Files and Errors

Outline

About Files

- What is a file?
  - A chunk of data stored on the hard disk
- Why do we need files?
  - Disks persist state across program runs
  - When a program is running, all its variables are in RAM
  - RAM is faster, but doesn't persist through restarts
    - Or when the program exits

About Files and Paths

- Files are arranged in a tree of *folders*
  - On my computer, e.g., this slideshow is at this *path*:
  - `/home/jcoa2018/CS050/week4/lec4-2.slides.pdf`
  - "The file `lec4-2.slides.pdf` in the folder `week4` of the folder `050` ..."
- The topmost folder is called "root" and written `/`
  - On Windows: `C:\` or whatever disk it's on
- Paths can be complete (*absolute*) paths from the root...
  - Or incomplete (*relative*) paths with respect to "the current directory"
  - Typically "where the program is running from"

Reading Files

We use the `open(filename, mode)` function to open files in Python.
We'll talk about the *read mode* `"r"` first:

```
file = open("assignment1.py", "r")
```

If we have a file open in read mode, we can read to it but not write to it.
Quick question: Is this path *absolute* or *relative*?

Files are Iterable

- Like strings and tuples, files support `for ... in file`
  - Other sequence operations like + and `x in file` don't work though

```
file = open("assignment1.py", "r")
for line in file:
    print(line)
```

`line` will include the ending "newline" or "carriage return" character, so you may want to use `line.rstrip()` to remove that.

Caveat

File objects, unlike strings, have a *current position*. So this code won't do what you expect:

```
file = open("assignment1.py", "r")
print("Print it once...")
for line in file:
    print(line)

print("Print it again...")
for line in file:
    print(line)
```

The first loop *uses up* the file.

File Read Methods

We have four key methods for reading files:

- `file.read()`
- `file.readline()`
- `file.seek(position)`
- `file.tell()`

Read and Readline

- `read()` reads as much of the file as possible into a string, which it returns.
  - You can give a parameter to read if you only want a few characters: `file.read(8)` will read at most 8 characters.
- `readline()` is a convenience over read that reads up to a newline character (`"\n"`).
- Calls to these functions push the current position forward by the number of characters read.

Exercise

Imagine `file.txt` is as follows:

```
this is my cool file
it has more lines than we'll read
it is a haiku
```

What will the values of `first_five`, `rest_of_line`, and `next_line` be after this code executes? What will the position of the `file` be (you don't need to count the character number for this exercise, just be able to point to where it ends up)?

```
file = open("file.txt", "r")
first_five = file.read(5)
rest_of_line = file.readline()
next_line = file.readline()
```

Exercise: Seek and Tell

We can move the file cursor without reading anything:

```
file = open("file.txt", "r")
first_five = file.read(5)
file.seek(0)
first_line = file.readline()
pos = file.tell()
print(f"ended up at {pos}")
```

What will the contents of `first_line` be?

Where will the file cursor be?

Closing Files

There's some coordination between your program and operating system here.

- Which files are open?
- Where are we in each file?
- What if another program modifies or deletes the file while we read it?
- Etc

Closing Files

This coordination has some overhead, so operating systems limit the number of open files and track which programs are reading which files.

When you are done with a file, close it:

```
file = open("test.txt", "r")
# do some reads...
file.close()
# can't read file anymore, all done!
```

Not closing a file properly is a bug, so remember to close it!

Remember to Close Files

- Anything that you need to *remember* to do is a bug waiting to happen.
  - Also: If you open a file in a function, you should close it before returning
  - Also also: If your function has an error and exits, you still need to close the file!
  - It's hard to keep track of this stuff
- One virtue for programmers is *laziness*:
  - Offload work for later
  - Do as little as possible
  - Automate, don't remember

Automatic Closing

Many scarce resources like files exist in programming: network or database connections, large allocations of memory, etc. Python gives us a tool for such cases.

```
with open("file.txt", "r") as file:
    first_line = file.readline()
    for line in file:
        print(line)
# file closes automatically here
print("All done!")
```

Python files and other scarce resources provide "context managers" for use with the `with` statement. In this class we'll really only use `with` for files.

File Formats

- What is a file format?
  - A set of rules we expect when reading the file
  - Sometimes, a special file extension in the path (e.g., `.py`)
- Text vs binary
  - We'll work with text files only in this class
  - But files can contain raw binary data too
    - The `bytes` type can be helpful here
- Common formats: .html, .pdf, .txt, .docx, .png, ...
  - text, binary, text, text (XML), binary, ...

File Modes

We've seen how to read files already–we pass `"r"` as the second argument of `open()`.

If we want to write a file, we use `"w"` instead.

Text Encodings

- Before we proceed: even text files are binary data!
  - Python uses a *text encoding* scheme to convert between characters and bytes.
  - By default it uses your "system encoding"
- It's a good idea to be explicit about the encoding used.
  - In almost all cases we want UTF-8

```
file = open("test.txt", "r", encoding="utf-8")
```

New syntax alert! We call `encoding` a *keyword argument*.

Writing Text

The primary method we need for writing text files is `write()`.

In fact, we can't call `read()` on a file open for writing.

```
# the open call actually **erases** output.txt!
with open("file.txt", "w", encoding="utf-8") as file:
    file.write("this is my cool file\n")
    # note the '\n's here
    file.write("it is also a haiku\n")
    file.write("poetry is fun")
```

`write()`, like `read()`, moves the file position.

Reading and Writing

What if we just want to replace a single line of the file?

We could read the whole thing with one `open`, then write it out with a second `open`–but that sounds annoying.

Instead, we can use the `"r+"` mode to open a file we can both read from and write to.

Be careful with this though—if your new line is shorter than the old line, some of the old line will remain; and if it's longer, it will run over into the following line!

Finally, if you have made the overall file shorter in the process of reading and writing, you'll want to use `file.truncate(length)` to eliminate text beyond the new "end" of the file.

Errors

Python programs can go wrong in a bunch of ways:

- `1 + 'hi'`
- ```
  def f(x):
      return x + x
  f()
  ```
- Writing to a read-only file
- Reading from a closed file
- Opening a file that doesn't exist
  ```

Exceptions

Python errors indicate exceptional circumstances—errors that Python can't resolve on its own.

Sometimes, howver, we as programmers have enough information to do something useful with the error. We can use the `try...except` syntax for this.

```
try:
    with open("file.txt", "r") as file:
        for line in file:
            print(line)
except:
    print("Couldn't open file!")
```

Exceptions

Sometimes we need to clean up whether or not an error occurred:

```
file = open("file.txt", "r")
try:
    do_something_with(file)
except:
    print("Failed to do something with file")
finally:
    file.close()
```

This is more or less how the `with` statement works.

Exception Types

There are many different kinds of exceptions that can occur, and we can match on the type of error to decide how to handle it:

```
x = "a string"
try:
    if x > 50:
        y = "hello"
    print(y)
except NameError:
    print("Variable y was not defined")
except:
    print("Some other issue")
```

Quiz