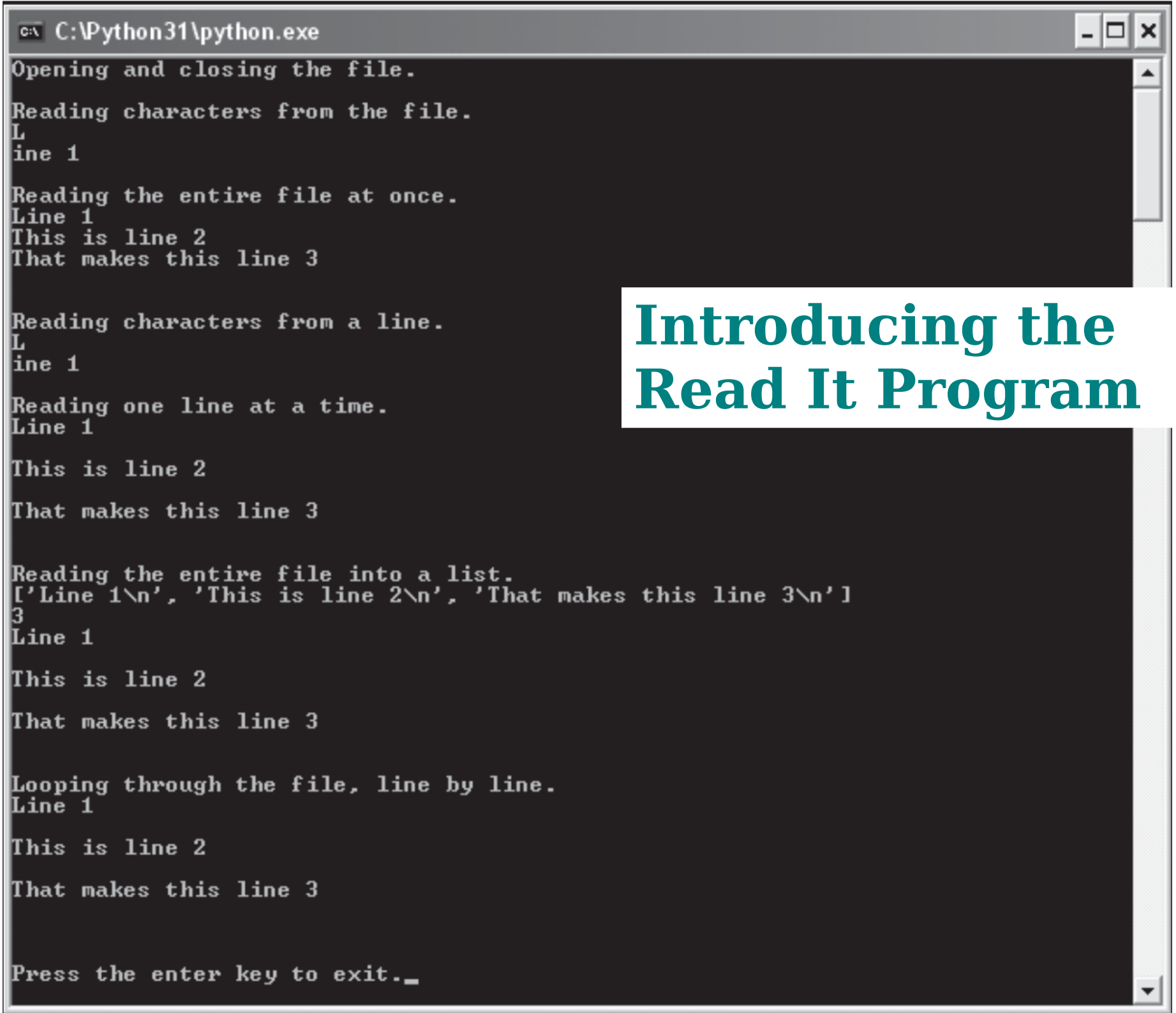
Chapter 7

Files and Exceptions: The Trivia Challenge Game



Introducing the Read It Program

read_it.py

```
# Read It
# Demonstrates reading from a text file

print("Opening and closing the file.")
text_file = open("read_it.txt", "r")
text_file.close()

print("\nReading characters from the file.")
text_file = open("read_it.txt", "r")
print(text_file.read(1))
print(text_file.read(5))
text_file.close()

print("\nReading the entire file at once.")
text_file = open("read_it.txt", "r")
whole_thing = text_file.read()
print(whole_thing)
text_file.close()
```

```
print("\nReading characters from a line.")
text_file = open("read_it.txt", "r")
print(text_file.readline(1))
print(text_file.readline(5))
text_file.close()

print("\nReading one line at a time.")
text_file = open("read_it.txt", "r")
print(text_file.readline())
print(text_file.readline())
print(text_file.readline())
text_file.close()

print("\nReading the entire file into a list.")
text_file = open("read_it.txt", "r")
lines = text_file.readlines()
print(lines)
print(len(lines))
for line in lines:
    print(line)
text_file.close()
```

```
print("\nLooping through the file, line by line.")
text_file = open("read_it.txt", "r")
for line in text_file:
    print(line)
text_file.close()

input("\n\nPress the enter key to exit.")
```

read_it.txt

Line 1
This is line 2
That makes this line 3

Opening and Closing a File

- *plain text files*: files made up of only ASCII characters.

- To open a file:

```
text_file = open("read_it.txt", "r")
```

- The 1st argument is the file name, the 2nd argument is the access mode:

Mode	Description
"r"	Read from a text file. If the file doesn't exist, Python will complain with an error.
"w"	Write to a text file. If the file exists, its contents are overwritten. If the file doesn't exist, it's created.
"a"	Append a text file. If the file exists, new data is appended to it. If the file doesn't exist, it's created.
"r+"	Read from and write to a text file. If the file doesn't exist, Python will complain with an error.
"w+"	Write to and read from a text file. If the file exists, its contents are overwritten. If the file doesn't exist, it's created.
"a+"	Append and read from a text file. If the file exists, new data is appended to it. If the file doesn't exist, it's created.

- After opening the file, we access it through the variable text_file, which represents a file object.

- The simplest file object methods is **close()**, which closes the file, sealing it off from further reading or writing until the file is opened again:

text_file.close()

Reading Characters from a File

- We can read a file's contents with the **read()** file object method. **read()** allows you to read a specified number of characters from a file, which the method returns as a string:

```
>>> text_file = open("read_it.txt", "r")
>>> print(text_file.read(1))
L
>>> print(text_file.read(5))
ine 1
```

- Notice that we read the 5 characters following the "L". Python remembers where we last left off. When you read to the end of a file, subsequent reads return the empty string.

- To start back at the beginning, you can close and open it:

```
>>> text_file.close()
>>> text_file = open("read_it.txt", "r")
```

- If you don’t specify the number of characters to be read, Python returns the entire file as a string:

```
>>> whole_thing = text_file.read()
>>> print(whole_thing)
Line 1
This is line 2
That makes this line 3
```

- If a file is small enough, reading the entire thing at once may make sense. But it is not a good idea if the file is big.

- Since we’ve read the entire file, any subsequent reads will just return the empty string.

Reading Characters from a Line

- **readline()** reads characters from the current line. You just pass the number of characters you want read from the current line and the method returns them as a string.

- If you don't pass a number, the method reads from the current position to the end of the line. Once you read all of the characters of a line, the next line becomes the current line.

```
>>> text_file = open("read_it.txt", "r")
>>> print(text_file.readline(1))
L
>>> print(text_file.readline(5))
ine 1
>>> text_file.close()
```

- **readline()** reads characters from the current line only, while read() reads characters from the entire file.

- **readline()** is usually invoked to read one line of text at a time.

```
>>> text_file = open("read_it.txt", "r")
>>> print(text_file.readline())
Line 1

>>> print(text_file.readline())
This is line 2

>>> print(text_file.readline())
That makes this line 3

>>> text_file.close()
```

- A blank line appears after each line. That's because each line in the text file ends with a newline character ("\n").

Reading All Lines into a List

- **readlines()** reads a text file into a **list**, where each line of the file becomes a string element in the list:

```
>>> text_file = open("read_it.txt", "r")
>>> lines = text_file.readlines()
>>> print(lines)
['Line 1\n', 'This is line 2\n', 'That makes this line 3\n']
>>> print(len(lines))
3
>>> for line in lines:
        print(line)
Line 1

This is line 2

That makes this line 3

>>> text_file.close()
```

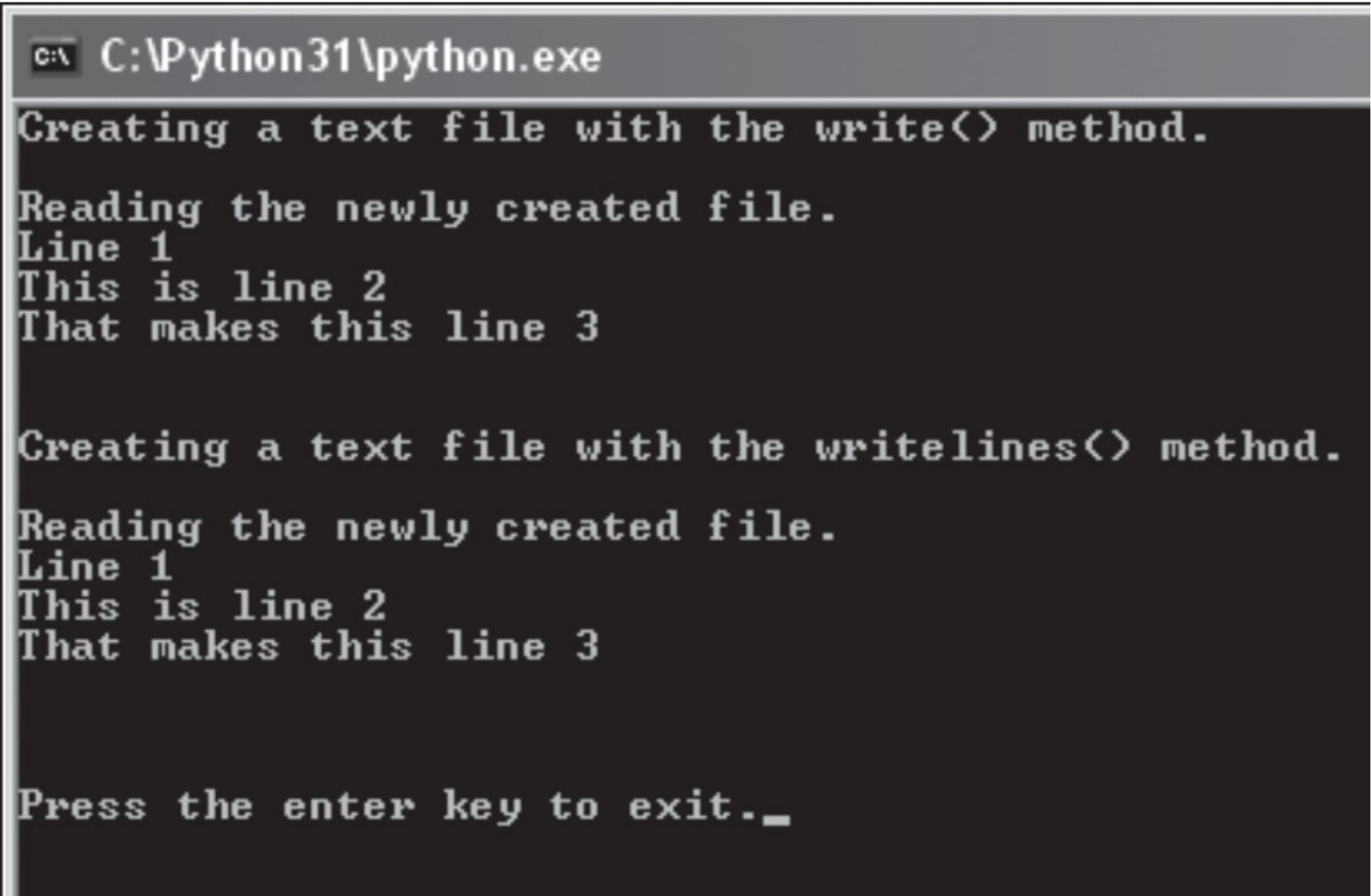
Looping through a File

- You can also loop directly through the lines of a file using a for loop:

```
>>> text_file = open("read_it.txt", "r")
>>> for line in text_file:
        print(line)

Line 1

This is line 2

That makes this line 3

>>> text_file.close()
```

- The loop variable, eg, line, gets each line of the file, in succession. This technique is the most elegant solution if you want to move through a file one line at a time.

Introducing the Write It Program



write_it.py

```
# Write It
# Demonstrates writing to a text file

print("Creating a text file with the write() method.")
text_file = open("write_it.txt", "w")
text_file.write("Line 1\n")
text_file.write("This is line 2\n")
text_file.write("That makes this line 3\n")
text_file.close()

print("\nReading the newly created file.")
text_file = open("write_it.txt", "r")
print(text_file.read())
text_file.close()
```

```
print("\nCreating a file with the writelines() method.")
text_file = open("write_it.txt", "w")
lines = ["Line 1\n",
         "This is line 2\n",
         "That makes this line 3\n"]
text_file.writelines(lines)
text_file.close()

print("\nReading the newly created file.")
text_file = open("write_it.txt", "r")
print(text_file.read())
text_file.close()

input("\n\nPress the enter key to exit.")
```

Writing Strings to a File

- To write strings to a file, we open a file in write mode:

```
text_file = open("write_it.txt", "w")
```

- The file write_it.txt springs into existence as an empty text file just waiting for the program to write to it.

- If write_it.txt had already existed, it would have been replaced with a brand-new, empty file and all of its original contents would have been erased.

- use **write()** to write a string to the file:

```
text_file.write("Line 1\n")
text_file.write("This is line 2\n")
text_file.write("That makes this line 3\n")
```

- **write()** does not automatically insert a newline character at the end of a string it writes. You have to put newlines in where you want them.

- Without the 3 newline characters, the program would write one, long line to the file.

- To achieve the same result, we could just as easily have stuck all 3 of the previous strings together to form one long string, "Line 1\n This is line 2\n That makes this line 3\n" , and written that string to the file with a single **write()**.

- To prove that the writing worked, we can read and print the entire contents of the file, as done in the code.

Writing a List of Strings to a File

- **writelines()** works with a list of strings and writes a list of strings to a file.

- We open the same file, write_it.txt, which means we wipe out the existing file and start with a new, empty one:

```
text_file = open("write_it.txt", "w")
```

- create a list of strings to be written, in order, to the file:

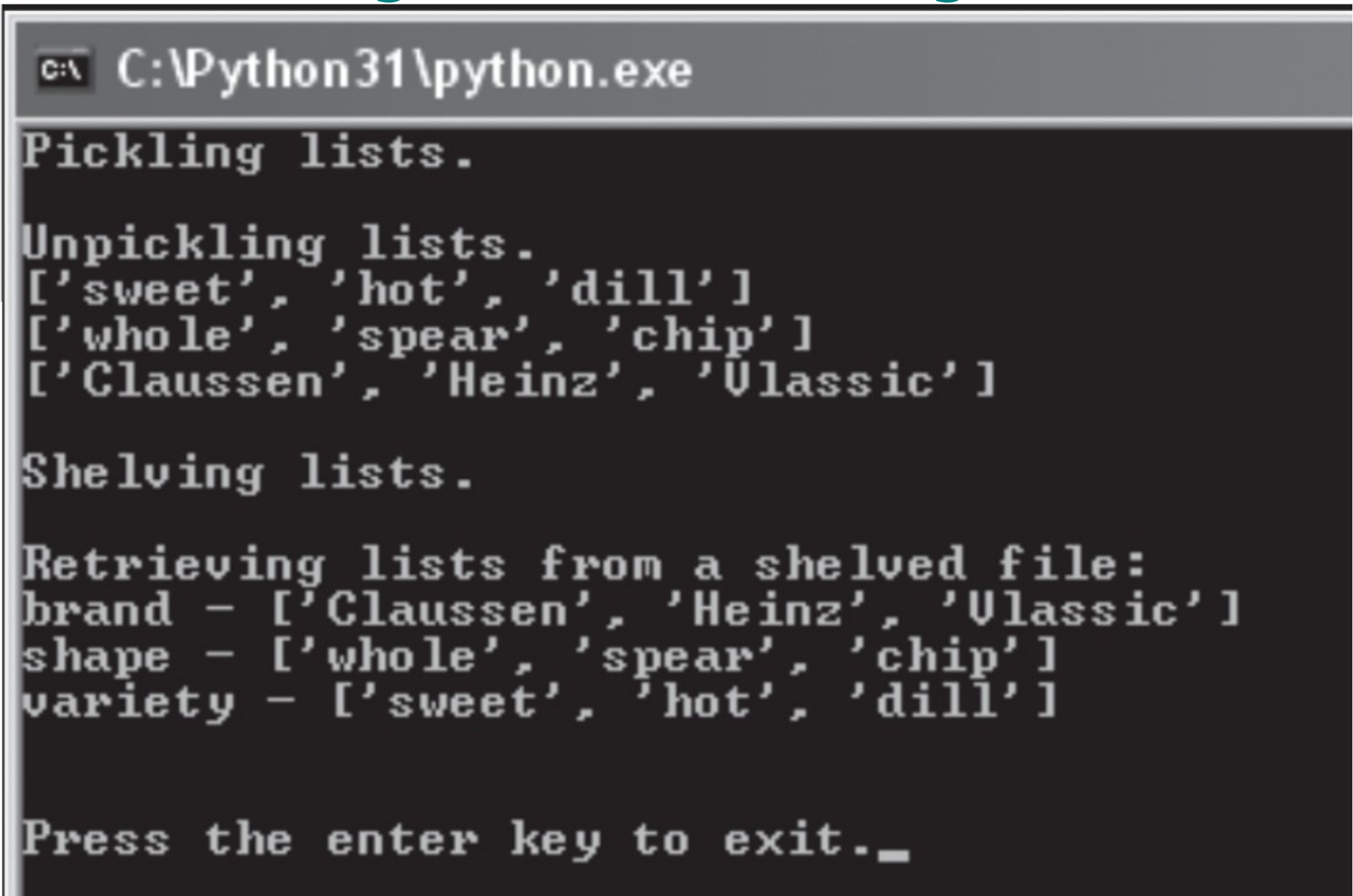
```
lines = ["Line 1\n",
         "This is line 2\n",
         "That makes this line 3\n"]
```

- write the entire lists of strings to the file with **writelines()**:

```
text_file.writelines(lines)
```

Method	Description
`close()`	Closes the file. A closed file cannot be read from or written to until opened again.
`read([`*`size`*`])`	Reads *`size`* characters from a file and returns them as a string. If size is not specified, the method returns all of the characters from the current position to the end of the file.
`readline([`*`size`*`])`	Reads *`size`* characters from the current line in a file and returns them as a string. If size is not specified, the method returns all of the characters from the current position to the end of the line.
`readlines()`	Reads all of the lines in a file and returns them as elements in a list.
`write(`*`output`*`)`	Writes the string *`output`* to a file.
`writelines(`*`output`*`)`	Writes the strings in the list *`output`* to a file.

Introducing the Pickle It Program



pickle_it.py

```
# Pickle It
# Demonstrates pickling and shelving data

import pickle, shelve

print("Pickling lists.")
variety = ["sweet", "hot", "dill"]
shape = ["whole", "spear", "chip"]
brand = ["Claussen", "Heinz", "Vlassic"]
f = open("pickles1.dat", "wb")
pickle.dump(variety, f)
pickle.dump(shape, f)
pickle.dump(brand, f)
f.close()
```

```
print("\nUnpickling lists.")
f = open("pickles1.dat", "rb")
variety = pickle.load(f)
shape = pickle.load(f)
brand = pickle.load(f)
print(variety)
print(shape)
print(brand)
f.close()

print("\nShelving lists.")
s = shelve.open("pickles2.dat")
s["variety"] = ["sweet", "hot", "dill"]
s["shape"] = ["whole", "spear", "chip"]
s["brand"] = ["Claussen", "Heinz", "Vlassic"]
s.sync()    # make sure data is written
```

```
print("\nRetrieving lists from a shelved file:")
print("brand -", s["brand"])
print("shape -", s["shape"])
print("variety -", s["variety"])
s.close()

input("\n\nPress the enter key to exit.")
```

Pickling Data and Writing It to a File

- The **pickle** module allows you to pickle and store more complex data in a file. The **shelve** module allows you to store and randomly access pickled objects in a file:

```
import pickle, shelve
```

- Instead of writing characters to a file, you can write a **pickled object** to a file. Pickled objects are stored in files much like characters; you can store and retrieve them sequentially:

```
variety = ["sweet", "hot", "dill"]
shape = ["whole", "spear", "chip"]
brand = ["Claussen", "Heinz", "Vlassic"]
```

- Then open the new file to store the pickled lists:

```
f = open("pickles1.dat", "wb")
```

- Pickled objects must be stored in a **binary** file—they can't be stored in a text file.

Mode	Description
`"rb"`	Read from a binary file. If the file doesn't exist, Python will complain with an error.
`"wb"`	Write to a binary file. If the file exists, its contents are overwritten. If the file doesn't exist, it's created.
`"ab"`	Append a binary file. If the file exists, new data is appended to it. If the file doesn't exist, it's created.
`"rb+"`	Read from and write to a binary file. If the file doesn't exist, Python will complain with an error.
`"wb+"`	Write to and read from a binary file. If the file exists, its contents are overwritten. If the file doesn't exist, it's created.
`"ab+"`	Append and read from a binary file. If the file exists, new data is appended to it. If the file doesn't exist, it's created.

- Then pickle and store the 3 lists variety, shape, and brand in the file pickles1.dat using the pickle.**dump()** function.

- pickle.**dump()** requires 2 arguments: the data to pickle and the file in which to store it.

```
pickle.dump(variety, f)
pickle.dump(shape, f)
pickle.dump(brand, f)
f.close()
```

- This code pickles the list referred to by variety and writes the whole thing as one object to pickles1.dat. Then shape. Then brand.

- You can pickle a variety of objects, including:

* Numbers
* Strings
* Tuples
* Lists
* Dictionaries

Reading Data from a File & Unpickling It

- Now we retrieve and unpickle the 3 lists with pickle.**load()**:

```
f = open("pickles1.dat", "rb")
variety = pickle.load(f)
shape = pickle.load(f)
brand = pickle.load(f)
```

- The code reads the 1st pickled object in the file, unpickles it to get the list [“sweet”, “hot”, “dill”], and assigns the list to variety. Then the code reads the next pickled object from the file, unpickles it to get the list [“whole”, “spear”, “chip”], and assigns the list to shape. Finally, the code reads the last one from the file, unpickles it to get the list [“Claussen”, “Heinz”, “Vlassic”] , and assigns the list to brand.

Function	Description
`dump(object, file, [,bin])`	Writes pickled version of `object` to `file`. If `bin` is `True`, `object` is written in binary format. If `bin` is `False`, `object` is written in less efficient, but more human-readable, text format. The default value of `bin` is equal to `False`.
`load(file)`	Unpickles and returns the next pickled object in `file`.

Using a Shelf to Store Pickled Data

- Using the **shelve** module, we create a *shelf* that acts like a dictionary, which provides random access to the lists.

- create a shelf, s:

s = shelve.open("pickles2.dat")

- **shelve.open()** works a lot like open(). But **shelve.open()** works with a file that stores pickled objects, not characters.

- When you call **shelve.open()**, Python may add an extension to the file name you specify. Python may also create additional files to support the newly created shelf.

- **shelve.open()** requires one argument: a file name. It also takes an optional access mode. If you don't supply an access mode, it defaults to "c" .

Mode	Description
"c"	Open a file for reading or writing. If the file doesn't exist, it's created.
"n"	Create a new file for reading or writing. If the file exists, its contents are overwritten.
"r"	Read from a file. If the file doesn't exist, Python will complain with an error.
"w"	Write to a file. If the file doesn't exist, Python will complain with an error.

- Add 3 lists to the shelf:

s["variety"] = ["sweet", "hot", "dill"]
s["shape"] = ["whole", "spear", "chip"]
s["brand"] = ["Claussen", "Heinz", "Vlassic"]

- The key "variety" is paired with ["sweet", "hot", "dill"]. The key "shape" is paired with ["whole", "spear", "chip"]. And the key "brand" is paired with ["Claussen", "Heinz", "Vlassic"].

- One important thing is that a shelf key can only be a string.

- Python writes changes to a shelf file to a buffer and then periodically writes the buffer to the file. To make sure the file reflects all the changes to a shelf, you can invoke **sync()**:

s.sync() # make sure data is written

Using a Shelf to Retrieve Pickled Data

- Since a shelf acts like a dictionary, you can randomly access pickled objects from it by supplying a key. To prove this, we access the pickled lists in s in reverse order:

```
print("brand -", s["brand"])
print("shape -", s["shape"])
print("variety -", s["variety"])
```

Handling Exceptions

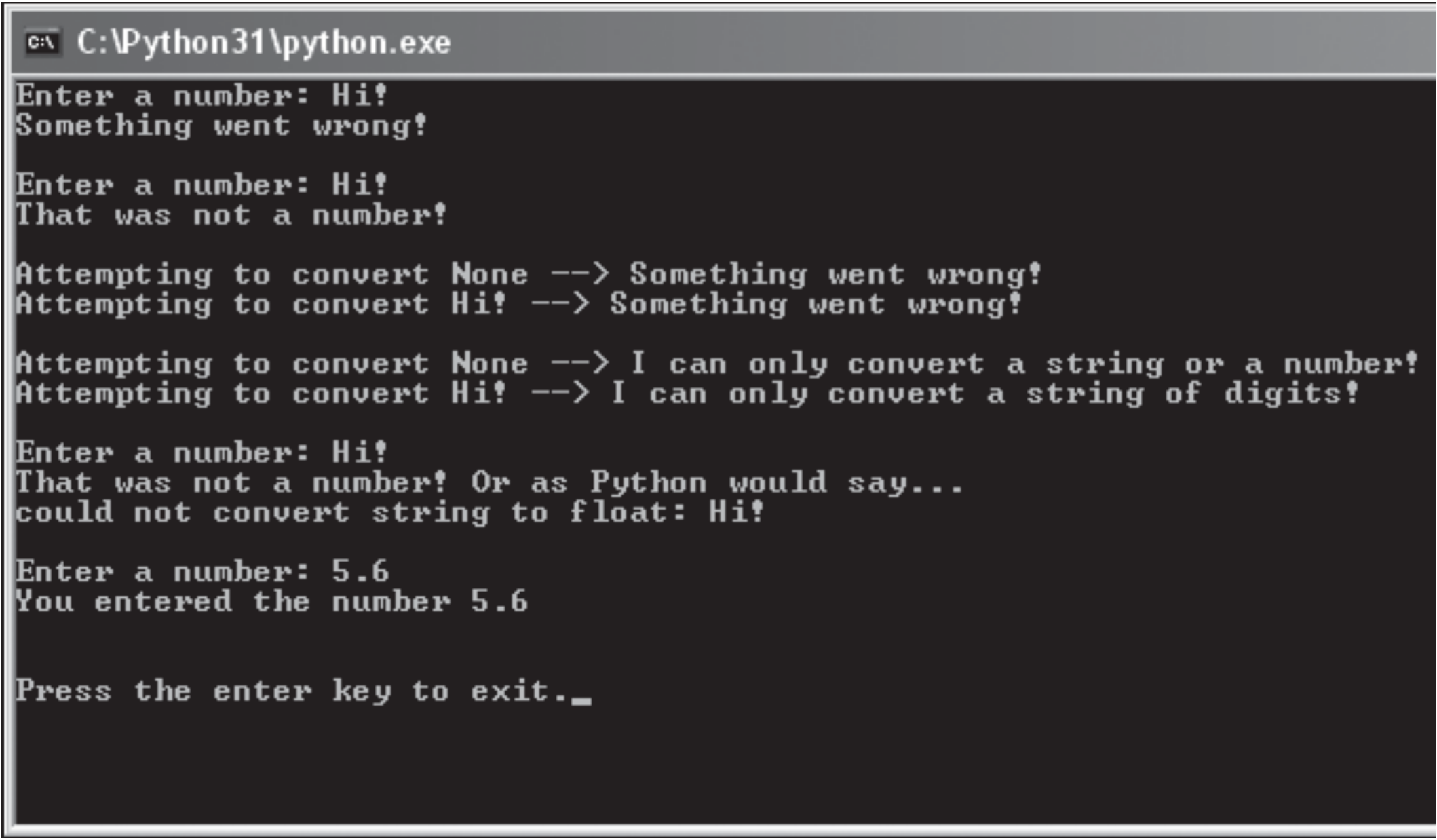
- When Python runs into an error, it stops the program and displays an error message. More precisely, it raises an *exception*, indicating that something exceptional has occurred

- If nothing is done with the exception, Python halts what it's doing and displays an error message detailing the exception:

```
>>> num = float("Hi!")
Traceback (most recent call last):
    File "<pyshell#1>", line 1, in <module>
        num = float("Hi!")
ValueError: could not convert string to float: Hi!
```

- Using Python's exception handling functionality, you can intercept and handle exceptions so that your program doesn't end abruptly.

Introducing the Global Reach Program



handle_it.py

```
# Handle It
# Demonstrates handling exceptions

# try/except
try:
    num = float(input("Enter a number: "))
except:
    print("Something went wrong!")

# specifying exception type
try:
    num = float(input("\nEnter a number: "))
except ValueError:
    print("That was not a number!")
```

```
# handle multiple exception types
print()
for value in (None, "Hi!"):
    try:
        print("Attempting to convert", value,"-->", end=" ")
        print(float(value))
    except (TypeError, ValueError):
        print("Something went wrong!")

print()
for value in (None, "Hi!"):
    try:
        print("Attempting to convert", value,"-->", end=" ")
        print(float(value))
    except TypeError:
        print("I can only convert a string or a number!")
    except ValueError:
        print("I can only convert a string of digits!")
```

```
# get an exception's argument
try:
    num = float(input("\nEnter a number: "))
except ValueError as e:
    print("That’s not a number! Or as it would say...")
    print(e)

# try/except/else
try:
    num = float(input("\nEnter a number: "))
except ValueError:
    print("That was not a number!")
else:
    print("You entered the number", num)

input("\n\nPress the enter key to exit.")
```

Using try statement with an except clause

- The most basic way to *handle* (or *trap*) exceptions is to use the **try** statement with an except clause.

- By using a **try** statement, you section off some code that could potentially raise an exception. Then, you write an **except** clause with a block of statements that are executed only if an exception is raised:

```
try:
    num = float(input("Enter a number: "))
except:
    print("Something went wrong!")
```

- If the call to float() raises an exception, the exception is caught and the user is informed that Something went wrong!

- If no exception is raised, num gets the number entered and the code skips the except clause, continuing to the next.

Specifying an Exception Type

- Different kinds of errors result in different types of exceptions. There are over two dozen exception types.

- The except clause lets you specify exactly which type of exceptions it will handle. To specify a single exception type, you just list the specific type of exception after except.

Exception Type	Description
`IOError`	Raised when an I/O operation fails, such as when an attempt is made to open a nonexistent file in read mode.
`IndexError`	Raised when a sequence is indexed with a number of a nonexistent element.
`KeyError`	Raised when a dictionary key is not found.
`NameError`	Raised when a name (of a variable or function, for example) is not found.
`SyntaxError`	Raised when a syntax error is encountered.
`TypeError`	Raised when a built-in operation or function is applied to an object of inappropriate type.
`ValueError`	Raised when a built-in operation or function receives an argument that has the right type but an inappropriate value.
`ZeroDivisionError`	Raised when the second argument of a division or modulo operation is zero.

```
try:
    num = float(input("\nEnter a number: "))
except ValueError:
    print("That was not a number!")
```

- Now print will only execute if a **ValueError** is raised.

- If any other type of exception is raised inside the try statement, the except clause won't catch it and the program will come to a halt.

- It's good programming practice to specify exception types so that you handle each individual case.

- In fact, it's dangerous to catch all exceptions the way we did in the 1st except clause. This is because the code could blindly run without a correct treatment of the exception.

Handling Multiple Exception Types

- One way to trap for multiple exception types is to list them in a single except clause as a comma-separated group enclosed in a set of parentheses:

```
for value in (None, "Hi!"):
    try:
        print("Attempting to convert",value,"-->",end=" ")
        print(float(value))
    except (TypeError, ValueError):
        print("Something went wrong!")
```

- This code tries to convert 2 different values to a real number. Both fail, but each raises a different exception type.

- float(None) raises a TypeError because the function can only convert strings and numbers. float("Hi!") raises a ValueError because, while "Hi!" is a string, the characters in the string are of the wrong value.

- Another way to catch multiple exceptions is with multiple except clauses:

```
for value in (None, "Hi!"):
    try:
        print("Attempting to convert",value,"-->",end=" ")
        print(float(value))
    except TypeError:
        print("I can only convert a string or a number!")
    except ValueError:
        print("I can only convert a string of digits!")
```

- Using multiple except clauses allows you to define unique reactions to different types of exceptions from the same try block.

Getting an Exception's Argument

- When an exception occurs, it may have an associated value, the exception's *argument*. The argument is usually an official message from Python describing the exception.

- You can receive the argument if you specify a variable after the exception type, preceded by the keyword **as**:

```
try:
    num = float(input("\nEnter a number: "))
except ValueError as e:
    print("That was not a number! Or as it would say...")
    print(e)
```

Adding an else Clause

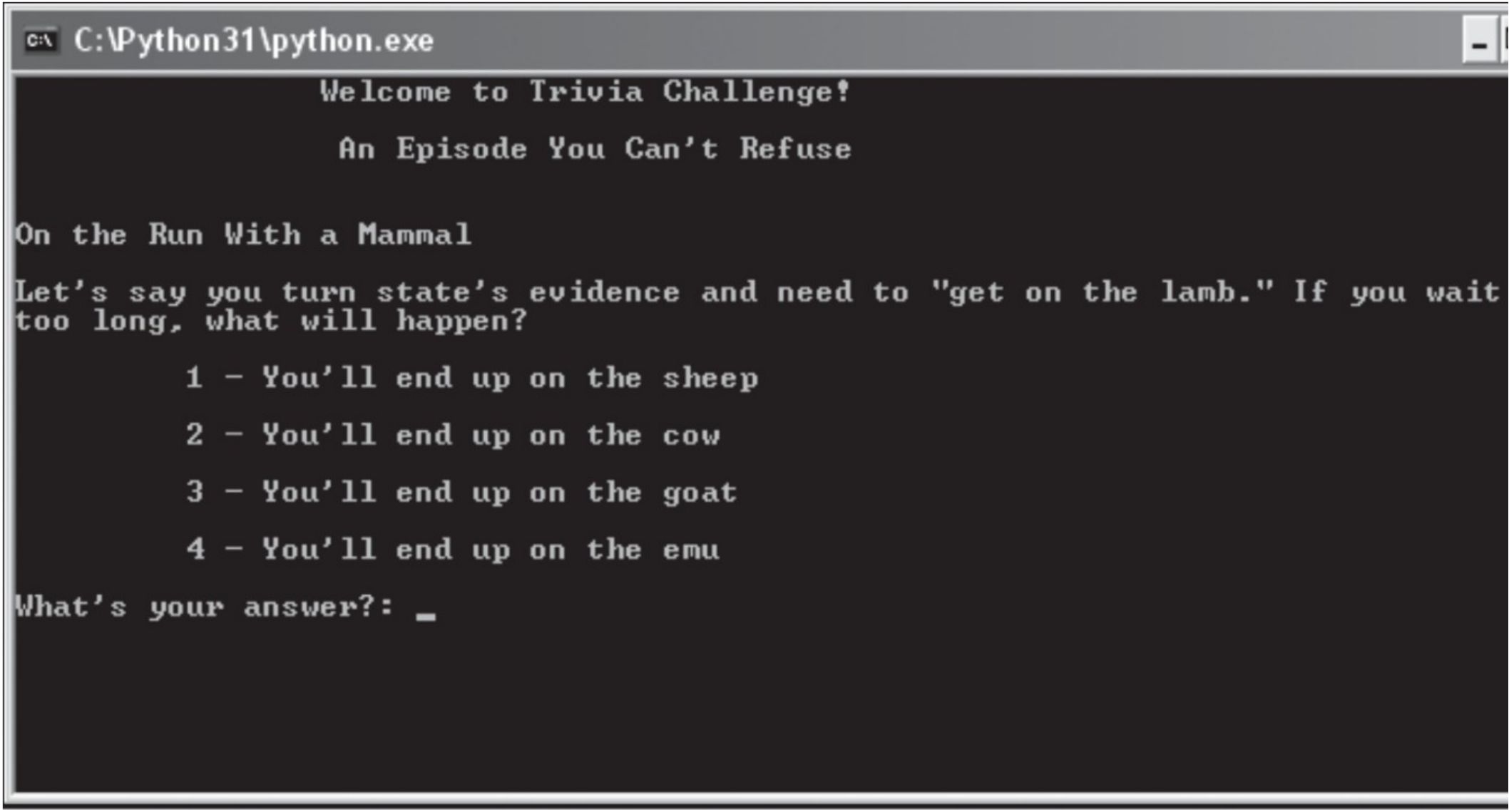
- You can add an **else** clause after all the except clauses in a try statement. The **else** block executes only if no exception is raised in the try block:

```
try:
    num = float(input("\nEnter a number: "))
except ValueError:
    print("That was not a number!")
else:
    print("You entered the number", num)
```

- num is printed in the **else** block only if the assignment statement in the try block doesn’t raise an exception.

- This is perfect because that means num will be printed only if the assignment statement was successful and the variable exists.

Introducing the Trivia Challenge Game



trivia_challenge.py

```
# Trivia Challenge
# Trivia game that reads a plain text file

import sys

def open_file(file_name, mode):
    """Open a file."""
    try:
        the_file = open(file_name, mode)
    except IOError as e:
        print("Unable to open the file", file_name,
              "Ending program.\n", e)
        input("\n\nPress the enter key to exit.")
        sys.exit()
    else:
        return the_file
```

```
def next_line(the_file):
    """Return next line from File trivia, formatted."""
    line = the_file.readline()
    line = line.replace("/", "\n")
    return line

def next_block(the_file):
    """Return the next block of data from File trivia."""
    category = next_line(the_file)

    question = next_line(the_file)

    answers = []
    for i in range(4):
        answers.append(next_line(the_file))

    correct = next_line(the_file)
    if correct:
        correct = correct[0]
```

```
    explanation = next_line(the_file)

    return category,question,answers,correct,explanation

def welcome(title):
    """Welcome the player and get his/her name."""
    print("\t\tWelcome to Trivia Challenge!\n")
    print("\t\t", title, "\n")

def main():
    trivia_file = open_file("trivia.txt", "r")
    title = next_line(trivia_file)
    welcome(title)
    score = 0

    # get first block
    category, question, answers, correct, explanation = \
    next_block(trivia_file)
```

```
while category:
    # ask a question
    print(category)
    print(question)
    for i in range(4):
        print("\t", i + 1, "-", answers[i])

    # get answer
    answer = input("What's your answer?: ")

    # check answer
    if answer == correct:
        print("\nRight!", end=" ")
        score += 1
    else:
        print("\nWrong.", end=" ")
    print(explanation)
    print("Score:", score, "\n\n")
```

```
        # get next block
        category,question,answers,correct,explanation=\
        next_block(trivia_file)

    trivia_file.close()

    print("That was the last question!")
    print("You're final score is", score)

main()
input("\n\nPress the enter key to exit.")
```

trivia.txt

An Episode You Can't Refuse
On the Run With a Mammal
Let's say you turn state's evidence and need to "get on the lamb." If you wait /too long, what will happen?
You'll end up on the sheep
You'll end up on the cow
You'll end up on the goat
You'll end up on the emu
1
A lamb is just a young sheep.
The Godfather Will Get Down With You Now
Let's say you have an audience with the Godfather of Soul. How would it be /smart to address him?
Mr. Richard
Mr. Domino
Mr. Brown
Mr. Checker
3
James Brown is the Godfather of Soul.

That's Gonna Cost Ya
If you paid the Mob protection money in rupees, what business would you most /likely be insuring?
Your tulip farm in Holland
Your curry powder factory in India
Your vodka distillery in Russian
Your army knife warehouse in Switzerland
2
The Rupee is the standard monetary unit of India.
Keeping It the Family
If your mother's father's sister's son was in "The Family," how are you /related to the mob?
By your first cousin once removed
By your first cousin twice removed
By your second cousin once removed
By your second cousin twice removed
1

Your mother's father's sister is her aunt -- and her son is your /mother's first cousin. Since you and your mother are exactly one generation /apart, her first cousin is your first cousin once removed.
A Maid Man
If you were to literally launder your money, but didn't want the green in your /bills to run, what temperature should you use?
Hot
Warm
Tepid
Cold
4
According to my detergent bottle, cold is best for colors that might run.

Understanding the Data File Layout

- The 1st line in trivia.txt is the title of the episode. The rest of the file consists of blocks of 7 lines for each question:

<category>
<question>
<answer 1>
<answer 2>
<answer 3>
<answer 4>
<correct answer>
<explanation>

- On the Run With a Mammal is the category of the 1st question. Let's say you turn state's evidence and need to "get on the lamb." If you wait /too long, what will happen?, is the 1st question in the game. The next 4 lines are the 4 possible answers from which the player will choose.

- The next line, 1, is the number of the correct answer. The next line, A lamb is just a young sheep., explains why the correct answer is correct.

- Include a forward slash (**/**) in 2 of the lines to represent a newline since Python does not automatically wrap text when it prints it.

The open_file() Function

- open_file() receives a file name and mode (both strings) and returns a corresponding file object.

- Use try & except to trap for an IOError exception for input-output errors, which would occur if the file doesn't exist.

- If there was a problem opening the trivia file, then there's no point in continuing the program, so we print a message and call **sys.exit()**.

- **sys.exit()** raises an exception resulting in the termination of the program. You should only use **sys.exit()** as a last resort, when you must end a program.

- We have to import the **sys** module to call **sys.exit()**:

```
import sys
```

```
def open_file(file_name, mode):
    """Open a file."""
    try:
        the_file = open(file_name, mode)
    except IOError as e:
        print("Unable to open the file", file_name, \
                "Ending program.\n", e)
        input("\n\nPress the enter key to exit.")
        sys.exit()
    else:
        return the_file
```

The next_line() Function

- next_line() receives a file object and returns the next line of text from it:

```
def next_line(the_file):
    line = the_file.readline()
    line = line.replace("/", "\n")
    return line
```

- Before its return, we replace all forward slashes with newline characters because Python does not automatically word wrap printed text.

The next_block() Function

- next_block() reads the next block of lines for one question.

It takes a file object and returns a string for the category, question, correct answer, and explanation as well as a list of 4 strings for the possible answers to the question:

```
def next_block(the_file):
    category = next_line(the_file)
    question = next_line(the_file)
    answers = []
    for i in range(4):
        answers.append(next_line(the_file))
    correct = next_line(the_file)
    if correct:
        correct = correct[0]
    explanation = next_line(the_file)
    return category, question, answers, correct, \
    explanation
```

The welcome() Function

- welcome() welcomes the player to the game and announces the episode's title:

```
def welcome(title):
    print("\t\tWelcome to Trivia Challenge!\n")
    print("\t\t", title, "\n")
```

Setting Up the Game

- main() houses the main game loop:

```
def main():
    trivia_file = open_file("trivia.txt", "r")
    title = next_line(trivia_file)
    welcome(title)
    score = 0
```

Asking a Question

- Next, we start the while loop, which will continue to ask questions as long as category is not the empty string.

- If category is the empty string, that means the end of the trivia file has been reached and the loop won't be entered.

```
# get first block
category, question, answers, correct, explanation = \
next_block(trivia_file)
while category:
    # ask a question
    print(category)
    print(question)
    for i in range(4):
        print("\t", i + 1, "-", answers[i])
```

Getting an Answer

```
# get answer
answer = input("What's your answer?: ")
```

Checking an Answer

- Compare the player's answer to the correct answer. If they match, the player is congratulated and his score is increased by 1. If they don't match, the player is told he is wrong. In either case, the explanation is displayed, so is the player's current score:

```
# check answer
if answer == correct:
    print("\nRight!", end=" ")
    score += 1
else:
    print("\nWrong.", end=" ")
print(explanation)
print("Score:", score, "\n\n")
```

Getting the Next Question

```
# get next block
category,question,answers,correct,explanation = \
next_block(trivia_file)
```