Lesson 7: Files, Databases, and Pickles

Persistence

So far, we have learned how to write programs and communicate our intentions to the ***Central Processing Unit*** using conditional execution, functions, and iterations. We have learned how to create and use data structures in the ***Main Memory***. The CPU and memory are where our software works and runs. It is where all of the "thinking" happens.

But once the power is turned off, anything stored in either the CPU or main memory is erased. So up to now, our programs have just been transient fun exercises to learn Python.

In this lesson, we start to work with ***Secondary Memory***. Secondary memory is not erased even when the power is turned off. Or in the case of a USB flash drive, the data we write from our programs can be removed from the system and transported to another system.

These programs are ***persistent***: they run for a long time (or all the time); they keep at least some of their data in permanent storage (a hard drive, for example); and if they shut down and restart, they pick up where they left off.

Examples of persistent programs are operating systems, which run pretty much whenever a computer is on, and web servers, which run all the time, waiting for requests to come in on the network.

One of the simplest ways for programs to maintain their data is by reading and writing text files.

An alternative is to store the state of the program in a database. In this lesson I will present a simple database and a module, `pickle`, that makes it easy to store program data.

We will primarily focus on reading and writing text files such as those we create in a text editor.

Files

A text file is a sequence of characters stored on a permanent medium like a hard drive, flash memory, or CD - ROM.

First Thing's First

For the examples in this lesson we need few files.

The first one is called `words.txt` and it is a list of 113,809 official crosswords; that is, words that are considered valid in crossword puzzles and other word games. This is part of the Moby lexicon project (see http://wikipedia.org/wiki/Moby_Project).

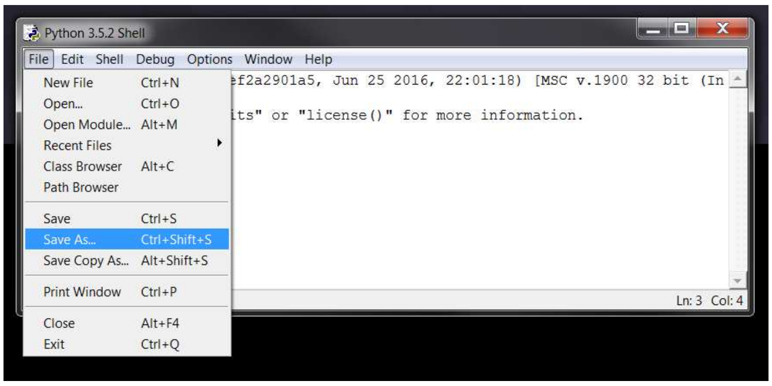
The second one is a list of emails from an open source coding project, called `mbox.txt`.

The third is from Act 2, Scene 2 of Romeo and Juliet, called `romeo-full.txt`.

You can download them here:

- https://online.cscc.edu/apps/python/book/words.txt
- https://online.cscc.edu/apps/python/book/mbox.txt
- https://online.cscc.edu/apps/python/book/romeo-full.txt

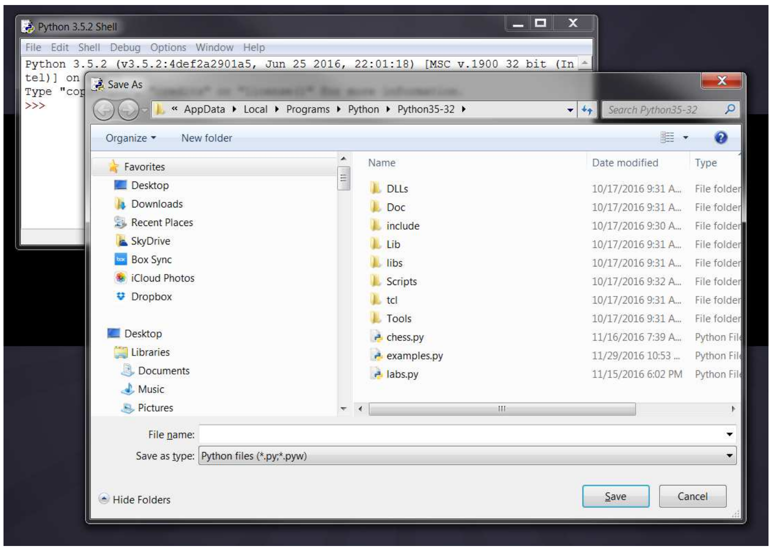
For ease, you will want to save these files in the same folder that you are in when you start Python. To find this folder, open IDLE and then go to *File > Save As*.



```
ef2a2901a5, Jun 25 2016, 22:01:18) [MSC v.1900 32 bit (In

its" or "license()" for more information.
```

The default folder which displays should be where you save this file.



```
Python 3.5.2 (v3.5.2:4def2a2901a5, Jun 25 2016, 22:01:18) [MSC v.1900 32 bit (In
tel)] on
Type "cop
>>>
```

Name	Date modified	Type
DLLs	10/17/2016 9:31 A...	File folder
Doc	10/17/2016 9:31 A...	File folder
include	10/17/2016 9:30 A...	File folder
Lib	10/17/2016 9:31 A...	File folder
libs	10/17/2016 9:31 A...	File folder
Scripts	10/17/2016 9:32 A...	File folder
tcl	10/17/2016 9:31 A...	File folder
Tools	10/17/2016 9:31 A...	File folder
chess.py	11/16/2016 7:39 A...	Python File
examples.py	11/29/2016 10:53 ...	Python File
labs.py	11/15/2016 6:02 PM	Python File

File name:
Save as type: Python files (*.py;*.pyw)

In the above example, I want to save the files in the `\AppData\Local\Programs\Python\Python35-32\` folder.

Opening Files

When we want to read or write a file, we first must open the file. Opening the file communicates with your operating system, which knows where the data for each file is stored. When you open a file, you are asking the operating system to find the file by name and make sure the file exists.

This file is in plain text, so you can open it with a text editor, but you can also read it from Python. The built - in function open takes the name of the file as a parameter and returns a file object you can use to read the file.

```
fin = open('words.txt')
```

`fin` is a common name for a file object used for input.

If we display the value of fin, we get this:

Code	Output

`fin = open('words.txt')` `print(fin)`	`<_io.TextIOWrapper name='words.txt' mode='r' encoding='cp1252'>`

If the open is successful, the operating system returns us a ***file handle***. The file handle is not the actual data contained in the file, but instead it is a "handle" that we can use to read the data. You are given a handle if the requested file exists and you have the proper permissions to read the file.

Go online to complete the
Opening a File Challenge

If the file does not exist, open will fail with a `traceback` and you will not get a handle to access the contents of the file:

Code	Output
`fin = open('stuff.txt')` `print(fin)`	`FileNotFoundError: [Errno 2] No such file or directory: stuff.txt'`

Later we will use try and except to deal more gracefully with the situation where we attempt to open a file that does not exist.

Text Files and Lines

A text file can be thought of as a sequence of lines, much like a Python string can be thought of as a sequence of characters. To break the file into lines, there is a special character that represents the "end of the line" called the newline character.

We've seen as far back as Lesson 1 that the newline character is `\n`. Remember that even though this looks like two characters, it is actually a single character. When we look at the variable by entering "stuff" in the interpreter, it shows us the `\n` in the string, but when we use print to show the string, we see the string broken into two lines by the newline character.

Code	Output
`stuff = '1\n2'` `print(stuff)`	`1` `2`

You can also see that the length of the string `'1\n2'` is three characters because the newline character is a single character.

Code	Output
`stuff = '1\n2'`	`3`

```
print(len(stuff))
```

So when we look at the lines in a file, we need to imagine that there is a special invisible character called the newline at the end of each line that marks the end of the line.

So the newline character separates the characters in the file into lines.

Reading Files

While the file handle does not contain the data for the file, it is quite easy to construct a `for` loop to read through and count each of the lines in a file:

Code

```
fin = open('words.txt')

count = 0

for line in fin:
    count += 1

print("Line Count", count)
```

Output

```
Line Count 113809
```

We can use the file handle as the sequence in our `for` loop. Our `for` loop simply counts the number of lines in the file and prints them out. The rough translation of the `for` loop into English is, "for each line in the file represented by the file handle, add one to the count variable."

The reason that the open function does not read the entire file is that the file might be quite large with many gigabytes of data. The open statement takes the same amount of time regardless of the size of the file. The `for` loop actually causes the data to be read from the file.

When the file is read using a `for` loop in this manner, Python takes care of splitting the data in the file into separate lines using the newline character. Python reads each line through the newline and includes the newline as the last character in the line variable for each iteration of the `for` loop.

Because the `for` loop reads the data one line at a time, it can efficiently read and count the lines in very large files without running out of main memory to store the data. The above program can count the lines in any size file using very little memory since each line is read, counted, and then discarded.

Go online to complete the Counting Lines Challenge

The readline Method

The file object provides several methods for reading, including `readline`, which reads characters from the file until it gets to a newline and returns the result as a string:

Code	Result

`fin = open('words.txt')`	`'aa\n'`
`fin.readline()`	

The first word in this particular list is "aa", which is a kind of lava. The sequence `\n` represents two whitespace characters, a carriage return and a newline, that separate this word from the next.

When we print the result `fin.readline()`, we cannot see the `\n` because it renders as a new line break. But if we immediately display another character, we can see the line break:

Code	Output
`fin = open('words.txt')`	`aa`
`print(fin.readline())`	`hi`
`print("hi")`	

Go online to complete the
Reading a Line Challenge

The file object keeps track of where it is in the file, so if you call `readline` again, you get the next word:

Code	Result
`fin.readline()`	`'aah\n'`

The next word is "aah", which is a perfectly legitimate word, so stop looking at me like that.

Go online to complete the
Reading a Second Line Challenge

If it's the whitespace that's bothering you, we can get rid of it with the string method `strip`:

Code	Result
`line = fin.readline()`	`'aahed'`
`word = line.strip()`	
`word`	

If you recall back to Lesson 5, strip will remove any whitespace (spaces, tabs, or newlines) from the beginning and end of a string.

Go online to complete the
Stripping Lines Challenge

You can also use a file object as part of a `for` loop. This program reads `words.txt` and prints each word, one per line:

Code

```
fin = open('words.txt')

for line in fin:
    word = line.strip()
    print(word)
```

Output

```
aa
aah
aahed
aahing
aahs
aal
aalii
aaliis
aals
aardvark
(...)
```

Go online to complete the
Using a for Loop with Lines Challenge

The readlines Method

If you want to store all lines in a list of lines, you can use the `readlines` method:

Code

```
fin = open('words.txt')

words = fin.readlines()
```

Result

```
['aa\n', 'aah\n', 'aahed\n',
'aahing\n', 'aahs\n', 'aal\n',
'aalii\n', 'aaliis\n', 'aals\n',
'aardvark\n', (...)]
```

Go online to complete the
Reading the Whole File Challenge

When we use the `len` function to see the length of the list called `words`, we get the full number of lines in the file `words.txt`:

Code	Result
`len(words)`	`113809`

Just like the number we got when we counted each line in the file!

So now, if we want to access the 15th line in this file, we can display the 14th index (remember that indices start with 0 so the 15th item is actually the 14th index):

Code	Result
`words[14]`	`'aasvogel\n'`

Because `words` is now a list, you can use all of the list methods (`find`, `replace`, slicing, etc) on it.

Go online to complete the
Indexing Lines Challenge

The read Method

If you know the file is relatively small compared to the size of your main memory, you can read the whole file into one string using the `read` method on the file handle.

Code	Output

`fin = open('words.txt')`
`words = fin.read()`
`print(len(words))`
	`1016714`

`print(words[:20])`	`aa`
	`aah`
	`aahed`
	`aahing`

In this example, the entire contents (all 1,016,714 characters) of the file `words.txt` are read directly into the variable `words`. We use string slicing to print out the first 20 characters of the string data stored in `words`.

When the file is read in this manner, all the characters including all of the lines and newline characters are one big string in the variable `words`. Remember that this form of the open function should only be used if the file data will fit comfortably in the main memory of your computer.

If the file is too large to fit in main memory, you should write your program to read the file in chunks using a `for` or `while` loop.

Go online to complete the
Using the read Method Challenge

Searching Through a File

When you are searching through data in a file, it is a very common pattern to read through a file, ignoring most of the lines and only processing lines which meet a particular condition. We can combine the pattern for reading a file wit string methods to build simple search mechanisms.

For these examples, we are going to use the file called `mbox.txt`. You can download it from https://online.cscc.edu/apps/python/book/mbox.txt. This file is a record of e-mail activity from various individuals in an open source project development team.

For example, if we wanted to read a file and only print out lines which started with the prefix "From:", we could use the string method `startswith` to select only those lines with the desired prefix:

Code

```
fin = open('mbox.txt')

for line in fin:
    if line.startswith("From:"):
        print(line)
```

Output

```
From: stephen.marquard@uct.ac.za

From: louis@media.berkeley.edu

From: zqian@umich.edu

From: rjlowe@iupui.edu

From: zqian@umich.edu

...
```

The output looks great since the only lines we are seeing are those which start with "From:", but why are we seeing the extra blank lines? Oh yeah – we forgot about that invisible newline character. Each of the lines ends with a newline, so the print statement prints the string in the variable line which includes a newline and then print adds another newline, resulting in the double spacing effect we see.

We could use line slicing to print all but the last character, but a simpler approach is to use the `strip` or `rstrip` method which strips whitespace from the right side of a string as follows:

Code

```
fin = open('mbox.txt')

for line in fin:

   line = line.strip()

   if line.startswith("From:"):
        print(line)
```

Output

```
From: stephen.marquard@uct.ac.za
From: louis@media.berkeley.edu
From: zqian@umich.edu
From: rjlowe@iupui.edu
From: zqian@umich.edu
...
```

As your file processing programs get more complicated, you may want to structure your search loops using `continue`. The basic idea of the search loop is that you are looking for "interesting" lines and effectively skipping "uninteresting" lines. And then when we find an interesting line, we do something with that line.

We can structure the loop to follow the pattern of skipping uninteresting lines as follows:

Code	Output

```
fin = open('mbox.txt')

for line in fin:

    line = line.strip()

    # Skip uninteresting line
    if not line.startswith("From:"):
        continue

    # Process 'interesting' line
    else:
        print(line)
```

```
From: stephen.marquard@uct.ac.za
From: louis@media.berkeley.edu
From: zqian@umich.edu
From: rjlowe@iupui.edu
From: zqian@umich.edu
...
```

The output of the program is the same. In English, the uninteresting lines are those which do not start with "From:", which we skip using continue. For the "interesting" lines (i.e., those that start with "From:") we perform the processing on those lines.

We can use the `find` string method to simulate a text editor search that finds lines where the search string is anywhere in the line. Since find looks for an occurrence of a string within another string and either returns the position of the string or `- 1` if the string was not found, we can write the following loop to show lines which contain the string "@uct.ac.za" (i.e., they come from the University of Cape Town in South Africa):

Code

```
fin = open('mbox.txt')

for line in fin:

    line = line.strip()

    # Skip uninteresting line
    if line.find("@uct.ac.za") ==  -
1:
        continue

    # Process 'interesting' line
    else:
        print(line)
```

Output

```
From stephen.marquard@uct.ac.za
Sat Jan  5 09:14:16 2008
X - Authentication - Warning:
nakamura.uits.iupui.edu: apache
set sender to
stephen.marquard@uct.ac.za
using –f
From:
stephen.marquard@uct.ac.za
Author:
stephen.marquard@uct.ac.za
From david.horwitz@uct.ac.za
Fri Jan  4 07:02:32 2008
...
```

Letting the User choose the File Name

We really do not want to have to edit our Python code every time we want to process a different file. It would be more usable to ask the user to enter the file name string each time the program runs so they can use our program on different files without changing the Python code.

This is quite simple to do by reading the file name from the user using `input` as follows:

```
file_name = input('Enter the file name: ')

fin = open(file_name)

count = 0
```

```
for line in fin:
    line = line.strip()

    if not line.startswith('Subject:') :
        continue

    else:
        count += 1

print("There were", count, "subject lines in", file_name)
```

We read the file name from the user and place it in a variable named `file`_name and open that file. Now we can run the program repeatedly on different files.

```
Enter the file name: mbox.txt
There were 1797 subject lines in mbox.txt

Enter the file name: mbox - short.txt
There were 27 subject lines in mbox.txt
```

Before peeking at the next section, take a look at the above program and ask yourself, "What could go possibly wrong here?" or "What might our friendly user do that would cause our nice little program to ungracefully exit with a traceback, making us look not - so - cool in the eyes of our users?"

What if our user types something that is not a file name?

```
Enter the file name: missing.txt
Traceback (most recent call last):
  File "examples.py", line 28, in <module>
    fin = open(file_name)
FileNotFoundError: [Errno 2] No such file or directory: 'missing.txt'
```

Do not laugh, users will eventually do every possible thing they can do to break your programs—either on purpose or with malicious intent. As a matter of fact, an important part of any software development team is a person or group called Quality Assurance (or QA for short) whose very job it is to do the craziest things possible in an attempt to break the software that the programmer has created.

The QA team is responsible for finding the flaws in programs before we have delivered the program to the end users who may be purchasing the software or paying our salary to write the software. So the QA team is the programmer's best friend.

So now that we see the flaw in the program, we can elegantly fix it using the try/except structure – but we will learn more about it later in this lesson. For now, let us move in to learn how to write our files.

Writing Files

To write a file, you have to open it with mode 'w' as a second parameter:

Code	Output

`fout = open('output.txt', 'w')`
 	
`print(fout)`
	`<_io.TextIOWrapper name='output.txt' mode='w' encoding='cp1252'>`

If the file already exists, opening it in write mode clears out the old data and starts fresh, so be careful!

If the file doesn't exist, a new one is created.

The `write` method of the file handle object puts data into the file. The `write` method accepts a string as its input:

```
line1 = "This here's the wattle,\n"
fout.write(line1)
```

Again, the file object keeps track of where it is, so if you call write again, it adds the new data to the end.

We must make sure to manage the ends of lines as we write to the file by explicitly inserting the newline character when we want to end a line. The `print` statement automatically appends a newline, but the `write` method does not add the newline automatically.

```
line2 = 'the emblem of our land.\n'
fout.write(line2)
```

When you are done writing, you have to close the file to make sure that the last bit of data is physically written to the disk so it will not be lost if the power goes off.

```
fout.close()
```

It is good programming practice to always close your files when you are finished with them. When we are writing files, we certainly want to explicitly close the files so as to leave nothing to chance.

The writelines Method

If you have a list of strings, you can write them all into the file in one fell swoop using the `writelines` method.

Code	Output

`my_list = ['Line 1', 'Line 2', 'Line 3']`
`fout = open('output.txt', 'w')`
`fout.writelines(my_list)`
`fout.close()`	

When we re-open `output.txt` in read mode and display the lines, this is what we get:

Code	Output

`fout = open('output.txt', 'r')`
`for line in fout:`
`    print(line)`
	`Line 1Line 2Line 3`

Notice that there are no line breaks. Something to keep in mind when using `writelines`: your list needs to have line breaks included if you want to separate your strings with line breaks.

Appending to Files

What if you want to merely append data to an existing file? You can use the append mode of `'a'`:

Code	Output
`fout = open('append.txt', 'a')`	`<_io.TextIOWrapper name='append.txt'`
`print(fout)`	`mode='a' encoding='cp1252'>`

Just like with the write mode, append will create the file if it does not already exist. Unlike `write` mode, append will leave all of the existing data intact, and will merely add data to the end of the file.

We add to the file using the `write` method:

```
line = "I am a lumberjack\n"
fout.write(line)
```

And we close the file the same way:

```
fout.close()
```

Later when we reopen the file and add another line to it, the original line will still be there:

Code	Output
`fout = open('append.txt', 'a')`	`I am a lumberjack`
`fout.write("and I'm okay")`	`and I'm okay`
`fout.readlines()`	

Available Modes

We have seen the modes `'r'`, `'w'`, and `'a'`. You can add a plus sign + to the mode to increase its functionality. It can get confusing, so here is a chart which might help:

Function	r	r+	w	w+	a	a+
read file	X	X		X		X
write to file		X	X	X	X	X
create file if it doesn't exist			X	X	X	X
truncate file (delete contents)			X	X		
start at beginning of file	X	X	X	X		

start at end of file	X	X

By default, open assumes that you are working with a text document. If you wish to work with a binary file, then you can pass the mode `'b'` into any of the above modes. You can also explicitly specify a text file by using the mode `'t'`:

Mode	Description
`'r+b'`	Read and write a binary file.
`'ab'`	Append to a binary file
`'w+t'`	Read and write a text file, truncating the file first and auto-creating it if it doesn't exist.

You should use the mode which provides the functionality you need, without providing too much functionality. If you only need to read a file, then don't use a mode which allows you to read and write to the file.

Dictionaries, Lists, and Tuples: File Example

One of the common uses of a dictionary is to count the occurrence of words in a file with some written text. Let's start with a very simple file of words taken from the text of Act 2, Scene 2 of Romeo and Juliet. Here is a small sample from `romeo-full.txt`:

```
Romeo and Juliet
Act 2, Scene 2

SCENE II. Capulet's orchard.

Enter ROMEO

ROMEO

He jests at scars that never felt a wound.
JULIET appears above at a window

But, soft! what light through yonder window breaks?
It is the east, and Juliet is the sun.
Arise, fair sun, and kill the envious moon,
Who is already sick and pale with grief,
That thou her maid art far more fair than she:
```

We will write a Python program to read through the lines of the file, break each line into a list of words, and then loop through each of the words in the line and count each word using a dictionary.

You will see that we have two for loops. The outer loop is reading the lines of the file and the inner loop is iterating through each of the words on that particular line. This is an example of a pattern called nested loops because one of the loops is the outer loop and the other loop is the inner loop.

Because the inner loop executes all of its iterations each time the outer loop makes a single iteration, we think of the inner loop as iterating "more quickly" and the outer loop as iterating more slowly.

The combination of the two nested loops ensures that we will count every word on every line of the input file.

```
fname = input('Enter the file name: ')

try:
    fhand = open(fname)

except:
    print('File cannot be opened:', fname)
    exit()

counts = dict()

for line in fhand:
    words = line.split()

    for word in words:
        if word not in counts:
            counts[word] = 1

        else:
            counts[word] += 1

print(counts)
```

When we run the program, we see a raw dump of all of the counts in unsorted hash order.

```
Enter the file name: romeo-full.txt

{'A': 2, 'name.': 1, 'sweet;': 1, 'foot,': 1, 'region': 1, 'that': 23,
'Sweet': 1, 'entreat': 1, 'think': 2, 'compliment!': 1, 'night.': 1,
'JULIET,': 2, 'walls': 1, 'Echo': 1, 'Exit': 2, 'come': 2, 'wanting': 1,
'Sweet,': 2, 'else,': 1, 'orchard': 1, 'sea,': 2, 'who': 3, 'one': 2,
'place?': 1, 'little': 1, 'May': 1, 'sight;': 1, 'was': 1, 'cheek!': 1,
'blessed': 2, ...}
```

It is a bit inconvenient to look through the dictionary to find the most common words and their counts, so we need to add some more Python code to get us the output that will be more helpful. We will use list sorting to make this happen by adding this code:

```
# Sort the dictionary by value
lst = list()

for key, val in counts.items():
    lst.append( (val, key) )

lst.sort(reverse=True)

for key, val in lst[:10] :
    print(key, val)
```

The first part of the program which reads the file and computes the dictionary that maps each word to the count of words in the document is unchanged. But instead of simply printing out counts and ending the program, we construct a list of `(val, key)` tuples and then sort the list in reverse order.

Since the value is first, it will be used for the comparisons. If there is more than one tuple with the same value, it will look at the second element (the key), so tuples where the value is the same will be further sorted by the alphabetical order of the key.

At the end we write a nice for loop which does a multiple assignment iteration and prints out the ten most common words by iterating through a slice of the list (`lst[:10]`).

So now the output finally looks like what we want for our word frequency analysis.

```
60 I
31 to
30 the
29 thou
28 JULIET
27 ROMEO
23 that
22 my
22 and
22 a
```

The fact that this complex data parsing and analysis can be done with an easy - to - understand 19 - line Python program is one reason why Python is a good choice as a language for exploring information.

Databases

A database is a file that is organized for storing data. Many databases are organized like a dictionary in the sense that they map from keys to values. The biggest difference between a database and a dictionary is that the database is on disk (or other permanent storage), so it persists after the program ends.

The module `dbm` provides an interface for creating and updating database files. As an example, I'll create a database that contains captions for image files.

Opening a database is similar to opening other files:

Code	Notes

`import dbm`
 	
`db = dbm.open('captions', 'c')`
	The mode `'c'` means that the database should be created if it doesn't already exist. The result is a database object that can be used (for most operations) like a dictionary.

When you create a new item, `dbm` updates the database file.

```
db['cleese.png'] = 'Photo of John Cleese.'
```

When you access one of the items, `dbm` reads the file:

Code	Result
`db['cleese.png']`	`b'Photo of John Cleese.'`

The result is a ***bytes object***, which is why it begins with `b`. A bytes object is similar to a string in many ways. When you get farther into Python, the difference becomes important, but for now we can ignore it.

If you make another assignment to an existing key, `dbm` replaces the old value:

Code	Result
`db['cleese.png'] = 'A silly walk'`	
`db['cleese.png']`	`b'A silly walk'`

Some dictionary methods, like `keys` and `items`, don't work with database objects. But iteration with a `for` loop works:

```
for key in db:
    print(key, db[key])
```

As with other files, you should close the database when you are done:

```
db.close()
```

Pickling

A limitation of `dbm` is that the keys and values have to be strings or bytes. If you try to use any other type, you get an error.

The `pickle` module can help. It's part of the Python standard library, so it's always available. It's fast; the bulk of it is written in C, like the Python interpreter itself. It can store arbitrarily complex Python data structures.

What can the pickle module store?

- All the native datatypes that Python supports: booleans, integers, floating point numbers, strings, bytes objects, byte arrays, and None.
- Lists, tuples, dictionaries, and sets containing any combination of native datatypes.
- Lists, tuples, dictionaries, and sets containing any combination of lists, tuples, dictionaries, and sets containing any combination of native datatypes (and so on, to the maximum nesting level that Python supports).
- Functions (with caveats).

First, we will use the `dumps` method to show you what pickling effectively does to an object.

`pickle.dumps` takes an object as a parameter and returns a string representation (`dumps` is short for "dump string"):

Code	Result

`import pickle`
`t = [1, 2, 3]`
`pickle.dumps(t)`
	`b'\x80\x03]q\x00(K\x01K\x02K\x03e.'`

So what just happened?

The `pickle` module takes a Python data structure and ***serializes*** the data structure using a data format called "the `pickle` protocol."

The `pickle` protocol is Python-specific; there is no guarantee of cross-language compatibility. You probably couldn't take the `shoplistfile` file you just created and do anything useful with it in Perl, PHP, Java, or any other language.

Not every Python data structure can be serialized by the `pickle` module. The `pickle` protocol has changed several times as new data types have been added to the Python language, but there are still limitations.

As a result of these changes, there is no guarantee of compatibility between different versions of Python itself. Newer versions of Python support the older serialization formats, but older versions of Python do not support newer formats (since they don't support the newer data types).

Unless you specify otherwise, the functions in the `pickle` module will use the latest version of the `pickle` protocol. This ensures that you have maximum flexibility in the types of data you can serialize, but it also means that the resulting file will not be readable by older versions of Python that do not support the latest version of the `pickle` protocol.

The latest version of the `pickle` protocol is a binary format. Be sure to open your `pickle` files in binary mode, or the data will get corrupted during writing.

Pickles and Saving to a File

So now that we used the `dumps` method to see what pickling is actually doing to our objects, let's see how we can use the `dump` method (note: no "s" in `dump`) to load it into a file.

The `dump` method accepts a minimum of two arguments: the object that we are pickling and the file in which we are storing the data, like this:

```
pickle.dump(object, file)
```

Here is an example where we take a shopping list and save it for later in a file.

Line	Code	Notes
1	`import pickle`	Import in the `pickle` module.
2	`shoplist = ['apple', 'mango', 'carrot']`	The list of things to buy. Notice that this is a Python `list` object.
3	`shoplistfile = 'shoplist.data'`	The name of the file where we will store the object
4	`fout = open(shoplistfile, 'wb')`	Open the file in write mode. Notice the b after the `'w'` mode – b is for "binary"
5	`pickle.dump(shoplist, fout)`	Dump the object to a file
6	`fout.close()`	Close the file.

Hooray! We successfully pickled the list.

Reading Pickle Data from a File

Now that we have a pickled file, we want to be able to get that data back. We can use the `load` method to accomplish this. First we need to open our file in binary mode. Then we can use the `load` method to "unpickle" the data and make it usable again.

The `load` method accepts the file handler as its argument. Let's continue with our shopping list example.

Line	Code	Notes
7	`fin = open(shoplistfile, 'rb')`	Read back from the storage
8	`storedlist = pickle.load(fin)`	Load the object from the file
9	`print(storedlist)`	Display the list.

Here is our output:

```
['apple', 'mango', 'carrot']
```

It's our original list! We can pick up where we left off and do whatever we want with it.

However, it needs to be noted that this is not identical to the original list – it is a different object. Here's an example where we pickle a list, then unpickle it.

Code	Result

`import pickle`
`t = [1, 2, 3]`
`s = pickle.dumps(t)`
`s`
	`b'\x80\x03]q\x00(K\x01K\x02K\x03e.'`

`t2 = pickle.loads(s)`
`t2`
	`[1, 2, 3]`

Although the new object has the same value as the old, it is not (in general) the same object:

Code	Result
`t1 == t2`	`True`
`t1 is t2`	`False`

In other words, pickling and then unpickling has the same effect as copying the object.

You can use `pickle` to store non-strings in a database. In fact, this combination is so common that it has been encapsulated in a module called `shelve`, although we don't have time to discuss it in this course.

Multiple Pickles in one File

We can store multiple pickles in a single file as well. The caveat here is that you have to remember which order they went in, because it follows a "first-in / first-out" rule:

Code	Result
`import pickle`	
`list1 = ['apple', 'mango', 'carrot']`	
`list2 = ['bread', 'bagel']`	
`list3 = ['cake', 'cookies', 'pie']`	
`fout = open('list.dat', 'wb')`	
`pickle.dump(list1, fout)`	
`pickle.dump(list2, fout)`	
`pickle.dump(list3, fout)`	
`fout.close()`	

That saved each list as a pickle into one file. Now let's extract them:

Code	Result
`fin = open('list.dat', 'rb')`	`['apple', 'mango', 'carrot']`
	`['bread', 'bagel']`
`pickled_list1 = pickle.load(fin)`	`['cake', 'cookies', 'pie']`
`pickled_list2 = pickle.load(fin)`	
`pickled_list3 = pickle.load(fin)`	
`print(pickled_list1)`	
`print(pickled_list2)`	
`print(pickled_list3)`	

First in - first out. You have to keep track of it.

Filenames and Paths

Files are organized into directories (also called "***folders***"). Every running program has a "***current directory***", which is the default directory for most operations. For example, when you open a file for reading, Python looks for it in the current directory.

The `os` module provides functions for working with files and directories ("os" stands for "operating system"). `os.getcwd` returns the name of the current directory:

Code	Output
`import os`	`C:\Python`

```
cwd = os.getcwd()

print(cwd)
```

cwd stands for "current working directory". The result in this example is `C:\Python`.

A string like `'C:\Python'` that identifies a file or directory is called a ***path***.

A simple filename, like `mbox.txt` is also considered a path, but it is a relative path because it relates to the current directory. If the current directory is `C:\Python`, the filename `mbox.txt` would refer to `C:\Python\mbox.txt`.

A path that begins with / does not depend on the current directory; it is called an ***absolute path***. To find the absolute path to a file, you can use `os.path.abspath`:

Code	Output

`import os`
 	
`path = os.path.abspath('mbox.txt')`
 	
`print(path)`
	`C:\Python\mbox.txt`

`os.path` provides other functions for working with filenames and paths. For example, `os.path.exists` checks whether a file or directory exists:

Code	Result

`import os`
 	
`os.path.exists('mbox.txt')`
	`True`

If it exists, `os.path.isdir` checks whether it's a directory:

Code	Result
`os.path.isdir('mbox.txt')`	`False`
`os.path.isdir('C:\\Python')`	`True`

A quick note: notice that the backslash is listed twice in the string `'C:\\Python'` above. Why is that? Well, remember that a backslash is the escape sequence for creating special characters like newlines `\n` and tabs `\t`. When Python sees a backslash, it assumes an escape sequence is coming. As a result, you need to use *two* backslashes for a backslash to display: one is the escape sequence and the other is the character to be displayed.

Similarly, `os.path.isfile` checks whether it's a file.

Code	Result
`os.path.isfile('words.txt')`	`True`
`os.path.isfile('C:\\Python')`	`False`

`os.listdir` returns a list of the files (and other directories) in the given directory:

Code	Result
`os.listdir('C:\\Python')`	`['append.txt', 'DLLs', 'Doc', 'examples.py', 'include', 'labs.py', 'Lib', 'libs', 'LICENSE.txt', 'mbox.txt', 'NEWS.txt', 'output.txt', 'python.exe', 'python3.dll', 'python35.dll', 'Scripts', vcruntime140.dll', 'words.txt']`

To demonstrate these functions, the following example "walks" through a directory, prints the names of all the files, and calls itself recursively on all the directories.

Line	Code	Notes
1	`import os`	Import in the os module.
2	`def walk(dirname):`	Define the `walk` function. It has a single argument: `dirname`.
3	`    for name in os.listdir(dirname):`	The method `listdir` produces a list. We're using a `for` loop to go through each element in the directory called `dirname`.
4	`        path = os.path.join(dirname, name)`	`os.path.join` takes a directory and a file name and joins them into a complete path. We assign this to the variable `path`.
5	`        if os.path.isfile(path):`	We check to see whether `path` is a file using `os.path.isfile`.
6	`            print(path)`	If it is a file, then we display the full path.
7	`        else:`	Otherwise…
8	`            walk(path)`	We run this same function for that directory.

Here is the output to this program when we use `'C:\\Python'` as `dirname`:

```
C:\Python\append.txt
C:\Python\DLLs\py.ico
C:\Python\DLLs\pyc.ico
C:\Python\DLLs\pyexpat.pyd
C:\Python\DLLs\select.pyd
C:\Python\DLLs\sqlite3.dll
(about 1,200 more lines)
```

Catching Exceptions with Files

As we saw from letting our end user type their own file name, a lot of things can go wrong when you try to read and write files. If you try to open a file that doesn't exist, you get an `IOError`:

Code	Result
`fin = open('bad_file')`	`IOError: [Errno 2] No such file or directory: 'bad_file'`

If you don't have permission to access a file:

Code	Result
`fout = open('/etc/passwd', 'w')`	`PermissionError: [Errno 13] Permission denied: '/etc/passwd'`

And if you try to open a directory for reading, you get

Code	Result
`fin = open('/home')`	`IsADirectoryError: [Errno 21] Is a directory: '/home'`

To avoid these errors, you could use functions like `os.path.exists` and `os.path.isfile`, but it would take a lot of time and code to check all the possibilities (if "`Errno 21`" is any indication, there are at least 21 things that can go wrong).

It is better to go ahead and try - and deal with problems if they happen - which is exactly what the `try` statement does. Here is how you might use it with opening a file:

```
try:
    fin = open('bad_file')

except:
    print('Something went wrong.')
```

Python starts by executing the `try` clause. If all goes well, it skips the `except` clause and proceeds. If an exception occurs, it jumps out of the `try` clause and runs the `except` clause.

Handling an exception with a `try` statement is called catching an exception. In this example, the `except` clause prints an error message that is not very helpful. In general, catching an exception gives you a chance to fix the problem, or try again, or at least end the program gracefully.

Go online to complete the Bad File! Challenge

Going back to our example, we need to assume that the open call might fail and add recovery code when the open fails as follows:

```
file_name = input('Enter the file name: ')

try:
    fin = open(file_name)

except:
    print("File cannot be opened:", file_name)

else:
    count = 0

    for line in fin:
        line = line.strip()

        if not line.startswith('Subject:') :
            continue

        else:
            count += 1

    print("There were", count, "subject lines in", file_name)
```

Now when our user (or QA team) types in silliness or bad file names, we "catch" them and recover gracefully:

```
Enter the file name: mbox.txt
There were 1797 subject lines in mbox.txt

Enter the file name: na na boo boo
File cannot be opened: na na boo boo
```

Protecting the open call is a good example of the proper use of `try` and `except` in a Python program. We use the term "Pythonic" when we are doing something the "Python way". We might say that the above example is the Pythonic way to open a file.

Once you become more skilled in Python, you can engage in repartee with other Python programmers to decide which of two equivalent solutions to a problem is "more Pythonic". The goal to be "more Pythonic" captures the notion that programming is part engineering and part art. We are not always interested in just making something work, we also want our solution to be elegant and to be appreciated as elegant by our peers.

Debugging

When you are reading and writing files, you might run into problems with whitespace. These errors can be hard to debug because spaces, tabs, and newlines are normally invisible:

Code

```
s = '1 2\t 3\n 4'
print(s)
```

Output

```
1 2      3
 4
```

The built - in function `repr` can help. It takes any object as an argument and returns a string representation of the object. For strings, it represents whitespace characters with backslash sequences:

Code

```
s = '1 2\t 3\n 4'
print(repr(s))
```

Output

```
'1 2\t 3\n 4'
```

This can be helpful for debugging.

One other problem you might run into is that different systems use different characters to indicate the end of a line. Some systems use a newline, represented `\n`. Others use a return character, represented `\r`. Some use both. If you move files between different systems, these inconsistencies might cause problems.

For most systems, there are applications to convert from one format to another. You can find them (and read more about this issue) at https://en.wikipedia.org/wiki/Newline. Or, of course, you could write one yourself.