FILE INPUT/OUTPUT

LECTURE 12-1

JIM FIX, REED COLLEGE CSCI 121

COURSE INFO

- **Today:** file input/output
- **Wednesday:** error handling
- **For both:** homework along with BSTs, due by end of semester.
- **Project 4:**
  - Can work on game in lab tomorrow.
  - Checkpoint of 15-20 points is due **Friday!**
- **Next Monday:** second midterm exam. Topics covered:
  - recursion
  - object orientation
  - inheritance
  - higher-order functions
  - linked lists

WHY USE FILES?

- Process data created by other programs.
- Produce data analyzed/displayed by other programs.
- Take snapshots of a program's or user's progress.
- Read program "assets": images, sounds, user info, &c.
- Produce documents in standard formats.

- Python was first designed to be a scripting and/or "glue" language for enlisting several programs, automating their work, and massaging their data files.
- Python makes file reading and writing fairly easy.

WHY USE FILES?

- Saving: a program can store info it has collected or computed.
  - Can read it back later.
  - Can share for use in other programs.
- Loading: a program can access data from other programs, data sources.

WHY USE FILES?

- **Writing**: a program can store info it has collected or computed.
  - Can read it back later.
  - Can share for use in other programs.
- **Reading**: a program can access data from other programs, data sources.

WHY USE FILES?

- **Writing**: a program can store info it has collected or computed.
  - Can read it back later.
  - Can share for use in other programs.
- **Reading**: a program can access data from other programs, data sources.

- Disk storage is effectively permanent.
- Python program's data in memory is not.
  - ➡ Program could crash with an error.
  - ➡ Or computer system could crash. Etc.

LET'S CREATE A FILE

- Consider this script:

```
f = open("testing.txt","w")
f.write("let's write words")
f.write("into a file")
f.write("and see what")
f.write("happens")
f.close()
```

LET'S CREATE A FILE

- Consider this script:

```
f = open("testing.txt","w")
f.write("let's write words")
f.write("into a file")
f.write("and see what")
f.write("happens")
f.close()
```

- Line 1 creates a file object `f`.
  - ➡ This leads to creation of an associated text file on disk.
  - ➡ That file object is set up so that the text file can be written to.
- Lines 2-5 write a series of characters into the text file.
- Line 6 closes the file. Saving all changes to the disk.

LET'S CREATE A FILE

- Consider this script:

```
f = open("testing.txt","w")
f.write("let's write words")
f.write("into a file")
f.write("and see what")
f.write("happens")
f.close()
```

- Let's examine the results in the terminal:

```
$ cat testing.txt
let's write wordsinto a fileand see whathappens$
```

INSTEAD WITH A NEWLINE CHARACTER

- Consider this script, instead:

```
f = open("testing.txt","w")
f.write("let's write words")
f.write("into a file\n")
f.write("and see what")
f.write("happens")
f.close()
```

INSTEAD WITH A NEWLINE CHARACTER

- Consider this script, instead:

```
f = open("testing.txt","w")
f.write("let's write words")
f.write("into a file\n")
f.write("and see what")
f.write("happens")
f.close()
```

- Let's examine the results in the terminal:

```
$ cat testing.txt
let's write wordsinto a file
and see whathappens$
```

WITH AN ENDING NEWLINE CHARACTER

- Consider this script, instead:

```
f = open("testing.txt","w")
f.write("let's write words")
f.write("into a file\n")
f.write("and see what")
f.write(“happens\n”)
f.close()
```

WITH AN ENDING NEWLINE CHARACTER

- Consider this script, instead:

```
f = open("testing.txt","w")
f.write("let's write words")
f.write("into a file\n")
f.write("and see what")
f.write("happens\n")
f.close()
```

- Let's examine the results in the terminal:

```
$ cat testing.txt
let's write wordsinto a file
and see whathappens
$
```

APPENDING

- Consider this script, instead:

```
f = open("testing.txt", "a")
f.write("and let's include this, too\n")
f.close()
```

APPENDING

- Consider this script, instead:

```
f = open("testing.txt", "a")
f.write("and let's include this, too\n”)
f.close()
```

- We run this script 3x and inspect the results.

```
$ python3 sample_append.py
$ python3 sample_append.py
$ python3 sample_append.py
$ cat testing.txt
let's write wordsinto a file
and see whathappens
and let’s include this, too
and let’s include this, too
and let’s include this, too
$
```

FILE PATH

- Let's examine the use of **`open`**:

  `f = open(`*file path and name*`,`*mode*`)`**

- The file path is what you have become used to typing, a series of folders ending with the file's name:

```
csci121/hw12/test.txt
some_file.txt
../../pr4/arcade/PlayAsteroids.out
/Users/jimfix/Desktop/csci121/my_output.txt
```

- On MacOS or Unix, if the first character is a **`/`** then it is an *absolute path* from the *root folder*. On Windows machines it is a disk designator (e.g. **`C:`**) followed by the slash character.
- Otherwise it is a *relative path*.

FILE OBJECT'S MODE

- Let's examine the use of **`open`**:

  `f = open(`*file path and name*`,`*mode*`)`**

- The *mode* is a string that designates what we want to do.
  - **`"w"`** creates a new file onto the disk, or overwrites the file if it already exists. (*WRITE* mode)
  - **`"a"`** sets it to write onto the end of an existing file. (*APPEND* mode)
  - **`"r"`** opens an existing file on the disk, and puts as at the start of the file so we can read its contents. (*READ* mode)
  - and also **`"r+"`** (*READ/WRITE* mode)

READING A TEXT FILE

- Consider, now, this script:

```
f = open("testing.txt","r")
first_line = f.readline()
print("Here is what is in the file:")
print(first_line)
print("And so that's what is in the file.”)
f.close()
```

- Here is the output

```
$ python3 sample_read.py
Here is what is in the file:
let's write wordsinto a file

And so that's what is in the file.
$
```

READING A TEXT FILE

- Consider a similar script that instead uses **`repr`**:

```
f = open("testing.txt","r")
first_line = f.readline()
print("Here is what is in the file:")
print(repr(first_line))
print("And so that's what is in the file.”)
f.close()
```

- Here is the output

```
$ python3 sample_read.py
Here is what is in the file:
“let's write wordsinto a file\n”
And so that's what is in the file.
$
```

READING A TEXT FILE, TWO LINES

```
f = open("testing.txt","r")
first_line = f.readline()
second_line = f.readline()
print("Here is what is in the file:")
print(repr(first_line))
print(repr(second_line))
print("And so that's what is in the file.")
f.close()
```

- Here is the output

```
$ python3 sample_read.py
Here is what is in the file:
"let's write wordsinto a file\n”
'and see whathappens\n'
And so that's what is in the file.
$
```

READING A TEXT FILE, TWO LINES

```
f = open("testing.txt","r")
first_line = f.readline()
second_line = f.readline()
print("Here is what is in the file:")
print(repr(first_line))
print(repr(second_line))
print("And so that's what is in the file.")
f.close()
```

- After the first **`readline`**, Python remembers its place within the file.
- A subsequent **`readline`** gives back a string that starts with characters that follow that first line.

READING ALL THE LINES

- If you perform a series of **`readline`** operations, these will grab a series of lines from the file in order.
- If you have read the last line, a subsequent **`readline`** will give back the empty string.

- The code below reads & prints all the lines in a file, then stops:

```
f = open("testing.txt","r")
print("Here are the contents of the file:")
line = f.readline()
while line != '':
    print(repr(line))
    line = f.readline()
print("And so those are the contents.")
f.close()
```

USING READLINES

- There are other ways to read the full file. You can use **`readlines`**:

```
f = open("testing.txt","r")
lines = f.readlines()
f.close()
print("Here are the contents of the file:")
print(lines)
print("And so those are the contents.")
```

- This gets the file contents as a list of strings, one for each line:

```
$ python3 sample_read.py
Here is what is in the file:
["let's write wordsinto a file\n", 'and see whathappens\n']
And so those are the contents.
$
```

USING READ

- There are other ways to read the full file. You can use **`readlines`**:

```
f = open("testing.txt","r")
everything = f.read()
f.close()
print("Here are the contents of the file:")
print(repr(everything))
print("And so those are the contents.")
```

- This gets the file contents as one (long) string:

```
$ python3 sample_read.py
Here is what is in the file:
"let's write wordsinto a file\nand see whathappens\n"
And so those are the contents.
$
```

USING READ THEN SPLIT

- There are other ways to read the full file. You can use **`readlines`**:

```
f = open("testing.txt","r")
everything = f.read()
f.close()
print("Here are the contents of the file:")
print(everything.split('\n'))
print("And so those are the contents.")
```

- Doing this makes a list of all the lines.

```
$ python3 sample_read.py
Here is what is in the file:
["let's write wordsinto a file\n", 'and see whathappens\n', '']
And so those are the contents.
$
```

USING FOR

- There are other ways to read the full file. You can use **`for`**:

```
f = open("testing.txt","r")
print("Here are the contents of the file:")
for line in f:
    print(repr(line))
print("And so those are the contents.")
f.close()
```

- This scans through the lines, one by one.

```
$ python3 sample_read.py
Here is what is in the file:
"let's write wordsinto a file\n"
'and see whathappens\n'
And so those are the contents.
$
```

USING SEEK AND TELL

- It's possible to skip around with **`seek`** and to **`read`** a limited amount:

```
f = open("testing.txt", "r")
print(f.tell())
f.seek(3)
print(f.read(6))
print(f.tell())
f.seek(7)
print(f.tell())
print(f.read(3))
print(f.tell())
```

- The **`tell`** method can be used to determine where we're at.

```
$ python3 sample_seek.py
0
's wri
9
7
rit
10
$
```

BINARY FILES

- We can also write to "binary files" instead of text files.
  - ➡ These aren't human-readable and contain "raw" ("machine") data.
  - ➡ Files are a sequence of bytes, rather than characters.
  - Image, audio, or video data; numeric data; compiled machine code; etc.
  - Need to know about Python's binary strings.

- Using `open`, we instead designate binary file modes like so:
  - ➡ `"wb"` for writing into a binary file
  - ➡ `"rb"` for reading from a binary file
  - ➡ `"ab"` for appending to the end of a binary file

- `pickle` is a commonly-used library for saving Python objects to be read later
  - ➡ has its own format for representing program objects; uses binary files