result
$ more result
    return a
    return True
$ _
```

  - Append by >> *file*

- Redirects `stdin` from a file using < *file*

```
$ wc -w < arg.py
240
$ _
```

Unix pipes

- another Unix feature
  - connect stdout of one program to stdin of another
  - syntax:
    $ *cmd1* | *cmd2* | … | *cmdN*
- Example

```
$ grep return *.py | wc
```

  - output of `grep return *.py` is fed into `wc` (word-count), which counts #lines, words, and characters

stderr: standard error output

- what happens on an error?
- without redirection

```
$ grep return me
grep: me: No such file or directory
```

- with redirection

```
$ grep return me > result
grep: me: No such file or directory
```

- error message did not get redirected to a file!
  - standard error is different (`sys.stderr`) and is not redirected, even though `sys.stdout` also goes to screen

To report error messages to stderr

- instead of calling `print(`*errorMsg*`)`, call `sys.stderr.write(`*errMsg*`)`

```
#!/usr/bin/env python3
import sys
fileName = 'me'
try:
    fh = open(fileName)
except:
    sys.stderr.write(f'cannot open file "{fileName}"\n')
    sys.exit(1)
print(f'file "{fileName}" opened successfully\n')
```

Example revisited: `myuniq.py`

```python
#!/usr/bin/env python3
import sys
numberOfArgs = len(sys.argv)
if numberOfArgs != 2:
    sys.stderr.write('Usage: %s inputFile\n' % sys.argv[0])
    sys.exit(1)
try:
    fh = open(sys.argv[1], 'r')
except:
    sys.stderr.write('cannot open input file %s\n' % sys.argv[1])
    sys.exit(2)
previousLine = ''  # initialize
for line in fh.readlines():
    if line != previousLine: # filter
        print(line, end='')
    previousLine = line  # update previous
fh.close()
```

Alternative code

file: altuniq.py

```
#!/usr/bin/env python3
import sys
numberOfArgs = len(sys.argv)
if numberOfArgs != 2:
    sys.stderr.write(f'Usage: {sys.argv[0]} inputFile\n')
    sys.exit(1)
try:
    with open(sys.argv[1], 'r') as fh:
        previousLine = ''  # initialize
        for line in fh.readlines():
            if line != previousLine: # filter
                print(line, end='')
            previousLine = line  # update previous
except OSError as err:
    sys.stderr.write(str(err)+'\n')
    sys.exit(2)
```

Demo running altuniq.py

- Purpose: show different error messages

```
$ python3 altuniq.py foo
[Errno 2] No such file or directory: 'foo'
$ touch foo          # create an empty file named foo
$ chmod -r foo       # remove read permission on foo
$ python3 altuniq.py foo
[Errno 13] Permission denied: 'foo'
$
```

Text vs. Binary files

- Text file: concept of newline
  - Unix: '\n', DOS/Windows: CRLF '\r\n'
  - Opening as text mostly takes care of newlines
  - Example: `.py` file, `.c`, `.h`, `.html`, ...
- Binary files: just bytes
  - no newline conversion
  - may be faster to work with, suitable for everything else (that is not a plain text file)
  - Example: `.mp3` file, `.mov`, `.jpg`, `.doc`, `.ppt`, `.png`, `.zip` ...

Text vs. Binary: open()

- Text
  - `open(`*filename*`, 'r')` to read
  - `open(`*filename*`, 'w')` to write
- Binary
  - `open(`*filename*`, 'rb')` to read,
  - `open(`*filename*`, 'wb')` to write

Text vs. Binary: read()

- Text:
  - `fh.read()` returns a `str` object (text)
    `str` = sequence of unicode characters
  - can call `readline()`, `readlines()`, etc
- Binary:
  - `fh.read()` returns a `bytes` object (binary)
    `bytes` = sequence of bytes (8-bit)

What is bytes data type?

- sequence of 8-bit data
  - each element is a byte, rather than a char
  - each char in Python `str` is a unicode character
- byte literal: `b'...'`
  - e.g., `b'hello world'` - content uses ASCII chars
  - each symbol is exactly one byte

bytes vs characters

- byte: exactly 8 bits each
- character:
  - ASCII: can fit in one byte
  - Unicode: may need 1, 2, 3, or 4 bytes!
- Encoding scheme:
  - UTF-8: variable-length encoding of Unicode
  - Represent a Unicode character as a sequence of bytes
  - if ASCII => 1 byte; European character => 2 bytes; Asian character => 3 or 4 bytes etc.

Conversion between `bytes` and `str`

- From raw `bytes` to `str`
  - *textString* = `str(`*rawBytesData*`, 'UTF8')`
- From `str` to `bytes`
  - *rawBytes* = `bytes(`textString`, 'UTF8')`

```
>>> data = bytes('你好', 'UTF8')
>>> data
b'\xe4\xbd\xa0\xe5\xa5\xbd'
>>> text = str(data, 'UTF8')
>>> text
'你好'
```

General file operations used by Unix shell

- Files
  - `cp`
  - `mv`
  - `rm`
- Permission
  - `chmod`
  - `chown`
  - `chgrp`
- Directories
  - `ls`
  - `cd`
  - `pwd`
  - `mkdir`
  - `rmdir`

These are services provided by the operating system (OS)! rather than Python language itself.

os module

- API to access services provided by OS
- File and directory API
  - os.chdir(path)
  - os.chmod(path, mode)
  - os.chown(path, uid, gid)
  - os.getcwd()
  - os.getcwdb()
  - os.link(s, d)
  - os.listdir(path)
  - os.mkdir(path)
  - os.remove(path)
  - os.rename(src, dst)
  - os.replace(src, dst)
  - os.rmdir(path)
  - os.truncate(path, length)

`os.path` module

- definitions specific to file paths
  - convert to absolute path
  - resolve alternative path names to a file
  - parse and join directory names and file names
  - get last access/modified time of a file