Files

adapted from material by Mike Scott and Bill Young at the University of Texas at Austin

Everything we've programmed so far will go away if you restart your computer.

If you want data to persist past a reboot, you need to store it in a **file**.

Files are stored in a memory known as the **file system**, which is typically arranged into a hierarchy of **directories.**

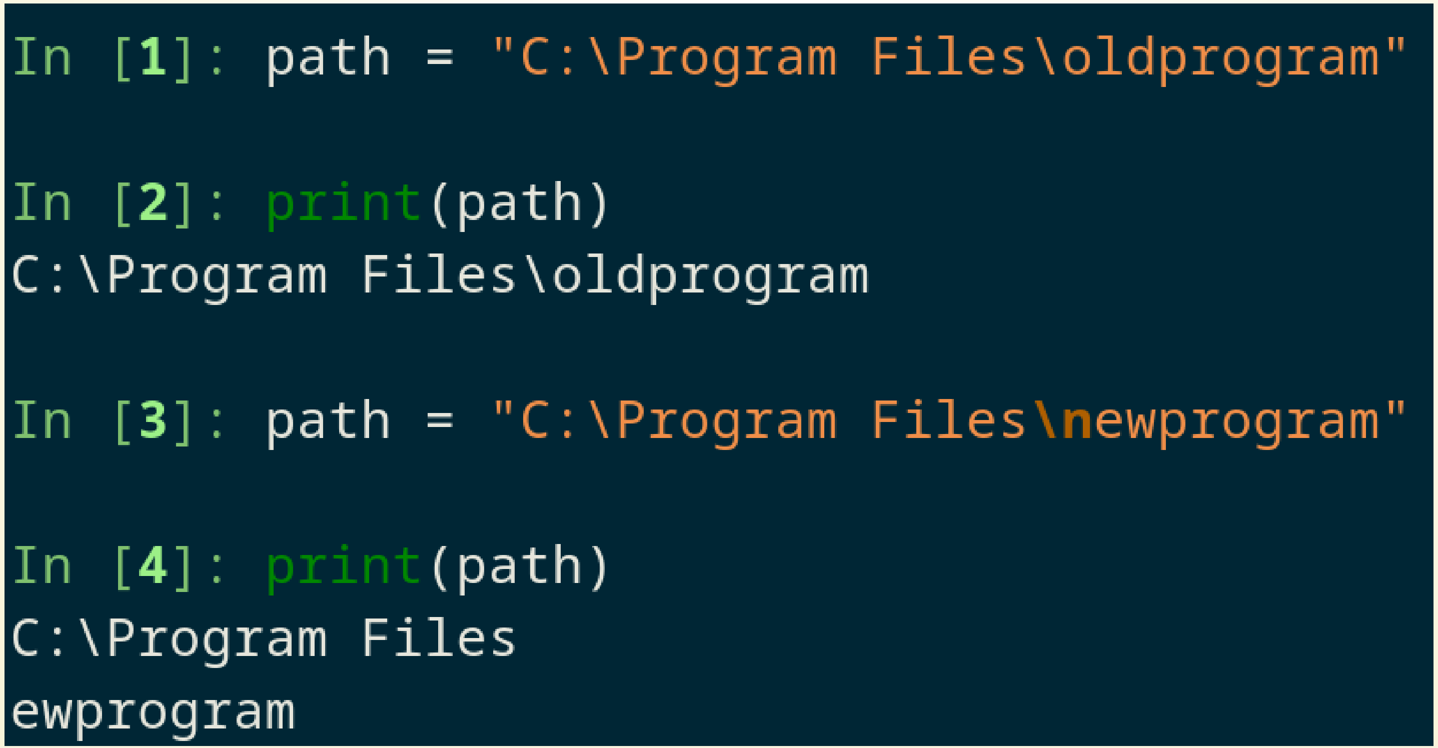
The **path** to a particular file details where the file is stored within the hierarchy.

Path format is determined by the OS.

Windows: C:\Users\user\Documents\program.py

MacOS: /Users/user/Documents/program.py

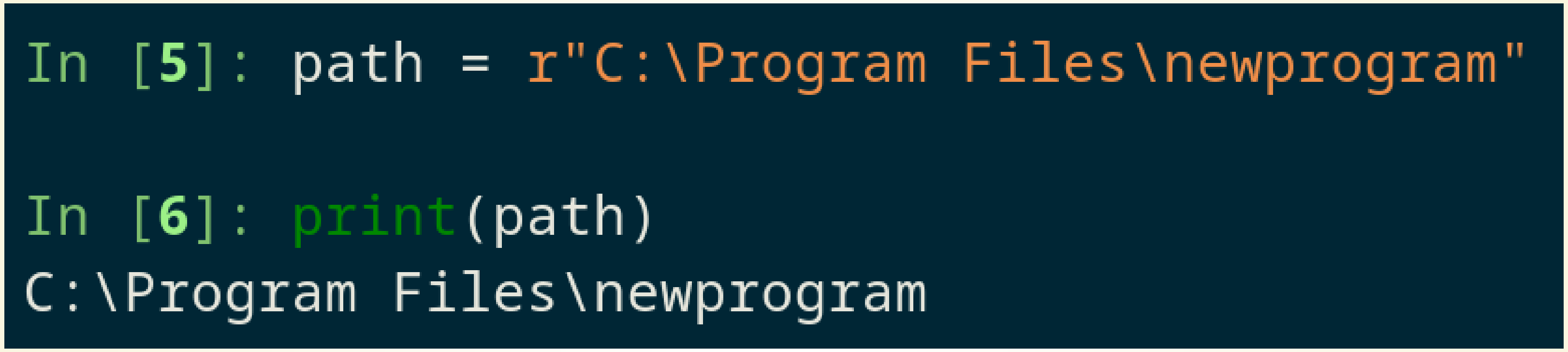
Note: Python uses backslash as an escape,
so this causes problems on Windows!



```
In [1]: path = "C:\Program Files\oldprogram"

In [2]: print(path)
C:\Program Files\oldprogram

In [3]: path = "C:\Program Files\newprogram"

In [4]: print(path)
C:\Program Files
ewprogram
```

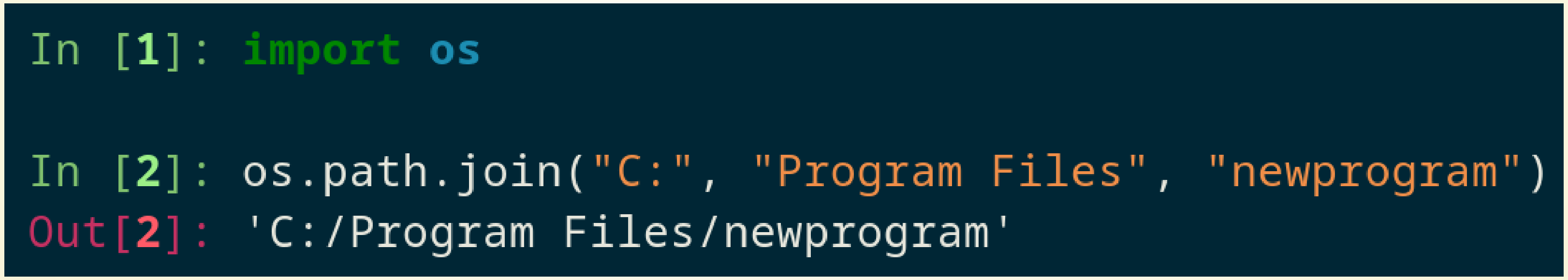
Solutions

- When inputting string literals, use a *raw string literal*.



```
In [5]: path = r"C:\Program Files\newprogram"

In [6]: print(path)
C:\Program Files\newprogram
```

- When joining paths, use os.path.join() instead of string append

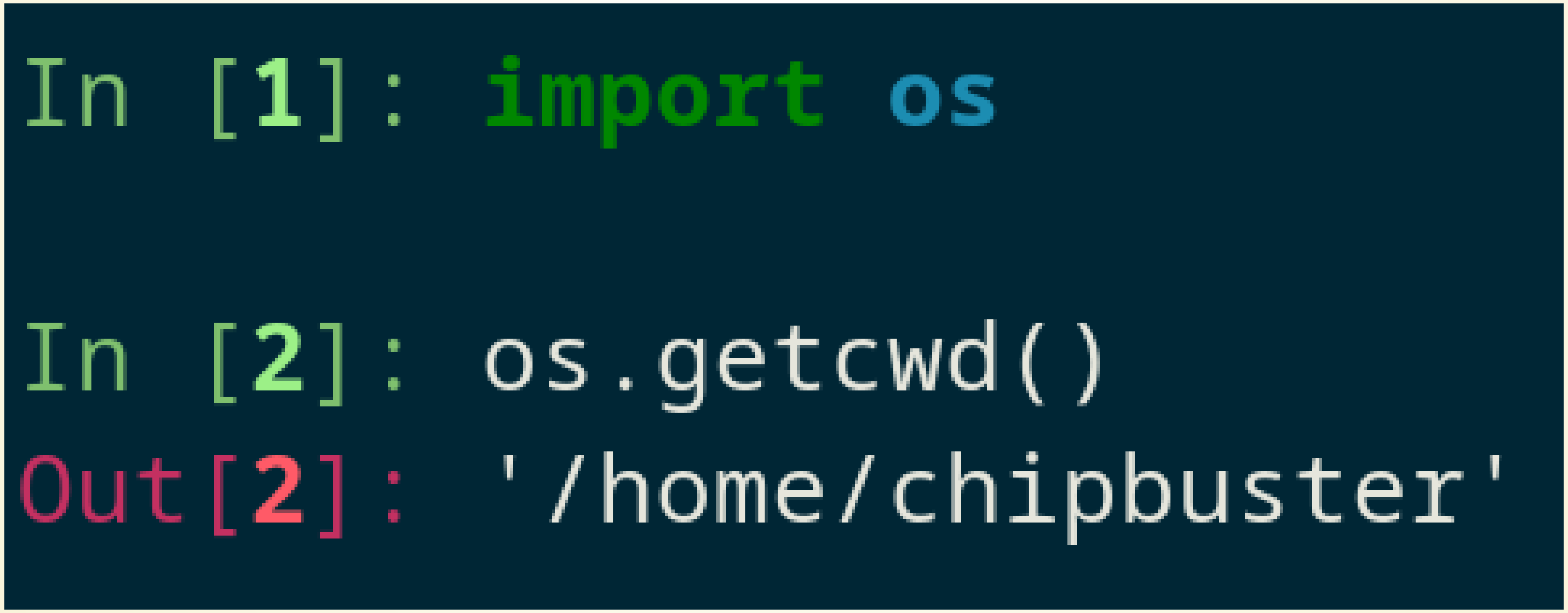


```
In [1]: import os

In [2]: os.path.join("C:", "Program Files", "newprogram")
Out[2]: 'C:/Program Files/newprogram'
```

A path can be *relative* or *absolute*.

If a path is relative, it is assumed to be relative to your *current working directory.*

You can find your cwd with the os module



```
In [1]: import os


In [2]: os.getcwd()
Out[2]: '/home/chipbuster'
```

Of course, your output will vary.

On systems, paths can be *relative* or *absolute*.

Again, what this looks like depends on the operating system. For absolute paths...

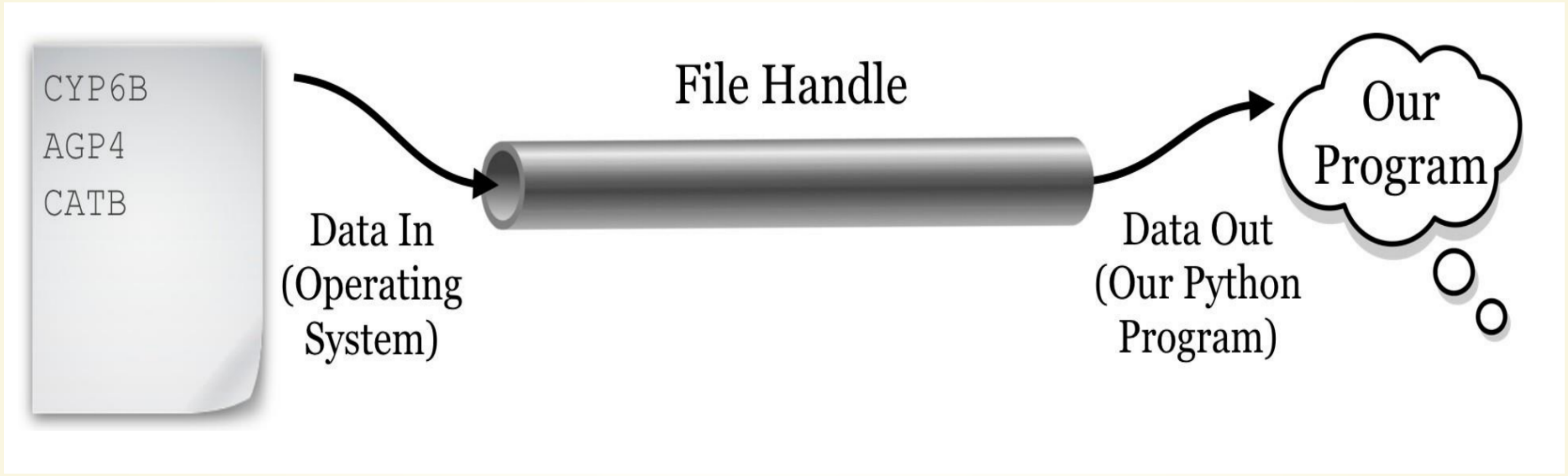
Windows: paths starts with a drive letter (e.g "C:\", "G:\"), or is a fabled UNC path (we won't discuss these much in this class)

MacOS: path starts from the *root directory* (i.e. has a "/" character at the front).

	Absolute	**Relative**
Windows	C:\Users\chip\files	chip\files
MacOS	/home/chip/files	chip/files

Interacting with Files

Python lets you interact with files via what are called "handles." You can think of these as hoses attached to the data source.



You also specify how you intend to interact with the file: reading, writing, and other options. These are known as the "file mode".

```python
infile = open("input_data.txt", "r")
contents = infile.read()

outfile = open("output_data.txt","w")
outfile.write(contents)
outfile.write("\n")
outfile.write("This is what I read from the input file."

infile.close()
outfile.close()
```

Things to note:

- We are calling a method on a variable. This behaves a little like calling a normal function, but with a different way of writing it (e.g. outfile.write(msg) is a little like write(outfile, msg)).
- write() returns a number. What do you think it is?

File Modes

Mode	**Description**
'r'	Open for reading
'w'	Open for reading. Delete any existing contents.
'a'	Open to append data at the end of the file.
'rb'	Open for reading binary data
'wb'	Open for writing binary data

You also have to have necessary permissions from the operating system.

We won't use the binary modes in this class.

Closing the file

You should *always* close a file when you're done with it.

File operations are often buffered for speed reasons (demo). These won't be visible until you close the file.

It's always a good idea to clean up after yourself.

This being said, the operating system will close all files for you once your program exits. It just might not do it correctly.

Opening with 'with'

Another python keyword. We can use 'with' to make sure the file gets automatically closed at the end of a block.

```
def demo_with(file_name):
  with open(file_name, "r") as in_file:
    print("File contents are: ")
    for line in in_file:
      print(line)
    print("Within with block. Is file closed?", in_file.close

  print("After with block. Is file closed?", in_file.closed)
```

Operations on Files

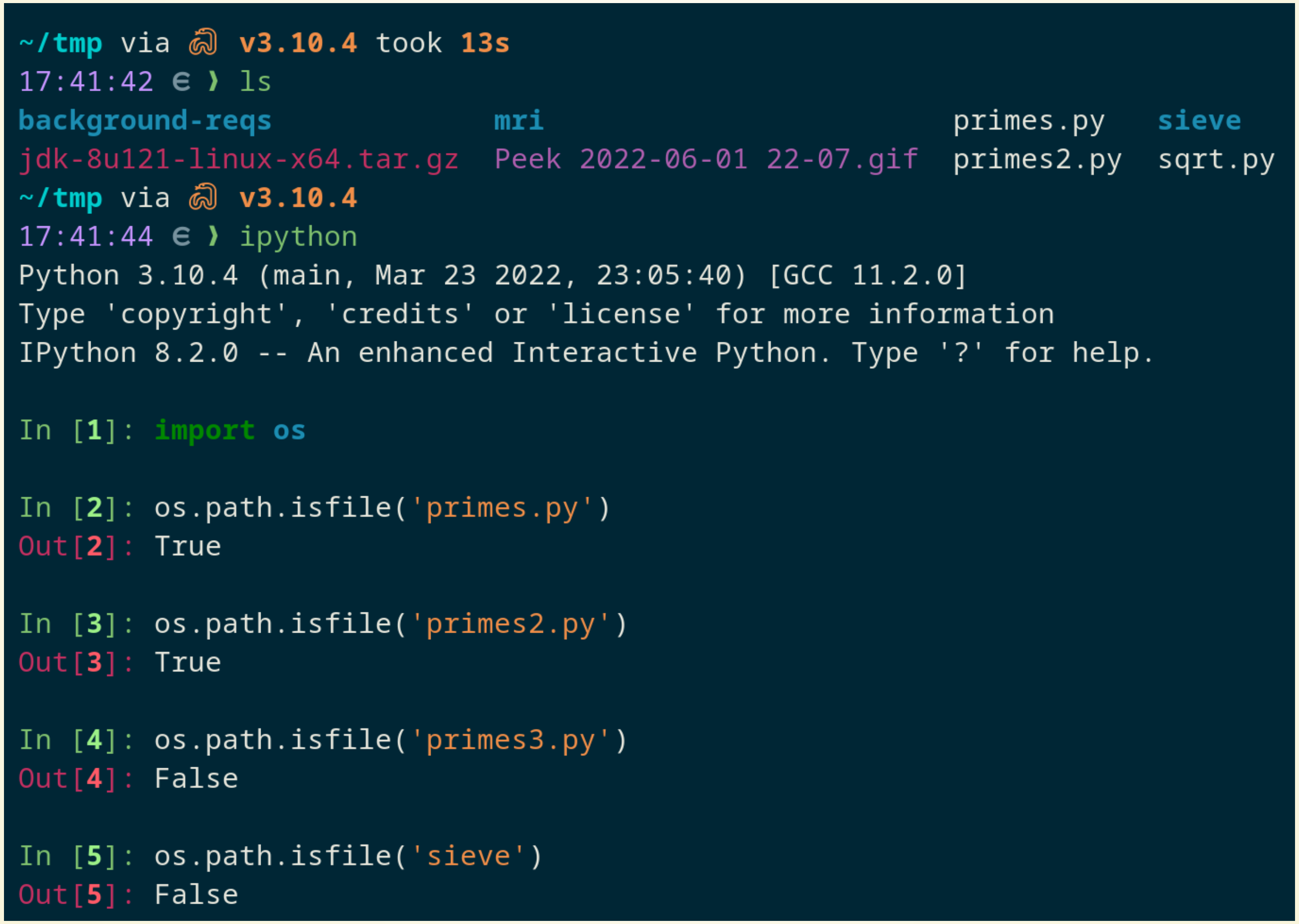
Various functions in python can be used to read/write data from file. These advance the internal *file pointer* (like a cursor in Word).

open() sets the file pointer to the start of the file.

(vim demo)

Function	**Description**
f.read()	Read to end of file.
f.read(k)	Read next k characters in the file
f.readline()	Read until the end of the current line.
f.readlines()	Read all remaining lines in the file as seq.
f.write(str)	Writes string to the file.

Testing File Existence



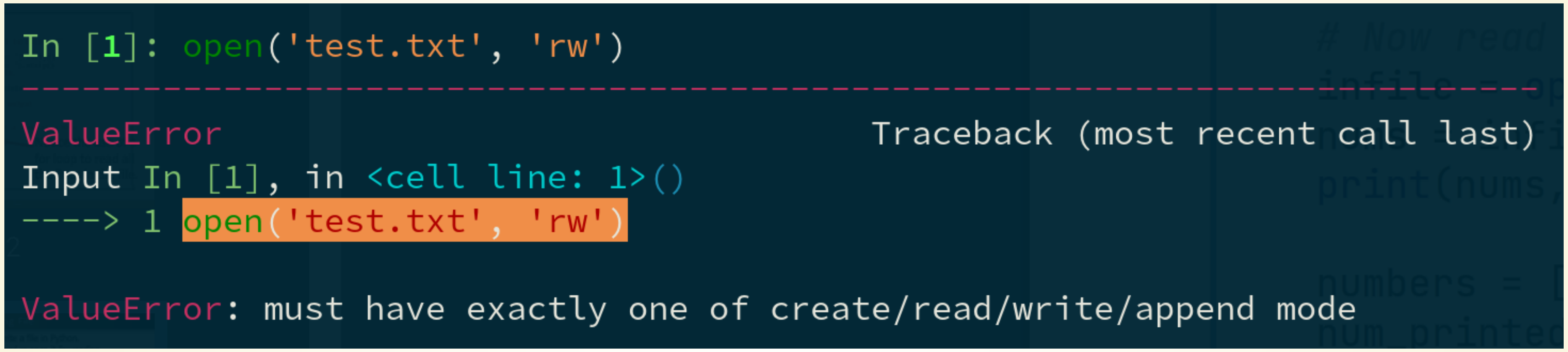
```
~/tmp via  v3.10.4 took 13s
17:41:42 ∈ ❱ ls
background-reqs             mri                        primes.py   sieve
jdk-8u121-linux-x64.tar.gz  Peek 2022-06-01 22-07.gif  primes2.py  sqrt.py
~/tmp via  v3.10.4
17:41:44 ∈ ❱ ipython
Python 3.10.4 (main, Mar 23 2022, 23:05:40) [GCC 11.2.0]
Type 'copyright', 'credits' or 'license' for more information
IPython 8.2.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]: import os

In [2]: os.path.isfile('primes.py')
Out[2]: True

In [3]: os.path.isfile('primes2.py')
Out[3]: True

In [4]: os.path.isfile('primes3.py')
Out[4]: False

In [5]: os.path.isfile('sieve')
Out[5]: False
```

Write a program which reads and numbers lines from a file

Write a program which prints out 10,000 coin flips to a file, with 50 results per line. Use 'H' for heads and 'T' for tails.

Reading and Writing

You **cannot** open a file for reading and writing at the same time.



```
In [1]: open('test.txt', 'rw')
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Input In [1], in <cell line: 1>()
----> 1 open('test.txt', 'rw')

ValueError: must have exactly one of create/read/write/append mode
```

However, you *can* open the file, write to it, close it, then re-open it for reading.