FILE INPUT AND OUTPUT

Ultimately, one goal of computer programming is to write interesting and useful programs, and then test them on realistic input data. For example, counting all the words in a book, or finding all the words that have exactly 9 letters in them, or the words that have 3 vowels, etc. "Interesting" programs tend to be those that have large amounts of input and/or output.

Output

If you have a program that needs to print out a lot of output, it's a good idea to instead write all the output to a file so that you can view it later. By "a lot" of output, I would say more output than you can easily see on a screen. When we perform file output, we actually have our computer program create a new file to contain the output. It turns out that a file is its own data type, meaning that we can create a variable in a Python program to refer to an actual file on the hard drive. So, now he have learned six data types: int, float, str, boolean, list, file.

Here are the steps to accomplishing file output, followed by an example program.

1. First, create the output file by using the built-in open( ) function. You generally should do this early in the program. You are free to choose whatever names you want for the name of the file, and the name of the variable referring to the file within your program. For example:
   ```
   outFile = open("output.txt", "w")
   ```
   Note that the 2nd parameter to the open( ) function is the letter "w" which stands for write. We are telling Python that we want to write to this file.

2. When you are ready to print something to the file, use the built-in write( ) function instead of the usual print( ) function. You need to specify the file variable as well as what string to write to it. As an example:
   ```
   outFile.write("a modern major general\n")
   ```
   Note that it is a good idea to end the string with \n, which is the newline character. This will allow any subsequent output to appear on the next line.

3. At the end of the program, tell Python to close the file, using the built-in close( ) function:
   ```
   outFile.close()
   ```
   By closing the file, we ensure that all of the output actually gets written to the file. Because of the way that the operating system usually handles files, if we forget to "close" the file, the last part of our output may get omitted from the file.

```
# output.py - Demo of file output
# Can we handle a lot of numbers?

# Create an output file.
outFile = open("numbers.txt", "w")

# Put the calculated values into the file.
for i in range(14, 23, 2):
    outFile.write(str(i) + "\n")

# We're done with the file.
print("I am done writing to the file.")
outFile.close()
```

Exercise:

1. In the above example program, what numbers appear in the output file?

2. In the above program, when writing the number i to the file, why was it necessary to call str(i)? In other words, why couldn't we say `outFile.write(i + "\n")`?

3. Write a program that prints the numbers 1-100 on the screen, one number per line.

4. Write a second program that prints the numbers 1-100 to a file. Call the file counting.txt.

<u>Input</u>

If your program needs a lot of input (e.g. adding 10 numbers), then rather than having the user type so much input each time the program is run, it is more convenient to use file input. The steps for accomplishing file input are analogous to what you saw for file output.

1. First, open the input file.
   ```
   inFile = open("input.txt", "r")
   ```

2. We read lines from the input file one by one. Each line is a string, just like the interactive input() function would provide us.
   ```
   for line in inFile:
       # do whatever you need with the input
   ```

3. When done reading input, close the file.
   ```
   inFile.close()
   ```

When doing file input, it is very useful to write a loop that reads each line of the file, like you saw in step 2 above:

```
for line in inFile:
```

Here, `inFile` is the file object, and `line` is a string containing one line from the file. There is an important technical point about these lines: the newline character is included at the end of the line! This can be slightly inconvenient. For example, let's suppose you wanted to open a text file, and simply print out all the lines. We would do it like this:

```
 1  filename = input("What is the file name?  ")
 2  file = open(filename, "r")
 3  for line in file:
 4      print(line)
```

The output will show all the lines in the input file … double spaced! This is because the print( ) statement always prints a newline character after printing its string argument. Often we want to get rid of the newline character at the end of our line variable. This is easy, if we remember that the index of the last character of the string is –1. The following statement will delete this last character, and would need to go before line 4 above:

```
line = line[0 : –1]
```

<u>Combining interactive and file I/O</u>

Can a program have both file I/O and interactive I/O? Yes! In fact, it's often a good idea. If a program needs to open a file and does not know the name, interactive input may be essential. On the other hand, if writing an output file takes a long time, interactive output can reassure the user that the program is actually doing useful work and not hanging (i.e. not suffering an infinite loop or waiting for interactive input).

For example, let's say you were writing a program to print the first one million primes. That could take a while. You could print a "." after each 1,000th prime is written. The appearance of the next dot on the screen would indicate to the user that another piece of the task has been completed. Alternatively, you could print more explicit information about the progress of the algorithm, such as occasionally printing to the screen how many primes have so far been written.

You need to strike a balance between providing too much or too little progress information. Too much (such as printing a message after every 5th prime number) might slow down your program or be unreadable to the user because it scrolls off the screen too rapidly. Too little (such as printing after every 100,000th prime) might not be reassuring to the user. It's a good rule of thumb to show some change on the screen every several seconds if the user is going to be at the terminal watching the program. If the entire process is going to take hours or days, then less frequent progress information would be appropriate.

Finally, now that you have seen both file input and file output, you should notice that the structure is the same. There are three steps: opening the file, reading or writing the file, and then closing the file. *Don't forget to close the file!* Otherwise, the last I/O operation you perform may be delayed by the operating system.

Exercises:

1. Create a text file called input.txt, and enter 10 numbers, one per line. This will be our test input.

2. Based on the steps listed above, write a program that reads input.txt and finds the sum of all the numbers in the input file. Your program should be written in such a way that it does not matter how long the input file is. In other words, don't tell your program to read only the first 10 numbers in the input file.

3. How would you find the average of the numbers in input.txt?

4. Perform frequency analysis of the letters in a text file.

5. What is wrong with the following code?
   ```
   filename = input("Enter file name:  ")
   file = open("filename", "r")
   for line in file:
       # program continues
   ```

6. What is wrong with the following code?
   ```
   filename = input("Enter file name:  ")
   file = open(filename, "r")
   for line in filename:
       # program continues
   ```

7. Show how we can use Python to read a text file, and report:
   a. The total number of words.
   b. The number of words that begin with a capital letter.
   c. The longest word.
   d. Whether any words appear more than once.
   e. Whether any lines are identical.
   f. Whether any lines are blank.

g. Whether any lines contain more than 80 characters.
h. The distribution of word lengths (i.e. the number of words of length n, for suitable values of n such as 1-12).

8. How would we read an input text file, and tell which words are misspelled?

9. A twin prime is a prime number that is 2 more than or 2 less than another prime number. For example, 17 and 19 are twin primes. Write a program to print to a file all twin primes less than 1,000,000. We can do so without testing the primality of each number more than once. For example, when considering the pair of numbers 89 and 91, we would discover that 89 is prime and 91 is not prime. It would be wasteful to consider the pair 91 and 93 beginning by asking if 91 is prime again. So, it might be simpler to create a list of primes, and later detect if consecutive primes differ by 2.

10. Suppose a text file contains a list of integers, one per line. There is no stray text besides the numbers. Show how we can use Python to determine:
    a. The largest number.
    b. How many numbers are even.
    c. How many numbers are between 100 and 999, inclusive.

11. Suppose a cash register receipt lists items one per line. The first 30 characters on each line identify a product. Beyond the 30th character is the price.
    a. How would we find the total of all the prices on the receipt?

    b. Let's suppose that taxable items appear on the receipt with the letter T in the 30th position on the line, indicating that 7% needs to be added for sales tax. How would you modify the program to handle tax?

EXCEPTIONS

You may recall that there are three broad types of errors in computing: syntax errors, run-time errors and logical errors. Let's consider run-time errors. A run-time error occurs when a Python statement asks the computer to do something inappropriate or impossible, such as dividing by zero. The program immediately halts and the Python system emits an error message identifying the errant instruction. In programming, we use the word *exception* to mean any operation that could result in a run-time error.

The good news is that we can write our programs to catch exceptions before they actually result in a run-time error killing our program. In other words, many run-time errors can be prevented by how we write the code. The Python language provides a mechanism for allowing us to handle the possibility of a run-time error occurring. This mechanism is called "try-except", named after the two keywords `try` and `except` that we employ for this purpose.

Try/except is analogous to an if-statement. In a nutshell, we would like to do something like this:

```
if an exception might occur:
   allow me to handle the situation so the program can recover
else:
    proceed normally
```

Here is how the words `try` and `except` are used in Python:

```
try:
    perform an operation that might cause a run-time error
except <name of exception>:
    exactly how we should respond to the exception
```

For beginners, a common mistake is to over-use the try/except mechanism. Try and except are not meant to be a replacement for an if-else statement. Try and except are only used for run-time exceptions. The following table lists some common exceptions that we can handle. You don't need to memorize this list.

Name of exception	Examples	Python's error message
ZeroDivisionError	5 / 0	division by zero

ValueError	math.sqrt(–5)
	math.asin(2)
	int("2+1")
	float("1/2")
		math domain error
		math domain error
		invalid literal for int() with base 10: '2+1'
		could not convert string to float: '1/2'

FileNotFoundError	open("null.txt")	No such file or directory: 'null.txt'
IndexError	L[500000] s[500000]	list index out of range string index out of range

As examples, let's see how we can recover from two of the most common problems: not able to open a file, and trying to convert a bad string to an integer. Here is a minimalist approach to handling a FileNotFoundError (with line numbers included for reference):

```
 1  fileName = input("What is the file name?  ")
 2
 3  # Attempt to open the file.
 4  try:
 5      file = open(fileName)
 6  except FileNotFoundError:
 7      print("Sorry, I cannot find the file", fileName)
 8
```

This code will avoid the run-time error associated with opening a nonexistent file. The problem with this solution is that after printing the error message, we still have no file. Supposedly, this program will next begin to read lines of the file. We need to provide a little more support in case we observed the exception. Here are some possibilities:

- Have the program exit after printing the error message.
- Create a boolean variable `success`, which is initially `False`, but set to `True` between lines 5 and 6. This way, we can write an statement `if success:` before trying to read lines of the file.
- Enclose the try/except in a loop, to allow the user to try again. This is reminiscent of how we handled error checking in the past.

For interactive programs, the most satisfying option may be to loop to ask the user to enter the file name again. Here is what the code looks like:

```
# Attempt to open the file.
success = False
while not success:
    try:
        fileName = input("What is the file name?  ")
        file = open(fileName)
        success = True
    except FileNotFoundError:
        print("Sorry, I cannot find the file", fileName)

```

Notice the changes – First, it was necessary to move the file name prompt inside the loop. Also notice the use of the boolean variable `success`.

If all goes well, and the user enters the correct file name the first time, we execute the lines 2-7 and then skip the `except` block on lines 8 and 9. But if the user makes a mistake, we execute lines 2-6, skip line 7 and go directly to lines 8 and 9, and restart the loop to give the user a second chance.

Inside the `try` block, there are three statements, and the second statement is the one that could cause the exception – opening the file. If the exception occurs, the program immediately jumps down to the `except` block corresponding to the name of the exception.

Let's look at the example of catching a `ValueError` arising from converting a bad string to an int. The structure of the code is similar to catching a bad file name:

```
1  # Read an integer, but be careful!
2  success = False
3  while not success:
4      try:
5          n = int(input("Enter an integer:  "))
6          success = True
7      except ValueError:
8          print("Sorry, I don't understand.  Try again.")
9
```

If the user enters a valid string that can be converted to an integer, then we execute lines 2-6 and skip lines 7 and 8 that handle the exception, and we exit the loop. On the other hand, if the user enters a bad string, then we execute lines 2-5, skip line 6, and go to lines 7 and 8, and iterate again because `success` is still False.

Exercises:

1. Practice handling the exception of dividing by zero. Write a program that asks the user for two numbers. Divide them. Complain if the second number is zero, and give the user another chance (in a while loop) to re-enter it.

2. Give an example of a situation where an IndexError exception could occur in a program. Write a try/except block to handle the exception.