## Persistence: working with text files

- so far our programs have been independent from rest
- they started with a blank slate of memory, and worked with input provided during that specific run of the program
- all results were lost when program ended
- This is for with calculation type programs
- Other types of programs work with existing data or need to store their results
- Example: Office software: we should be able to resume editing
- Example: Game soft: game state should be saved so that  user can resume their game

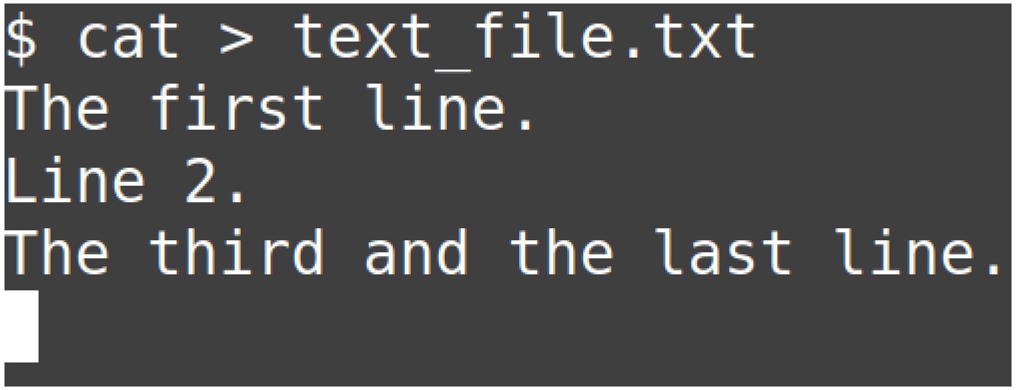
## Persistence: working with text files

- **Many software need ability to move data out of RAM and put in a lasting storage area, i.e. disk**
- **RAM storage is volatile: it disappears when program ends or when computer is turned off**
- **Disk storage is persistent**
- **Many software require ‘data persistence’**
- **In this course, we will focus on storing data as:**
  - local text file
  - local binary file
- **Other options are: storing data in a database, or remotely**

## Text vs binary files

- **Anything on a disk is stored as 0’s and 1’s: how come there are two types: text vs binary?**
- **The difference is the way 0’s 1’s <u>interpreted</u>!**
- **Text files are human readable**

# Reading from text files



```
$ cat > text_file.txt
The first line.
Line 2.
The third and the last line.
```

we can visualize this ond the disk as:

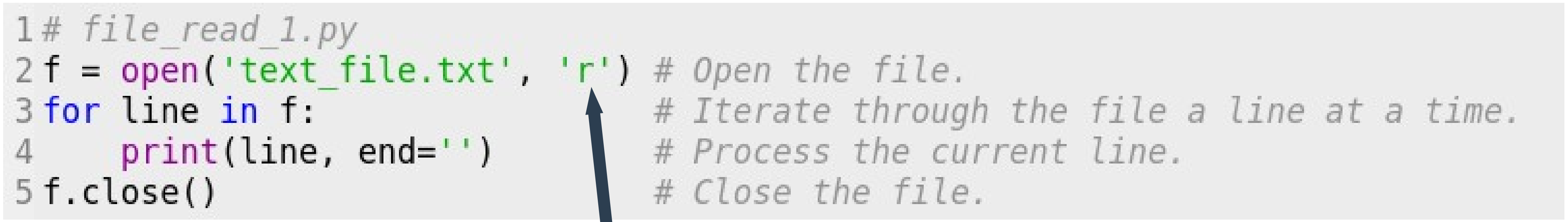
| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| T | h | e | | f | i | r | s | t | | l | i | n | e | . | \n | L | i | n | e | | 2 | . | \n | T | h | e | | t | h | i | r | d | | a | n | d | | l | a | s | t | | l | i | n | e | . | \n | EOF |

The main difference between what we write and how its stored is **\n**
**\n** denotes the **newline** character

There are three common ways of reading in a text file:
1. one line at a time (preferred)
2. the whole file into a string
3. the whole file into a list of strings, one entry per line

Option 1 is preferred since it occupies less memory as one line at a time kept in mem

# Reading from text files:
# 1. One line at a time



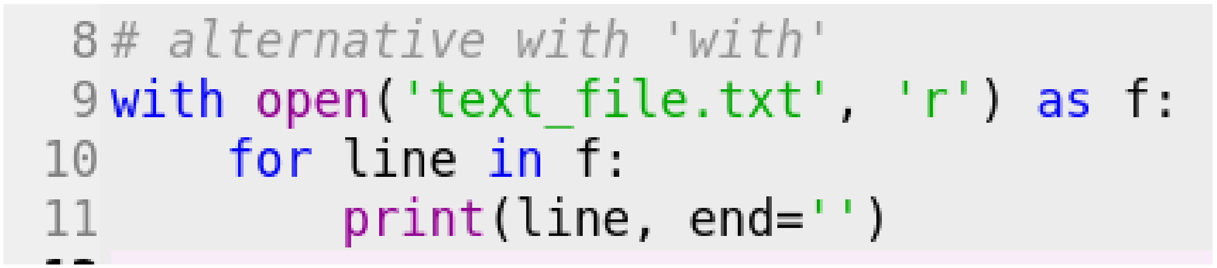
```python
# file_read_1.py
f = open('text_file.txt', 'r') # Open the file.
for line in f:                  # Iterate through the file a line at a time.
    print(line, end='')         # Process the current line.
f.close()                       # Close the file.
```

‘r’ for ‘read’ mode

- file object ‘f’ is iterable: hence can used in **for** statement

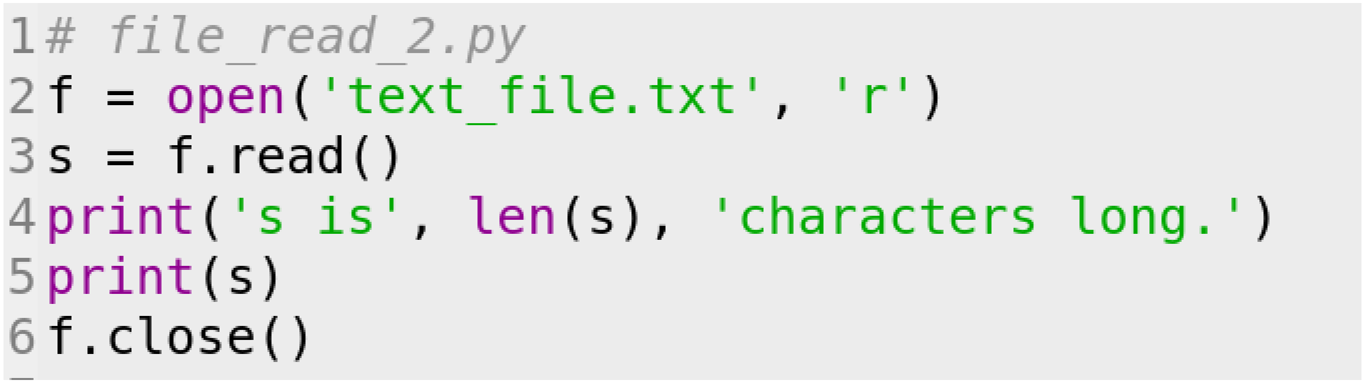
- default iteration is to grab one line at a time

- Always close files via **f.close()** when you’re finished and hence release memory.

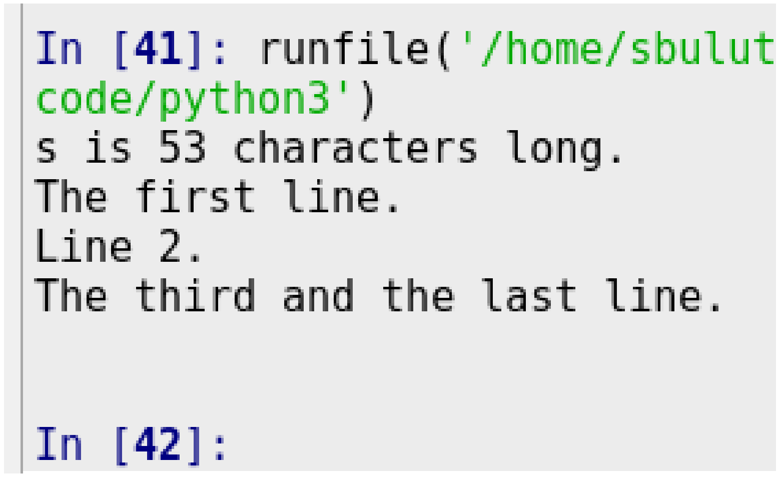


```python
# alternative with 'with'
with open('text_file.txt', 'r') as f:
    for line in f:
        print(line, end='')
```

# Reading from text files:
# 2. The whole file into string



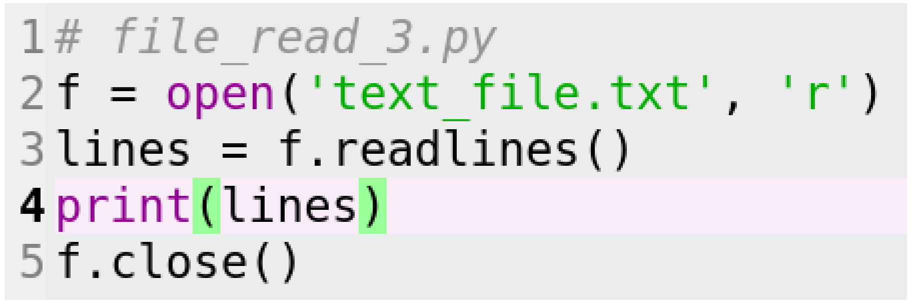
```python
# file_read_2.py
f = open('text_file.txt', 'r')
s = f.read()
print('s is', len(s), 'characters long.')
print(s)
f.close()
```



```
In [41]: runfile('/home/sbulut
code/python3')
s is 53 characters long.
The first line.
Line 2.
The third and the last line.


In [42]:
```

- **read** method reads entire file as string including \n characters
- '\n' characters are interpreted, hence file looks same as reading one line at a time
- f.read(n) reads only n characters

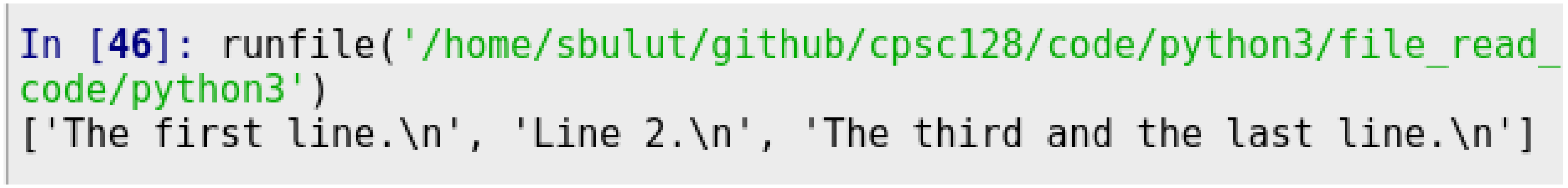
## Reading from text files:
## 3. The whole file into list of strings



```python
# file_read_3.py
f = open('text_file.txt', 'r')
lines = f.readlines()
print(lines)
f.close()
```

Another option is:

lines = list(f)



```
In [46]: runfile('/home/sbulut/github/cpsc128/code/python3/file_read_
code/python3')
['The first line.\n', 'Line 2.\n', 'The third and the last line.\n']
```

Note the newline character. If you don’t want it just strip it!

## Example: search log files

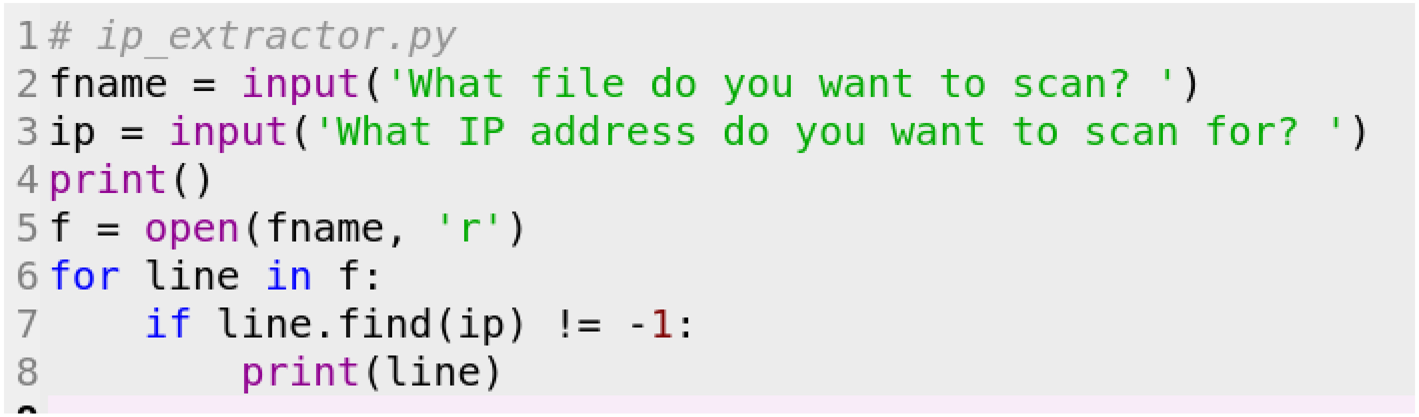
Suppose as part of a security audit we want to display all the lines from our web server log file containing the IP address 199.247.232.110. The file name is **access.log.1** and the first ten lines of the file look like this,

Typically such files are quite large 100’s of Megabytes.
So, read one line at a time!

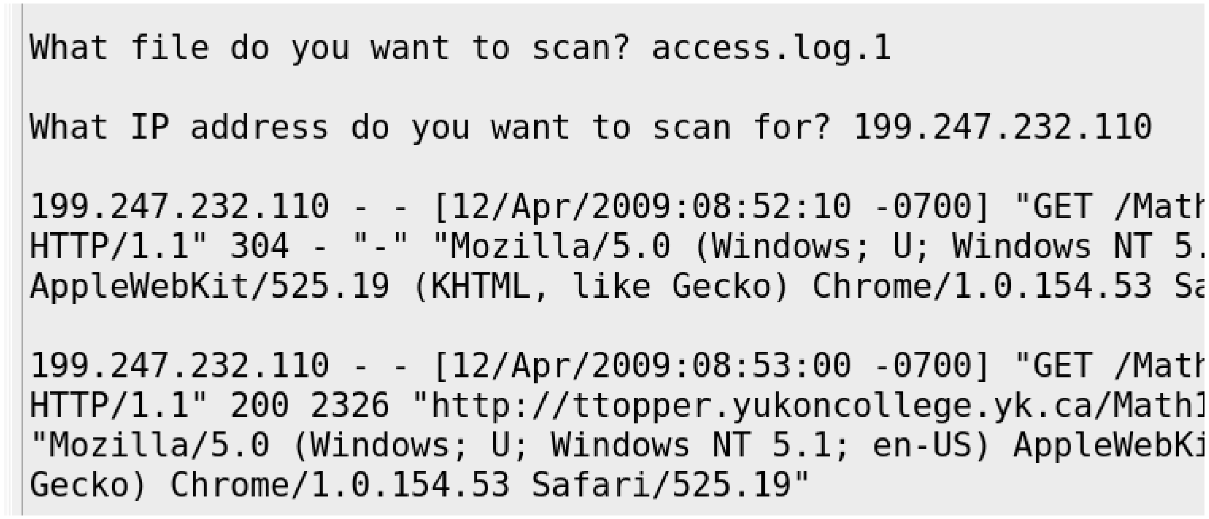
Pseudocode

```
Get the name of the log file
Get the name of the IP address to scan for
Open the file for reading
For each line in the file
    If the line contains the IP address
        Display the line
```

## Example: search log files



```python
# ip_extractor.py
fname = input('What file do you want to scan? ')
ip = input('What IP address do you want to scan for? ')
print()
f = open(fname, 'r')
for line in f:
    if line.find(ip) != -1:
        print(line)
```



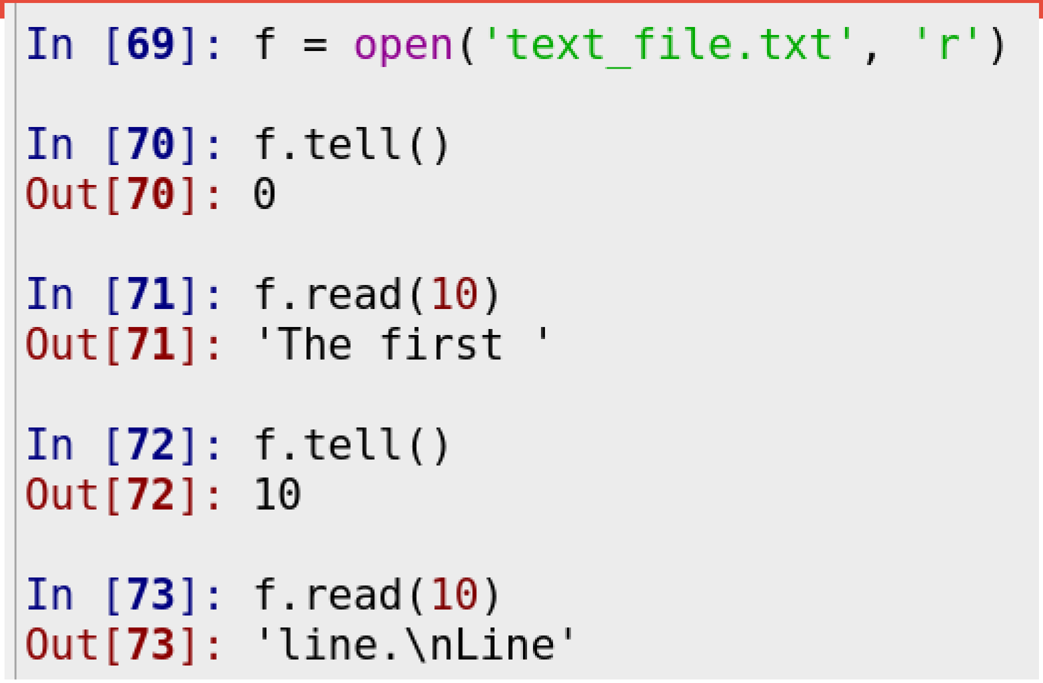
```
What file do you want to scan? access.log.1

What IP address do you want to scan for? 199.247.232.110

199.247.232.110 - - [12/Apr/2009:08:52:10 -0700] "GET /Math
HTTP/1.1" 304 - "-" "Mozilla/5.0 (Windows; U; Windows NT 5.
AppleWebKit/525.19 (KHTML, like Gecko) Chrome/1.0.154.53 Sa

199.247.232.110 - - [12/Apr/2009:08:53:00 -0700] "GET /Math
HTTP/1.1" 200 2326 "http://ttopper.yukoncollege.yk.ca/Math1
"Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US) AppleWebKi
Gecko) Chrome/1.0.154.53 Safari/525.19"
```

## Files are sequential

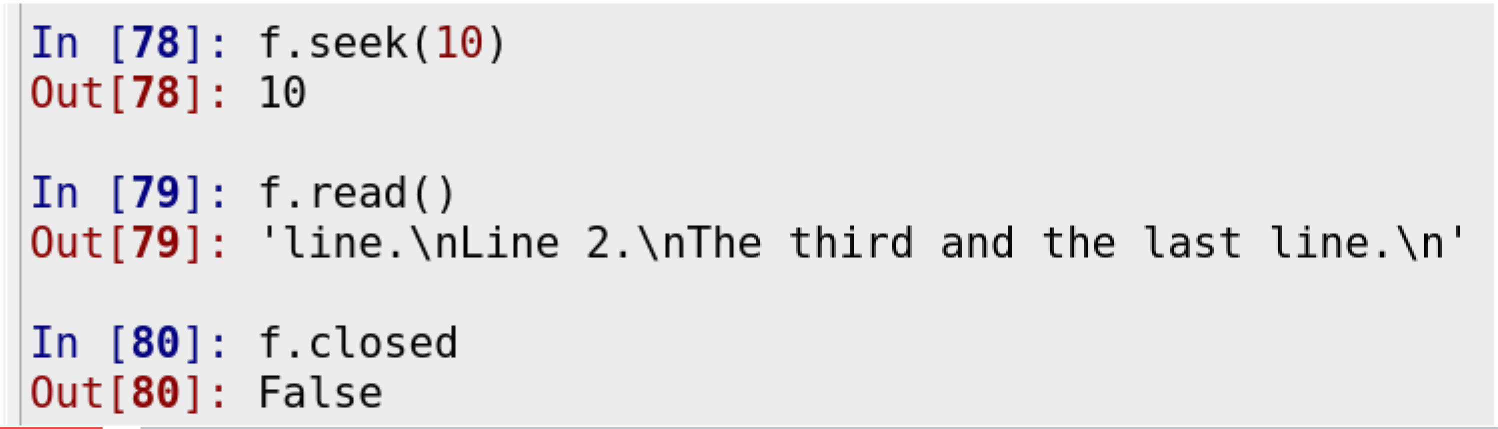
In Python, a file's content are read in sequentially similar to the physical tapes.

```
>>> f = open('text_file.txt','r')
>>> s = f.read()
>>> s
'The first line.\nLine 2.\nThe third and last line.\n'
>>> p = f.read()
>>> p
''
```

## Files are sequential: other methods



```
In [69]: f = open('text_file.txt', 'r')

In [70]: f.tell()
Out[70]: 0

In [71]: f.read(10)
Out[71]: 'The first '

In [72]: f.tell()
Out[72]: 10

In [73]: f.read(10)
Out[73]: 'line.\nLine'
```



```
In [78]: f.seek(10)
Out[78]: 10

In [79]: f.read()
Out[79]: 'line.\nLine 2.\nThe third and the last line.\n'

In [80]: f.closed
Out[80]: False
```

## Writing to text files

Suppose we need to store a list of coordinates to a text file

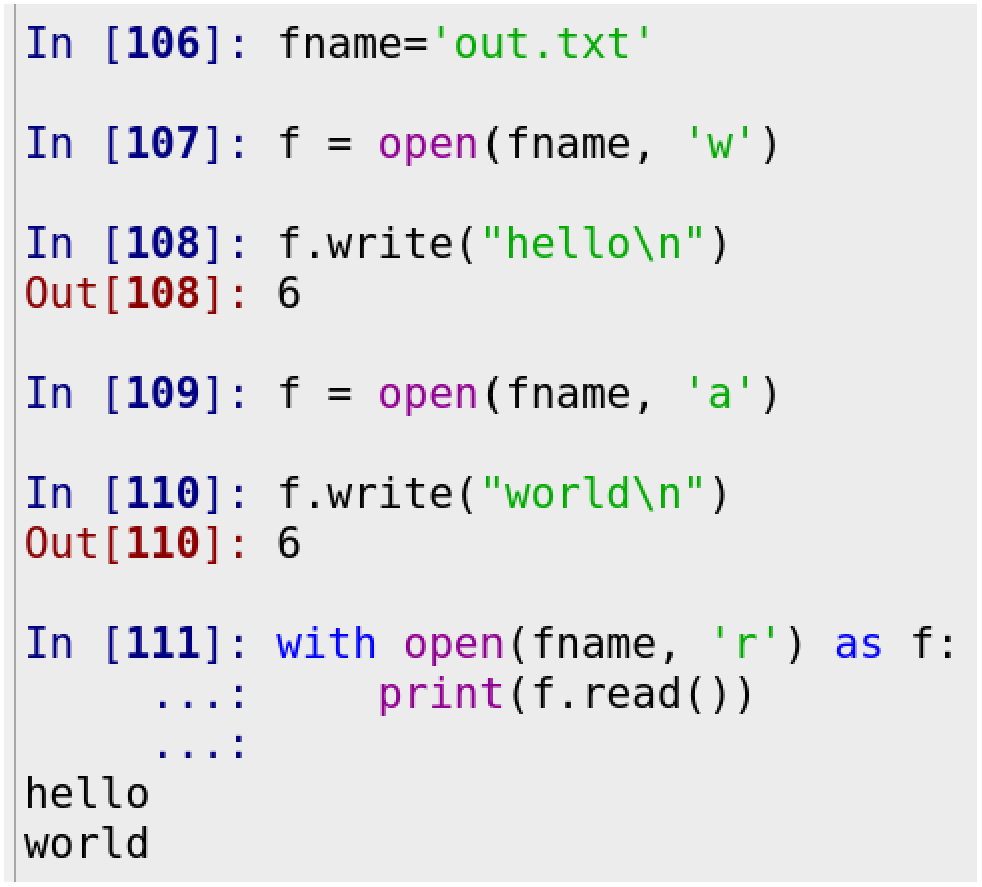
```
coords = [[12, 31], [75, 19], [28, 51]]
```

```
# write_coords.py
coords = [[12, 31], [75, 19], [28, 51]]
fname = input('Name of file to create? ')
f = open(fname, 'w')
for coord in coords:
    f.write(str(coord[0])+' '+str(coord[1])+'\n')
f.close()
```

- Note ‘w’ for ‘write’: we open file in write mode
- Opening a file in write mode deletes a file if it already exists!
- We direct output to file using **write** method
- **write** accepts **str** type as argument
- Hence convert the list to string, and also add **\n**
- remember to close the file

# Appending to files

- What if we want to append to a file?
- We can’t use **write** since it destroys a file
- So use ‘a’ (append) mode instead!

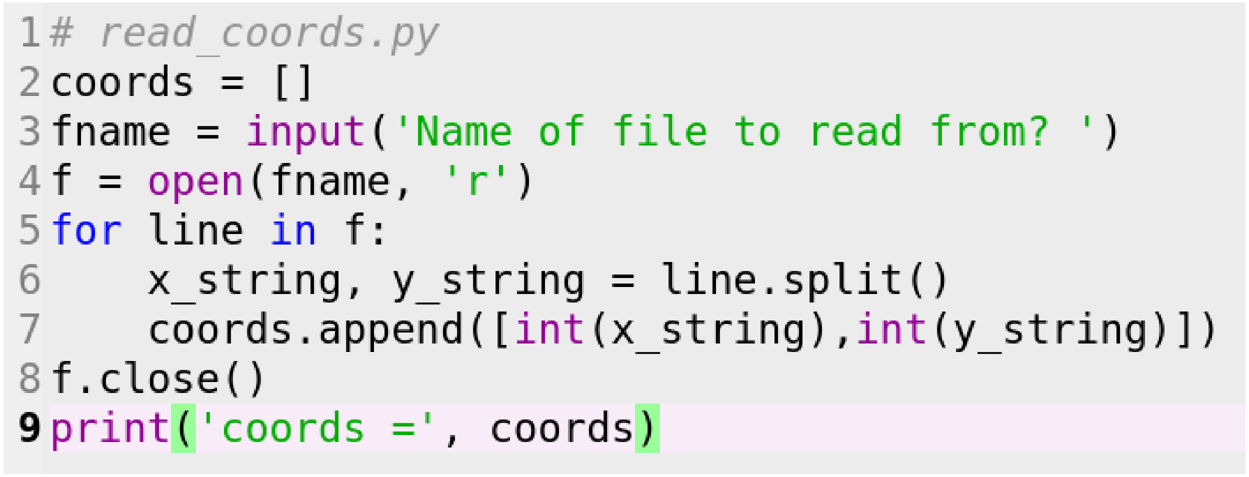


```
In [106]: fname='out.txt'

In [107]: f = open(fname, 'w')

In [108]: f.write("hello\n")
Out[108]: 6

In [109]: f = open(fname, 'a')

In [110]: f.write("world\n")
Out[110]: 6

In [111]: with open(fname, 'r') as f:
     ...:     print(f.read())
     ...:
hello
world
```

## Reading in numerical data

```
Open the file
Initialize coords to an empty list
Read it a line at a time
    Split the line into parts at blanks
    Convert each part into an integer value
    Append the integer values to the list coords
```

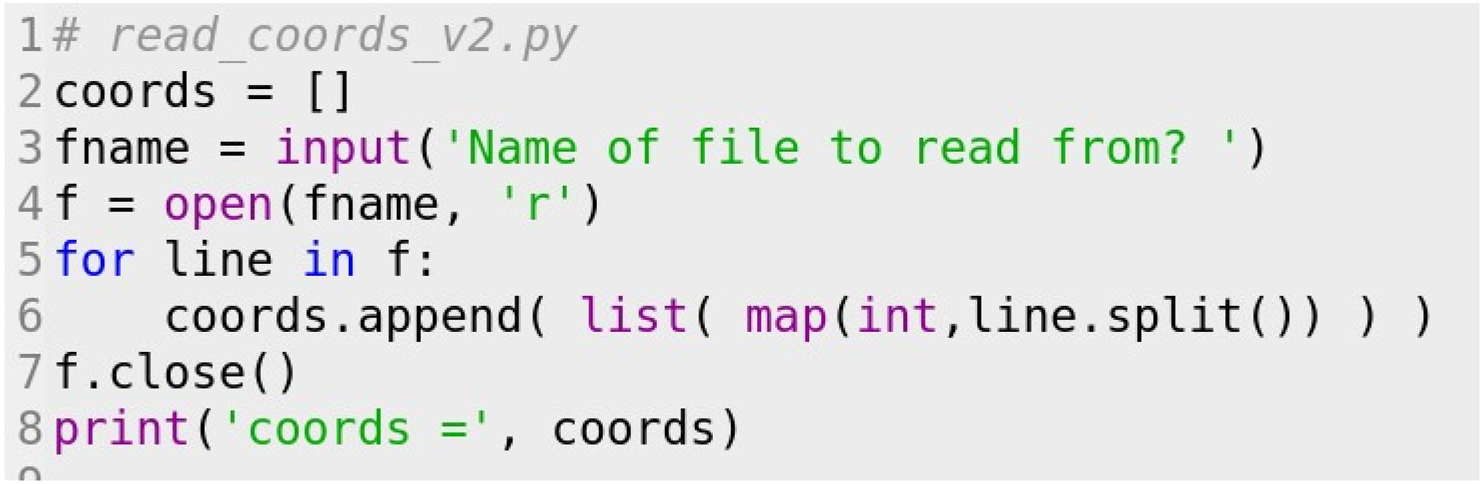
## Reading in numerical data



```python
# read_coords.py
coords = []
fname = input('Name of file to read from? ')
f = open(fname, 'r')
for line in f:
    x_string, y_string = line.split()
    coords.append([int(x_string),int(y_string)])
f.close()
print('coords =', coords)
```

Note the multiple assignment!

Aside:



```python
# read_coords_v2.py
coords = []
fname = input('Name of file to read from? ')
f = open(fname, 'r')
for line in f:
    coords.append( list( map(int,line.split()) ) )
f.close()
print('coords =', coords)
```

## Designing File Formats

- In the previous example, we stored coordinates to: one pair per line separated by single space
- instead, we could have put all in one line and separate items by commas
- one line vs many lines, spaces vs commas etc are in this case cosmetic
- Sometimes different approaches can be quite different in nature

## Conway’s game of life!

A simple, fun problem that will show us pros/cons of different data storage appr’chs

The universe in Conway's game of life is a grid of cells each of which can be in one of two states: alive or dead. A natural way to represent it in Python would be as a list of lists of cells (like a very large tic-tac-toe board):

```
universe = [ [0, 0, 0, 0, 0, 0, 0, 0],
             [0, 0, 0, 0, 0, 0, 0, 0],
             [0, 0, 0, 1, 0, 0, 0, 0],
             [0, 0, 0, 0, 1, 0, 0, 0],
             [0, 0, 1, 1, 1, 0, 0, 0],
             [0, 0, 0, 0, 0, 0, 0, 0],
             [0, 0, 0, 0, 0, 0, 0, 0],
             [0, 0, 0, 0, 0, 0, 0, 0]
           ]
```

How can we put this in a text file?

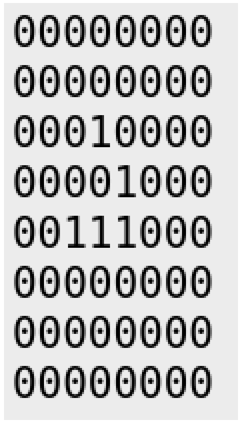
## Storing universe in Conway's game of life Option 1:

- space separated 0's 1's
- storing each character costs 1 byte (8bits)
- 64 0/1 characters = 64 bytes
- 64 spaces to separate them = 64 bytes
- 8 new line character = 8 bytes
- total = 64 + 64 + 8 = 136 bytes!
- for n x n system: **2n**2 + n bytes**!

```
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 1 0 0 0 0
0 0 0 0 1 0 0 0
0 0 1 1 1 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
```

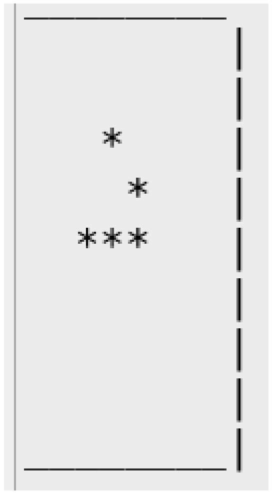
## Storing universe in Conway's game of life Option 2:

- get rid of spaces
- 8x8 bytes + 8 '\n' = 72 bytes
- general case: **n x n +n bytes**



```
00000000
00000000
00010000
00001000
00111000
00000000
00000000
00000000
```

use blanks as dead cells, '*' as alive



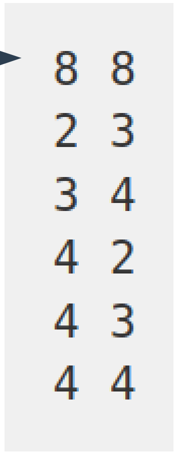
```
________
        |
        |
   *    |
    *   |
  ***   |
        |
        |
        |
________|
```

# Storing universe in Conway's game of life Option 3:

- The universe is usually sparse: relatively few live cells among many dead cells
- A consequence of the rules of the game: if an area is overcrowded, cells die
- Could we wasting space by storing dead cells? (Yes!)
- So just store locations of live cells!

Size of the universe

Coordinates of live cells!



```
8 8
2 3
3 4
4 2
4 3
4 4
```

Storage cost for general case:
4 bytes per live cell (per row)
+ 4 byte (1 row) to store the size of the universe, hence

**4n + 4 bytes!**

It takes 5*4 + 4 = 24 bytes so store this specific state!

## Storing universe in Conway’s game of life Option 4:

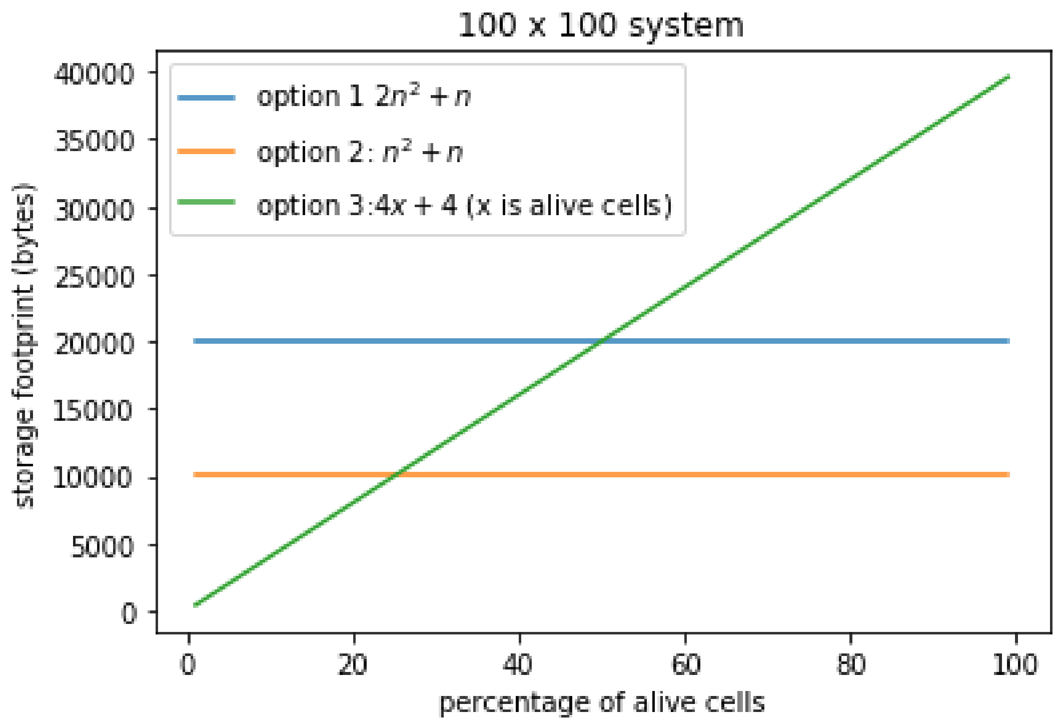
Since 0’s and 1’s can represent dead or alive cells,

store everything in bits as opposed to bytes

Python’s binary operators might enable us to do this.

However this is beyond the context of this course.

# Storing universe in Conway’s game of life Comparing various options



for 25% or less

alive cells,

option 3 is superior

to others

Note: For systems larger than 256x256,
storage requirement per pixel will be bigger.

## Pickling

```
for coord in coords:
    f.write(str(coord[0])+' '+str(coord[1])+'\n')
```

Recall this code?

as straightforward as it is, it seems to be a lot work

can there be a better way?

# Pickling

Enter **pickle** module:
it preserves the object
when writing to disk

- almost any python object can be pickled

- given so convenient, why not always use it?

- pickled representation is not human readable

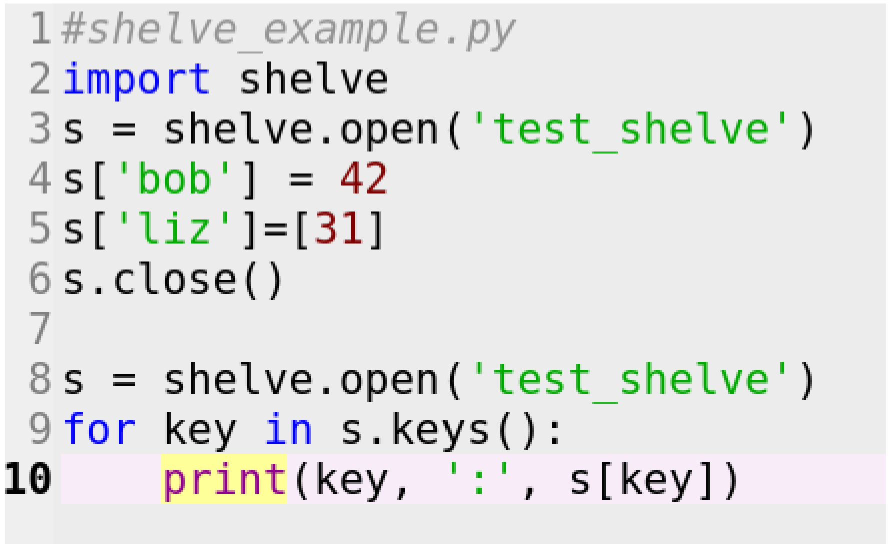
```python
# life.py
universe = [ [0, 0, 0, 0, 0, 0, 0, 0],
             [0, 0, 0, 0, 0, 0, 0, 0],
             [0, 0, 0, 1, 0, 0, 0, 0],
             [0, 0, 0, 0, 1, 0, 0, 0],
             [0, 0, 1, 1, 1, 0, 0, 0],
             [0, 0, 0, 0, 0, 0, 0, 0],
             [0, 0, 0, 0, 0, 0, 0, 0],
             [0, 0, 0, 0, 0, 0, 0, 0]
           ]
import pickle
f = open( 'pickled_universe.pickle', 'wb')
pickle.dump(universe, f)
f.close()
f = open('pickled_universe.pickle', 'rb')
u = pickle.load(f)
f.close()
print(u)
```

## Shelves

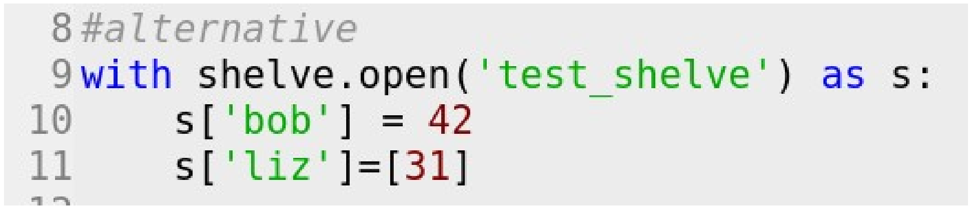
- Pickles can be inefficient with some data types, such as dictionaries
- Dictionary can be pickled, but in order to access a single element,
  - entire dictionary must be read into memory
  - it needs to be unpickled
  - This is a huge waste of time and space
- For this common problem, Python provides **shelve** module

A **shelve** is a dictionary that is efficiently stored in a disk

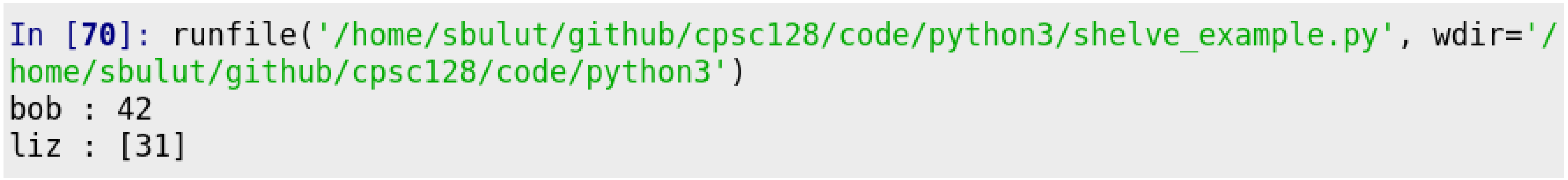
# Shelves



```python
#shelve_example.py
import shelve
s = shelve.open('test_shelve')
s['bob'] = 42
s['liz']=[31]
s.close()

s = shelve.open('test_shelve')
for key in s.keys():
    print(key, ':', s[key])
```



```python
#alternative
with shelve.open('test_shelve') as s:
    s['bob'] = 42
    s['liz']=[31]
```



```
In [70]: runfile('/home/sbulut/github/cpsc128/code/python3/shelve_example.py', wdir='/
home/sbulut/github/cpsc128/code/python3')
bob : 42
liz : [31]
```

## Gotcha: Shelves update on assignment not mutation!

```
bob : 42
liz : [31]
>>> s['bob'] = 43
>>> s['liz'][0] = 30
>>> s.close()
>>> s = shelve.open('test_shelve')
>>> for key in s.keys():
print key, ':', s[key]

bob : 43
liz : [31]
```

## Gotcha: Shelves update on assignment not mutation! Solution 1

```
>>> s = shelve.open('test_shelve', writeback=True)
>>> s['liz'][0] = 1
>>> s.close()
>>> s = shelve.open('test_shelve')
>>> for key in s.keys():
print key, ':', s[key]

bob : 43
liz : [1]
```

## Gotcha: Shelves update on assignment not mutation! Solution 2

```
>>> s = shelve.open('test_shelve', writeback=True)
>>> tmp = s['liz']
>>> tmp[0] = 2
>>> s['liz'] = tmp
>>> s.close()
>>> s = shelve.open('test_shelve')
>>> for key in s.keys():
print key, ':', s[key]

bob : 43
liz : [2]
```