stdin, stdout, stderr

stdout and stderr

Many programs make output to "standard out" and "standard error" (e.g. the print command goes to standard out, error messages go to standard error).

By default, both are written to the screen, but you can redirect each of them (next slide).

They are often referred to as "streams" (information streams).

These two streams are available directly to you via the `sys` module: `sys.stdout` and `sys.stderr`.

Writing and redirecting

Write to stdout and stderr with file write-like statements:

```
sys.stdout.write("blah blah\n")
sys.stderr.write("read 6 sequences, analysis complete\n")
```

When you use a program with these outputs, you can direct each stream into files as follows (`stdout` to fileA and `stderr` to fileB):

```
python myprog.py > fileA 2> fileB
```

Users of command line programs often expect to find these two sorts of outputs, which typically correspond to the main intended output of the program (`stdout`) and error or progress messages (`stderr`).

Standard input

standard in is analogous to standard out.

`sys.stdin` is a data stream that you can use in your program
as if it were reading from a file:

```
for line in sys.stdin:
    # do something with each line
```

Under some circumstances it is ideal to allow a user to provide input EITHER
from a file OR from stdin:

```
if fileName != None:
    inStream = open(fileName)
else:
    inStream = sys.stdin
# after this the code is identical for either data source
```

[the same pattern can be used for writing to a file or stdout]

Why bother with stdin?

Writing and reading files is often a slow step in program execution.

Though you can treat `sys.stdin` as if it were an open file, in fact it is sitting in RAM.

Suppose I have a tree-building program that can take a sequence alignment from `stdin` or a file, and a program that can write a sequence alignment either to `stdout` or to a file. These are equivalent:

```
align_prog.py infile > alnfile
tree_prog alnfile > outtree
```

or

```
align_prog.py infile | tree_prog > outtree
```

stdout piped directly to stdin

The first version has to write the alignment to a file, then read it again. The second version never reads or writes the alignment. That "`|`" symbol is read "pipe" and connects the two programs via `stdout` and `stdin`.

Redirection syntax

prog > file	write stdout to file
prog >> file	append stdout to file
prog 2> file	write stderr to file
prog 2>> file	append stderr to file
progA | progB	pipe stdout progA to stdin progB
prog < file	read from file to stdin prog

Running programs from inside a Python program

You can easily run other programs from inside a Python program:

```
import os

os.system("someProg myFile.txt > outFile.txt")

# the Python program pauses until someProg is complete
# someProg can be any kind of program (Python or something else)

# now that someProg is done, we can access its output
of = open("outFile.txt")
for line in of:
   < blah blah >
```

string that is exactly what you would have typed at the command line to run someProg

[fyi there are much more complex ways of controlling external programs, including setting parameters, tracking progress, handling errors etc.]

Sample Problem 1

Write a program that reads a file and makes two kinds of output: the number of words in the file on `stdout` and the number of lines in the file on `stderr`. Do NOT read the whole file at once in case it is very large.

If you finish early, try running your program and redirecting one or both output streams to files.

Answer 1

```
import sys

lineNum = 0; wordNum = 0    # this syntax is allowed, handy
of = open(sys.argv[1])
for line in of:
   lineNum += 1
   wordNum += len(line.split())
of.close()

outPrefix = "file " + sys.argv[1] + " has "        # another handy trick
sys.stdout.write(outPrefix + str(wordNum) + " words\n")
sys.stderr.write(outPrefix + str(lineNum) + " lines\n")
```

Sample Problem 2

Write a program that reads either a file or `stdin` and writes to `stdout` a list of all the words in the input in alphabetical order (one per line). TIP – use a `dict`.

Set it up so that if the program gets a command line argument, it expects it to be a file name, and if NOT it reads `stdin`.

You can use the following command to make sure it works right:

```
cat filename | python myprog.py
```

This should give the same result as `python myprog.py filename`

Answer 2

```
import sys

if len(sys.argv) > 1:
   instream = open(sys.argv[1])
else:
   instream = sys.stdin

wordDict = {}
for line in instream:
   for word in line.split():
      wordDict[word] = None    # we won't use value so don't waste memory
instream.close()

words = wordDict.keys()
words.sort()
for word in words:
   print word                  # or sys.stdout.write(word + "\n")
```

remember each key appears once in a `dict` - a word that appears multiple times in the stream will write over the key after the first appearance

Challenge Problems

1) Write a program that takes EITHER the output of your program in Problem 2 OR a specified file and makes as output a count of the number of words found.

2) In problem 2, instead of making a list of all the words, count how many times each word appears and make as output each word and its count (use a `dict` with count as value).