POWERING
POTENTIAL

Python 3 Programming

Data Storage and File Handling

DATA STORAGE AND FILE HANDLING

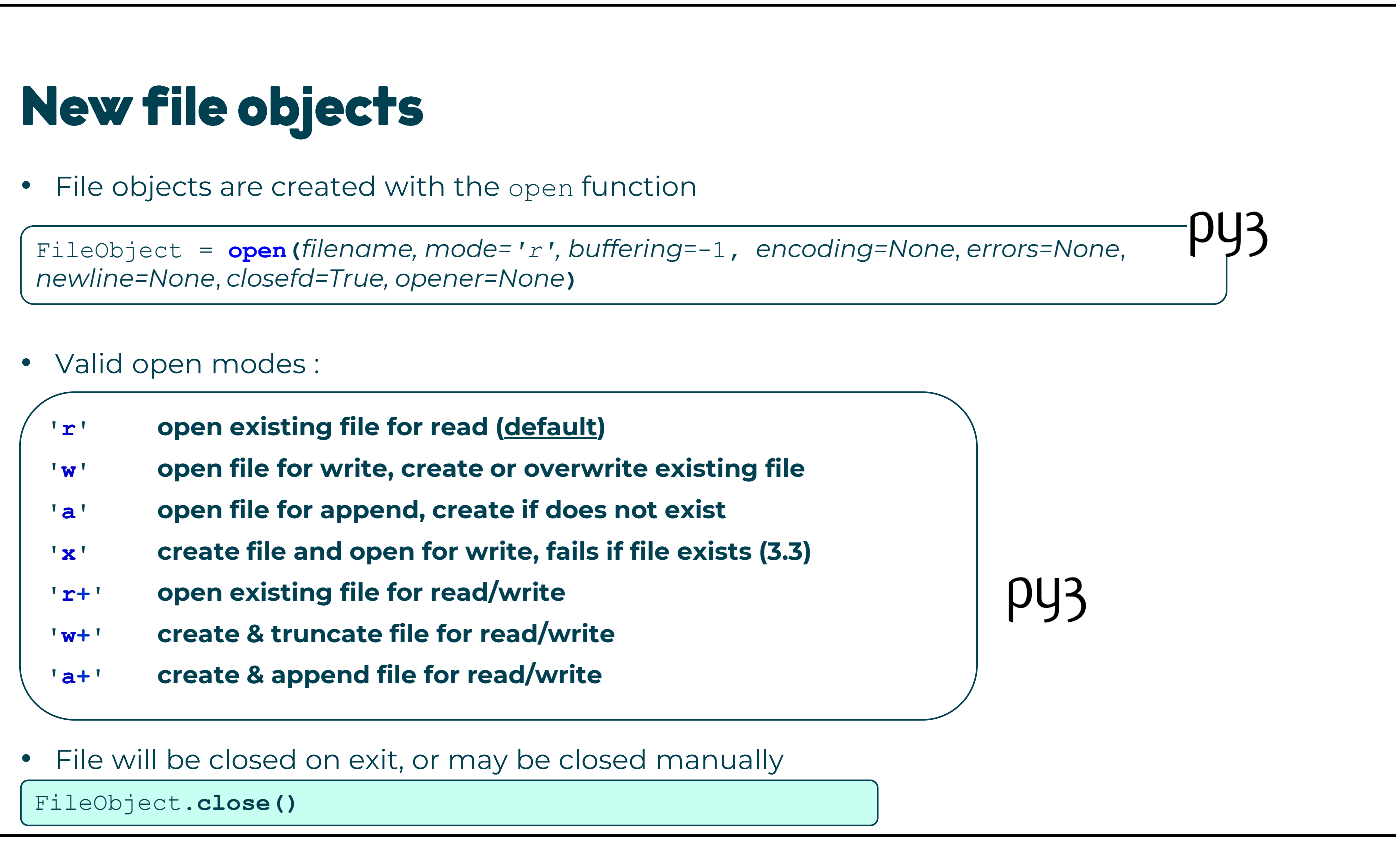
Contents

- File objects
- Reading files
- Writing files
- Standard streams
- More tricks
- Random access
- Pickles
- Shelves
- Compression

Summary

- Database interface overview
- Binary files: pack/unpack

Python has input/output features which are very easy to use, with syntax reminiscent of C and PHP.



New file objects

- File objects are created with the `open` function

```
FileObject = open(filename, mode='r', buffering=-1,  encoding=None, errors=None,
newline=None, closefd=True, opener=None)
```

py3

- Valid open modes :

`'r'`	**open existing file for read (default)**
`'w'`	**open file for write, create or overwrite existing file**
`'a'`	**open file for append, create if does not exist**
`'x'`	**create file and open for write, fails if file exists (3.3)**
`'r+'`	**open existing file for read/write**
`'w+'`	**create & truncate file for read/write**
`'a+'`	**create & append file for read/write**

py3

- File will be closed on exit, or may be closed manually

```
FileObject.close()
```

Files are accessed through file objects, which are created using the **open** function. A '**b**' must be appended to the mode if the file is non-text, and optionally '**t**' may be appended for text files (which is the default).

All files are automatically closed (and flushed) when the program exits, or the file object is destroyed (drops out of scope). It is usually considered to be good programming practice to explicitly close the file as soon as possible, and this may be done by calling the **close** method.

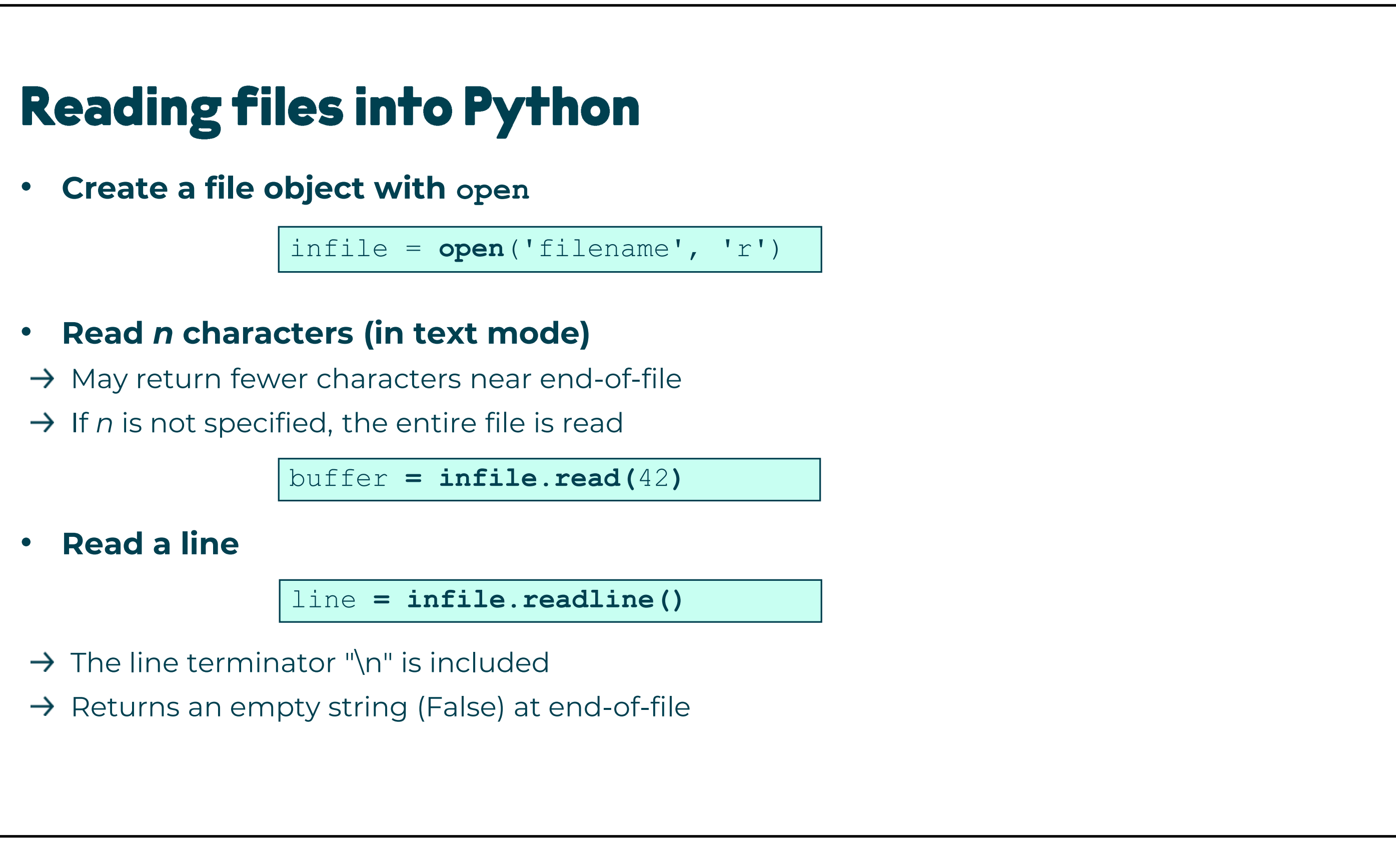
The mode 'x' was introduced at Python 3.3 and is used to exclusively create and open the file. If the file already exists then this will fail with `FileExistsError: [Errno 17] File exists`.

Python 3 also supports the 'U' (universal newline) mode, but that is provided for compatibility reasons only - it is no longer required and is deprecated.

The buffering argument can be set to zero on binary files, which means buffering is switched off. A value of one (1) means line-buffering, and a greater value gives the actual size of the buffer required. The default, -1, will choose a suitable buffer size for the system in use.

There are other parameters available to **open**, but they are not often needed: *encoding=None*, *errors=None*, *newline=None*, *closefd=True*). See the on-line documentation for details.

A further argument, *opener=None,* was added at 3.3. This allows us to provide an alternative method for opening the file.



Reading files into Python

- **Create a file object with `open`**

```
infile = open('filename', 'r')
```

- **Read *n* characters (in text mode)**

→ May return fewer characters near end-of-file

→ If *n* is not specified, the entire file is read

```
buffer = infile.read(42)
```

- **Read a line**

```
line = infile.readline()
```

→ The line terminator "\n" is included

→ Returns an empty string (False) at end-of-file

Once a file object has been created, we can read from the file using a method, and there are several to choose from.
With **`read`** we can specify the number of characters to be read, starting at the current file position. In binary mode (see later), the number given is the number of bytes, not Python characters (which are multi-byte). Related to this, we also have **`seek`**, which moves the current file position to a specified offset, and **`tell`**, which returns the current file position.
Reading a text line, a record terminated by a new-line character, may be done using **readline**, and this is a very common way of accessing text files from Python. Notice that the record terminator is included in the buffer, and can be removed using a slice **`[:-1]`**.

Reading tricks

Reading the whole file into a variable

- Be careful of the file size

```
lines = open('brian.txt').read()
llines= open('brian.txt').read().splitlines()
linelist = open('brian.txt').readlines()
```

Reading a file sequentially in a loop

- Inefficient

```
for line in open('lines.txt').readlines():
    print(line, end="")
```

- Use the file object iterator

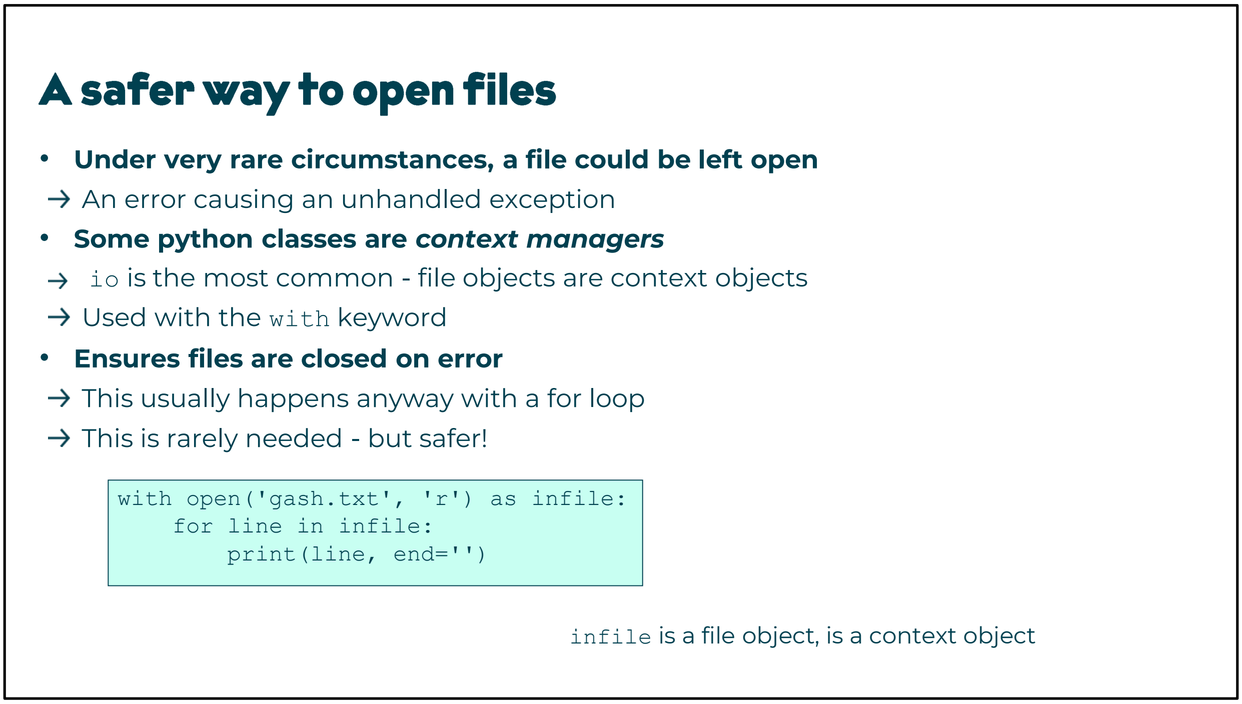
```
for line in open('lines.txt') :
    print(line, end="")
```

With the first three examples, the first (**lines**) reads the whole file into one string. The second (**llines**) reads the whole file into a string, but then produces a list from the **splitlines**. The third example (**linelist**) is similar in that it produces a list, but the new-line characters remain.

When reading each line in a loop (a common requirement), it is generally better to invoke the file iterator. Calling **readlines** in a `for` loop will read the entire file into a temporary list in-memory then iterate through that, which is rather inefficient.

Note the **end** parameter in the **print** statements. This prevents an additional new-line from being printed - we already have one at the end of each record.

Incidentally, *do not* use **seek** (random access) in the **for** loop - it upsets the file iterator and you can end-up in an infinite loop.



A safer way to open files

- **Under very rare circumstances, a file could be left open**

→ An error causing an unhandled exception

- **Some python classes are *context managers***

→ `io` is the most common - file objects are context objects

→ Used with the `with` keyword

- **Ensures files are closed on error**

→ This usually happens anyway with a for loop

→ This is rarely needed - but safer!

```
with open('gash.txt', 'r') as infile:
    for line in infile:
        print(line, end='')
```

`infile` is a file object, is a context object

Many Pythonistas prefer this method of iterating through a file. Using a context manager gets over reliability issues with destructors (see later). However, this level of protection is rarely needed for file handling, and it is very difficult to find a scenario when it is. However, you will see this commonly used with `open`.

So you probably don't need to use this, but do you want to take the risk? We suggest that you don't change existing code to use this mechanism, but consider using it for new code.

Writing to files from Python

Open a file handle with `open`

- Specifying write or append

```
output = open('myfile', 'w')
append = open('logfile', 'a')
```

- Write a string
- Append "\n" to make it a line
- Returns the number of chars (text) or bytes (binary) written

```
num = output.write("Hello\n")
```

py3

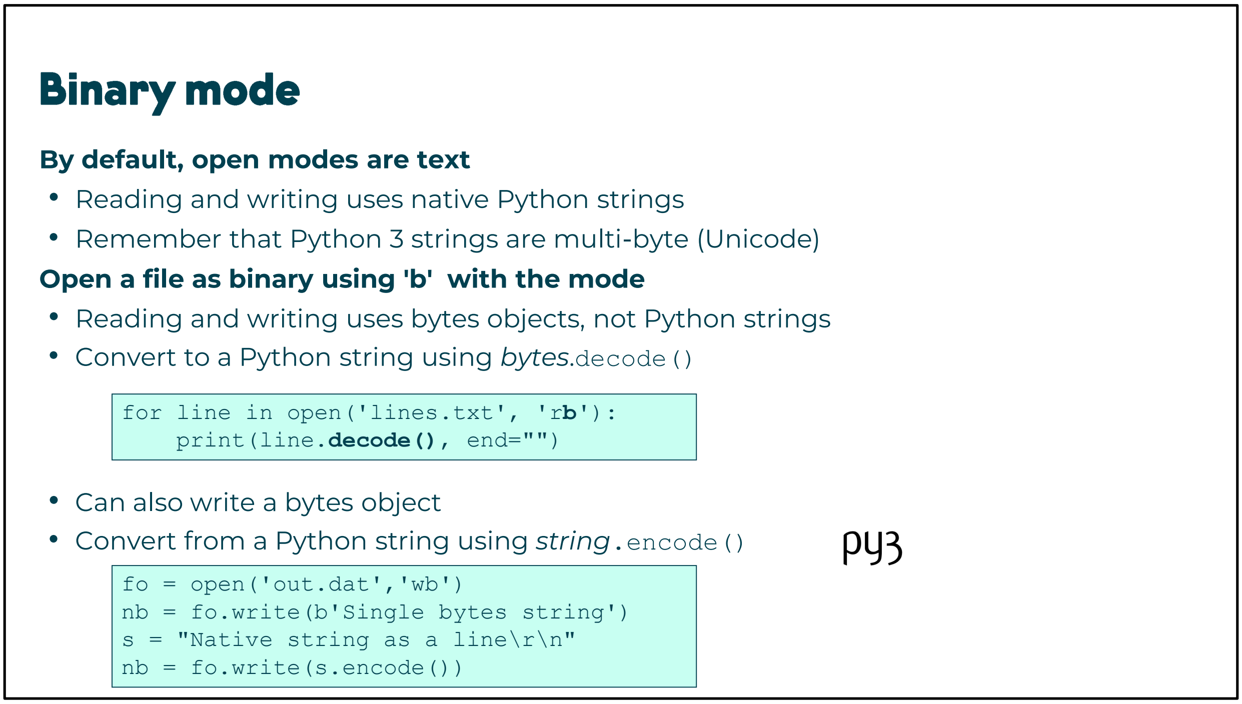
- Write strings from a list
- Append "\n" to each element to make lines

```
output.writelines(list)
```

Just as there are several ways of reading from a file, there are a couple of ways of writing. The **`write`** method can be used to write characters to a file at the current position, but remember to include the new-line terminator when writing to text files.

In Python 2, **`write`** did not return anything, but in Python 3 it returns the number of characters or bytes written, depending on how the file has been opened.

Writing records from a list uses the **`writelines`** method. Its name is a little misleading, it does not add line terminators (new-lines) to the end of each line, they must already be in the data.



Binary mode

By default, open modes are text

- Reading and writing uses native Python strings
- Remember that Python 3 strings are multi-byte (Unicode)

Open a file as binary using 'b' with the mode

- Reading and writing uses bytes objects, not Python strings
- Convert to a Python string using *bytes*.`decode()`

```
for line in open('lines.txt', 'rb'):
    print(line.decode(), end="")
```

- Can also write a bytes object
- Convert from a Python string using *string*`.encode()`

```
fo = open('out.dat','wb')
nb = fo.write(b'Single bytes string')
s = "Native string as a line\r\n"
nb = fo.write(s.encode())
```

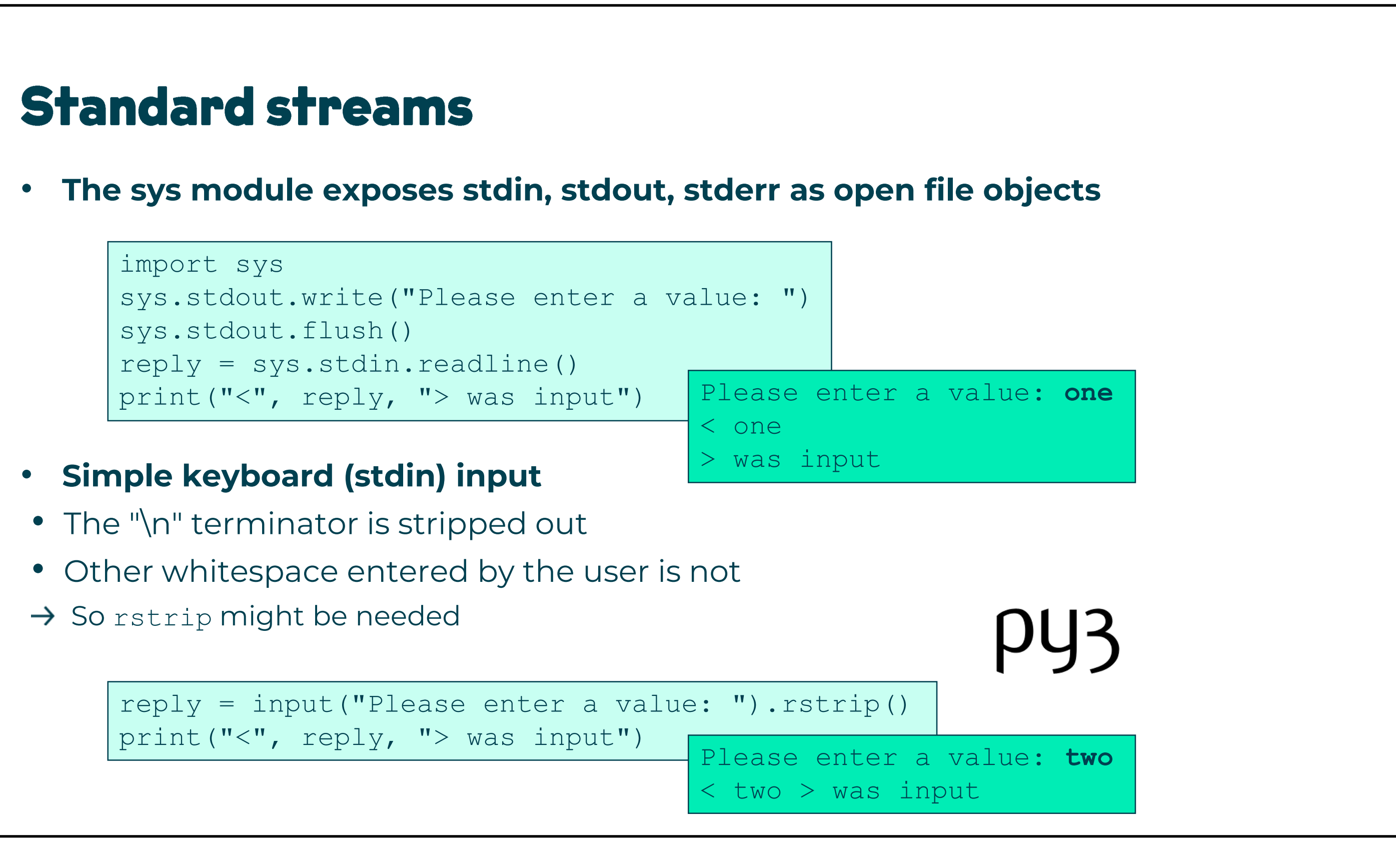
py3

Many programming languages, such as C and Perl, support a binary open mode on Windows. This is because the Windows operating system supports text and binary files differently. On UNIX, with those *other* languages, we just open the file for read or write, there is no difference between text and binary.

Not so with *this* language. With old fashioned languages text is single-byte character based, with multi-byte characters being the exception. In Python 3, (Java, and .Net) strings are multi-byte, and that is what is used if a file is opened as text, which is the default. By the way, writing a Python string still looks like ordinary text in the file if the characters are within a single-byte character set, like ISO Latin 1.

Setting binary mode, *even on UNIX*, forces single-byte characters, a **`byte`** object. Fortunately, it is easy to convert between bytes and strings, and the methods which can be applied to a **`byte`** object are similar to those for strings.

Text mode (the default) also handles line-endings relevant for the platform you are running on. With binary mode, there is no line-ending handling, so if you want end-of-line you have to explicitly say so.



Standard streams

- **The sys module exposes stdin, stdout, stderr as open file objects**

```
import sys
sys.stdout.write("Please enter a value: ")
sys.stdout.flush()
reply = sys.stdin.readline()
print("<", reply, "> was input")
```

```
Please enter a value: one
< one
> was input
```

- **Simple keyboard (stdin) input**
  - The "\n" terminator is stripped out
  - Other whitespace entered by the user is not

→ So `rstrip` might be needed

py3

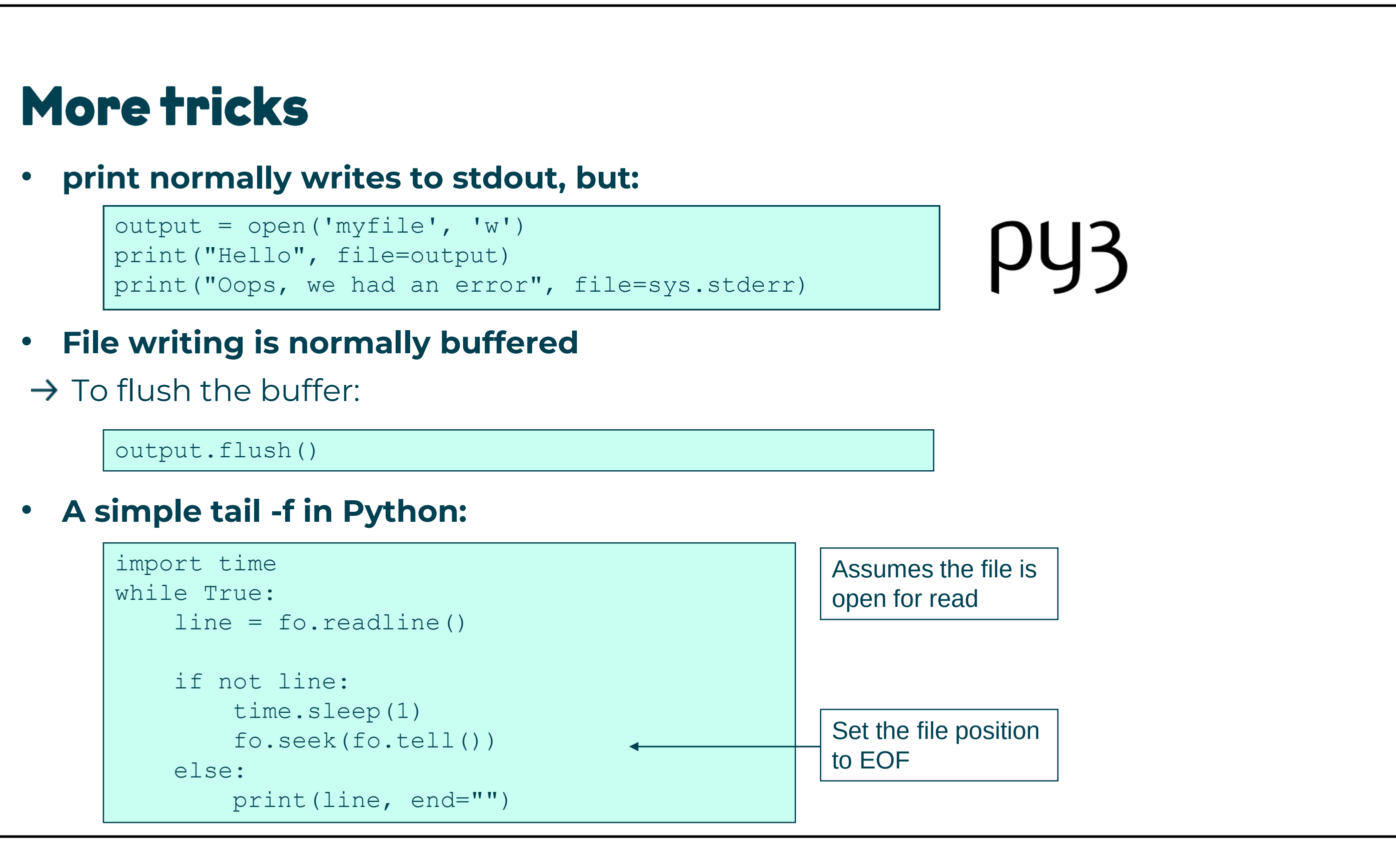
```
reply = input("Please enter a value: ").rstrip()
print("<", reply, "> was input")
```

```
Please enter a value: two
< two > was input
```

The three standard IO streams, stdin, stdout, and stderr, can be accessed directly through the **`sys`** module. Each one is a file object by that name, and may be used as any other file object.
In addition, the **`input`** statement reads from stdin, which is usually the keyboard but may have been redirected. If the **`readline`** module is imported first, then **`input`** will use it to provide line editing and history features. If you have seen the **`input`** function in Python 2, then please note that the Python 3 version is different. The **py3** icon is on the slide because **`input`** was called **`raw_input`** in Python 2.
On Windows cmd.exe, line endings are "`\r\n`", and in Python 3.2.0 a bug striped out the new-line but not the "`\r`". Fortunately, this bug was fixed in 3.2.1., but the work-around, using `rstrip()`, is sometimes a good idea anyway in case the user adds trailing spaces.
Normally standard streams like stdin and stdout are used in text mode, but if you wish to use them in binary mode then use the -u command-line option, for example:

```
python -u myprog.py
```

Other tricks with standard streams will be revealed later in the course.



More tricks

- **print normally writes to stdout, but:**

```
output = open('myfile', 'w')
print("Hello", file=output)
print("Oops, we had an error", file=sys.stderr)
```

py3

- **File writing is normally buffered**

→ To flush the buffer:

```
output.flush()
```

- **A simple tail -f in Python:**

```
import time
while True:
    line = fo.readline()

    if not line:
        time.sleep(1)
        fo.seek(fo.tell())
    else:
        print(line, end="")
```

Assumes the file is open for read

Set the file position to EOF

In Python 3, the **`print`** function may be used for writing to a file using the **`file=`** parameter. In earlier versions of Python "redirection" notation was used, and the **`print`** shown would have looked like this:

```
print >> output, "Hello"       # Python 2
```

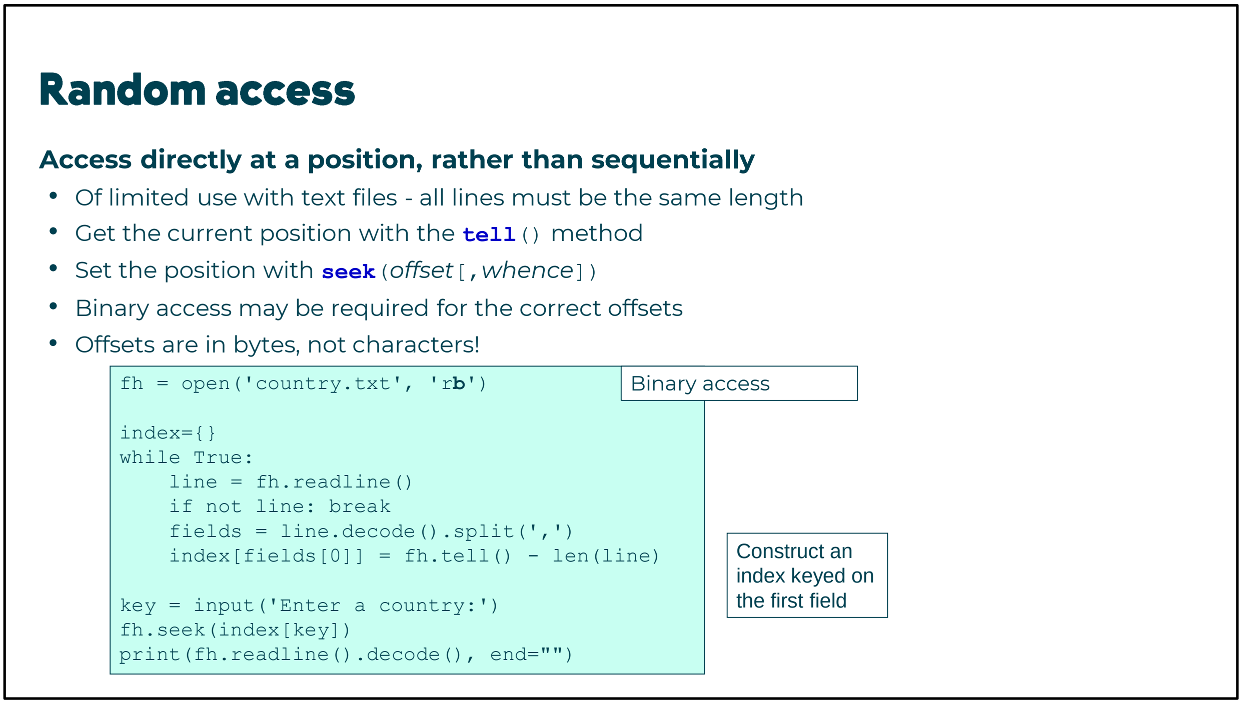
Buffering can also be turned off using the -u command-line option to python, or by setting the `PYTHONUNBUFFERED` environment variable to a non-empty string.

The final example requires some explanation: **`tail -f`** is a UNIX command which displays lines as they are appended. It works by reading to the end-of-file then waiting for one second. If a record has been added, then it will be displayed, otherwise we wait for one second again. There are several ways of improving this example!

Possibly the most peculiar aspect of this example is setting (`seek`) the current file position. When we hit end-of file the current file position becomes invalid, the line

```
fo.seek(fo.tell())
```

restores the file position to (physically) the same position as before.



Random access

Access directly at a position, rather than sequentially

- Of limited use with text files - all lines must be the same length
- Get the current position with the **tell**() method
- Set the position with **seek**(*offset*[,*whence*])
- Binary access may be required for the correct offsets
- Offsets are in bytes, not characters!

```
fh = open('country.txt', 'rb')

index={}
while True:
    line = fh.readline()
    if not line: break
    fields = line.decode().split(',')
    index[fields[0]] = fh.tell() - len(line)

key = input('Enter a country:')
fh.seek(index[key])
print(fh.readline().decode(), end="")
```

Binary access

Construct an index keyed on the first field

Most file IO, particularly with text files, is sequential. To get to a specific record in this way can be slow, particularly if we are doing this many times with the same data. Instead we can access the file randomly, or by position.

Unless we can calculate the position in some way, we need to know where each record is, so an index can be constructed - the obvious way to do that is to use a dictionary (which can be saved in a pickle).

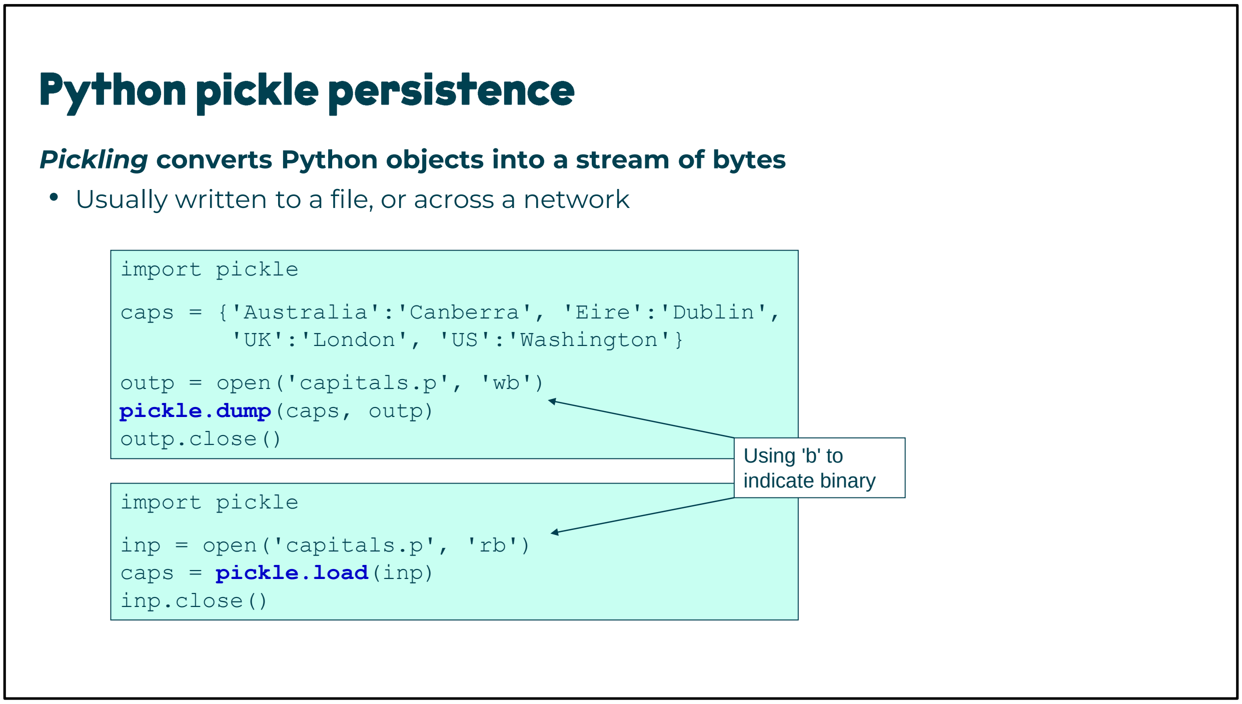
Notice that we open the file in **binary** mode. This is important on Windows since the '\r' is hidden in text mode and would mean our offsets would be one byte out. means that byte objects are read, rather than strings.

You may be wondering why we have used a convoluted **while** loop to read the file and create the index. This is because **tell()** is disabled inside a **for** loop based on **readline** or **open**. The iterator (a method called next()) may read-ahead into an internal buffer for efficiency, so the current file position might not actually be where you think - for example a small file might be read into memory in one go. Therefore, the method shown might be slower than using a **for** loop and accessing the file sequentially!

When we have a position, we can **seek** to that point, then read or write. By default the position (*offset*) is relative to beginning of file, but we can specify a different **whence** value:

0	offset is relative to beginning of file
1	offset is relative to the current file position can supply a negative offset to overwrite a record just read

2	offset is relative to end of file it is not an error to seek beyond end-of-file on most file systems



Python pickle persistence

***Pickling* converts Python objects into a stream of bytes**

- Usually written to a file, or across a network

```
import pickle

caps = {'Australia':'Canberra', 'Eire':'Dublin',
        'UK':'London', 'US':'Washington'}

outp = open('capitals.p', 'wb')
pickle.dump(caps, outp)
outp.close()
```

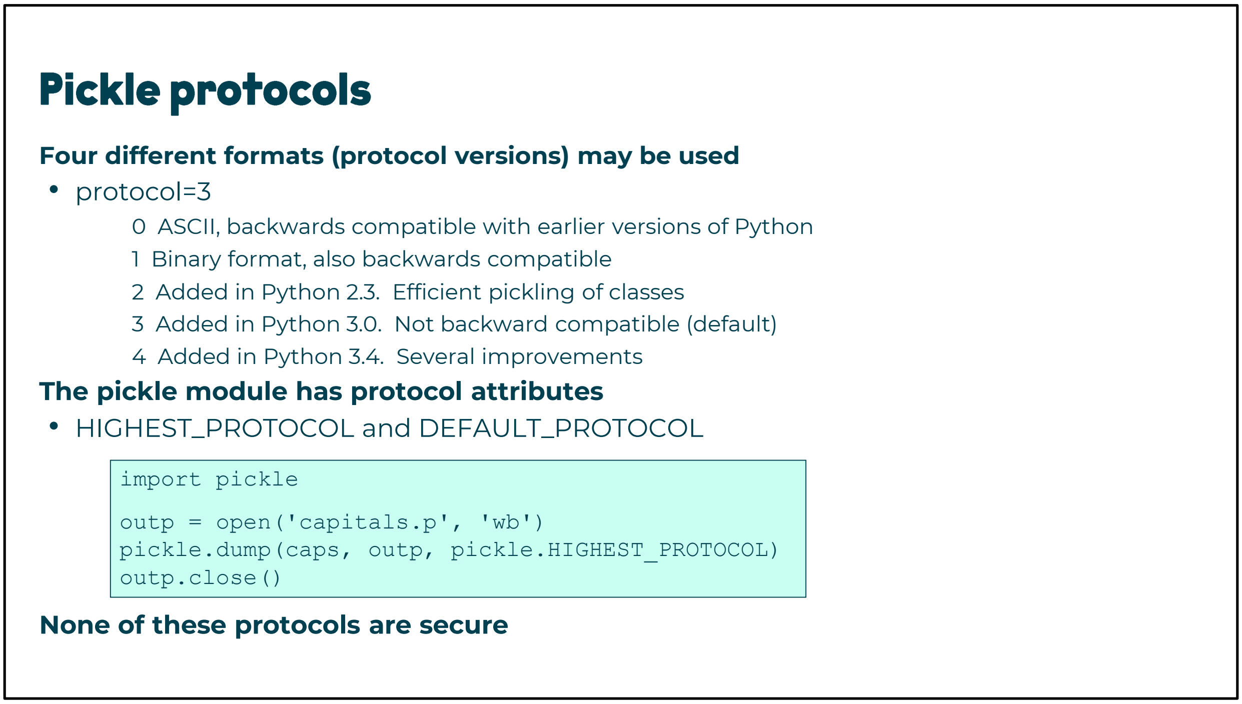
```
import pickle

inp = open('capitals.p', 'rb')
caps = pickle.load(inp)
inp.close()
```

Using 'b' to indicate binary

Pickling is a Python way of storing its own object types in a file.
Make sure the file is opened as binary.
What can be pickled?

None, **True**, and **False**
integers, floating point numbers, complex numbers
strings, bytes, bytearrays
tuples, lists, sets, and dictionaries containing only picklable objects
classes that are defined at the top level of a module

The python documentation also says that functions can be pickled, but this is misleading because only the function names survive. Code itself cannot be pickled easily, that is what a module is for. In fact, the .pyc compiled module file is very similar to a pickle. Python internal serialization (pyc files) can be exposed by the `marshal` module. That is more primitive than pickle, and offers fewer features.



Pickle protocols

Four different formats (protocol versions) may be used

- protocol=3

  0 ASCII, backwards compatible with earlier versions of Python
  1 Binary format, also backwards compatible
  2 Added in Python 2.3. Efficient pickling of classes
  3 Added in Python 3.0. Not backward compatible (default)
  4 Added in Python 3.4. Several improvements

The pickle module has protocol attributes

- HIGHEST_PROTOCOL and DEFAULT_PROTOCOL

```
import pickle

outp = open('capitals.p', 'wb')
pickle.dump(caps, outp, pickle.HIGHEST_PROTOCOL)
outp.close()
```

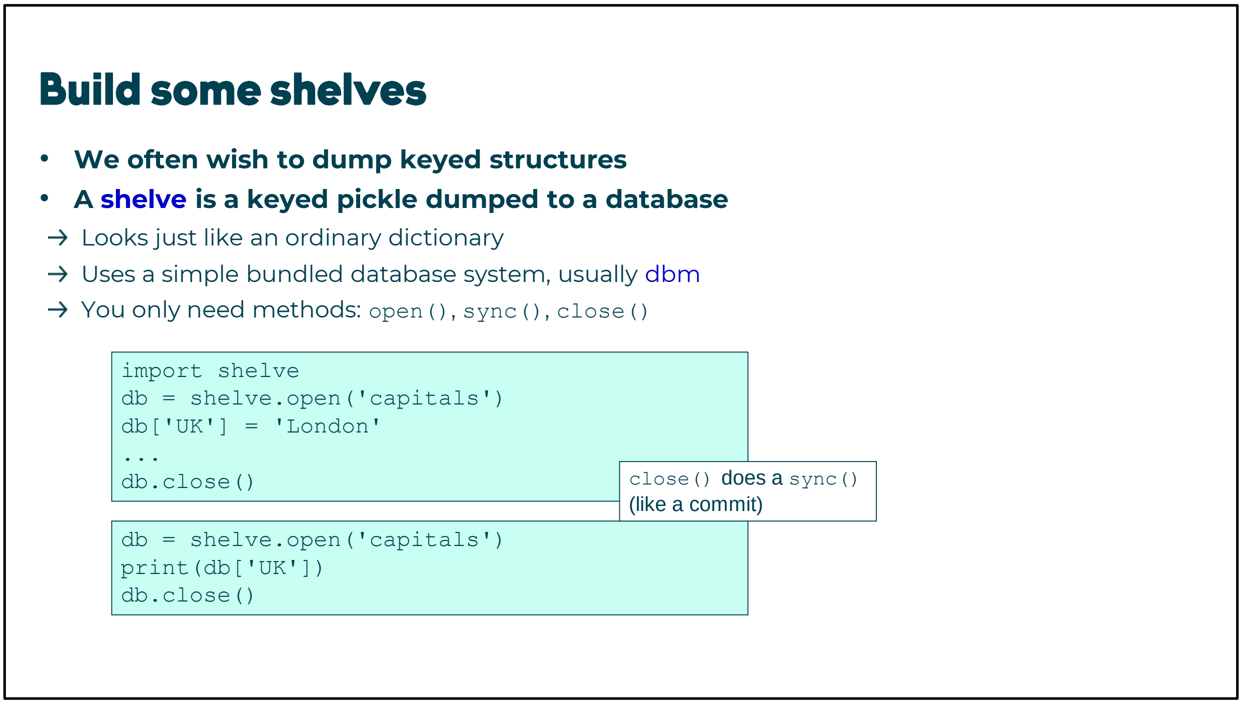
None of these protocols are secure

Pickle files can be written in a number of formats. If in doubt, use the default.

If ASCII is used, you do not need to open the file as binary, but the same mode should be used for reading and writing. ASCII is useful for debugging, and binary format is useful if you require earlier Python programs to read your files. Otherwise, the default (3) is your best bet.

Pickle had problems in Python 3.0 when creating pickles for Python 2 (protocols 0-2). This was fixed in Python 3.1, but the fix meant that protocols 0-2 cannot be passed between Python 3.0 and 3.1. So, if passing between Python 3 versions use protocol 3, if passing between Python 3 and Python 2 then make sure you upgrade to 3.1.

Pickle protocol 4 was added at Python 3.4, and is not backward compatible. Its improvements include better handling of large data. See PEP 3154. Note that even on Python 3.4, protocol 3 is still the default.



Build some shelves

- **We often wish to dump keyed structures**
- **A shelve is a keyed pickle dumped to a database**

→ Looks just like an ordinary dictionary

→ Uses a simple bundled database system, usually dbm

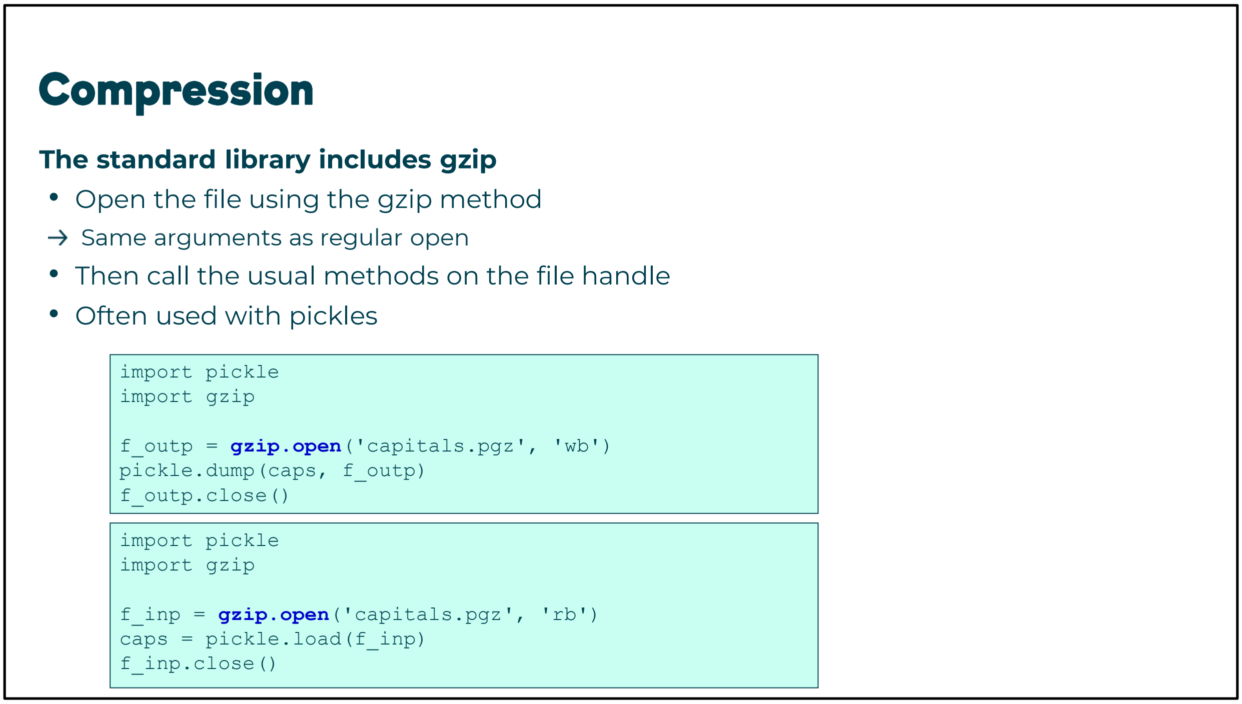
→ You only need methods: `open()`, `sync()`, `close()`

```
import shelve
db = shelve.open('capitals')
db['UK'] = 'London'
...
db.close()
```

`close()` does a `sync()` (like a commit)

```
db = shelve.open('capitals')
print(db['UK'])
db.close()
```

The **`shelve`** module is worth considering if you are going to use pickling on a large scale. It uses a database (usually **dbm**, depending on the environment) to store pickled objects by a string key. You should not consider the database to be portable, nor should you assume that it can be used by other (non-shelve) tools. Once we open the database, (different pickle protocols can be specified) then it is exposed as if it was an ordinary dictionary, for both read and write.
Note that you should not attempt to have more than one program (or thread) reading or writing the file at the same time. This will be prevented by file locking on most operating systems.



Compression

The standard library includes gzip

- Open the file using the gzip method

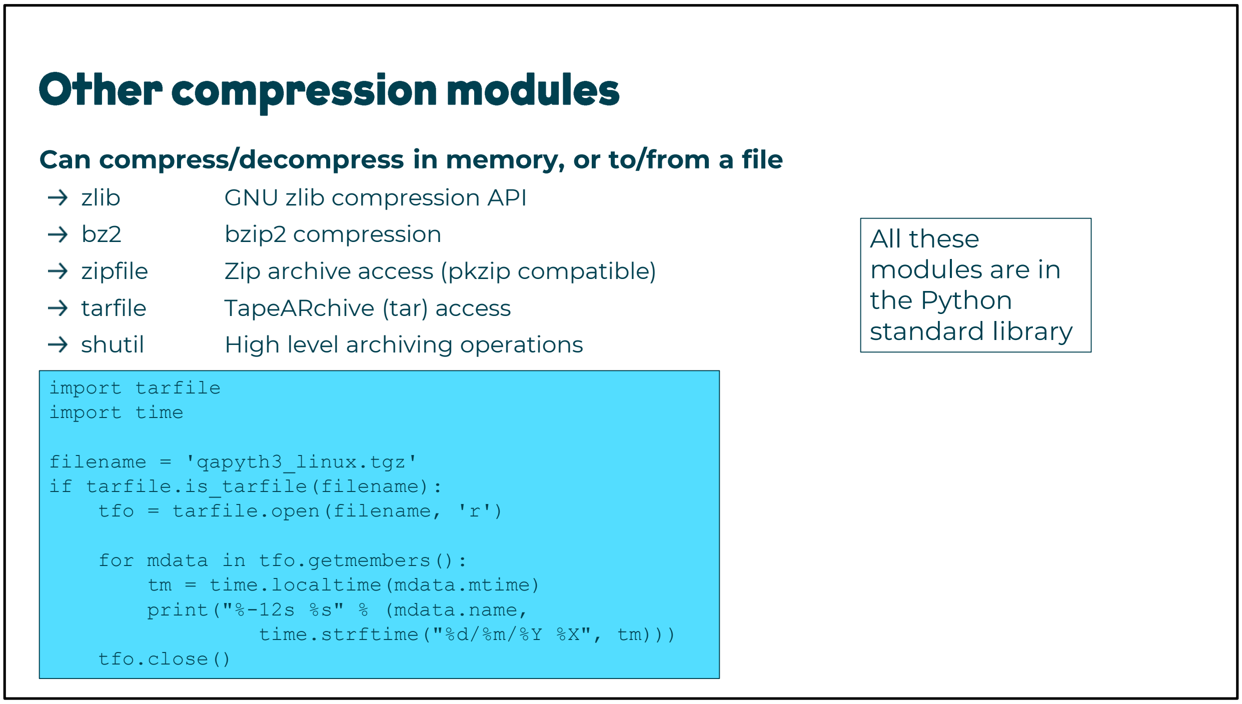
→ Same arguments as regular open

- Then call the usual methods on the file handle
- Often used with pickles

```
import pickle
import gzip

f_outp = gzip.open('capitals.pgz', 'wb')
pickle.dump(caps, f_outp)
f_outp.close()
```

```
import pickle
import gzip

f_inp = gzip.open('capitals.pgz', 'rb')
caps = pickle.load(f_inp)
f_inp.close()
```

For small amounts of data the file might not be any smaller (it might even be slightly larger). Notice that the zip files are open as binary, that means we get byte streams from zip files.
By the way, if you see a built-in function called `zip`, it has nothing to do with this. The `zip` built-in is for advanced combining of collections.



Other compression modules

Can compress/decompress in memory, or to/from a file

→ zlib	GNU zlib compression API
→ bz2	bzip2 compression
→ zipfile	Zip archive access (pkzip compatible)
→ tarfile	TapeARchive (tar) access
→ shutil	High level archiving operations

All these modules are in the Python standard library

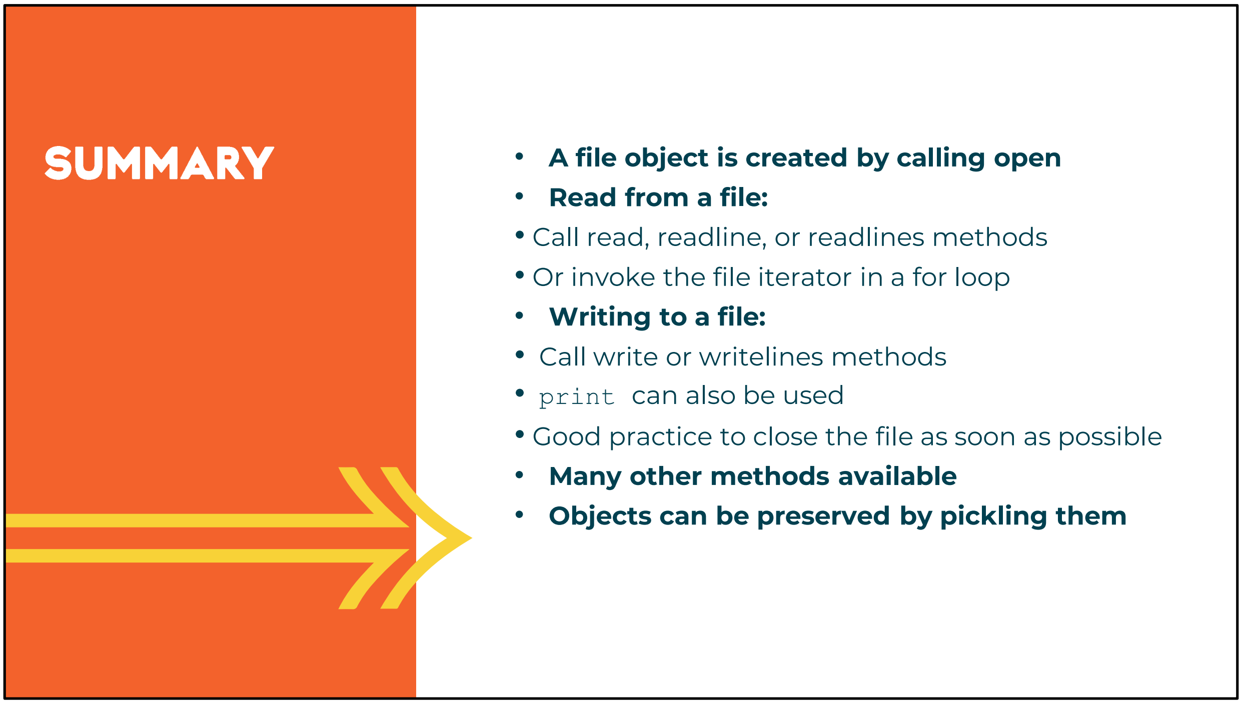
```
import tarfile
import time

filename = 'qapyth3_linux.tgz'
if tarfile.is_tarfile(filename):
    tfo = tarfile.open(filename, 'r')

    for mdata in tfo.getmembers():
        tm = time.localtime(mdata.mtime)
        print("%-12s %s" % (mdata.name,
                 time.strftime("%d/%m/%Y %X", tm)))
    tfo.close()
```

The `zlib` module includes checksum functions (adler32 and crc32) are intended to check data integrity and are not cryptographically secure.

bz2 supports a file-like class called bz2.BZ2File, which is used in a similar way to the open() built-in.

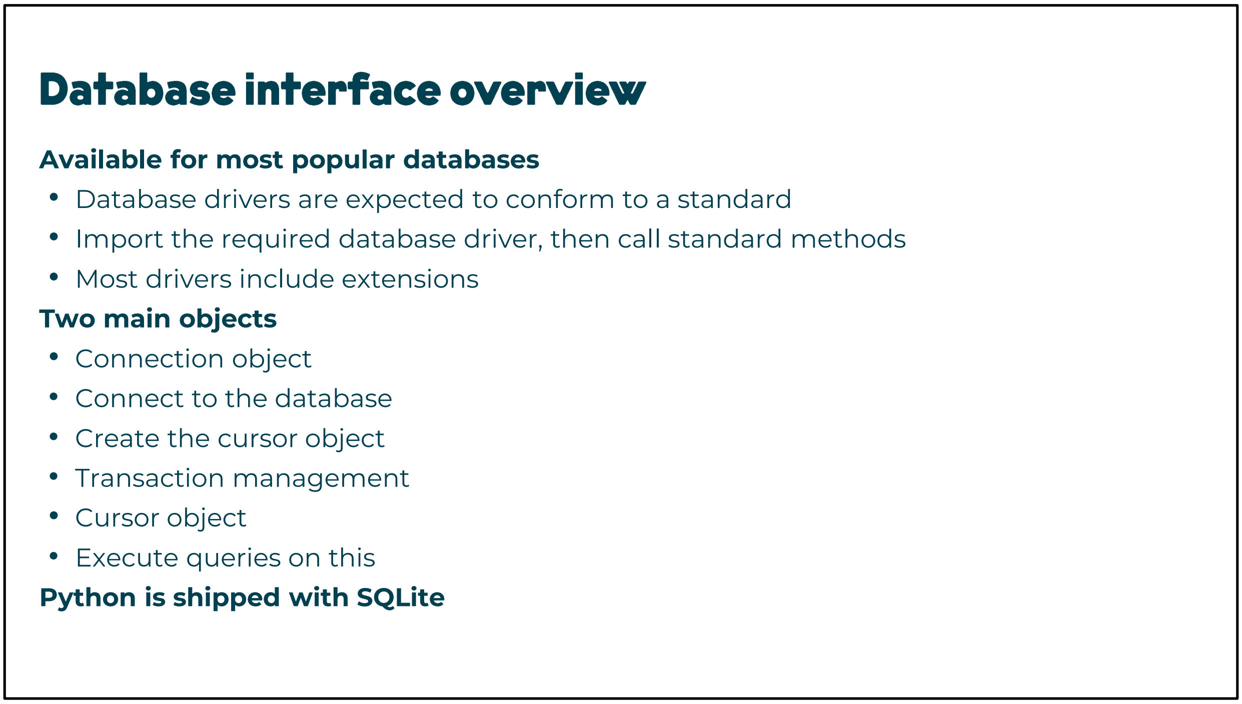
The `tarfile` module supports the usual expected mechanisms, in addition `getmembers()` returns all the metadata for files in an archive, not just the file name. The example shown was run on Windows, using a tar file with -z compression which came from Linux.

A higher level (simpler) generic interface is provided by the `shutil` module. This is a general module which includes archiving operations.



SUMMARY

- **A file object is created by calling open**
- **Read from a file:**
  - Call read, readline, or readlines methods
  - Or invoke the file iterator in a for loop
- **Writing to a file:**
  - Call write or writelines methods
  - `print` can also be used
  - Good practice to close the file as soon as possible
- **Many other methods available**
- **Objects can be preserved by pickling them**



Database interface overview

Available for most popular databases

- Database drivers are expected to conform to a standard
- Import the required database driver, then call standard methods
- Most drivers include extensions

Two main objects

- Connection object
- Connect to the database
- Create the cursor object
- Transaction management
- Cursor object
- Execute queries on this

Python is shipped with SQLite

Python database device drivers are available for most popular relational databases, and are likely to be bundled in with your release. They are written to a common standard - Python Database API Specification v2.0 (DB-API 2.0).

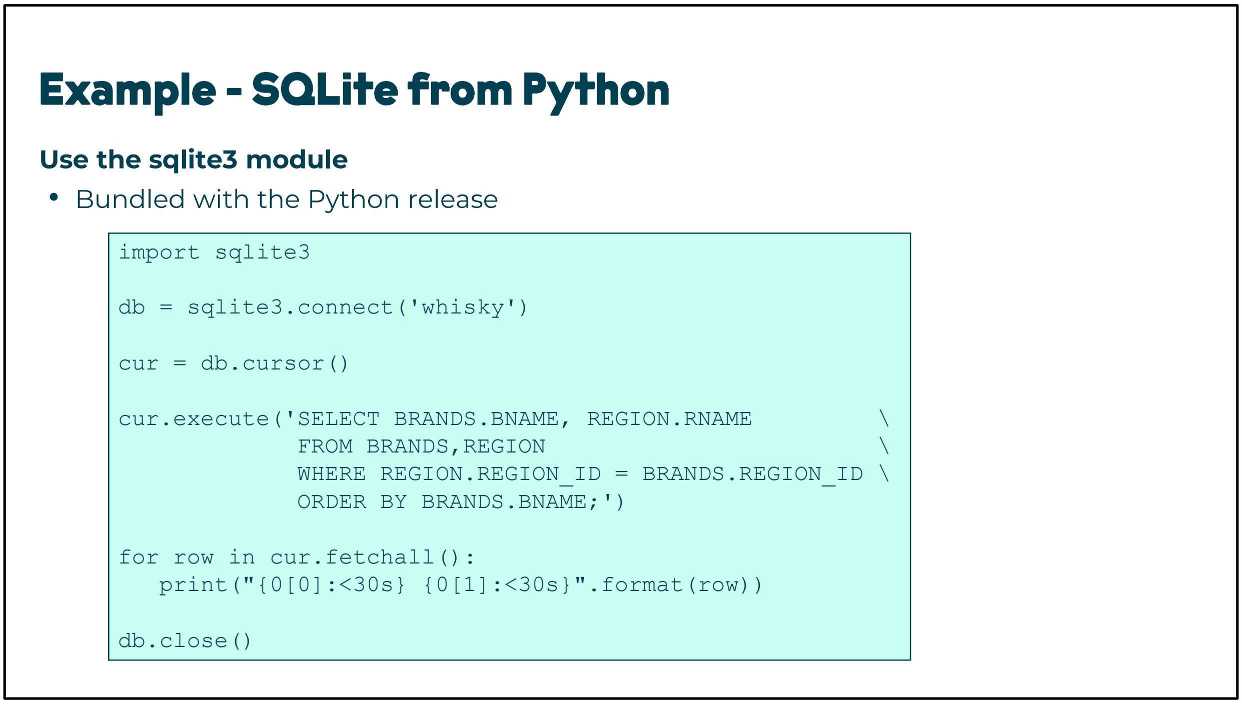
To use a database, we first have to connect with it, and that requires a connection object. That is then used to create a cursor, and for transaction management (commit and rollback).

The cursor object is the main workhorse, and a summary of the methods is shown below. Check the online documentation for details.

Cursor methods:

arraysize	Number of rows fetched by fetchmany
close	Close the cursor
description	Table meta-data
execute	Execute an SQL statement (supports placeholders)
executemany	Execute a sequence
fetchall	Get rows remaining
fetchmany	Get a number of rows
fetchone	Get next row
rowcount	Number of rows in the last operation

Python (like many languages) is shipped with SQLite (formally sqlite) which is a simple single file database.



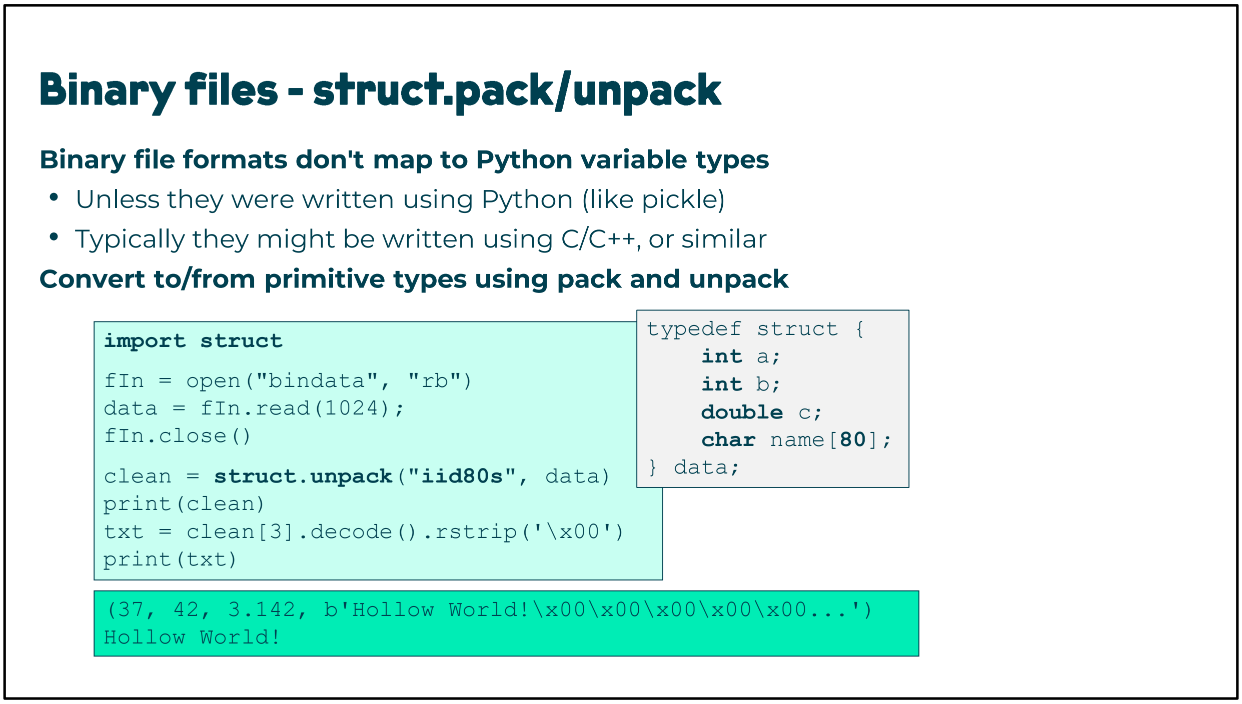
Example - SQLite from Python

Use the sqlite3 module

- Bundled with the Python release

```
import sqlite3

db = sqlite3.connect('whisky')

cur = db.cursor()

cur.execute('SELECT BRANDS.BNAME, REGION.RNAME         \
             FROM BRANDS,REGION                        \
             WHERE REGION.REGION_ID = BRANDS.REGION_ID \
             ORDER BY BRANDS.BNAME;')

for row in cur.fetchall():
   print("{0[0]:<30s} {0[1]:<30s}".format(row))

db.close()
```

Python uses cursors, which is traditionally the method used with embedded SQL in languages like COBOL and C. Neither PHP nor Perl support cursors at this time, but their use is being considered. The overall strategy for the SQL statement is not too dissimilar, however.



Binary files - struct.pack/unpack

Binary file formats don't map to Python variable types

- Unless they were written using Python (like pickle)
- Typically they might be written using C/C++, or similar

Convert to/from primitive types using pack and unpack

```
import struct

fIn = open("bindata", "rb")
data = fIn.read(1024);
fIn.close()

clean = struct.unpack("iid80s", data)
print(clean)
txt = clean[3].decode().rstrip('\x00')
print(txt)
```

```
typedef struct {
     int a;
     int b;
     double c;
     char name[80];
} data;
```

```
(37, 42, 3.142, b'Hollow World!\x00\x00\x00\x00\x00...')
Hollow World!
```

The first problem when reading binary data is to determine the format. Unless the data was written by Python (or, for example, using the Python API from C), then the data will not be directly compatible with Python types. All high-level dynamic programming languages have this problem.

The example shown is of a simple C struct which has been written to a file. Even if you have never seen C before, it is fairly obvious that the data consists of two integers (int), followed by a double precision number (double) followed by an array of 80 characters (that's a C string, and has a binary zero marking the end of the text).

To convert this to Python, we can use unpack from the **struct** module (part of the Python standard library). This function takes as its first parameter, a string which describes the block of data. The example shows **"iid80s"** - 2 integers, followed by a double, followed by an 80 character string.

The C string is single byte, so it is converted to multi-byte using **decode()**.

Note: when looking at C struct's be careful of padding bytes. By default, C compilers will align on item (often word) boundaries, and might insert pad bytes. This will vary between 64-bit and 32-bit compilers, and can even be adjusted (`#pragma pack`) by the program itself. You should also be aware of 'endian' issues well described in the documentation for the Python **struct** module.