Files and Exceptions

7

The programs that we have created so far have read all of their input from the keyboard. As a result, it has been necessary to re-type all of the input values each time the program runs. This is inefficient, particularly for programs that require a lot of input. Similarly, our programs have displayed all of their results on the screen. While this works well when only a few lines of output are printed, it is impractical for larger results that move off the screen too quickly to be read, or for output that requires further analysis by other programs. Writing programs that use files effectively will allow us to address all of these concerns.

Files are relatively permanent. The values stored in them are retained after a program completes and when the computer is turned off. This makes them suitable for storing results that are needed for an extended period of time, and for holding input values for a program that will be run several times. You have previously worked with files such as word processor documents, spreadsheets, images, and videos, among others. Your Python programs are also stored in files.

Files are commonly classified as being *text files* or *binary files*. Text files only contain sequences of bits that represent characters using an encoding system such as ASCII or UTF-8. These files can be viewed and modified with any text editor. All of the Python programs that we have created have been saved as text files.

Like text files, binary files also contain sequences of bits. But unlike text files, those sequences of bits can represent any kind of data. They are not restricted to characters alone. Files that contain image, sound and video data are normally binary files. We will restrict ourselves to working with text files in this book because they are easy to create and view with your favourite editor. Most of the principles described for text files can also be applied to binary files.

B. Stephenson, *The Python Workbook*, Texts in Computer Science,
https://doi.org/10.1007/978-3-030-18873-3_7

7.1 Opening a File

A file must be opened before data values can be read from it. It is also necessary to open a file before new data values are written to it. Files are opened by calling the `open` function.

The `open` function takes two arguments. The first argument is a string that contains the name of the file that will be opened. The second argument is also a string. It indicates the *access mode* for the file. The access modes that we will discuss include read (denoted by `"r"`), write (denoted by `"w"`) and append (denoted by `"a"`).

A *file object* is returned by the `open` function. As a result, the `open` function is normally called on the right side of an assignment statement, as shown below:

```
inf = open("input.txt", "r")
```

Once the file has been opened, methods can be applied to the file object to read data from the file. Similarly, data is written to the file by applying appropriate methods to the file object. These methods are described in the sections that follow. The file should be closed once all of the values have been read or written. This is accomplished by applying the `close` method to the file object.

7.2 Reading Input from a File

There are several methods that can be applied to a file object to read data from a file. These methods can only be applied when the file has been opened in read mode. Attempting to read from a file that has been opened in write mode or append mode will cause your program to crash.

The `readline` method reads one line from the file and returns it as a string, much like the input function reads a line of text typed on the keyboard. Each subsequent call to `readline` reads another line from the file sequentially from the top of the file to the bottom of the file. The `readline` method returns an empty string when there is no further data to read from the file.

Consider a data file that contains a long list of numbers, each of which appears on its own line. The following program computes the total of all of the numbers in such a file.

```
# Read the file name from the user and open the file
fname = input("Enter the file name: ")
inf = open(fname, "r")

# Initialize the total
total = 0

# Total the values in the file
line = inf.readline()
while line != "":
  total = total + float(line)
  line = inf.readline()
```

```
# Close the file
inf.close()

# Display the result
print("The total of the values in", fname, "is", total)
```

This program begins by reading the name of the file from the user. Once the name has been read, the file is opened for reading and the file object is stored in `inf`. Then `total` is initialized to 0, and the first line is read from the file.

The condition on the `while` loop is evaluated next. If the first line read from the file is non-empty, then the body of the loop executes. It converts the line read from the file into a floating-point number and adds it to `total`. Then the next line is read from the file. If the file contains more data then the `line` variable will contain the next line in the file, the `while` loop condition will evaluate to `True`, and the loop will execute again causing another value to be added to the total.

At some point all of the data will have been read from the file. When this occurs the `readline` method will return an empty string which will be stored into `line`. This will cause the condition on the `while` loop to evaluate to `False` and cause the loop to terminate. Then the program will go on and display the total.

Sometimes it is helpful to read all of the data from a file at once instead of reading it one line at a time. This can be accomplished using either the `read` method or the `readlines` method. The `read` method returns the entire contents of the file as one (potentially very long) string. Then further processing is typically performed to break the string into smaller pieces. The `readlines` method returns a list where each element is one line from the file. Once all of the lines are read with `readlines` a loop can be used to process each element in the list. The following program uses `readlines` to compute the sum of all of the numbers in a file. It reads all of the data from the file at once instead of adding each number to the total as it is read.

```
# Read the file name from the user and open the file
fname = input("Enter the file name: ")
inf = open(fname, "r")

# Initialize total and read all of the lines from the file
total = 0
lines = inf.readlines()

# Total the values in the file
for line in lines:
  total = total + float(line)

# Close the file
inf.close()

# Display the result
print("The total of the values in", fname, "is", total)
```

7.3 End of Line Characters

The following example uses the `readline` method to read and display all of the lines in a file. Each line is preceded by its line number and a colon when it is printed.

```
# Read the file name from the user and open the file
fname = input("Enter the name of a file to display: ")
inf = open(fname, "r")

# Initialize the line number
num = 1

# Display each line in the file, preceded by its line number
line = inf.readline()
while line != "":
  print("%d: %s" % (i, line))

  # Increment the line number and read the next line
  num = num + 1
  line = inf.readline()

# Close the file
inf.close()
```

When you run this program you might be surprised by its output. In particular, each time a line from the file is printed, a second line, which is blank, is printed immediately after it. This occurs because each line in a text file ends with one or more characters that denote the end of the line.[1] Such characters are needed so that any program reading the file can determine where one line ends and the next one begins. Without them, all of the characters in a text file would appear on the same line when they are read by your program (or when loaded into your favourite text editor).

The end of line marker can be removed from a string that was read from a file by calling the `rstrip` method. This method, which can be applied to any string, removes any whitespace characters (spaces, tabs, and end of line markers) from the right end of a string. A new copy of the string with such characters removed (if any were present) is returned by the method.

An updated version of the line numbering program is shown below. It uses the `rstrip` method to remove the end of line markers, and as a consequence, does not include the blank lines that were incorrectly displayed by the previous version.

```
# Read the file name from the user and open the file
fname = input("Enter the name of a file to display: ")
inf = open(fname, "r")

# Initialize the line number
num = 1

# Display each line in the file, preceded by its line number
line = inf.readline()
while line != "":
  # Remove the end of line marker and display the line preceded by its line number
  line = line.rstrip()
  print("%d: %s" % (i, line))
```

[1] The character or sequence of characters used to denote the end of a line in a text file varies from operating system to operating system. Fortunately, Python automatically handles these differences and allows text files created on any widely used operating system to be loaded by Python programs running on any other widely used operating system.

```
  # Increment the line number and read the next line
  num = num + 1
  line = inf.readline()

# Close the file
inf.close()
```

7.4 Writing Output to a File

When a file is opened in write mode, a new empty file is created. If the file already exists then the existing file is destroyed and any data that it contained is lost. Opening a file that already exists in append mode will cause any data written to the file to be added to the end of it. If a file opened in append mode does not exist then a new empty file is created.

The `write` method can be used to write data to a file opened in either write mode or append mode. It takes one argument, which must be a string, that will be written to the file. Values of other types can be converted to a string by calling the `str` function. Multiple values can be written to the file by concatenating all of the items into one longer string, or by calling the `write` method multiple times.

Unlike the `print` function, the `write` method does not automatically move to the next line after writing a value. As a result, one has to explicitly write an end of line marker to the file between values that are to reside on different lines. Python uses `\n` to denote the end of line marker. This pair of characters, referred to as an *escape sequence*, can appear in a string on its own, or `\n` can appear as part of a longer string.

The following program writes the numbers from 1 up to (and including) a number entered by the user to a file. String concatenation and the `\n` escape sequence are used so that each number is written on its own line.

```
# Read the file name from the user and open the file
fname = input("Where will the numbers will be stored? ")
outf = open(fname, "w")

# Read the maximum value that will be written
limit = int(input("What is the maximum value? "))

# Write the numbers to the file with one number on each line
for num in range(1, limit + 1):
  outf.write(str(num) + "\n")

# Close the file
outf.close()
```

7.5 Command Line Arguments

Computer programs are commonly executed by clicking on an icon or selecting an item from a menu. Programs can also be started from the command line by typing an appropriate command into a terminal or command prompt window. For example, on

many operating systems, the Python program stored in `test.py` can be executed by typing either `test.py` or `python test.py` in such a window.

Starting a program from the command line provides a new opportunity to supply input to it. Values that the program needs to perform its task can be part of the command used to start the program by including them on the command line after the name of the `.py` file. Being able to provide input as part of the command used to start a program is particularly beneficial when writing scripts that use multiple programs to automate some task, and for programs that are scheduled to run periodically.

Any command line arguments provided when the program was executed are stored into a variable named `argv` (argument vector) that resides in the `sys` (system) module. This variable is a list, and each element in the list is a string. Elements in the list can be converted to other types by calling the appropriate type conversion functions like `int` and `float`. The first element in the argument vector is the name of the Python source file that is being executed. The subsequent elements in the list are the values provided on the command line after the name of the Python file (if any).

The following program demonstrates accessing the argument vector. It begins by reporting the number of command line arguments provided to the program and the name of the source file that is being executed. Then it goes on and displays the arguments that appear after the name of the source file if such values were provided. Otherwise a message is displayed that indicates that there were no command line arguments beyond the `.py` file being executed.

```
# The system module must be imported to access the command line arguments
import sys

# Display the number of command line arguments (including the .py file)
print("The program has", len(sys.argv), \
      "command line argument(s).")

# Display the name of the .py file
print("The name of the .py file is", sys.argv[0])

# Determine whether or not there are additional arguments to display
if len(sys.argv) > 1:
  # Display all of the command line arguments beyond the name of the .py file
  print("The remaining arguments are:")
  for i in range(1, len(sys.argv)):
    print(" ", sys.argv[i])
else:
  print("No additional arguments were provided.")
```

Command line arguments can be used to supply any input values to the program that can be typed on the command line, such as integers, floating-point numbers and strings. These values can then be used just like any other values in the program. For example, the following lines of code are a revised version of our program that sums all of the numbers in a file. In this version of the program the name of the file is provided as a command line argument instead of being read from the keyboard.

```
# Import the system module
import sys

# Ensure that the program was started with one command line argument beyond the name
# of the .py file
if len(sys.argv) != 2:
  print("A file name must be provided as a command line", \
        "argument.")
  quit()

# Open the file listed immediately after the .py file on the command line
inf = open(sys.argv[1], "r")

# Initialize the total
total = 0

# Total the values in the file
line = inf.readline()
while line != "":
  total = total + float(line)
  line = inf.readline()

# Close the file
inf.close()

# Display the result
print("The total of the values in", sys.argv[1], "is", total)
```

7.6 Exceptions

There are many things that can go wrong when a program is running: The user can supply a non-numeric value when a numeric value was expected, the user can enter a value that causes the program to divide by 0, or the user can attempt to open a file that does not exist, among many other possibilities. All of these errors are *exceptions*. By default, a Python program crashes when an exception occurs. However, we can prevent our program from crashing by catching the exception and taking appropriate actions to recover from it.

The programmer must indicate where an exception might occur in order to catch it. He or she must also indicate what code to run to handle the exception when it occurs. These tasks are accomplished by using two keywords that we have not yet seen: `try` and `except`. Code that might cause an exception that we want to catch is placed inside a `try` block. The `try` block is immediately followed by one or more `except` blocks. When an exception occurs inside a `try` block, execution immediately jumps to the appropriate `except` block without running any remaining statements in the `try` block.

Each `except` block can specify the particular exception that it catches. This is accomplished by including the exception's type immediately after the `except` keyword. Such a block only executes when an exception of the indicated type occurs. An `except` block that does not specify a particular exception will catch any type

of exception (that is not caught by another `except` block associated to the same `try` block). The `except` blocks only execute when an exception occurs. If the `try` block executes without raising an exception then all of the `except` blocks are skipped and execution continues with the first line of code following the final `except` block.

The programs that we considered in the previous sections all crashed when the user provided the name of a file that did not exist. This crash occurred because a `FileNotFoundError` exception was raised without being caught. The following code segment uses a `try` block and an `except` block to catch this exception and display a meaningful error message when it occurs. This code segment can be followed by whatever additional code is needed to read and process the data in the file.

```
# Read the file name from the user
fname = input("Enter the file name: ")

# Attempt to open the file
try:
  inf = open(fname, "r")
except FileNotFoundError:
  # Display an error message and quit if the file was not opened successfully
  print("'%s' could not be opened.  Quitting...")
  quit()
```

The current version of our program quits when the file requested by the user does not exist. While that might be fine in some situations, there are other times when it is preferable to prompt the user to re-enter the file name. The second file name entered by the user could also cause an exception. As a result, a loop must be used that runs until the user enters the name of a file that is opened successfully. This is demonstrated by the following program. Notice that the `try` block and the `except` block are both inside the `while` loop.

```
# Read the file name from the user
fname = input("Enter the file name: ")

file_opened = False
while file_opened == False:
  # Attempt to open the file
  try:
    inf = open(fname, "r")
    file_opened = True
  except FileNotFoundError:
    # Display an error message and read another file name if the file was not
    # opened successfully
    print("'%s' wasn't found.  Please try again.")
    fname = input("Enter the file name: ")
```

When this program runs it begins by reading the name of a file from the user. Then the `file_opened` variable is set to `False` and the loop runs for the first time. Two lines of code reside in the `try` block inside the loop's body. The first attempts to open

the file specified by the user. If the file does not exist then a `FileNotFoundError` exception is raised and execution immediately jumps to the `except` block, skipping the second line in the `try` block. When the `except` block executes it displays an error message and reads another file name from the user.

Execution continues by returning to the top of the loop and evaluating its condition again. The condition still evaluates to `False` because the `file_opened` variable is still `False`. As a result, the body of the loop executes for a second time, and the program makes another attempt to open the file using the most recently entered file name. If that file does not exist then the program progresses as described in the previous paragraph. But if the file exists, the call to `open` completes successfully, and execution continues with the next line in the `try` block. This line sets `file_opened` to `True`. Then the `except` block is skipped because no exceptions were raised while executing the `try` block. Finally, the loop terminates because `file_opened` was set to `True`, and execution continues with the rest of the program.

The concepts introduced in this section can be used to detect and respond to a wide variety of errors that can occur as a program is running. By creating `try` and `except` blocks your programs can respond to these errors in an appropriate manner instead of crashing.

7.7 Exercises

Many of the exercises in this chapter read data from a file. In some cases any text file can be used as input. In other cases appropriate input files can be created easily in your favourite text editor. There are also some exercises that require specific data sets such as a list of words, names or chemical elements. These data sets can be downloaded from the author's website:

http://www.cpsc.ucalgary.ca/~bdstephe/PythonWorkbook

Exercise 149: Display the Head of a File

(*Solved, 40 Lines*)

Unix-based operating systems usually include a tool named `head`. It displays the first 10 lines of a file whose name is provided as a command line argument. Write a Python program that provides the same behaviour. Display an appropriate error message if the file requested by the user does not exist, or if the command line argument is omitted.

Exercise 150: Display the Tail of a File

(*Solved, 35 Lines*)

Unix-based operating systems also typically include a tool named `tail`. It displays the last 10 lines of a file whose name is provided as a command line argument. Write a Python program that provides the same behaviour. Display an appropriate error message if the file requested by the user does not exist, or if the command line argument is omitted.

There are several different approaches that can be taken to solve this problem. One option is to load the entire contents of the file into a list and then display its last 10 elements. Another option is to read the contents of the file twice, once to count the lines, and a second time to display its last 10 lines. However, both of these solutions are undesirable when working with large files. Another solution exists that only requires you to read the file once, and only requires you to store 10 lines from the file at one time. For an added challenge, develop such a solution.

Exercise 151: Concatenate Multiple Files

(*Solved, 28 Lines*)

Unix-based operating systems typically include a tool named `cat`, which is short for concatenate. Its purpose is to display the concatenation of one or more files whose names are provided as command line arguments. The files are displayed in the same order that they appear on the command line.

Create a Python program that performs this task. It should generate an appropriate error message for any file that cannot be displayed, and then proceed to the next file. Display an appropriate error message if your program is started without any command line arguments.

Exercise 152: Number the Lines in a File

(*23 Lines*)

Create a program that reads lines from a file, adds line numbers to them, and then stores the numbered lines into a new file. The name of the input file will be read from the user, as will the name of the new file that your program will create. Each line in the output file should begin with the line number, followed by a colon and a space, followed by the line from the input file.

Exercise 153: Find the Longest Word in a File

(*39 Lines*)

In this exercise you will create a Python program that identifies the longest word(s) in a file. Your program should output an appropriate message that includes the length of the longest word, along with all of the words of that length that occurred in the file. Treat any group of non-white space characters as a word, even if it includes digits or punctuation marks.

Exercise 154: Letter Frequencies

(*43 Lines*)

One technique that can be used to help break some simple forms of encryption is frequency analysis. This analysis examines the encrypted text to determine which characters are most common. Then it tries to map the most common letters in English, such as `E` and `T`, to the most commonly occurring characters in the encrypted text.

Write a program that initiates this process by determining and displaying the frequencies of all of the letters in a file. Ignore spaces, punctuation marks, and digits as you perform this analysis. Your program should be case insensitive, treating `a` and `A` as equivalent. The user will provide the name of the file to analyze as a command line argument. Your program should display a meaningful error message if the user provides the wrong number of command line arguments, or if the program is unable to open the file indicated by the user.

Exercise 155: Words that Occur Most

(*37 Lines*)

Write a program that displays the word (or words) that occur most frequently in a file. Your program should begin by reading the name of the file from the user. Then it should process every line in the file. Each line will need to be split into words, and any leading or trailing punctuation marks will need to be removed from each word. Your program should also ignore capitalization when counting how many times each word occurs.

> Hint: You will probably find your solution to Exercise 117 helpful when completing this task.

Exercise 156: Sum a Collection of Numbers

(*Solved, 26 Lines*)

Create a program that sums all of the numbers entered by the user while ignoring any input that is not a valid number. Your program should display the current sum after each number is entered. It should display an appropriate message after each non-numeric input, and then continue to sum any additional numbers entered by the user. Exit the program when the user enters a blank line. Ensure that your program works correctly for both integers and floating-point numbers.

> Hint: This exercise requires you to use exceptions without using files.

Exercise 157: Both Letter Grades and Grade Points

(*106 Lines*)

Write a program that converts from letter grades to grade points and vice-versa. Your program should allow the user to convert multiple values, with one value entered on each line. Begin by attempting to convert each value entered by the user from a number of grade points to a letter grade. If an exception occurs during this process then your program should attempt to convert the value from a letter grade to a number of grade points. If both conversions fail then your program should output a message indicating that the supplied input is invalid. Design your program so that it continues performing conversions (or reporting errors) until the user enters a blank line. Your solutions to Exercises 52 and 53 may be helpful when completing this exercise.

Exercise 158: Remove Comments

(*Solved, 53 Lines*)

Python uses the # character to mark the beginning of a comment. The comment continues from the # character to the end of the line containing it. Python does not provide any mechanism for ending a comment before the end of a line.

In this exercise, you will create a program that removes all of the comments from a Python source file. Check each line in the file to determine if a # character is present. If it is then your program should remove all of the characters from the # character to the end of the line (we will ignore the situation where the comment character occurs inside of a string). Save the modified file using a new name. Both the name of the input file and the name of the output file should be read from the user. Ensure that an appropriate error message is displayed if a problem is encountered while accessing either of the files.

Exercise 159: Two Word Random Password

(*Solved, 39 Lines*)

While generating a password by selecting random characters usually creates one that is relatively secure, it also generally gives a password that is difficult to memorize. As an alternative, some systems construct a password by taking two English words and concatenating them. While this password may not be as secure, it is normally much easier to memorize.

Write a program that reads a file containing a list of words, randomly selects two of them, and concatenates them to produce a new password. When producing the password ensure that the total length is between 8 and 10 characters, and that each word used is at least three letters long. Capitalize each word in the password so that the user can easily see where one word ends and the next one begins. Finally, your program should display the password for the user.

Exercise 160: Weird Words

(*67 Lines*)

Students learning to spell in English are often taught the rhyme "I before E except after C". This rule of thumb advises that when an I and an E are adjacent in a word, the I will precede the E, unless they are immediately preceded by a C. When preceded by a C the E will appear ahead of the I. This advice holds true for words without an immediately preceding C such as believe, chief, fierce and friend, and is similarly true for words with an immediately preceding C such as ceiling and receipt. However, there are exceptions to this rule, such as weird.

Create a program that processes a file containing lines of text. Each line in the file may contain many words (or no words at all). Any words that do not contain an E adjacent to an I should be ignored. Words that contain an adjacent E and I (in either order) should be examined to determine whether or not they follow the "I before E except after C" rule. Construct and report two lists: One that contains all of the words that follow the rule, and one that contains all of the words that violate the rule. Neither of your lists should contain any repeated values. Report the lengths of the lists at the end of your program so that one can easily determine the proportion of the words in the file that respect the "I before E except after C" rule.

Exercise 161: What's that Element Again?

(*59 Lines*)

Write a program that reads a file containing information about chemical elements and stores it in one or more appropriate data structures. Then your program should read and process input from the user. If the user enters an integer then your program should display the symbol and name of the element with the number of protons entered. If the user enters a non-integer value then your program should display the number of protons for the element with that name or symbol. Your program should display an appropriate error message if no element exists for the name, symbol or number of protons entered. Continue to read input from the user until a blank line is entered.

Exercise 162: A Book with No E...

(*Solved, 50 Lines*)

The novel "Gadsby" is over 50,000 words in length. While 50,000 words is not normally remarkable for a novel, it is in this case because none of the words in the book use the letter E. This is particularly noteworthy when one considers that E is the most common letter in English.

Write a program that reads a list of words from a file and determines what proportion of the words use each letter of the alphabet. Display this result for all 26 letters and include an additional message that identifies the letter that is used in the smallest proportion of the words. Your program should ignore any punctuation marks that are present in the file and it should treat uppercase and lowercase letters as equivalent.

A lipogram is a written work that does not use a particular letter (or group of letters). The letter that is avoided is often a common vowel, though it does not have to be. For example, The Raven by Edgar Allan Poe is a poem of more than 1,000 words that does not use the letter Z, and as such, is a lipogram. "La Disparition" is another example of a lipogrammatic novel. Both the original novel (written in French), and its English translation, "A Void", occupy approximately 300 pages without using the letter E, other than in the author's name.

Exercise 163: Names that Reached Number One

(*Solved, 54 Lines*)

The baby names data set consists of over 200 files. Each file contains a list of 100 names, along with the number of times each name was used. Entries in the files are ordered from most frequently used to least frequently used. There are two files for each year: one containing names used for girls and the other containing names used for boys. The data set includes data for every year from 1900 to 2012.

Write a program that reads every file in the data set and identifies all of the names that were most popular in at least one year. Your program should output two lists: one containing the most popular names for boys and the other containing the most popular names for girls. Neither of your lists should include any repeated values.

Exercise 164: Gender Neutral Names

(*56 Lines*)

Some names, like Ben, Jonathan and Andrew are normally only used for boys while names like Rebecca and Flora are normally only used for girls. Other names, like Chris and Alex, may be used for both boys and girls.

Write a program that determines and displays all of the baby names that were used for both boys and girls in a year specified by the user. Your program should generate an appropriate message if there were no gender neutral names in the selected year. Display an appropriate error message if you do not have data for the year requested by the user. Additional details about the baby names data set are included in Exercise 163.

Exercise 165: Most Births in a given Time Period

(*76 Lines*)

Write a program that uses the baby names data set described in Exercise 163 to determine which names were used most often within a time period. Have the user supply the first and last years of the range to analyze. Display the boy's name and the girl's name given to the most children during the indicated years.

Exercise 166: Distinct Names

(*41 Lines*)

In this exercise, you will create a program that reads every file in the baby names data set described in Exercise 163. As your program reads the files, it should keep track of every distinct name used for a boy and every distinct name used for a girl. Then your program should output each of these lists of names. Neither of the lists should contain any repeated values.

Exercise 167: Spell Checker

(*Solved, 58 Lines*)

A spell checker can be a helpful tool for people who struggle to spell words correctly. In this exercise, you will write a program that reads a file and displays all of the words in it that are misspelled. Misspelled words will be identified by checking each word in the file against a list of known words. Any words in the user's file that do not appear in the list of known words will be reported as spelling mistakes.

The user will provide the name of the file to check for spelling mistakes as a command line argument. Your program should display an appropriate error message if the command line argument is missing. An error message should also be displayed if your program is unable to open the user's file. Use your solution to Exercise 117 when creating your solution to this exercise so that words followed by a comma, period or other punctuation mark are not reported as spelling mistakes. Ignore the capitalization of the words when checking their spelling.

> Hint: While you could load all of the English words from the words data set into a list, searching a list is slow if you use Python's `in` operator. It is much faster to check if a key is present in a dictionary, or if a value is present in a set. If you use a dictionary, the words will be the keys. The values can be the integer 0 (or any other value) because the values will never be used.

Exercise 168: Repeated Words

(*61 Lines*)

Spelling mistakes are only one of many different kinds of errors that might appear in a written work. Another error that is common for some writers is a repeated word. For example, an author might inadvertently duplicate a word, as shown in the following sentence:

```
At least one value must be entered
entered in order to compute the average.
```

Some word processors will detect this error and identify it when a spelling or grammar check is performed.

In this exercise you will write a program that detects repeated words in a text file. When a repeated word is found your program should display a message that contains the line number and the repeated word. Ensure that your program correctly handles the case where the same word appears at the end of one line and the beginning of the following line, as shown in the previous example. The name of the file to examine will be provided as the program's only command line argument. Display an appropriate error message if the user fails to provide a command line argument, or if an error occurs while processing the file.

Exercise 169: Redacting Text in a File

(*Solved, 52 Lines*)

Sensitive information is often removed, or redacted, from documents before they are released to the public. When the documents are released it is common for the redacted text to be replaced with black bars.

In this exercise you will write a program that redacts all occurrences of sensitive words in a text file by replacing them with asterisks. Your program should redact sensitive words wherever they occur, even if they occur in the middle of another word. The list of sensitive words will be provided in a separate text file. Save the redacted version of the original text in a new file. The names of the original text file, sensitive words file, and redacted file will all be provided by the user.

> You may find the `replace` method for strings helpful when completing this exercise. Information about the `replace` method can be found on the Internet.

For an added challenge, extend your program so that it redacts words in a case insensitive manner. For example, if `exam` appears in the list of sensitive words then redact `exam`, `Exam`, `ExaM` and `EXAM`, among other possible capitalizations.

Exercise 170: Missing Comments

(*Solved, 48 Lines*)

When one writes a function, it is generally a good idea to include a comment that outlines the function's purpose, its parameters and its return value. However, sometimes comments are forgotten, or left out by well-intentioned programmers that plan to write them later but then never get around to it.

Create a Python program that reads one or more Python source files and identifies functions that are not immediately preceded by a comment. For the purposes of this exercise, assume that any line that begins with `def`, followed by a space, is the beginning of a function definition. Assume that the comment character, `#`, will be the first character on the previous line when the function has a comment. Display the names of all of the functions that are missing comments, along with the file name and line number where the function definition is located.

The user will provide the names of one or more Python files as command line arguments, all of which should be analyzed by your program. An appropriate error message should be displayed for any files that do not exist or cannot be opened. Then your program should process the remaining files.

Exercise 171: Consistent Line Lengths

(*45 Lines*)

While 80 characters is a common width for a terminal window, some terminals are narrow or wider. This can present challenges when displaying documents containing paragraphs of text. The lines might be too long and wrap, making them difficult to read, or they might be too short and fail to make good use of the available space.

Write a program that opens a file and displays it so that each line is as full as possible. If you read a line that is too long then your program should break it up into words and add them to the current line until it is full. Then your program should start a new line and display the remaining words. Similarly, if you read a line that is too short then you will need to use words from the next line of the file to finish filling the current line of output. For example, consider a file containing the following lines from "Alice's Adventures in Wonderland":

```
Alice was
beginning to get very tired of sitting by her
sister
on the bank, and of having nothing to do: once
or twice she had peeped into the book her sister
was reading, but it had
no
pictures or conversations in it,"and what is
the use of a book," thought Alice, "without
pictures or conversations?"
```

When formatted for a line length of 50 characters, it should be displayed as:

```
Alice was beginning to get very tired of sitting
by her sister on the bank, and of having nothing
to do: once or twice she had peeped into the book
her sister was reading, but it had no pictures or
conversations in it, "and what is the use of a
book," thought Alice, "without pictures or
conversations?"
```

Ensure that your program works correctly for files containing multiple paragraphs of text. You can detect the end of one paragraph and the beginning of the next by looking for lines that are empty once the end of line marker has been removed.

Hint: Use a constant to represent the maximum line length. This will make it easier to update the program when a different line length is needed.

Exercise 172: Words with Six Vowels in Order

(*56 Lines*)

There is at least one word in the English language that contains each of the vowels A, E, I, O, U and Y exactly once and in order. Write a program that searches a file containing a list of words and displays all of the words that meet this constraint. The user will provide the name of the file that will be searched. Display an appropriate error message and exit the program if the user provides an invalid file name, or if something else goes wrong while your program is searching for words with six vowels in order.