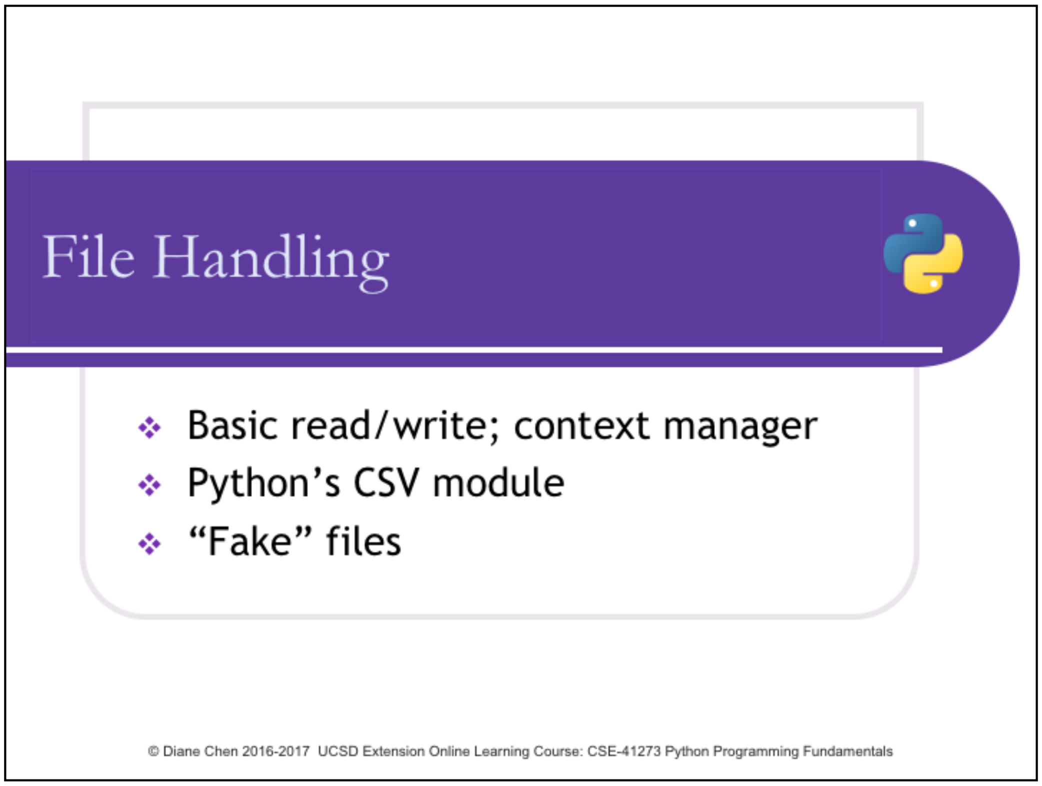
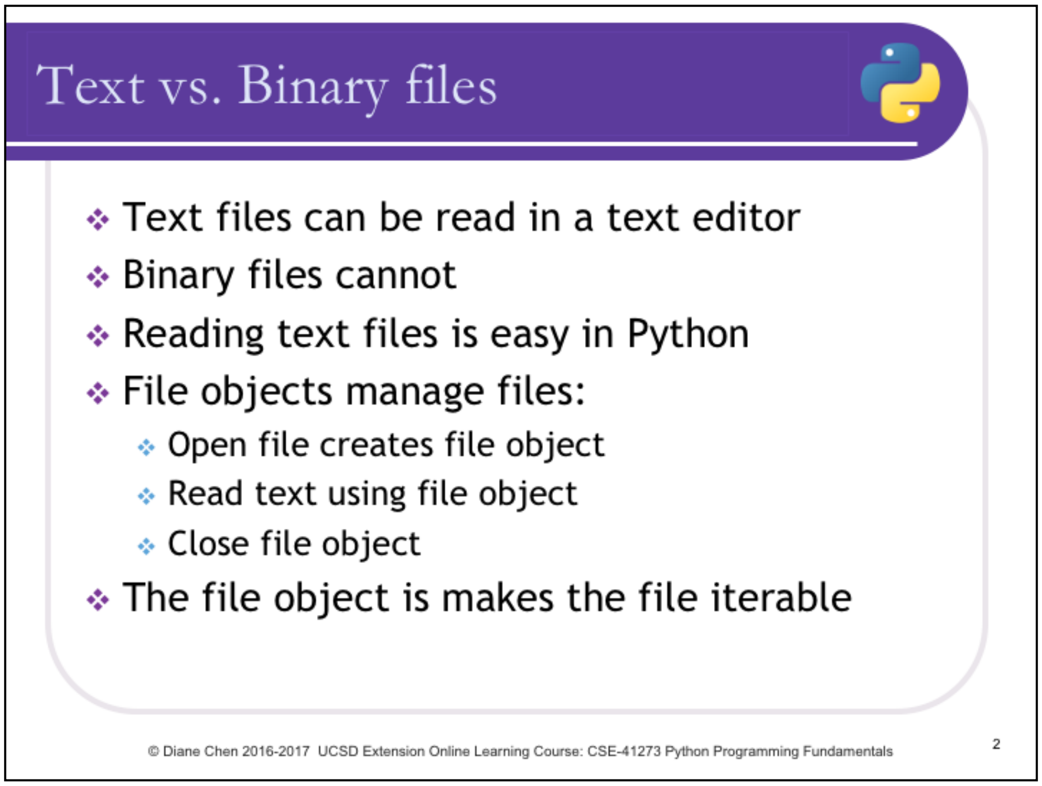


File Handling

- Basic read/write; context manager
- Python's CSV module
- "Fake" files

In this lecture we'll cover basic file handling, Python's context manager for files, and using Python's built-in module for reading/writing CSV files. I will cover the basics of file handling here. I'm not going to discuss dealing with binary files like image files, as that is beyond the scope of the class. I will also discuss "fake" files, namely treating strings as if they were files.



Text vs. Binary files

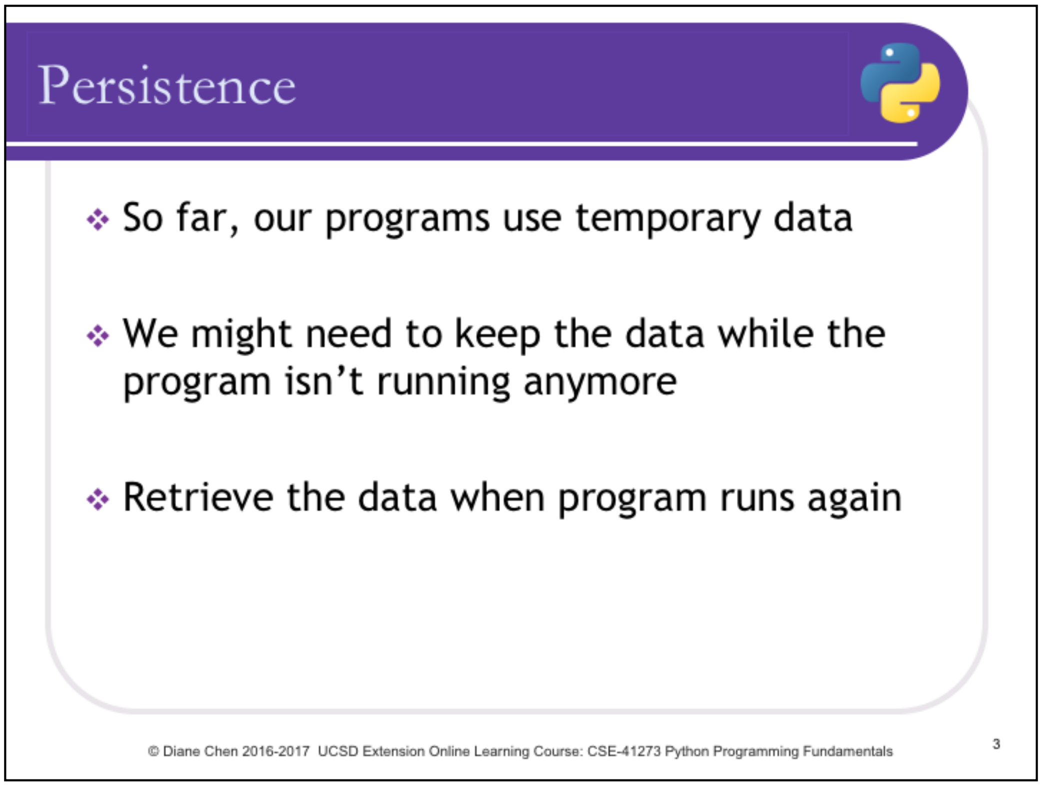
- Text files can be read in a text editor
- Binary files cannot
- Reading text files is easy in Python
- File objects manage files:
  - Open file creates file object
  - Read text using file object
  - Close file object
- The file object is makes the file iterable

2

We're going to primarily cover text files. Text files are files that can be read in a text editor, just like your code. Binary files are files like image files, and cannot be read in a text editor – that is, perhaps the editor will read it in, but it doesn't make any sense to the editor, because the data is just binary bytes rather than readable text.

You can think of files as streams of data. For text files, the data is all characters. Binary files contain binary bytes of raw data that has to be specially interpreted by whatever program is reading the file.

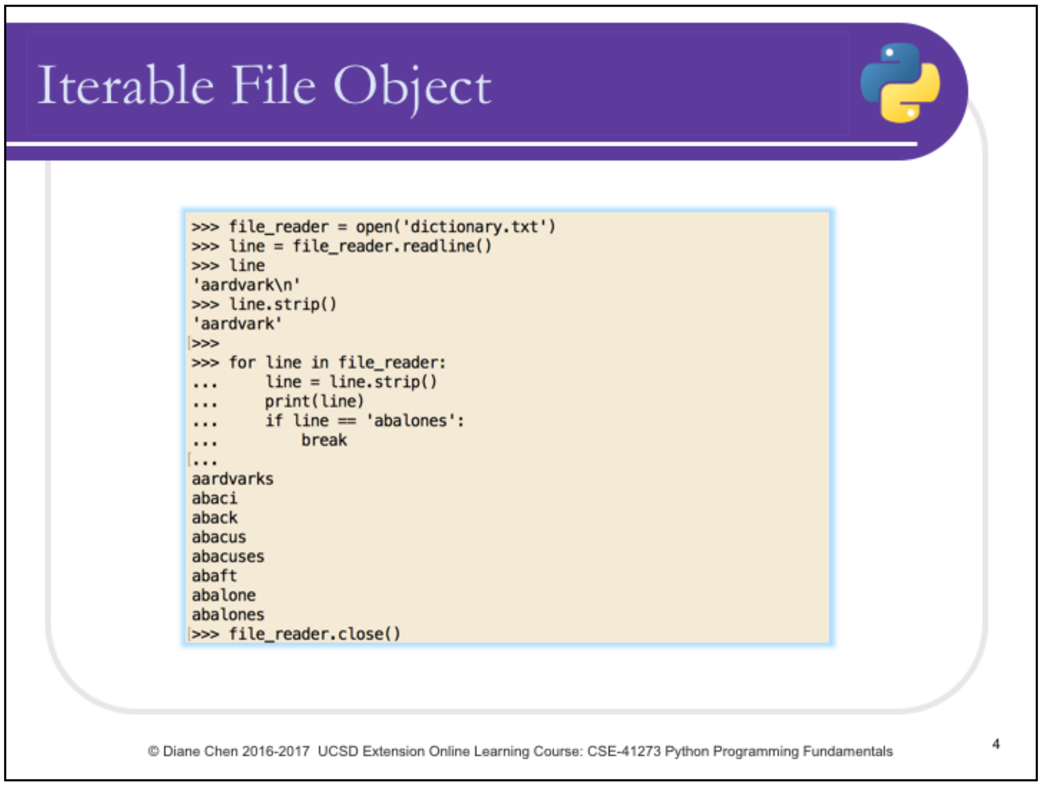
Reading text files is easy in Python. Python has file objects that manage the file and its data. All types of files use file objects, though our examples will be text files. When a file is opened, Python creates the file object. Using the file object, we read the text from the file using one or more of the methods of the file object. When the file is no longer needed, the file is closed using the file object. The cool thing about the file object is that it makes the file iterable.



Persistence

- So far, our programs use temporary data
- We might need to keep the data while the program isn't running anymore
- Retrieve the data when program runs again

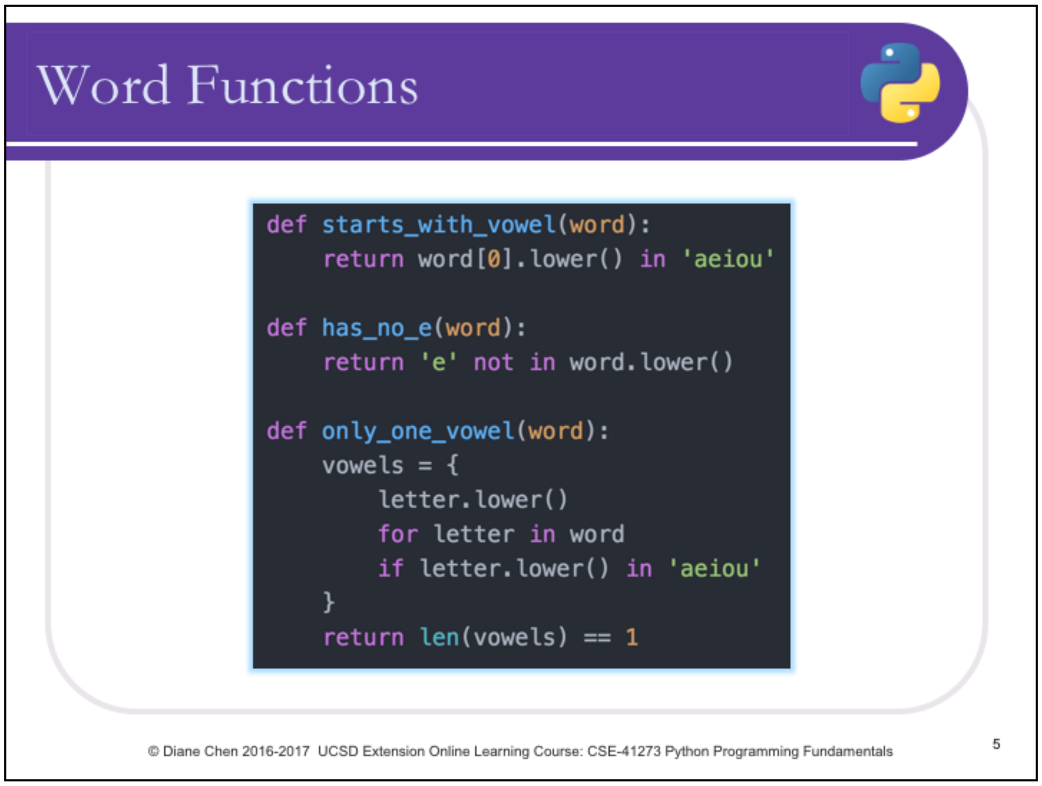
So far, all the programs we have written have just used temporary data or data from the user. What if we have data that we want to be persistent? We want to be able to save our data to files so it will still be there when we start the program again later. Or perhaps the data file is our program's end product, and the file we produce will be read later by some other program or programs. This is where we need to be able to manage the reading and writing of files. I am not going to talk about databases, because that could be a class in itself! However, I will talk about manipulating files of text data.



Iterable File Object

```
>>> file_reader = open('dictionary.txt')
>>> line = file_reader.readline()
>>> line
'aardvark\n'
>>> line.strip()
'aardvark'
>>>
>>> for line in file_reader:
...     line = line.strip()
...     print(line)
...     if line == 'abalones':
...         break
...
aardvarks
abaci
aback
abacus
abacuses
abaft
abalone
abalones
>>> file_reader.close()
```

Here is a simple example. I have a file 'dictionary.txt' that contains over 75,000 words. To open the file, we create the file object file_reader. One of the methods of the file object is readline(). It reads the file until it finds a line break. Here we see that the line as read in contains the word 'arardvark' with the newline character represented by backslash n. If you are on a Windows machine, you might see backslash r and backslash n together, as this is the default line endings for Windows. The string method strip() removes whitespace from the beginning and end of the string. Newlines and line terminators are treaded as whitespace. So if we use strip() on the line, we get just the word 'aardvark'. Now we iterate over the file object to get the words one at a time. Note we still have to call the strip() method on each line we get. Since I didn't want to print all of the words in the file, I just arbitarily check for 'abalones' and then stop. Note that it started printing words after 'aardvark'. The file_reader object acts as an iterator. If I repeated the for loop, it would start after 'abalones' and print all the rest of the words in the file. We don't want that! When we're done, we call file_reader.close() to close the file.

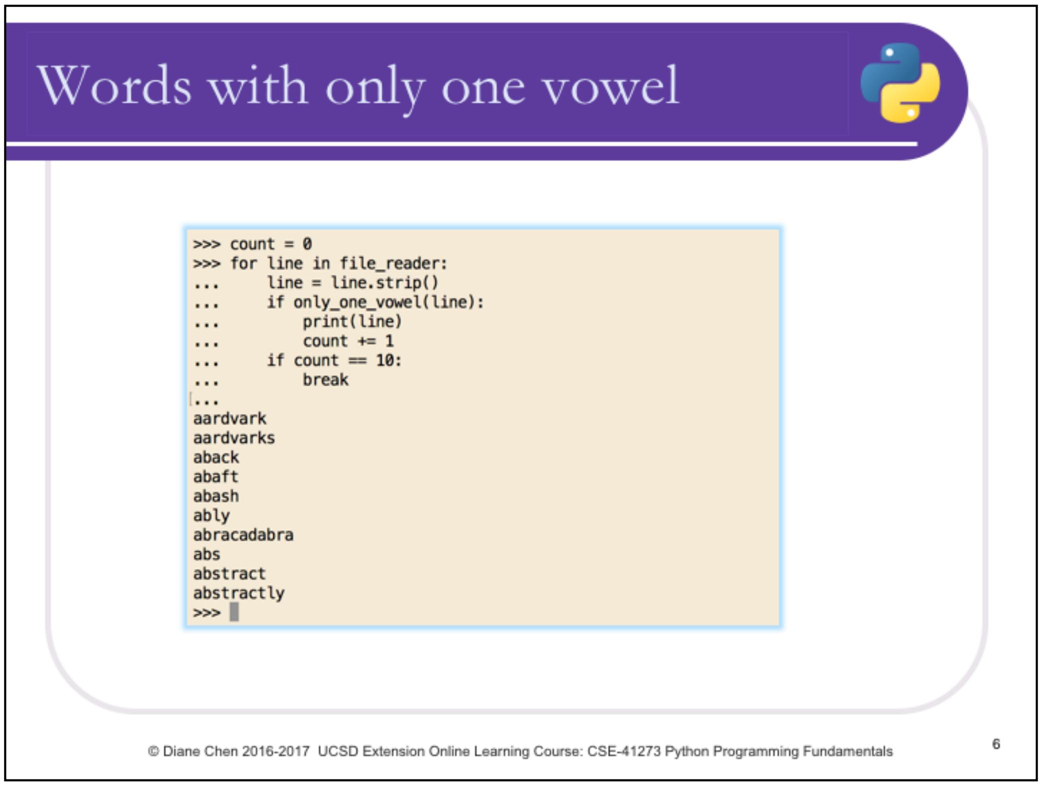


Word Functions

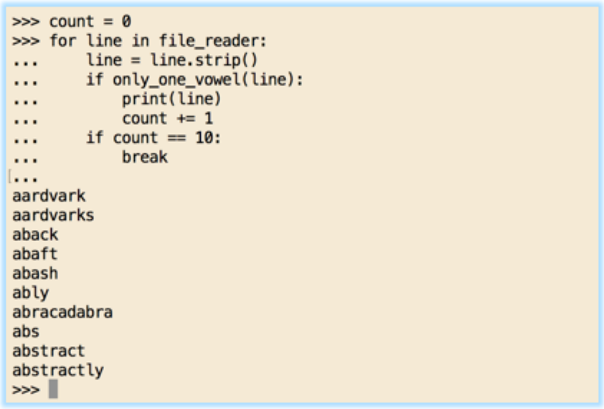
```python
def starts_with_vowel(word):
    return word[0].lower() in 'aeiou'

def has_no_e(word):
    return 'e' not in word.lower()

def only_one_vowel(word):
    vowels = {
        letter.lower()
        for letter in word
        if letter.lower() in 'aeiou'
    }
    return len(vowels) == 1
```

© Diane Chen 2016-2017 UCSD Extension Online Learning Course: CSE-41273 Python Programming Fundamentals 5

Here I've created some simple functions that take a word as input and return True or False based on some test. So I have starts_with_vowel(), has_no_e(), and only_one_vowel(), meaning that the word has only one vowel letter, although it can have more than one of that letter. For only_one_vowel, note the use of a set comprehension. I put any vowel I find into a set, and then check the size of the set. Not shown here, I also created has_5_vowels(), which is just like only_one_vowel() except for the test on the last line. I use word.lower() to make the functions work with any input, even though the dictionary file contains only lower case words. Instead of using the lower() method, I could have made the string we check against contain both lower and upper case vowels. I can't really say which way is better, I am used to working with words by lower-casing them, so it was normal to do that for me.

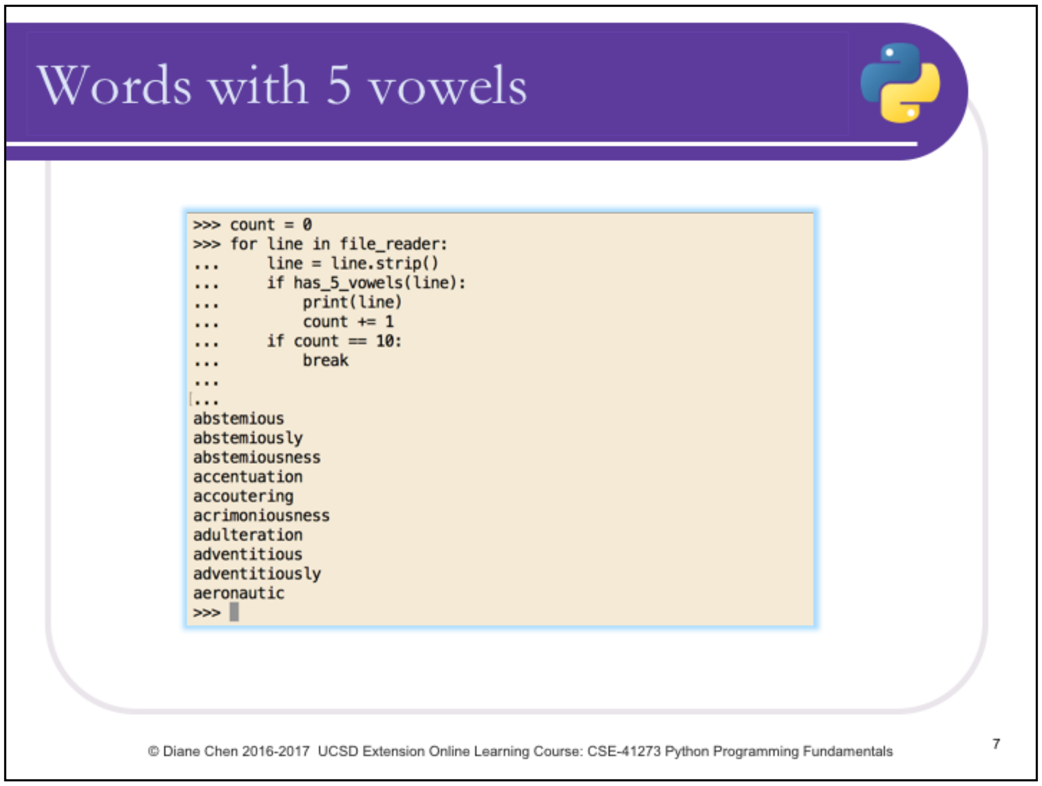


Words with only one vowel

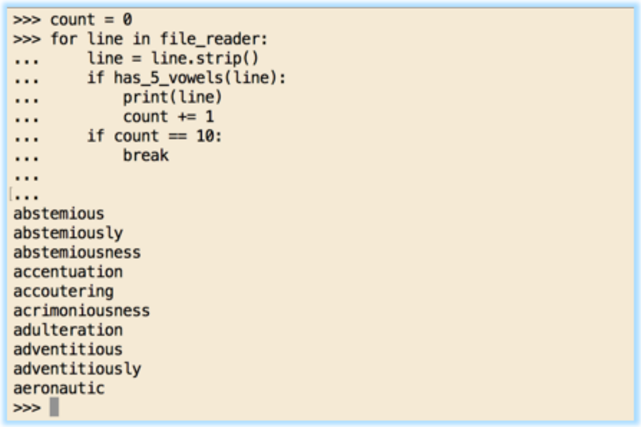


```
>>> count = 0
>>> for line in file_reader:
...     line = line.strip()
...     if only_one_vowel(line):
...         print(line)
...         count += 1
...     if count == 10:
...         break
...
aardvark
aardvarks
aback
abaft
abash
ably
abracadabra
abs
abstract
abstractly
>>> 
```

Here we see the first 10 words we find that have only one vowel.



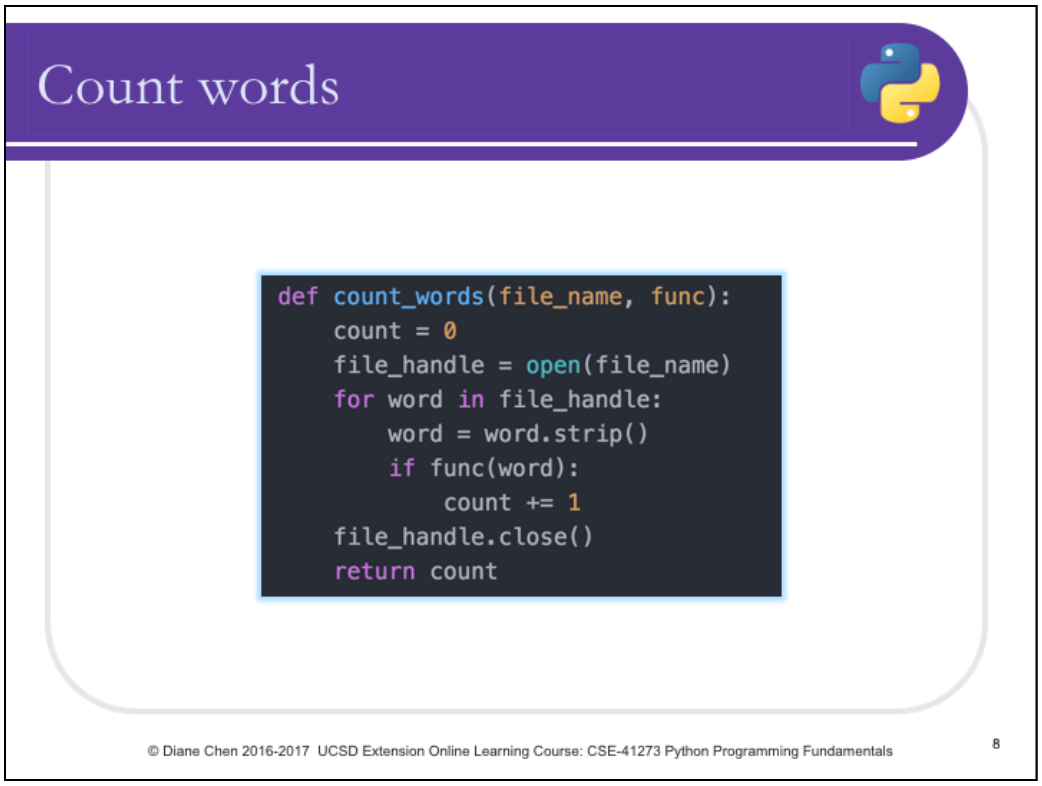
Words with 5 vowels



```
>>> count = 0
>>> for line in file_reader:
...     line = line.strip()
...     if has_5_vowels(line):
...         print(line)
...         count += 1
...     if count == 10:
...         break
...
...
abstemious
abstemiously
abstemiousness
accentuation
accoutering
acrimoniousness
adulteration
adventitious
adventitiously
aeronautic
>>> 
```

7

Here we see the first 10 words that have all 5 vowels. Note that I closed and re-opened the file between slides, so that we would start from the beginning.

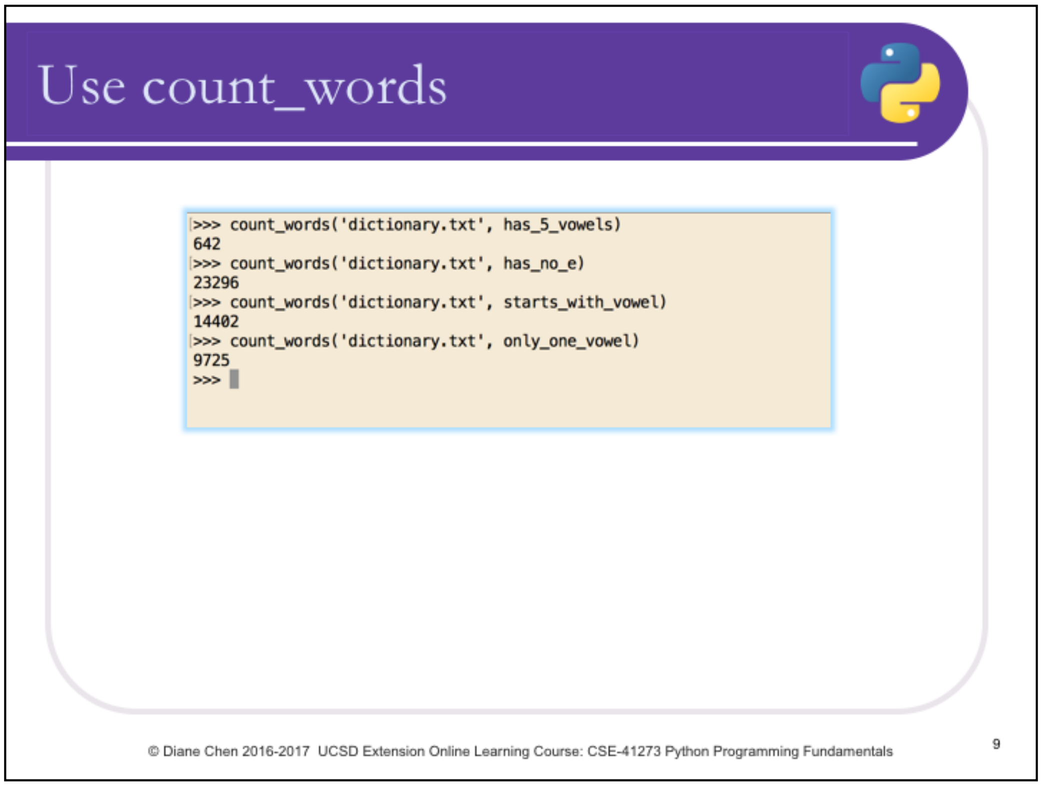


Count words

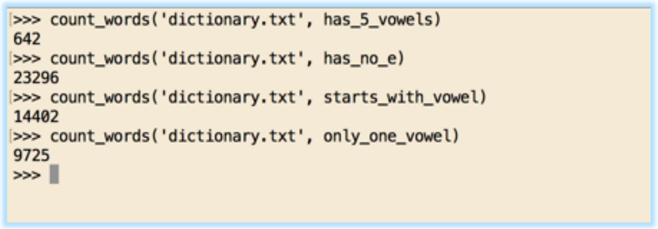
```python
def count_words(file_name, func):
    count = 0
    file_handle = open(file_name)
    for word in file_handle:
        word = word.strip()
        if func(word):
            count += 1
    file_handle.close()
    return count
```

© Diane Chen 2016-2017 UCSD Extension Online Learning Course: CSE-41273 Python Programming Fundamentals 8

Just printing the first 10 words we find that fit our criteria isn't very interesting, so here we have a function, count_words, that takes a file name and a function as input. The function should take a word as input and return True or False - True if the word should be counted, False if not. It opens the file, counts the words that should be counted, then closes the file and returns the count. Now this is a handy function for us to use.

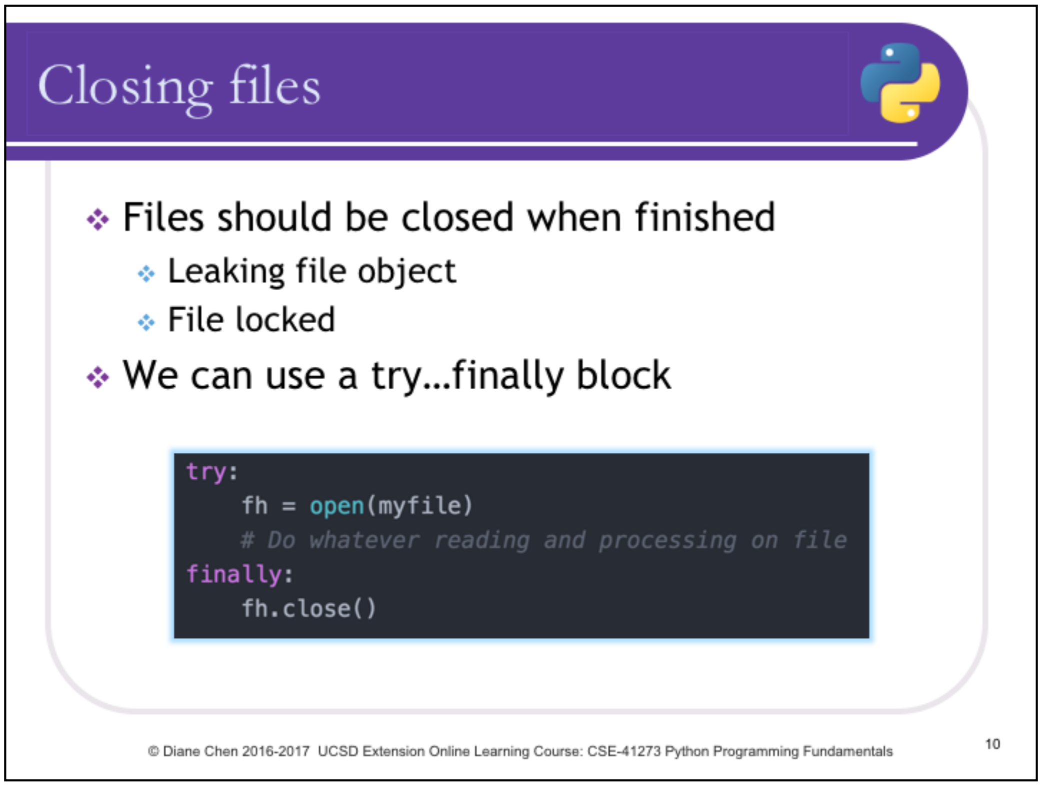


Use count_words



```
>>> count_words('dictionary.txt', has_5_vowels)
642
>>> count_words('dictionary.txt', has_no_e)
23296
>>> count_words('dictionary.txt', starts_with_vowel)
14402
>>> count_words('dictionary.txt', only_one_vowel)
9725
>>>
```

Here we see the counts of the number of words that have all 5 vowels, have no letter e, start with a vowel, and have only one vowel.



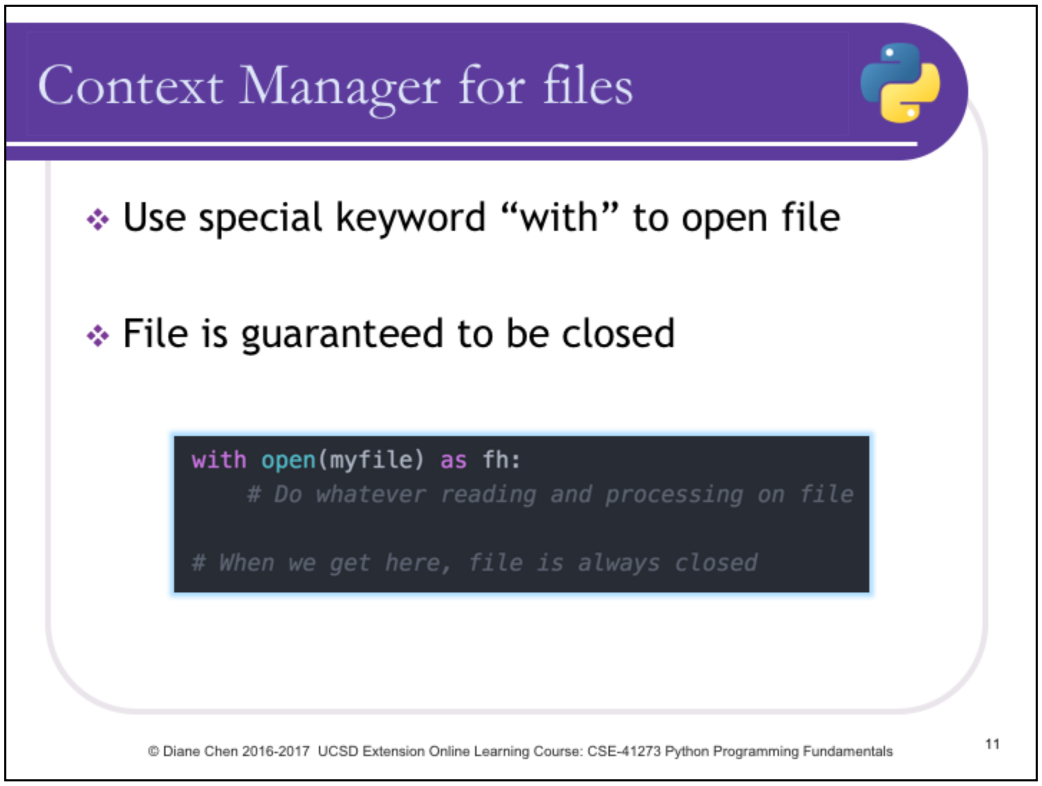
Closing files

- Files should be closed when finished
  - Leaking file object
  - File locked
- We can use a try…finally block

```
try:
    fh = open(myfile)
    # Do whatever reading and processing on file
finally:
    fh.close()
```

© Diane Chen 2016-2017 UCSD Extension Online Learning Course: CSE-41273 Python Programming Fundamentals 10

It's important when manipulating files to make sure that the file gets closed when we are done with it. This needs to happen even – and perhaps especially – when an error occurs during processing of the file, either the reading or the writing of the file. In a large system where we might be using lots of files, not closing a file when done with it means that Python can't delete the file object because it doesn't know you're done with it. It may cause the program's memory usage to bloat if we use a file multiple times, but have some error that causes the file not to get closed properly. This can be a big problem, we call this memory leaks. In addition, depending on how the the operating system works, it may lock the file if it is in write mode, and then no one else can open it. That's OK if the program just opens the file for write access, writes something, and closes the file. But if there is an error and the file isn't closed, it's really annoying to have to go manually fix the file access. We could use a try... finally block, since if there is some error, the finally block will always be executed. But it is redundant to always be including the finally block just to close a file when we know we will always be doing that.



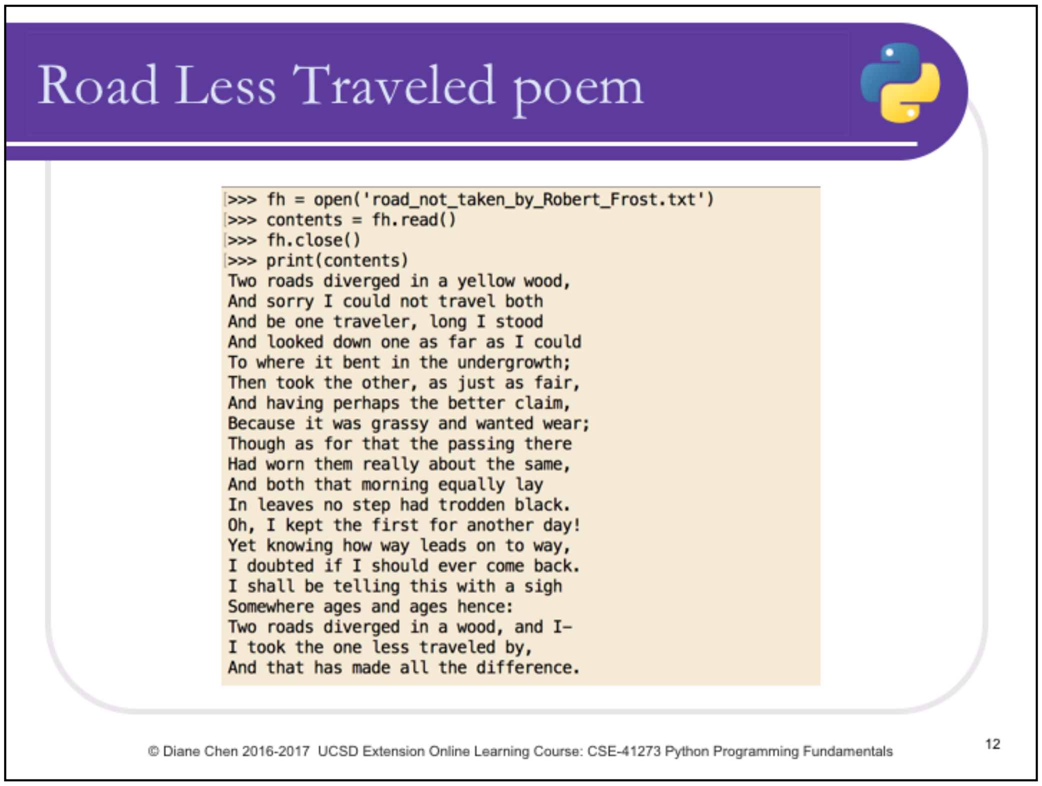
Context Manager for files

- Use special keyword "with" to open file
- File is guaranteed to be closed

```python
with open(myfile) as fh:
    # Do whatever reading and processing on file

# When we get here, file is always closed
```

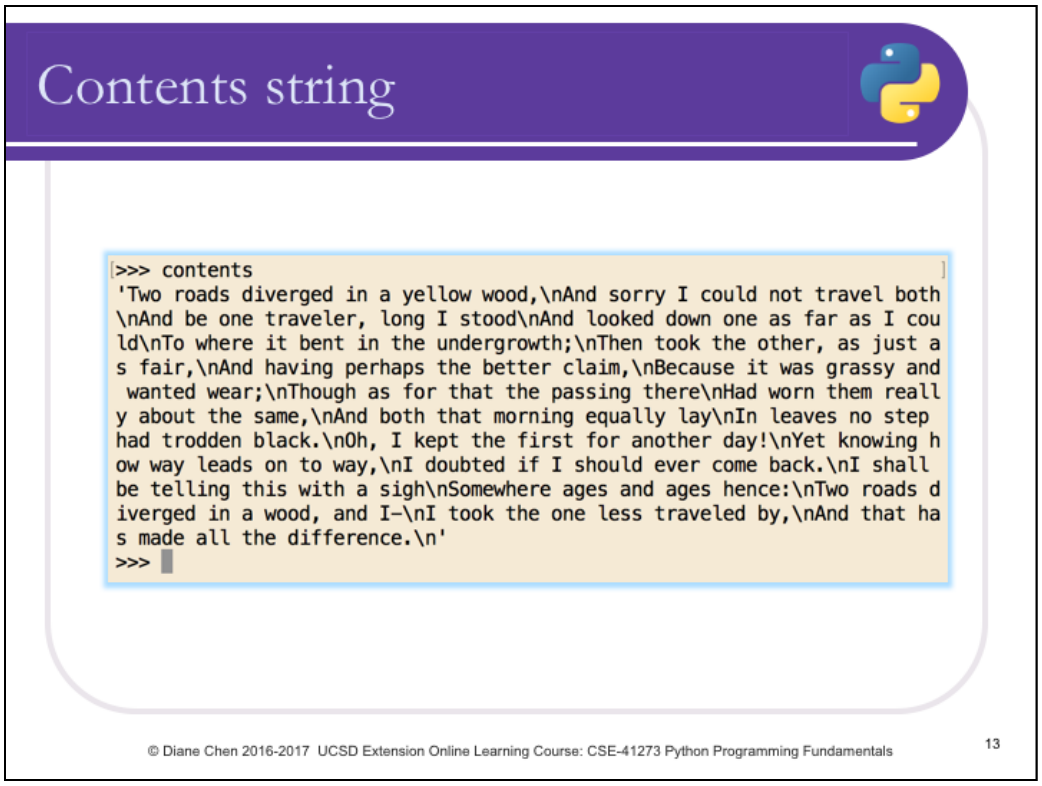
© Diane Chen 2016-2017 UCSD Extension Online Learning Course: CSE-41273 Python Programming Fundamentals 11

Python has it covered! Python has what is called context managers. We briefly saw a context manager in the lesson about unit testing. A context manager is used wherever you need some cleanup (such as closing a file, disconnecting from a database, or handling the test errors properly), and need to guarantee it always happens, even when unexpected errors occur. The file open function has a context manager built in. The keyword "with" tells Python we have a context manager. The file open as we used it in the previous slide returns a file object. This is what gets put into the variable following the "as" keyword – it is getting the file object from open. All of the file handling code goes into the block following the "with" statement. The file open() context manager always makes sure that the file gets closed when leaving the "with" code block. Best practices for file handling is to have the file open as little as possible to get the job done. Namely, get everything ready, open the file with the context manager, do whatever you need to do with the file, then exit the "with" code block. It's also possible to make your own context manager whenever you might need one. That's a bit of an advanced topic so we won't go into that for this class.


Road Less Traveled poem

```
>>> fh = open('road_not_taken_by_Robert_Frost.txt')
>>> contents = fh.read()
>>> fh.close()
>>> print(contents)
Two roads diverged in a yellow wood,
And sorry I could not travel both
And be one traveler, long I stood
And looked down one as far as I could
To where it bent in the undergrowth;
Then took the other, as just as fair,
And having perhaps the better claim,
Because it was grassy and wanted wear;
Though as for that the passing there
Had worn them really about the same,
And both that morning equally lay
In leaves no step had trodden black.
Oh, I kept the first for another day!
Yet knowing how way leads on to way,
I doubted if I should ever come back.
I shall be telling this with a sigh
Somewhere ages and ages hence:
Two roads diverged in a wood, and I—
I took the one less traveled by,
And that has made all the difference.
```

Here I've opened a file containing a poem by Robert Frost. The read() method reads the whole file in one gulp, saving it into the string contents. When I print contents, the poem looks normal. But it actually is just one long string with line breaks in it.

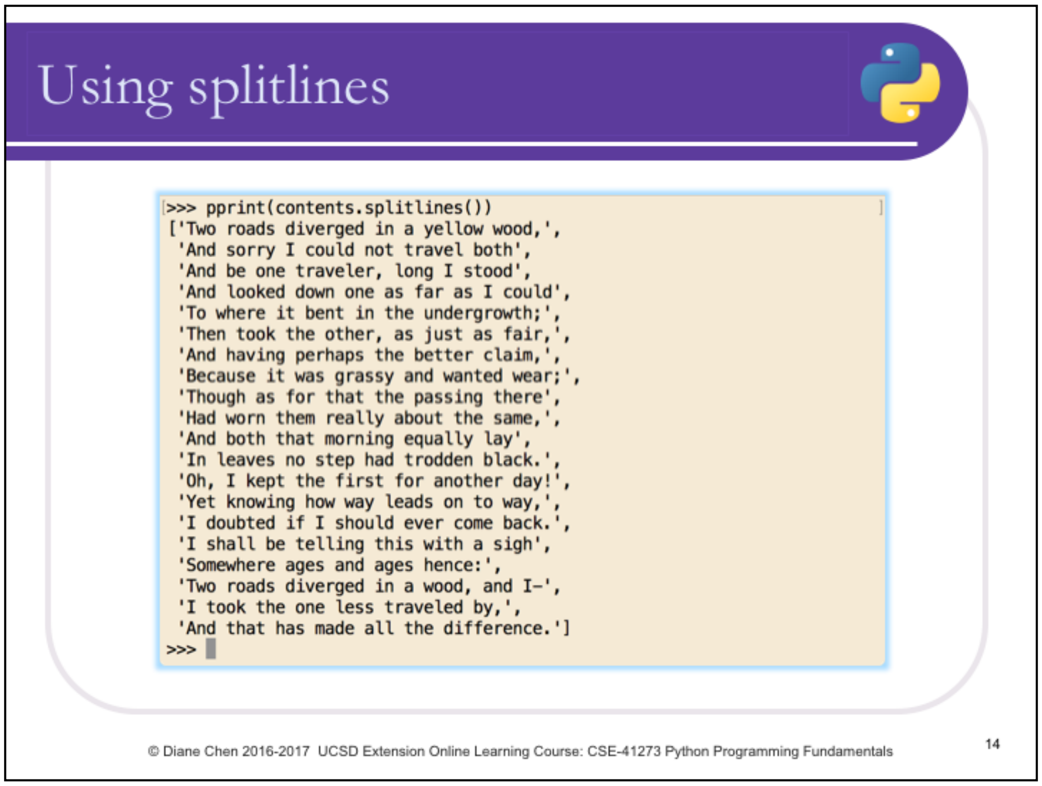


Contents string

```
>>> contents
'Two roads diverged in a yellow wood,\nAnd sorry I could not travel both
\nAnd be one traveler, long I stood\nAnd looked down one as far as I cou
ld\nTo where it bent in the undergrowth;\nThen took the other, as just a
s fair,\nAnd having perhaps the better claim,\nBecause it was grassy and
 wanted wear;\nThough as for that the passing there\nHad worn them reall
y about the same,\nAnd both that morning equally lay\nIn leaves no step
had trodden black.\nOh, I kept the first for another day!\nYet knowing h
ow way leads on to way,\nI doubted if I should ever come back.\nI shall
be telling this with a sigh\nSomewhere ages and ages hence:\nTwo roads d
iverged in a wood, and I-\nI took the one less traveled by,\nAnd that ha
s made all the difference.\n'
>>> 
```

13

Here is the __repr__ display of the string that displays the exact value of the variable contents. As we saw before, the backslash-n characters are newlines.



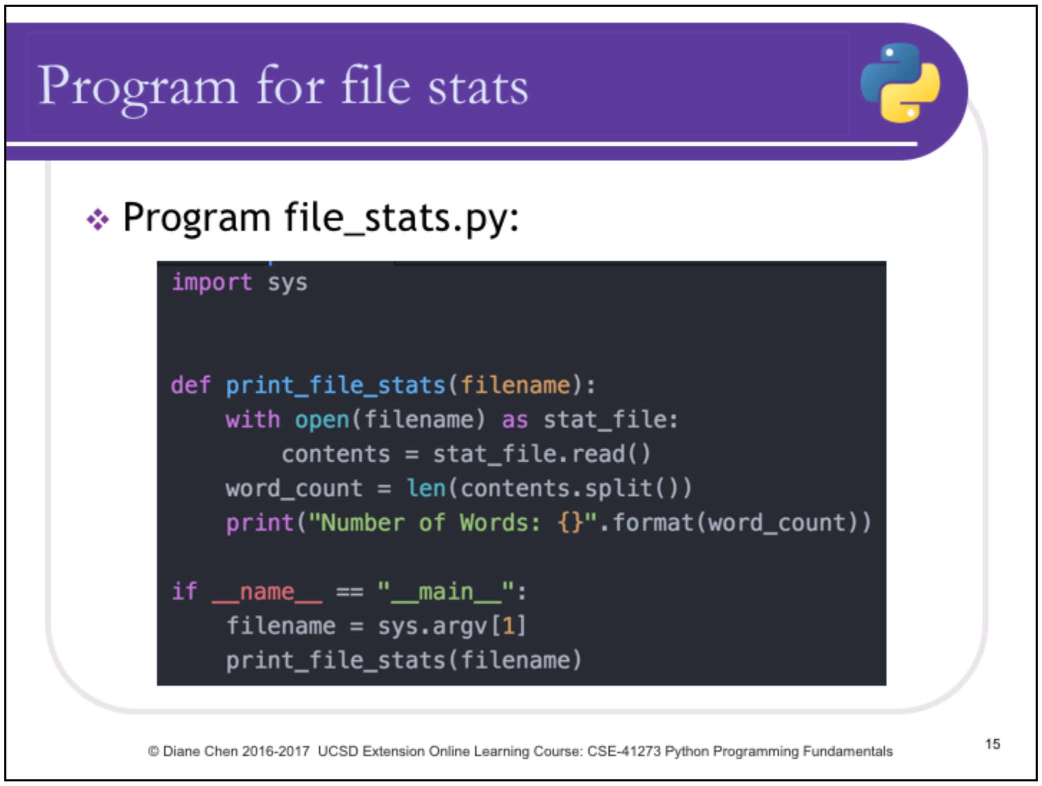
Using splitlines

```
>>> pprint(contents.splitlines())
['Two roads diverged in a yellow wood,',
 'And sorry I could not travel both',
 'And be one traveler, long I stood',
 'And looked down one as far as I could',
 'To where it bent in the undergrowth;',
 'Then took the other, as just as fair,',
 'And having perhaps the better claim,',
 'Because it was grassy and wanted wear;',
 'Though as for that the passing there',
 'Had worn them really about the same,',
 'And both that morning equally lay',
 'In leaves no step had trodden black.',
 'Oh, I kept the first for another day!',
 'Yet knowing how way leads on to way,',
 'I doubted if I should ever come back.',
 'I shall be telling this with a sigh',
 'Somewhere ages and ages hence:',
 'Two roads diverged in a wood, and I-',
 'I took the one less traveled by,',
 'And that has made all the difference.']
>>> 
```

14

There is a string method, splitlines() that takes a string and returns a list where each element of the list is one line of the string. We could have done this ourselves by iterating over the file using readlines(), like what we did with count_words.py, and adding each line to a list, stripping the newlines.

Here I've printed it out using pretty print.



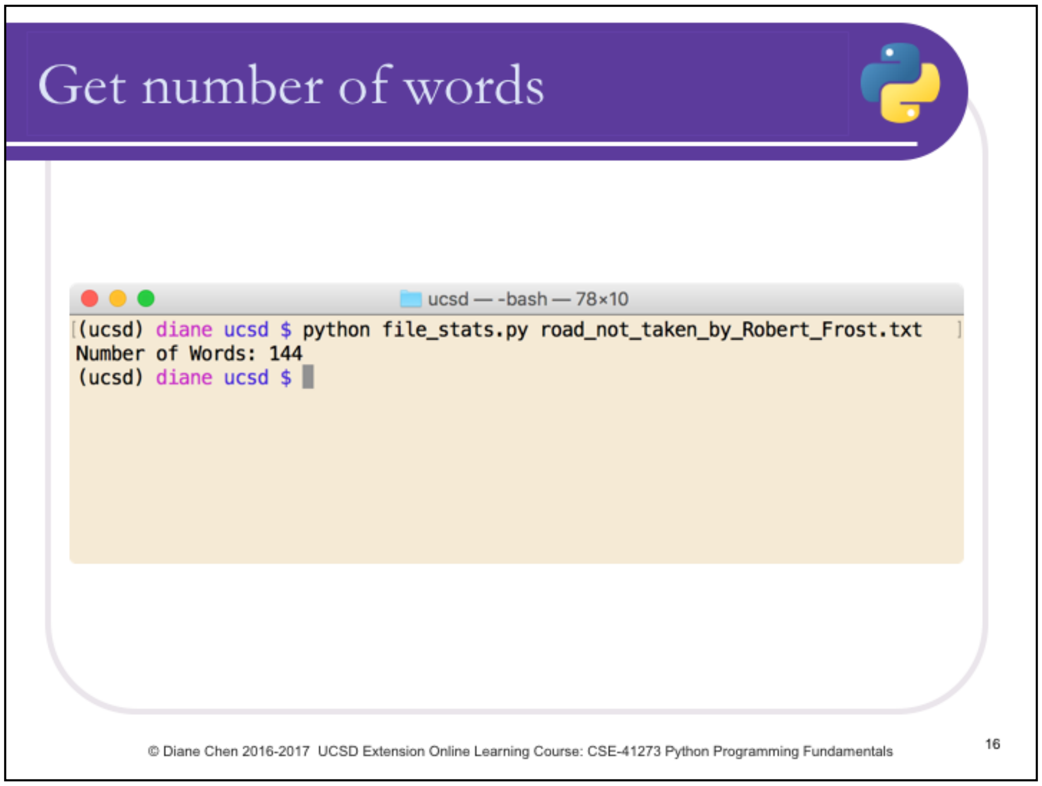
Program for file stats

- Program file_stats.py:

```
import sys


def print_file_stats(filename):
    with open(filename) as stat_file:
        contents = stat_file.read()
    word_count = len(contents.split())
    print("Number of Words: {}".format(word_count))

if __name__ == "__main__":
    filename = sys.argv[1]
    print_file_stats(filename)
```

15

Here I've made a command line program, file_stats.py, to print the number of words in a given file. Again, we read the whole file into one variable. We use the split() method because it splits on any whitespace. Newlines are considered whitespace.

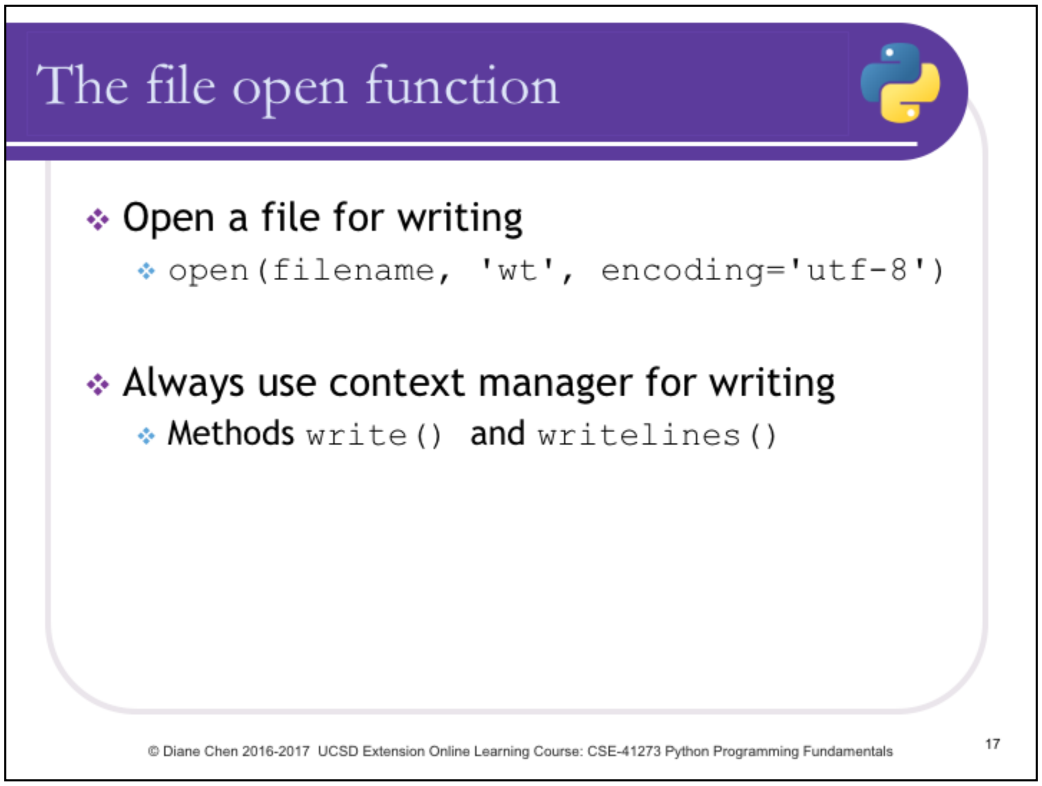


Get number of words

```
ucsd — -bash — 78×10
(ucsd) diane ucsd $ python file_stats.py road_not_taken_by_Robert_Frost.txt
Number of Words: 144
(ucsd) diane ucsd $
```

16

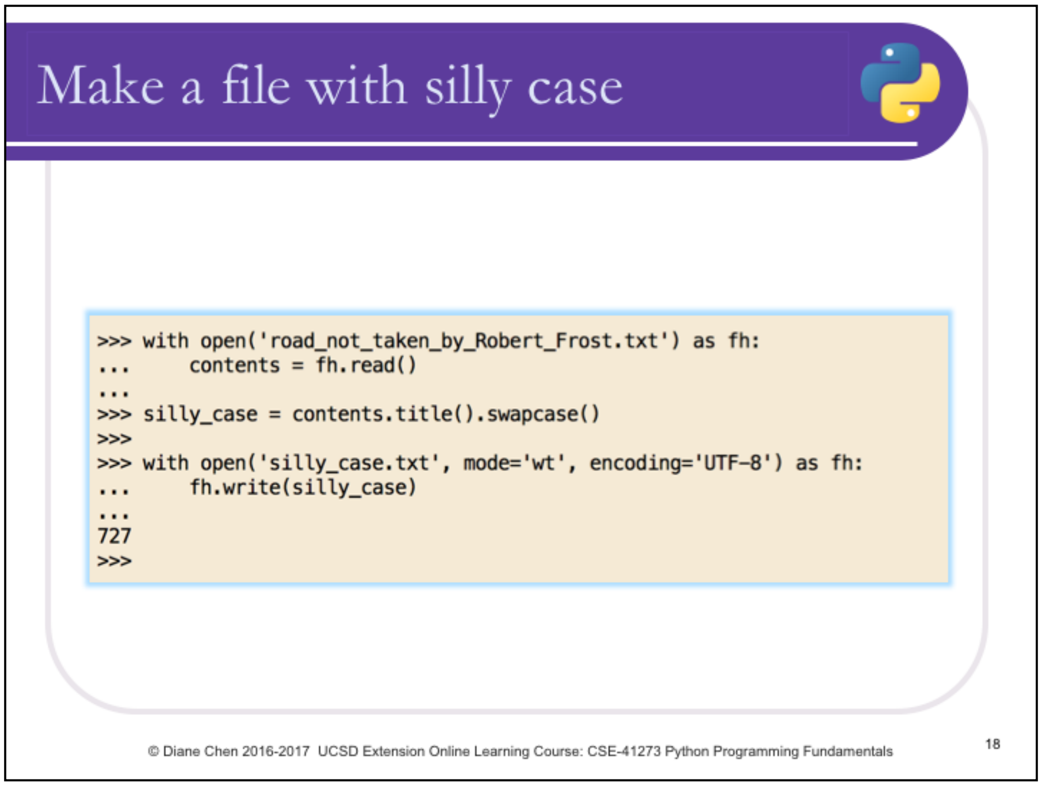
And, here is the output from file_stats.py when we run it on the poem file. It tells us there are 144 words in the poem.



The file open function

- Open a file for writing
  - `open(filename, 'wt', encoding='utf-8')`
- Always use context manager for writing
  - Methods `write()` and `writelines()`

When we want to open a file to write to it, we need to add the mode argument of ‘wt’, which stands for ‘write text’. The default mode is ‘r’ which is a synonym for ‘rt’ or ‘read text’. That’s why we didn’t use the mode parameter before, because we were just reading text files. File encoding can be confusing and sometimes is a mess. Fortunately Python 3 uses Unicode by default for strings. This is one of the big changes from Python 2 to Python 3. Python 2 did not use Unicode by default. The file encoding of ‘utf-8’ is pretty standard these days. I highly recommend sticking with it unless you have good reasons for using something else. Always use a context manager for writing to files. The basic methods to write with are write() and writelines(). Write() writes one string while writelines writes to the file from an iterable containing strings.

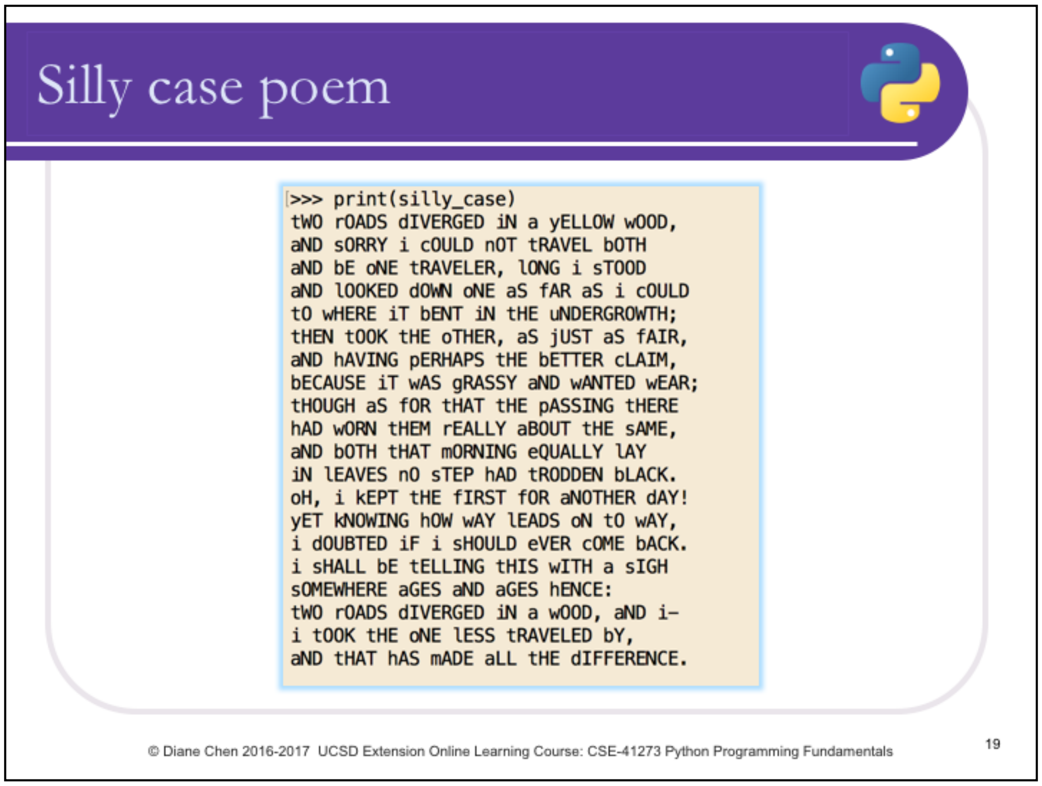


Make a file with silly case

```
>>> with open('road_not_taken_by_Robert_Frost.txt') as fh:
...     contents = fh.read()
...
>>> silly_case = contents.title().swapcase()
>>>
>>> with open('silly_case.txt', mode='wt', encoding='UTF-8') as fh:
...     fh.write(silly_case)
...
727
>>>
```

18

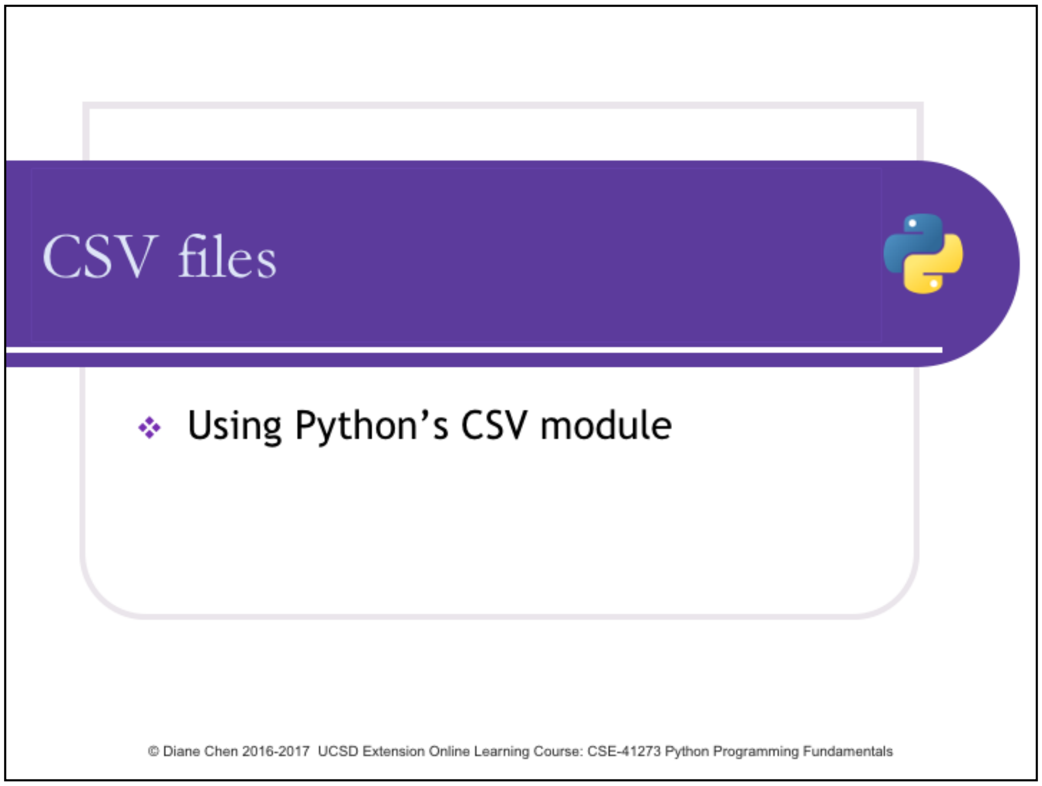
Let's take the poem file and make a file silly_case.txt, where all the words have been converted to start with a lowercase letter and have all the other letters uppercase. We read the file into one string with the read() method. Then create the silly_case variable as a string from contents where the title() and swapcase() methods have been applied to it. Then we use the write() method to write the whole string at one time. The number 727 comes from the write method – it is telling us that it wrote 727 characters.



Silly case poem

```
>>> print(silly_case)
tWO rOADS dIVERGED iN a yELLOW wOOD,
aND sORRY i cOULD nOT tRAVEL bOTH
aND bE oNE tRAVELER, lONG i sTOOD
aND lOOKED dOWN oNE aS fAR aS i cOULD
tO wHERE iT bENT iN tHE uNDERGROWTH;
tHEN tOOK tHE oTHER, aS jUST aS fAIR,
aND hAVING pERHAPS tHE bETTER cLAIM,
bECAUSE iT wAS gRASSY aND wANTED wEAR;
tHOUGH aS fOR tHAT tHE pASSING tHERE
hAD wORN tHEM rEALLY aBOUT tHE sAME,
aND bOTH tHAT mORNING eQUALLY lAY
iN lEAVES nO sTEP hAD tRODDEN bLACK.
oH, i kEPT tHE fIRST fOR aNOTHER dAY!
yET kNOWING hOW wAY lEADS oN tO wAY,
i dOUBTED iF i sHOULD eVER cOME bACK.
i sHALL bE tELLING tHIS wITH a sIGH
sOMEWHERE aGES aND aGES hENCE:
tWO rOADS dIVERGED iN a wOOD, aND i-
i tOOK tHE oNE lESS tRAVELED bY,
aND tHAT hAS mADE aLL tHE dIFFERENCE.
```

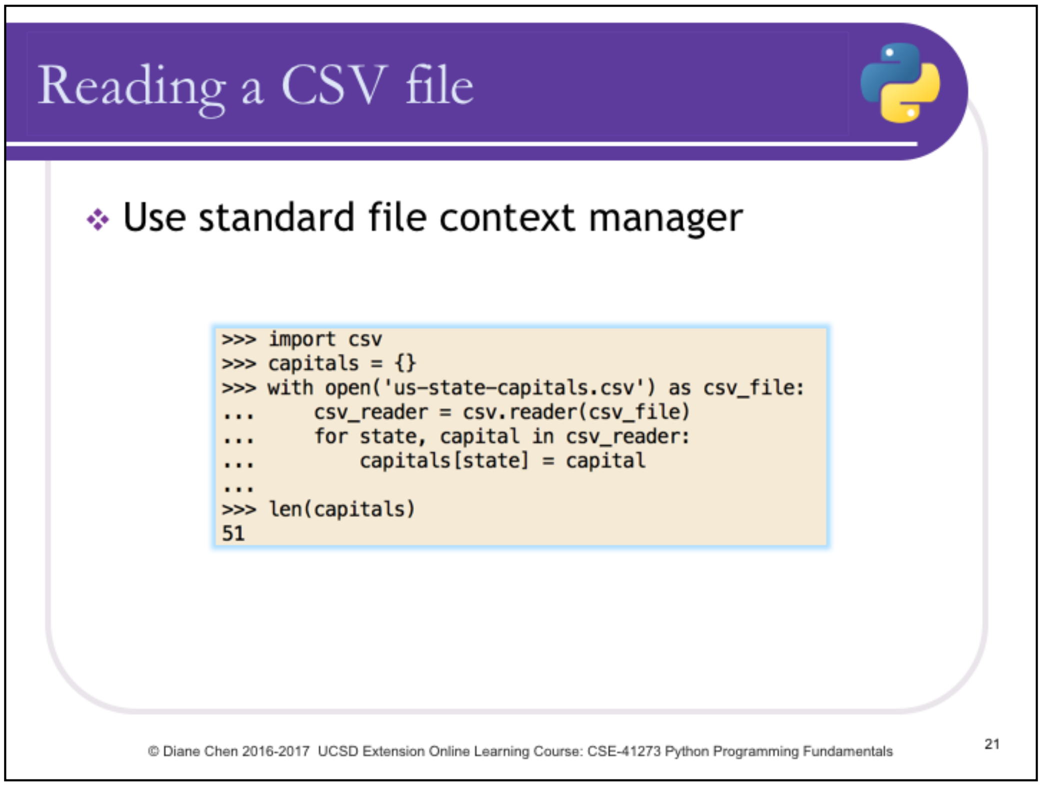
And, just for fun, because it is very silly, here's the poem in sillly case.



CSV files

- Using Python's CSV module

Python has a csv module that greatly aids in reading and writing csv files. CSV stands for "Comma-Separated Values", and is a format that is typically used for storing tabular data, such as data in a spreadsheet. Although the CSV format is widely used, it is not completely formalized, but generally each record (a "row") contains a fixed number of items or fields that are separated by commas (the default delimiter). It is possible to have csv files that use a different delimiter, typically a space or tab character, but it can be anything. There is also a need for a quote character, in case the item in a field contains the delimiter. The default for that is a double quote. And remember, as it is a text file, everything in the file is characters.

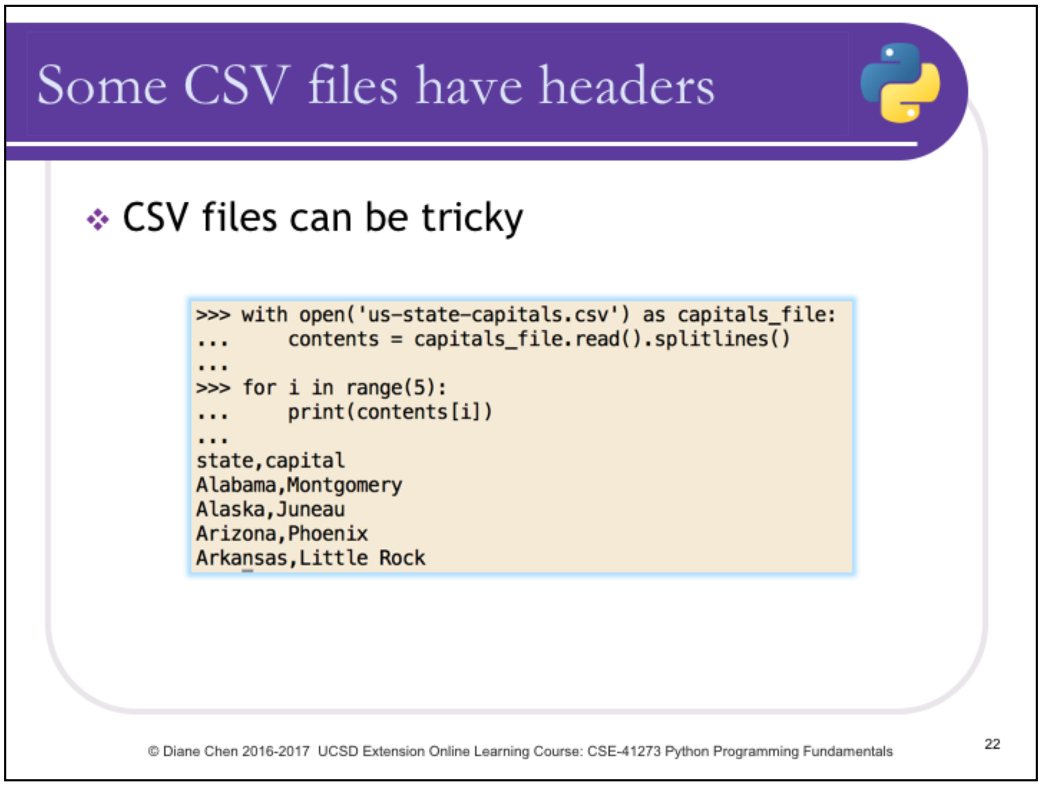


Reading a CSV file

- Use standard file context manager

```
>>> import csv
>>> capitals = {}
>>> with open('us-state-capitals.csv') as csv_file:
...     csv_reader = csv.reader(csv_file)
...     for state, capital in csv_reader:
...         capitals[state] = capital
...
>>> len(capitals)
51
```

I have a csv file containing all the US states and their capitals. To read a csv file using the csv module, use the standard file context manager. Then, within the context manager, we use the csv.reader() function to make a csv_reader object, which is an iterator. The csv_reader object handles all of the parsing of the lines in the file and separating the fields based on the delimiters, etc. Then, we only need to iterate over the csv_reader object, extracting the state and capital from each entry and putting them into a dictionary. But what's this? When we are done, we have 51 states and capitals? What is going on here?

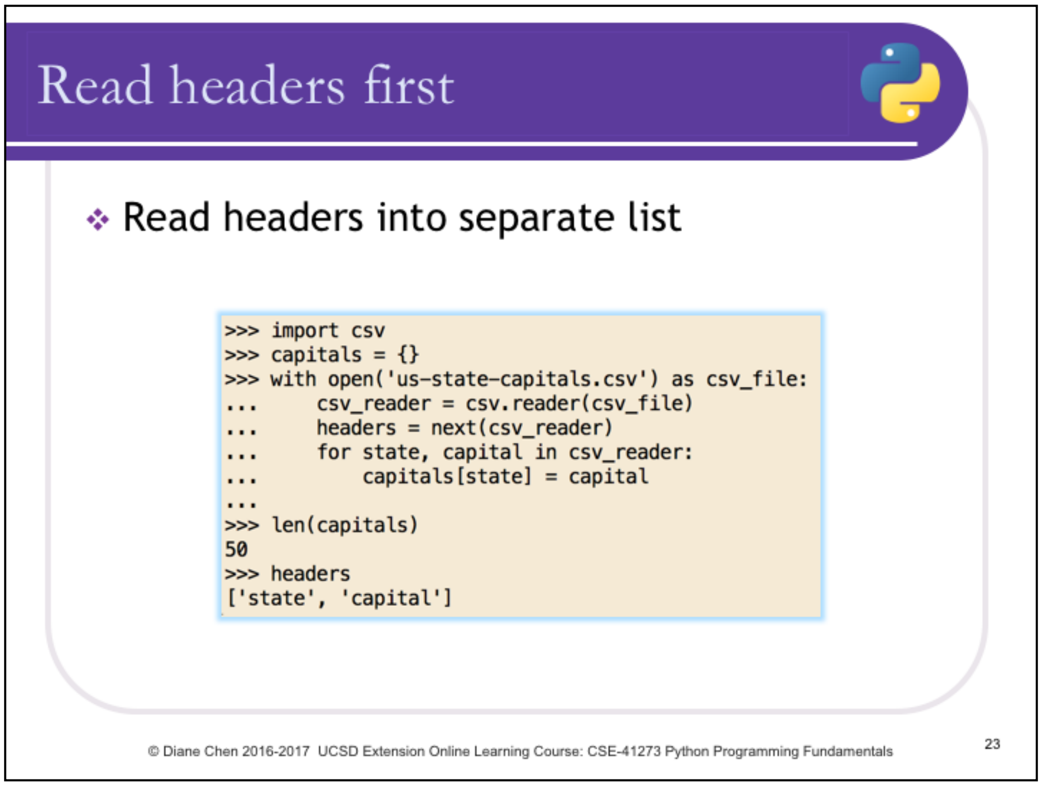


Some CSV files have headers

- CSV files can be tricky

```
>>> with open('us-state-capitals.csv') as capitals_file:
...     contents = capitals_file.read().splitlines()
...
>>> for i in range(5):
...     print(contents[i])
...
state,capital
Alabama,Montgomery
Alaska,Juneau
Arizona,Phoenix
Arkansas,Little Rock
```

Let’s look at the first 5 lines of the file. We read the entire file with the read() method, then use splitlines on it. Then we print the first 5 lines. This csv file has what we call a “header” containing the field names for the fields in the remaining file. This is very common in csv files.

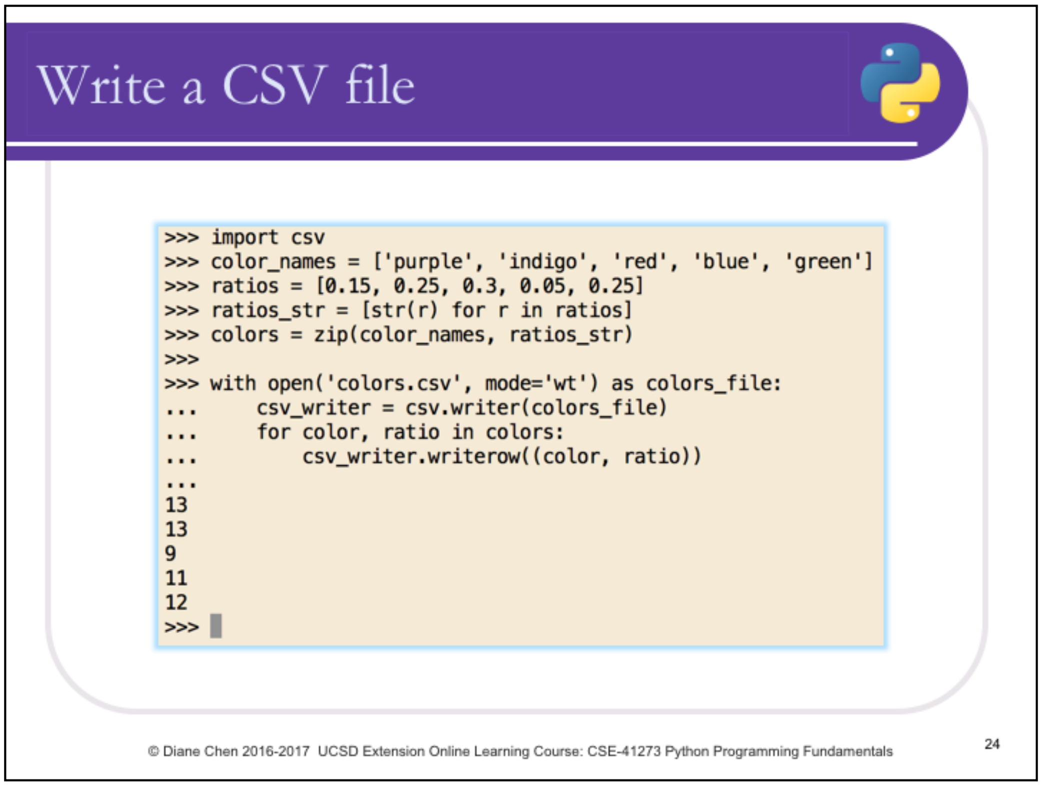


Read headers first

- Read headers into separate list

```
>>> import csv
>>> capitals = {}
>>> with open('us-state-capitals.csv') as csv_file:
...     csv_reader = csv.reader(csv_file)
...     headers = next(csv_reader)
...     for state, capital in csv_reader:
...         capitals[state] = capital
...
>>> len(capitals)
50
>>> headers
['state', 'capital']
```

Because the csv_reader object is an iterator, we can call next() on it and get the first row as the headers. Then we can fill the capitals dictionary and have the correct number of elements.



Write a CSV file

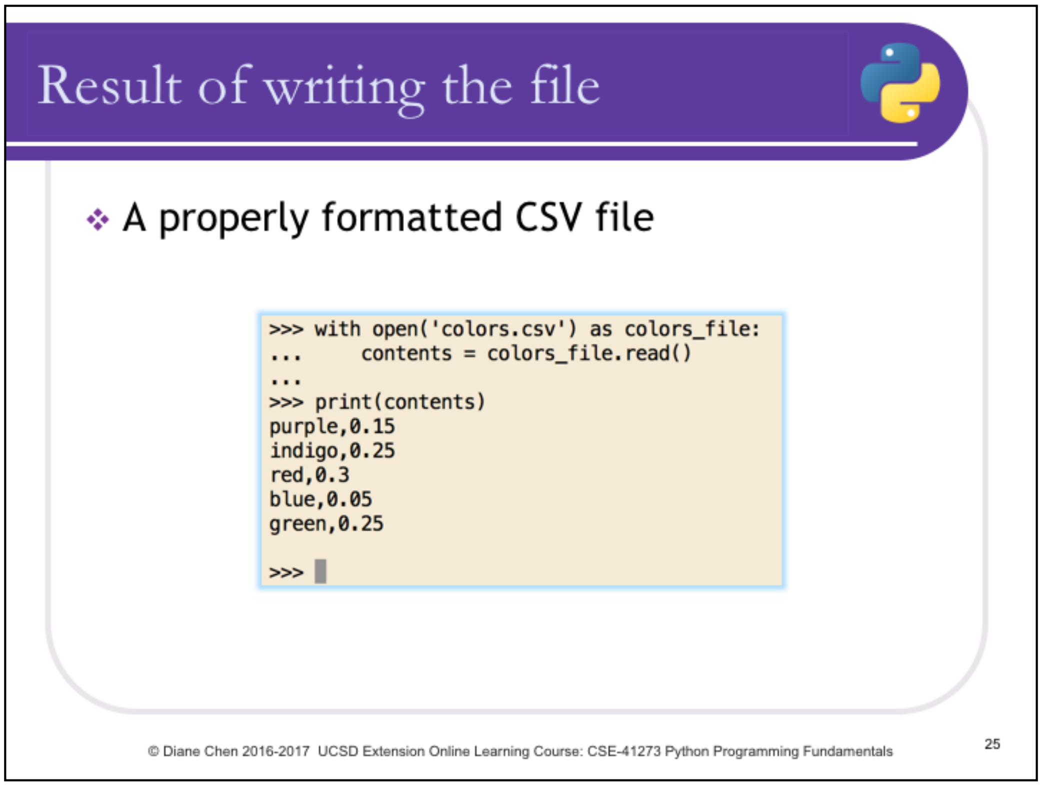
```
>>> import csv
>>> color_names = ['purple', 'indigo', 'red', 'blue', 'green']
>>> ratios = [0.15, 0.25, 0.3, 0.05, 0.25]
>>> ratios_str = [str(r) for r in ratios]
>>> colors = zip(color_names, ratios_str)
>>>
>>> with open('colors.csv', mode='wt') as colors_file:
...     csv_writer = csv.writer(colors_file)
...     for color, ratio in colors:
...         csv_writer.writerow((color, ratio))
...
13
13
9
11
12
>>> 
```

© Diane Chen 2016-2017 UCSD Extension Online Learning Course: CSE-41273 Python Programming Fundamentals 24

Now let's see how to create a csv file. Suppose we have some data that consists of a list of colors, and another list of numbers containing the ratios of each color. Because a csv file is a text file, we have to convert the ratio numbers into strings in the list ratios_str. Then we can use the handy zip function to make an iterator that will yield the color names and ratios in tuples.

In a similar manner to writing a text file, we use the context manager to open the csv file with the mode for writing text. We create the csv_writer object on the colors_file. Then we iterate over the color and ratio tuples and call writerow with the color and ratio for each one. Again, the csv module handles all the work of making sure the fields are written correctly with delimiters, etc.

The numbers printed out are from the writerow() function; it returns the number of characters written.



Result of writing the file

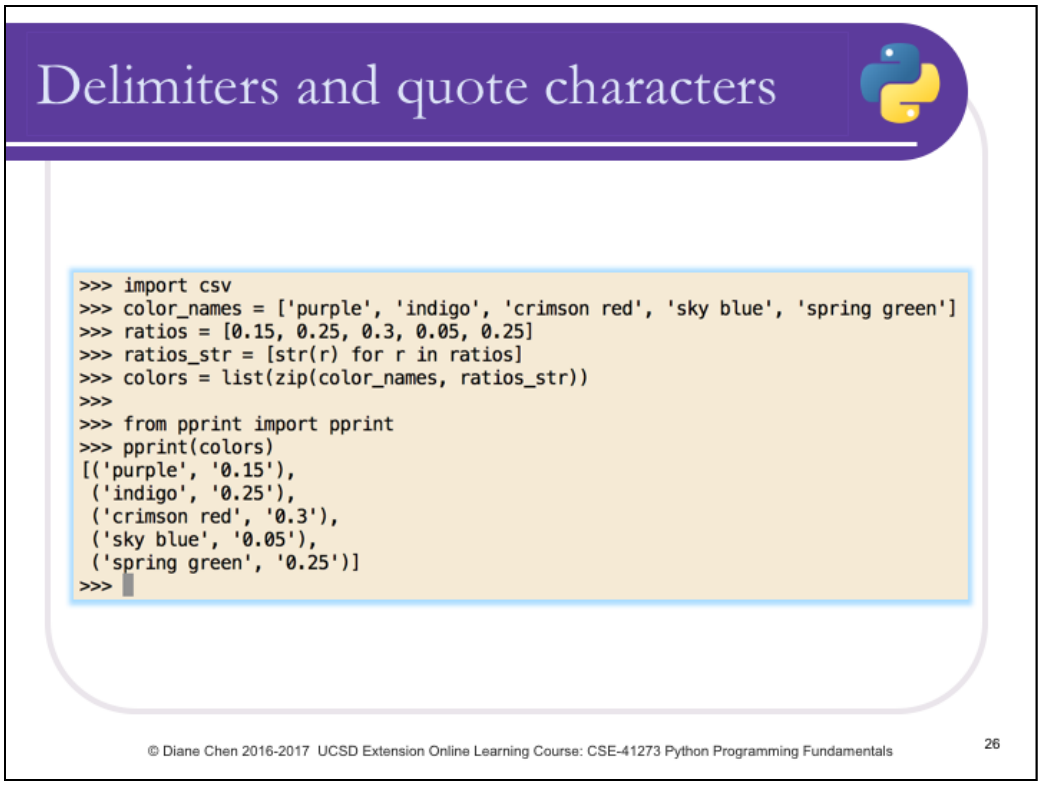
- A properly formatted CSV file

```
>>> with open('colors.csv') as colors_file:
...     contents = colors_file.read()
...
>>> print(contents)
purple,0.15
indigo,0.25
red,0.3
blue,0.05
green,0.25

>>> 
```

© Diane Chen 2016-2017 UCSD Extension Online Learning Course: CSE-41273 Python Programming Fundamentals

25

And here is the result. A properly formatted csv file, with data and delimiters.

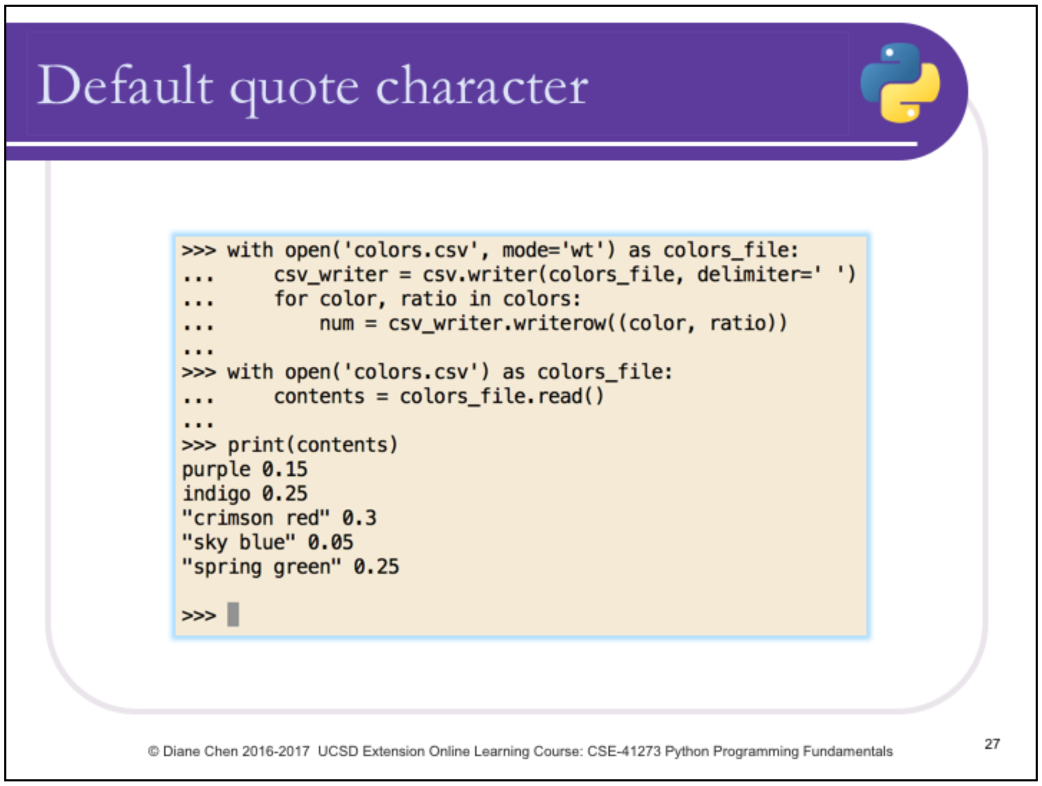


Delimiters and quote characters

```
>>> import csv
>>> color_names = ['purple', 'indigo', 'crimson red', 'sky blue', 'spring green']
>>> ratios = [0.15, 0.25, 0.3, 0.05, 0.25]
>>> ratios_str = [str(r) for r in ratios]
>>> colors = list(zip(color_names, ratios_str))
>>>
>>> from pprint import pprint
>>> pprint(colors)
[('purple', '0.15'),
 ('indigo', '0.25'),
 ('crimson red', '0.3'),
 ('sky blue', '0.05'),
 ('spring green', '0.25')]
>>> 
```

26

Sometimes we want to use different delimiters. A space is a common delimiter in addition to commas. Here I modified some of the color names to contain a space and create a new list of colors tuples to use for our next csv file example to show how delimiters and quote characters work.

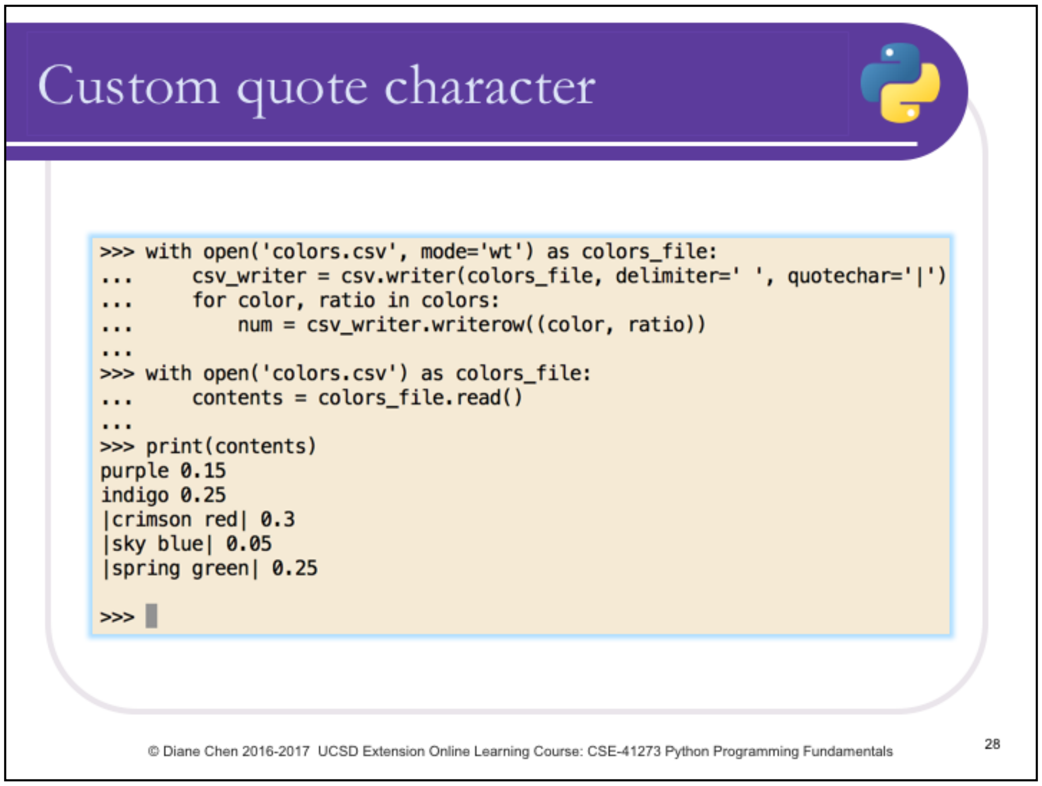


Default quote character

```
>>> with open('colors.csv', mode='wt') as colors_file:
...     csv_writer = csv.writer(colors_file, delimiter=' ')
...     for color, ratio in colors:
...         num = csv_writer.writerow((color, ratio))
...
>>> with open('colors.csv') as colors_file:
...     contents = colors_file.read()
...
>>> print(contents)
purple 0.15
indigo 0.25
"crimson red" 0.3
"sky blue" 0.05
"spring green" 0.25

>>> 
```

© Diane Chen 2016-2017 UCSD Extension Online Learning Course: CSE-41273 Python Programming Fundamentals 27

Here we add the parameter of "delimiter=' '" to the csv_writer object creation. I added the variable num to catch the return values from writerow, even though we don't care; it just saves space on the slide. This time our file is different. Since we are using space as a delimiter, the space inside some of the color names would be interpreted as a delimiter when the file is read. So the csv module automatically adds the quote character around those color names that need it. Here it is using the default quote character that is double-quote.



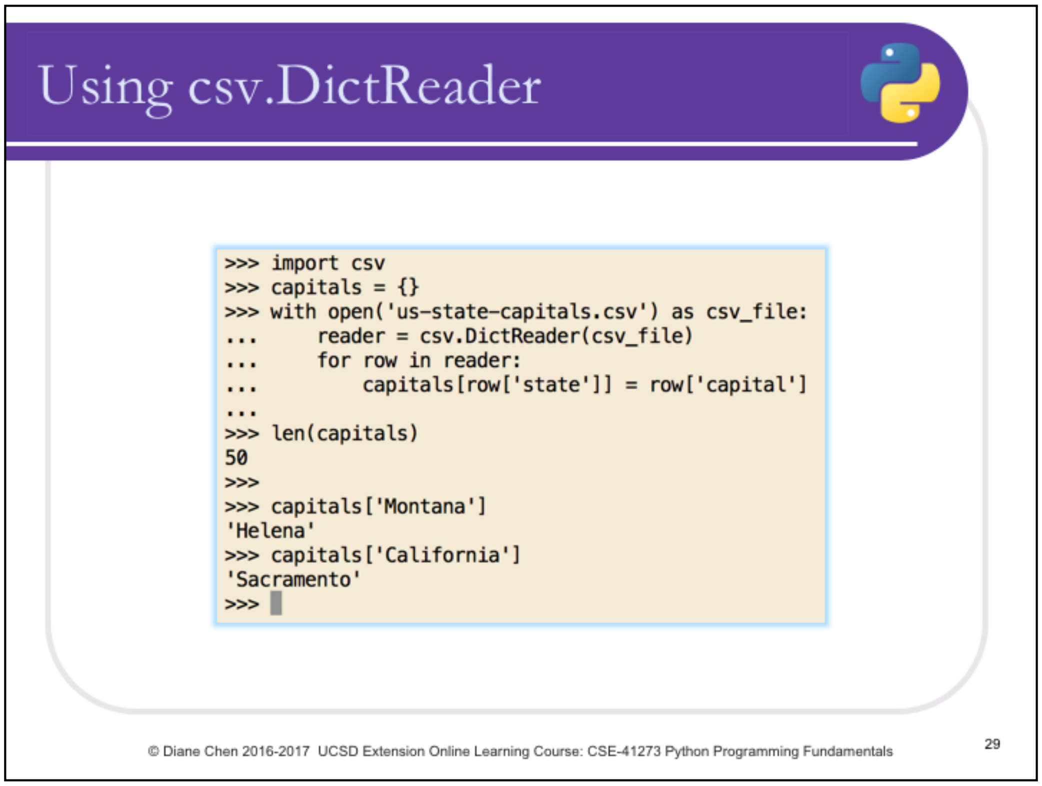
Custom quote character

```
>>> with open('colors.csv', mode='wt') as colors_file:
...     csv_writer = csv.writer(colors_file, delimiter=' ', quotechar='|')
...     for color, ratio in colors:
...         num = csv_writer.writerow((color, ratio))
...
>>> with open('colors.csv') as colors_file:
...     contents = colors_file.read()
...
>>> print(contents)
purple 0.15
indigo 0.25
|crimson red| 0.3
|sky blue| 0.05
|spring green| 0.25

>>> 
```

© Diane Chen 2016-2017 UCSD Extension Online Learning Course: CSE-41273 Python Programming Fundamentals

28

If we want, we can also specify a custom quote character. Here I used the vertical bar, and the csv system puts that around the color names that have spaces.

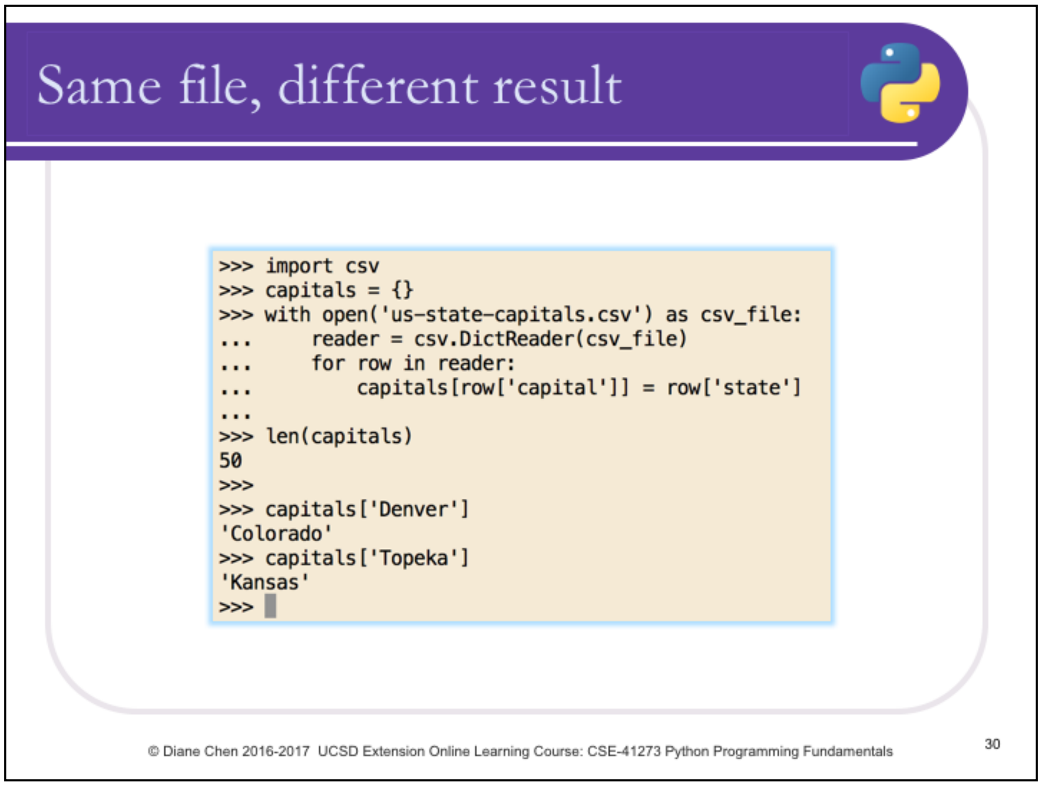


Using csv.DictReader

```
>>> import csv
>>> capitals = {}
>>> with open('us-state-capitals.csv') as csv_file:
...     reader = csv.DictReader(csv_file)
...     for row in reader:
...         capitals[row['state']] = row['capital']
...
>>> len(capitals)
50
>>>
>>> capitals['Montana']
'Helena'
>>> capitals['California']
'Sacramento'
>>> 
```

29

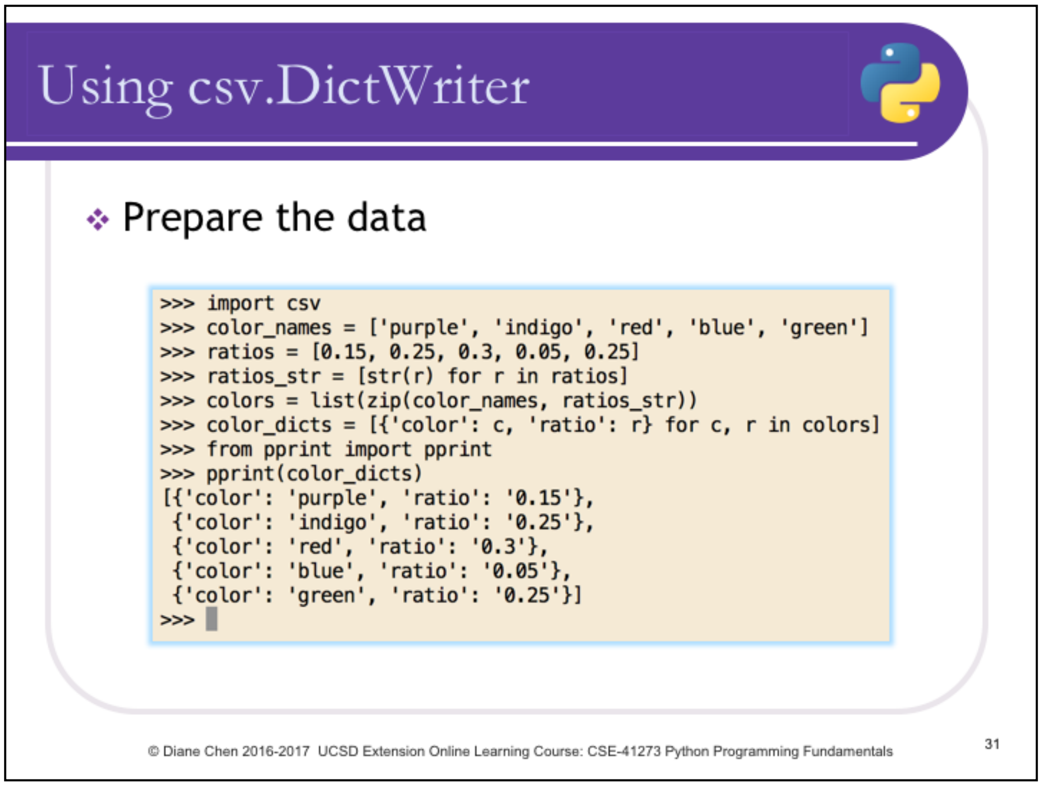
The csv module also has DictReader and DictWriter objects. When you have a file to read that has a header, you can use DictReader to read it. Here, each row is a dictionary whose keys are the field names and the values are the corresponding fields from the file. We can fill in our capitals file using the values for the state and the capital.



Same file, different result

```
>>> import csv
>>> capitals = {}
>>> with open('us-state-capitals.csv') as csv_file:
...     reader = csv.DictReader(csv_file)
...     for row in reader:
...         capitals[row['capital']] = row['state']
...
>>> len(capitals)
50
>>>
>>> capitals['Denver']
'Colorado'
>>> capitals['Topeka']
'Kansas'
>>>
```

In this example, we read the file the same way, but fill our capitals dictionary in reverse, so the capitals are the keys and the states are the values.



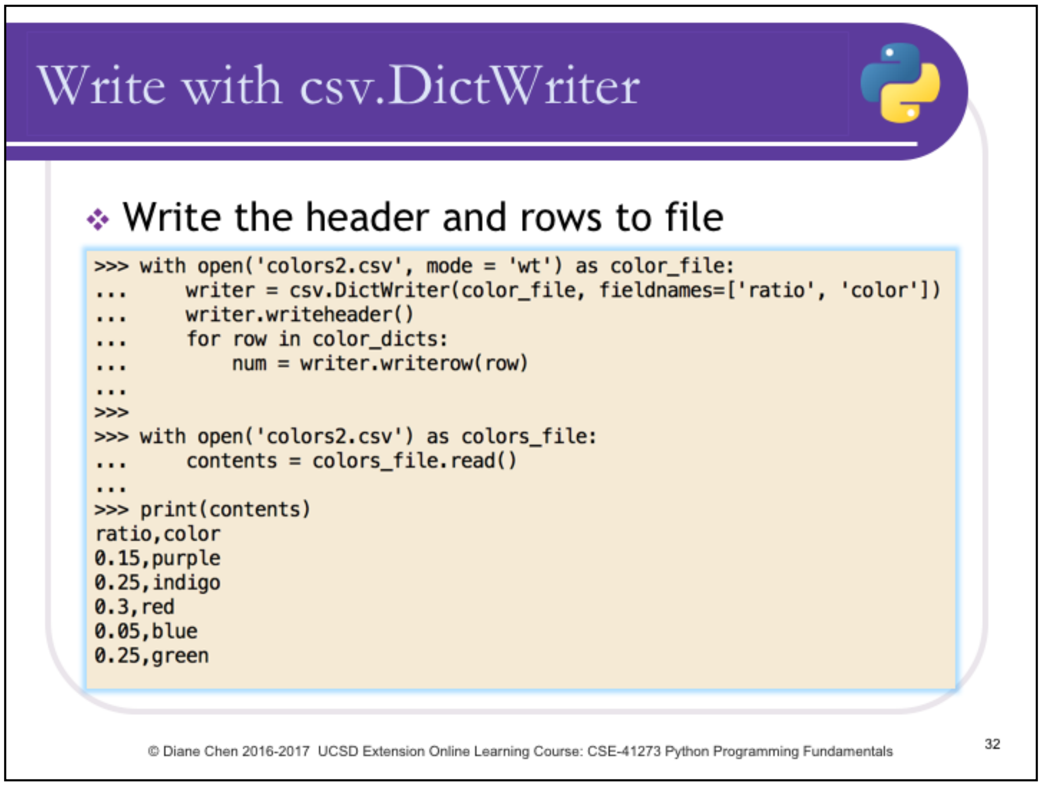
Using csv.DictWriter

- Prepare the data

```
>>> import csv
>>> color_names = ['purple', 'indigo', 'red', 'blue', 'green']
>>> ratios = [0.15, 0.25, 0.3, 0.05, 0.25]
>>> ratios_str = [str(r) for r in ratios]
>>> colors = list(zip(color_names, ratios_str))
>>> color_dicts = [{'color': c, 'ratio': r} for c, r in colors]
>>> from pprint import pprint
>>> pprint(color_dicts)
[{'color': 'purple', 'ratio': '0.15'},
 {'color': 'indigo', 'ratio': '0.25'},
 {'color': 'red', 'ratio': '0.3'},
 {'color': 'blue', 'ratio': '0.05'},
 {'color': 'green', 'ratio': '0.25'}]
>>> 
```

31

To use the csv DictWriter, we need to prepare the data. The DictWriter object's method writerow() expects to receive a dictionary where the keys are the field names and the values are the fields to use. I create the color_dicts list of dictionaries with a list comprehension. I print it out here with pretty print so you can see what it looks like.

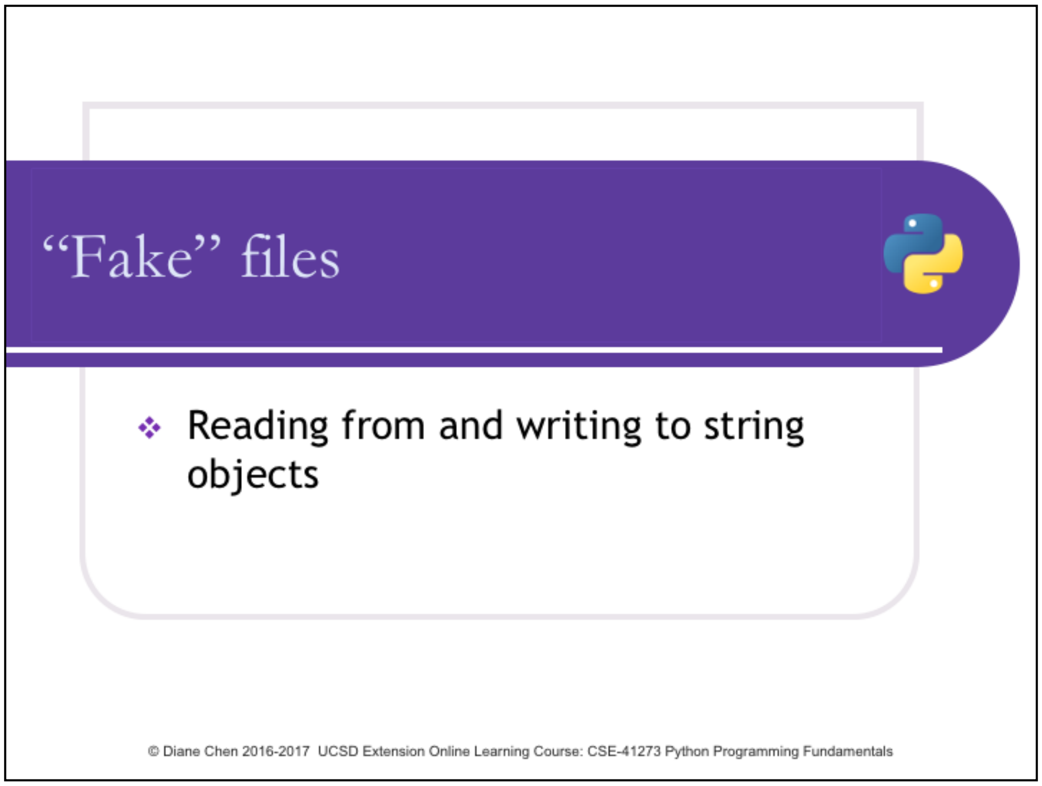


Write with csv.DictWriter

- Write the header and rows to file

```
>>> with open('colors2.csv', mode = 'wt') as color_file:
...     writer = csv.DictWriter(color_file, fieldnames=['ratio', 'color'])
...     writer.writeheader()
...     for row in color_dicts:
...         num = writer.writerow(row)
...
>>>
>>> with open('colors2.csv') as colors_file:
...     contents = colors_file.read()
...
>>> print(contents)
ratio,color
0.15,purple
0.25,indigo
0.3,red
0.05,blue
0.25,green
```

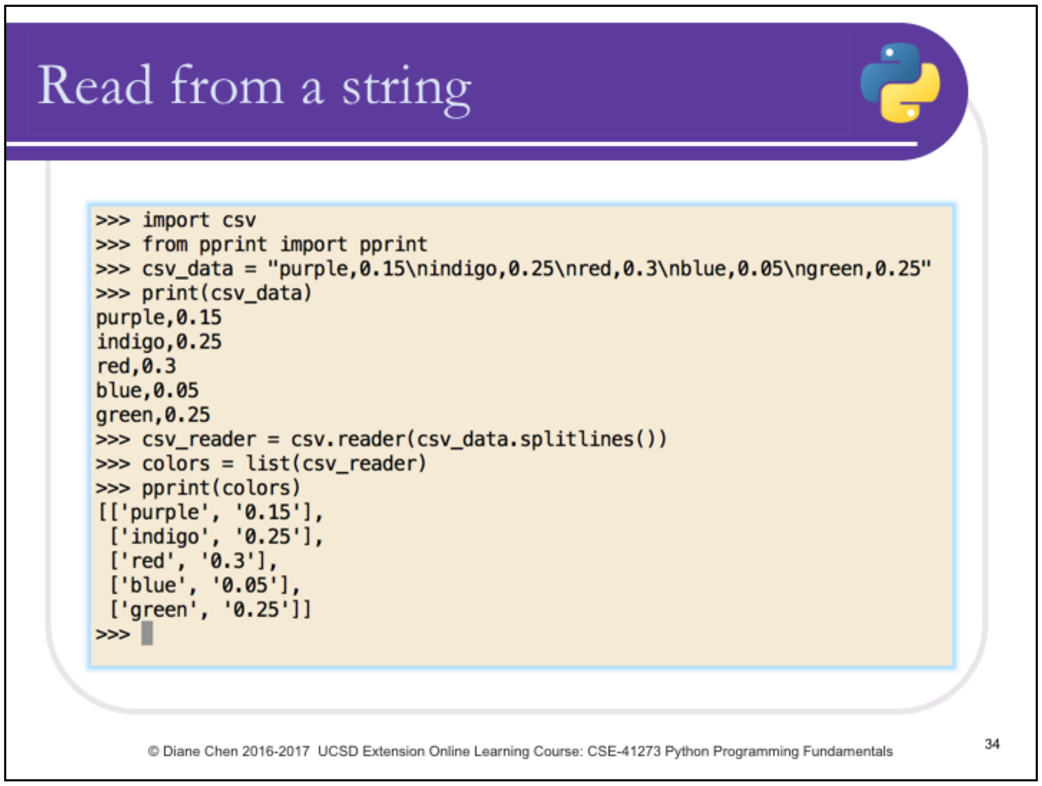
Now that we have the data prepared, we create the DictWriter object, giving it the fieldnames to use. It has a method writeheader() that writes the first row to the file. When we iterate over color_dicts, each “row” that we get from it is a dictionary from the list we made. When we print out the file, you can see that this file has a header row.



“Fake” files

- Reading from and writing to string objects

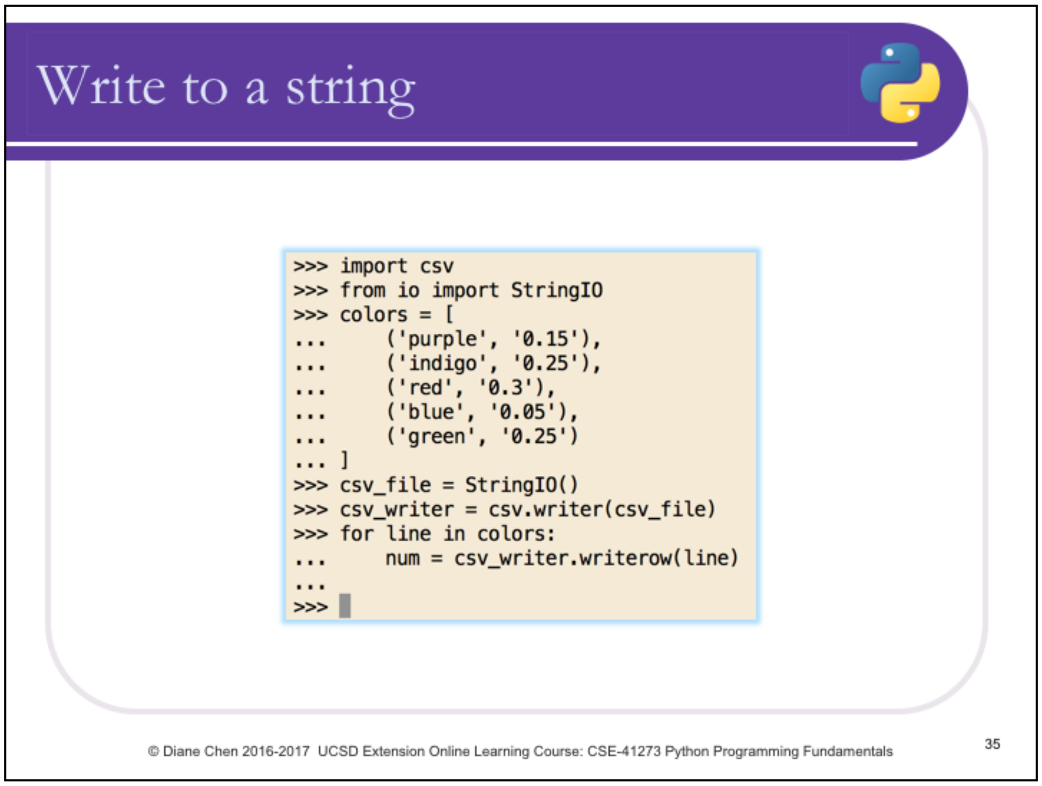
Now I’m just going to mention what I like to call “fake” files. Since all IO is reading and writing a stream of characters, we can use string objects if we want.



Read from a string

```
>>> import csv
>>> from pprint import pprint
>>> csv_data = "purple,0.15\nindigo,0.25\nred,0.3\nblue,0.05\ngreen,0.25"
>>> print(csv_data)
purple,0.15
indigo,0.25
red,0.3
blue,0.05
green,0.25
>>> csv_reader = csv.reader(csv_data.splitlines())
>>> colors = list(csv_reader)
>>> pprint(colors)
[['purple', '0.15'],
 ['indigo', '0.25'],
 ['red', '0.3'],
 ['blue', '0.05'],
 ['green', '0.25']]
>>> 
```

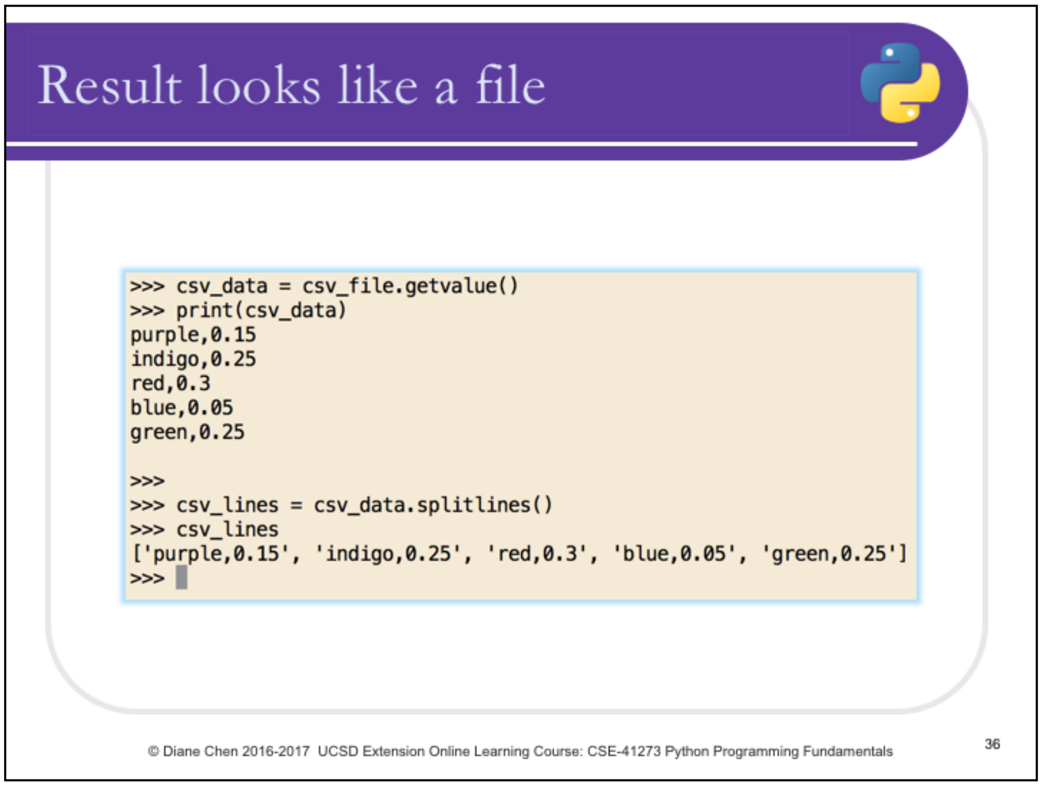
What if we wanted to read csv data from a string? I've created a string object called csv_data with the values from the color/ratio example earlier. In reality, I created the string by reading the entire file we created previously into one string, then I displayed it in the REPL and just copied the string. The backslash n characters are newlines, so when I use print on it, it displays like a file would. I call splitlines on the string and feed it to the csv.reader method which creates the iterator csv_reader. Then I feed csv_reader into the list function, that iterates over the csv_reader to create the colors object. Printing that out, we see we have a list of lists of the data from the original string.



Write to a string

```
>>> import csv
>>> from io import StringIO
>>> colors = [
...     ('purple', '0.15'),
...     ('indigo', '0.25'),
...     ('red', '0.3'),
...     ('blue', '0.05'),
...     ('green', '0.25')
... ]
>>> csv_file = StringIO()
>>> csv_writer = csv.writer(csv_file)
>>> for line in colors:
...     num = csv_writer.writerow(line)
...
>>> 
```

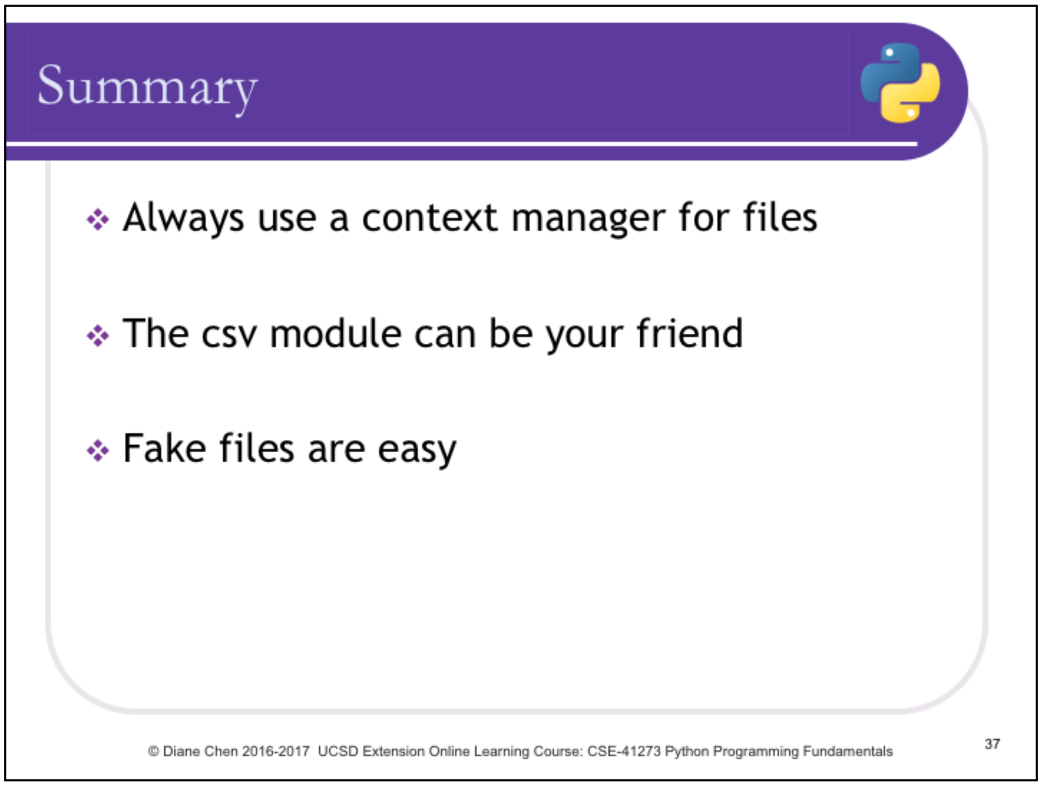
Now let's see writing to a string instead of a file. The csv.writer method expects a file-like object when we go to use it for writing our csv data. Fortunately there is the StringIO object in the io module that behaves like a file object. Thanks to Duck Typing, we can use it in place of a real file. The colors iterable here I got from when we used zip on the color names and ratio strings. We could just as easily use the list of lists from the previous slide. We create our “csv file” object from StringIO. Then we can create a csv_writer and write to it just as if it was a file.



Result looks like a file

```
>>> csv_data = csv_file.getvalue()
>>> print(csv_data)
purple,0.15
indigo,0.25
red,0.3
blue,0.05
green,0.25

>>>
>>> csv_lines = csv_data.splitlines()
>>> csv_lines
['purple,0.15', 'indigo,0.25', 'red,0.3', 'blue,0.05', 'green,0.25']
>>> 
```

Let's see what we have now. The method getvalue() gets the whole amount of data from the fake csv file object. When we print it, we see it looks just like a file. We can use splitlines to separate each line of the file into a string in a list. Notice that these strings each represent one line of the fake file data, because we are just reading it in like a regular text file, not handling it like csv data.



Summary

- Always use a context manager for files
- The csv module can be your friend
- Fake files are easy

This has been kind of a whirlwind tour of reading and writing text files. Remember to always use a context manager when you open files for reading or writing. If you have tabular data you need to save, you might find that a csv file is the way to go, even if you never plan to get near a spreadsheet. It's pretty easy to make and use fake files if you need something like that behavior but don't need to actually save things permanently.