File, Directory and External Program I/O

Introduction: File, Directory and External Program I/O; Exceptions

Our programs don't always work in isolation -- they can read directories, write to files and run external programs.

This unit is about the "outside world" of your computer's operating system -- its files and directories and other programs running on it, the *STDOUT* and *STDIN* data streams that can pass data between them, as well as the *command line*, which is the prompt from which we have been running our Python programs.

The data we have been parsing has been accessed by us from a specific location, but often we are called upon to marshal data from many locations in our filesystem. We may also need to search for these files.

We also sometimes need to be able to read data produced by programs that reside on our filesystem. Our python scripts can run Unix or Windows utilities, installed programs, and even other python programs from within our running Python script, and capture the output.

Objectives for the Unit: File, Directory and External Program I/O

- take input at the command line with *sys.argv*
- write to and append to files with the **file** object
- list files in a directory with *os.listdir()*
- learn about file metatdata using *os.path.isfile()* and *os.path.isdir()*, *os.path.getsize()*
- traverse a tree of directories and files with *os.walk()*
- interact with files and other programs at the command line with *STDIN* ("standard in") and *STDOUT* ("standard out")

- launch external programs with *subprocess*

Summary task: writing and appending to files using the file object

Files can be opened for writing or appending; we use the **file** object and the **file write()** method.

```
fh = open('new_file.txt', 'w')
fh.write("here's a line of text\n")
fh.write('I add the newlines explicitly if I want to write to the file\n'
fh.close()

lines = open('new_file.txt').readlines()
print lines
  # ["here's a line of text\n", 'I add the newlines explicitly if I want
```

Note that we are explicitly adding newlines to the end of each line. The **write()** method doesn't do this for us.

Summary task: redirecting the STDOUT data stream

STDOUT is the 'output pipe' from our program (usually the screen, but it can be redirected to a file or other program).

hello.py: print a greeting

```
#!/usr/bin/env ptython

print 'hello, world!'
```

redirecting STDOUT to a file at the command line:

```
mycomputer$  python hello.py
hello, world!                                  # default:  to the screen
mycomputer$  python hello.py > newfile.txt     # > redirect to a file (not


mycomputer$  cat newfile.txt                   # cat spits out a file's con
hello, world!

C:\$  type newfile.txt                         # same, but for Windows
hello, world!
```

STDOUT is something we have been using all along. Any **print** statement sends string data to **STDOUT**. It is simply the conduit that allows us to send data out of our program.

In the above example, we first run the program (which prints a greeting). We see that this prints to the screen. Next, we use the **>** *redirection operator* (which is an operating system command, not a Python feature) to redirect **STDIN** to a file instead of the screen. This is why we don't see the output to the screen - it has been redirected.

redirecting STDOUT to the input of another program at the command line:

```
mycomputer$  python hello.py
hello, world!                       # default:  to the screen

mycomputer$  python hello.py | wc   # "piped" redirect to the wc utility

       1       2      14            # the output of wc
```

In the above example, we are *piping* the output of **hello.py** to another program: the **wc** utility, which counts lines, words and letters. **wc** can work with a filename, but it can also work with **STDIN**. In this case we see that **hello, world!** has **1** line, **2** words and **14** 14 characters.

Summary task: writing to STDOUT in different ways

We can use **print**, **print** with a comma, or **sys.stdout.write()** to write to **STDOUT**

print: print a newline after each statement

```
print 'hello, world!'
print 'how are you?'

                          # hello, world!
                          # how are you?
```

print with a comma: print a space after each statement

```
print 'hello, world!',
print 'how are you?'

                         # hello, world! how are you?
```

sys.stdout.write(): print nothing after each statement

```
import sys

sys.stdout.write('hello, world!')
sys.stdout.write('how are you?')

                         # hello, world!how are you?
```

Summary task: reading and redirecting the STDIN data stream

STDIN is the 'input pipe' to our program (usually the keyboard, but can be redirected to read from a file or other program).

```
import sys

for line in sys.stdin.readlines():
  print line

## or, alternatively:
## filetext = sys.stdin.read()
```

A program like the above could be called this way:

```
mycomputer$ python readfile.py < filetoberead.txt
```

The **stdin** datastream can also come from another program and *piped* into the program:

```
mycomputer$ ls -l | python readfile.py     # unix

mycomputer$ dir | python readfile.py       # windows
```

In this command, the **ls** unix utility (which lists files) outputs to **STDOUT**, and the *pipe* (the vertical bar) passes this data to **readfile.py**'s **STDIN**. It's common practice in unix to pass one program's output to another's input.

Summary task: read directories with os.listdir()

os.listdir() can read any directory, but the filename must be appended to the directory path in order for Python to find it.

```
import os                                           # os ('operating system'
mydirectory = '/Users/dblaikie'

for item in os.listdir(mydirectory):

    item_path = os.path.join(mydirectory, item)

    print item_path                                 # /Users/dblaikie/photos
                                                    # /Users/dblaikie/backup
                                                    # /Users/dblaikie/colleg
                                                    # /Users/dblaikie/notes.
                                                    # /Users/dblaikie/financ
```

Here we see all the files in my home direcctory on my mac (**/Users/dblaikie**). We must use **os.path.join()** to join the path to the file to see the whole path to the file.

os.path.join() is designed to take any two or more strings and insert a directory slash between them. It is preferred over regular string joining or concatenation because it is aware of the operating system type and inserts the correct slash (forward slash or backslash) for the operating system.

Summary task: read directory listing type with os.path.isfile() and os.path.isdir()

os.path.isdir() and **os.path.isfile()** return **True** or **False** depending on whether a listing is a file or directory.

```
import os                                               # os ('operating system'
mydirectory = '/Users/dblaikie'

for item in os.listdir(mydirectory):

    item_path = os.path.join(mydirectory, item)

    if os.path.isdir(item_path):
        print "{}:  {}".format(item, 'directory')
    elif os.path.isfile(item_path):
        print "{}:  {}".format(item 'file')
                                                        # photos:  directory
                                                        # backups:  directory
                                                        # college_letter.docx:
                                                        # notes.txt:  file
                                                        # finances.xlsx:  file
```

Summary task: read file size with os.path.getsize()

os.path.getsize() takes a filename and returns the size of the file in bytes

```
import os                                               # os ('operating system'
mydirectory = '/Users/dblaikie'

for item in os.listdir(mydirectory):
    item_path = os.path.join(mydirectory, item)
    item_size = os.path.getsize(item_path)
    print "{}:  {} bytes".format(item_path, item_size)
```

Keep in mind that Python won't be able to find a file unless its path is prepended. This is why **os.path.join()** is so important.

Summary exception: OSError with os.listdir() (and a bad directory)

Python will raise an **OSError** exception if we try to read a directory or file that doesn't exist, or we don't have permissions to read.

```
import os

user_file = raw_input('please enter a filename:  ')    # user enters a fi

file_size = os.listdir(os.path.getsize(user_file))


Traceback (most recent call last):
    File "getsize.py", line 5, in
OSError:  No such file or directory:  'mispeld.txt'
```

How to handle this exception? Test to see if the file exists first, or trap the exception (see "Exceptions", coming up).

Summary module: launch an external program with *subprocess*

The **subprocess** module allows us to spawn new processes, connect to their input/output/error pipes, and obtain their return codes.

subprocess.call(): execute a command and output to STDOUT

The first argument is a list containing the command and any arguments. Here's a Unix example:

```
import subprocess

# output to STDOUT (usually the screen)
subprocess.call(['ls', '-l'])                  # Unix-specific
```

For Windows, we can instead call the **dir** Windows utility:

```
import subprocess

# output to STDOUT (usually the screen)
subprocess.call(['dir', '.'], shell=True)
```

Each of the arguments **stdout=**, **stderr=** and **stdin=** can point to an open filehandle for reading or writing. **stderr=STDOUT** will redirect stderr output to stdout.

```
# output to an open filehandle
subprocess.call(['ls', '-l'], stdout=open('outfile.txt', 'w'), stderr=STD

# read from a file
subprocess.call(['wc'], stdin=open('readfile.txt'))
```

shell=True will execute directly through the shell. In this case the entire shell command including arguments must be passed in a single argument. Using this flag and executing the command through the shell means that shell expansions (like * and any other shell behaviors can be accessed.

```
subprocess.call(['ls *'], shell=True)           # Unix-specific; use dir
```

However, never use **shell=True** if the command is coming from an untrusted source (like web input) as shell access means that arbitrary commands may be executed.

subprocess.check_output(): execute a command and return the output to a string

```
var = subprocess.check_output(["echo", "Hello World!"])  # Unix-specific
print var                                                 # Hello World!

out = subprocess.check_output(['wc', 'test.txt'])        # Unix-specific
print out           #       43     112     845 test.py   (this is wc outp
```

Windows:

```
var = subprocess.check_output(["dir", "."], shell=True)
print var          # prints file listing for the current directory
```

Many of the optional arguments are the same; **check_output()** simply returns the output rather than sending it to **STDOUT**.

Sidebar -- summary function: traverse a directory tree with os.walk()

os.walk<() visits every directory in a directory tree so we can list files and folders.

```
import os
root_dir = '/Users'
for root, dirs, files in os.walk(root_dir):    # root string, dirs list,
  for dir in dirs:                             # loop through directories
    print os.path.join(root, dir)              # print full path to dir
  for file in files:                           # loop thgrough files in t
    print os.path.join(root, file)             # print full path to file
```

os.walk does something magical (and invisible): it traverses each directory, descending to each subdirectory in turn. Every subdirectory beneath the root directory is visited in turn. So for each loop of the outer **for** loop, we are seeing the contents of one particular directory. Each loop gives us a new list of files and directories to look at; this represents the content of this particular directory. We can do what we like with this information, until the end of the block. Looping back, **os.walk** visits another directory and allows us to repeat the process.