

# **Chapter XVII**

## **Sequential Text Files**

### **Chapter XVII Topics**

- 17.1 Introduction
- 17.2 Different Types of Files
- 17.3 Writing To and Reading From Text Files
- 17.4 Files of Numbers
- 17.5 Files of Multiple Data Types
- 17.6 Reading & Writing Simultaneously
- 17.7 Appending to an Existing File
- 17.8 Using Text Files with Graphics

## 17.1 Introduction

Random Access Memory (*RAM*) is great while it lasts, but it is only temporary. Rebooting the computer, flipping the power switch, tripping over the power cord and enjoying a friendly electrical storm can quickly delete or corrupt the information stored in your computer's dynamic memory.

Long ago, you became familiar with the solution to this problem. Use external storage to permanently store any important information. Most students learn several lessons about the virtue of saving their work on a regular basis. Common software programs like word processors, spreadsheets, and program languages, like Java, create user data files that are stored externally.

This means that the program you write is in fact a data file of another program. This may seem odd. You probably are comfortable thinking that a word processing document is stored as a data file. However, the programs you have written seem like programs, not data files. Both are true simultaneously. You have written fully functional Java programs. At the same time, the programs you have written were data files of some text editor program, like Notepad or a special Integrated Development Environment (*IDE*) program, like **jGRASP** or **JCreator**.

Program languages in general - and Java specifically - have the ability to create their own data files. This means that data entered by a program user can be stored for future processing. It also means that data only needs to be entered once. In some cases, data never needs to be entered. It is quite possible that your teacher, or somebody else, has created a data file of information. It becomes your job to retrieve that information in your program and process the data. This approach is very convenient and used quite extensively in college computer science classes.

### Programs and Data Files Notes

Data information processed by word processing, spreadsheet and similar application programs are stored as data files. Python programs, written with a text editor or some Integrated Development Environment (*IDE*), are also data files.

Programs have the capability to create and retrieve their own data files. This means that data entered in any one of your programs can be saved for later use.

The most fundamental justification for using a computer is to process information or data. Frequently, data need to be transferred from some external storage to internal memory or newly processed internal data need to be stored externally for future processing needs. The word *data* is tossed around in computer science in general and it certainly is mentioned every other sentence already in this chapter. What exactly is *data*, and how does it fit in with this file business? Computer data forms an organized hierarchy of information. Data starts at the very humble *bit* and works up the ranks to *byte*, *field*, *record* and *file*. Let's look at each stage.

Data Organization	
<b>bit</b>	This is the fundamental building block of all computer information – a <b>binary digit</b> – which stores only a <b>1</b> or <b>0</b> .
<b>byte</b>	One <i>byte</i> equals 8 <i>bits</i> . The <i>ASCII</i> format uses 1 byte to store a character. <i>UTF-8</i> uses between 1 and 4 bytes to store a character.
<b>field</b>	A <i>field</i> is one specific unit of information. Examples: <b>name</b> , <b>address</b> , <b>gpa</b> , <b>salary</b> , <b>diagnosis</b>
<b>record</b>	A <i>record</i> consists of a set of <i>fields</i> for a specific purpose. These <i>fields</i> can be of the same or different data types. Examples of the fields contained by different records: <b>student</b> record: <b>name</b> , <b>address</b> , <b>gpa</b> , <b>classRank</b> <b>employee</b> record: <b>name</b> , <b>address</b> , <b>salary</b> , <b>position</b> <b>patient</b> record: <b>name</b> , <b>address</b> , <b>diagnosis</b> , <b>allergies</b>
<b>file</b>	A data base <i>file</i> consists of a sequence of <i>records</i> . Examples: A school can have a <i>file</i> of <b>student</b> records. A business can have a <i>file</i> of <b>employee</b> records. A hospital can have a <i>file</i> of <b>patient</b> records.

The transfer of data between internal memory and external storage is quite a complicated process. Modern languages like C++, Java and Python use *streams* to handle data transfer. A *stream* is a set of bytes or characters that *flow* much like water flows in a river. (In the old days there was no distinction between bytes and characters. Now that *Unicode*, with its *two-byte* characters is in use, there is a distinction.) Think of a stream as a pipeline that contains a neatly ordered set of data. Streams can go in two directions; for input and for output. Input streams can come from the keyboard, from an external storage source, or mouse-click. Output streams can go to the monitor, an external storage location or the printer. Streams allow consistent input/output processing for many different types of sources and destinations.

## 17.2 Different Types of Files

It is very important to understand the difference between a *file* and a *file data structure*. This difference was actually mentioned back when *Data Structures* were first introduced in chapter XIII:

### File Data Structure Definition Review

A *file data structure* is an *internal* data structure - with an unspecified number of elements of the same type - assigned to an *external* file name. The file data structure allows transfer of data between internal and external storage.

You will see examples of “internal file data structures” once we get to the program examples; but first, we need to understand what these internal file data structures are assigned to – specifically, the “external files”.

### External File Definition

An *external file* is a sequence of information stored as bytes on a hard drive, disk, tape, CD, DVD, jump drive, or some other external storage device.

There can be some confusion because the term *file* is used to describe several different things. One might even say that it is *overloaded*. As with overloading, you need to look at the context to understand what type of file is being discussed. If this is a *file variable* that is inside a program, then it is an *internal file data structure*. If there are documents in a folder on your hard drive or jump drive, then these are *external files*. They are “external” because they exist outside of your program. If this external file is organized in such a way that it contains a sequence of records and the user can make search queries like **gpa** > **3.5** (which would display all records where the **gpa** field was **3.5** or higher) then you have a *data base* file.

## Text Files vs. Binary Files

If a file contains only a sequence of characters, it is a *text file*. Text files are nice and simple and can be edited by any text editor like Notepad or jGRASP. A text file is not the same thing as a *word processing document* which is far more complicated. Word processing documents allow different styles like **bold**, underline, and *italics* or different fonts like Arial, Courier New, and Times New Roman. They also allow tables, margins, images and many other features. This is why we use a simple text editor like jGRASP to edit our Java program.

Word processing documents, presentation files, electronic spreadsheets, data base file and many others are all *binary files*. Unlike text files, binary files do not simply store text. Because of this, they cannot be edited with a text editor. Binary files have a specific format which requires a specific application for editing. **.docx** files can be edited with *Microsoft Word*; **.pptx** files can be edited with *Microsoft PowerPoint*; **.xlsx** files can be edited with *Microsoft Excel*; and **.accdb** files be edited with *Microsoft Access* (a data base manager). Keep in mind that binary files are VERY specific. Just because you have a spread sheet document, that does not mean you can load it with just any Electronic Spreadsheet software. A file created with *Google Sheets* might not be completely compatible with *Microsoft Excel*. In the same way, a file created with *Microsoft PowerPoint* might not be completely compatible with *Google Slides*.

### **Text Files and Binary Files**

*Text files* contain only characters and can be edited by any text editor like Notepad or jGRASP. All Java programs are text files.

*Binary files* have a specific format which requires a specific application for editing.

## Sequential Files vs. Random Access Files

One way to classify file is by what they store. That what we looked at when we compared text files and binary files. Another way to classify files is by how the information is accessed. To properly understand this, we need a brief history of external storage.

Early storage devices used long tapes on large reels. Accessing information on such tapes was only possible in sequence. Reading or writing information with tape storage required *sequential access*. Most students are familiar with this process. Pop an audiotape in a player and try to play the third song. It is necessary to listen to the first two songs first or fast forward to your desired selection. Either way, it is not possible to access the requested song instantly. Access on a tape is sequential. The same is true for VCRs (yes I know they are old fashion, but you still remember them). There is no instant way to instantly jump to your favorite scene in a movie stored on video tape. Because of this, early computers, with tape storage, developed programs that could only use sequential file data access.

With time, external storage switched to disks, CDs, DVDs, jump drives and hard drives. Such devices allow direct access of certain files. This type of access is called *random access*. You can randomly access any file information. This type of access is similar to an old record album player or CD player or DVD player. It is possible to jump directly to the third song or any other desired song on the album or CD or to jump to any scene in the movie on a DVD.

Random access of files may be handled in a future computer science course. In this first file introduction we will only concentrate on sequential files. After you have worked with sequential files, it will be a comfortable transition to add the additional processing required to manage random access files.

## Sequential Access and Random Access

Files can have sequential access or random access.

*Sequential access* files allow data access only in the sequence that the data is stored in the file.

*Random access* files allow data access in any random pattern, regardless of how the data is stored.

**NOTE:** In this chapter, we will strictly be working with *sequential text files*.

## 17.3 Writing To and Reading From Text Files

This section is called *Writing To and Reading From Text Files*. Program **TextFiles01.py**, in Figure 17.1, accomplishes the first part of this. The program sends one line of text to the file "TextFiles01.txt".

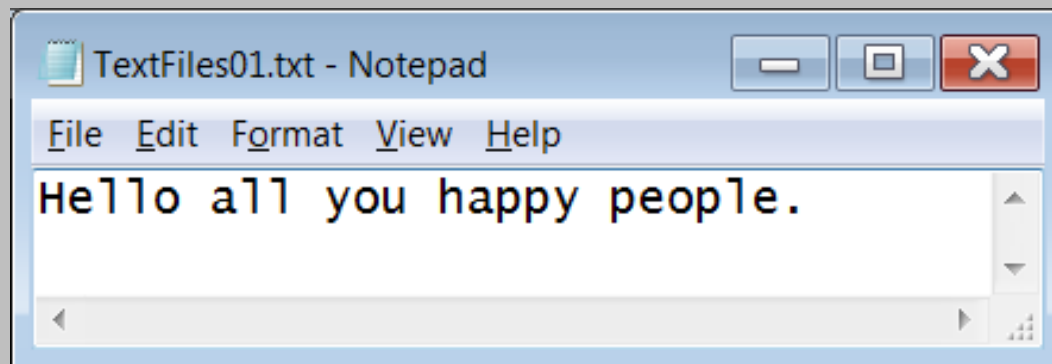
Figure 17.1

```
1 # TextFiles01.py
2 # This program demonstrates how to create a new
3 # text file and <write> some text to it.
4
5
6 file = open("TextFiles01.txt",'w')
7
8 file.write("Hello all you happy people.")
9
10 file.close()
11
```

Monitor Output: *None*

```
----jGRASP exec: python TextFiles01.py
----jGRASP: operation complete.
```

File Output:



While this is a very short, 3 lines program, we need to understand what each of the 3 lines does. Let us take a closer look.

On line 6 we are creating our *file data structure*, with a not-so-imaginative name, **file**. Note the 2 parameters of the **open** function. The first is **"TextFiles01.txt"**. This is the *external file* that our *file data structure* will work with. The second parameter is **'w'**. This indicates that we are *writing* information to the file. If the file **"TextFiles01.txt"** does not exist, this program statement will create it. If the file already exists, this program statement will erase it and replace it with this new file.

On line 8 we are sending one line of text to the file. This may look somewhat like a **print** command. Truth be told, the process is very similar. Both **print** and **write** generate *output*. The difference is **print** generates *monitor output* and **write** generates *file output*. This means that on quizzes and test, I cannot simply say “What is the output of this program?” I will need to say, “What is the monitor output of this program?” or “What is the file output of this program?”

On line 10 we are done with the file, so we close it. One very important rule in computer science – one that I even called a “commandment” way back in Chapter III – is that anything you open, you must also close. Remember this:





Program **TextFiles02.py**, in Figure 17.2, accomplishes the second part of the title of this section, specifically “Reading From a Text File”. The program accesses the file **"TextFiles01.txt"** – the same file created by the previous program. The difference is that now it reads the line of text from the file and displays it on the monitor.

**Figure 17.2**

```
1 # TextFiles02.py
2 # This program demonstrates how to <read> the text
3 # from the file created by the previous program.
4 # NOTE: If the previous program, TextFiles01.py,
5 # has not been executed, this program will crash.
6
7
8 file = open("TextFiles01.txt",'r')
9
10 text = file.readline()
11
12 print()
13 print(text)
14
15 file.close()
16
```

```
----jGRASP exec: python TextFiles02.py
Hello all you happy people.
----jGRASP: operation complete.
```

This program also is very short, but like the previous one it bear further scrutiny.

On line 8 we are creating our *file data structure*; this is almost identical to the previous program except the second parameter in the **open** function is now an **'r'**. This indicates that we are *reading* information from the file. If the file **"TextFiles01.txt"** does not exist, the program will actually crash with a **FileNotFoundException**.

On line 10 we are reading one line of text from the file and storing it in the variable **text**. This may look somewhat like an **input** command. As before, the process is very similar. Both **input** and **readline** receive *input*. The difference is **input** receives *keyboard input* and **readline** receives *file input*.

On line 13 a **print** command is used to display the value of the **text** variable. This means that this program has *file input* and *monitor output*.

On line 15 we **close** the **file**. Whether we are writing to or reading from a text file, when we are done, the file needs to be closed.

Program **TextFiles03.py**, in Figure 17.3, demonstrates a couple useful functions from the **os** (*Operating System*) library. The first is **os.path.exists**. This returns **True** or **False** and lets you know if a particular file “exists”. This is very useful. Before you start to read information from a file, you can first confirm its existence rather than risk crashing the program by trying to read from a non-existent file. The second function, **os.path.getsize**, will return the “size” of the file in terms of how many bytes it contains.

**Figure 17.3**

```
1 # TextFiles03.py
2 # This program demonstrates a way to check
3 # if a file <exists> and also a way to see
4 # how many bytes a file contains.
5
6
7 import os
8
9
10 print()
11 if os.path.exists("TextFiles01.txt"):
12     print("TextFiles01.txt stores",
13           os.path.getsize("TextFiles01.txt"),
14           "bytes of data.")
15 else:
16     print("TextFiles01.txt does not exist.")
17
18 print()
```

```

19 if os.path.exists("qwerty.txt"):
20     print("qwerty.txt stores",
21           os.path.getsize("qwerty.txt"),
22           "bytes of data.")
23 else:
24     print("qwerty.txt does not exist.")
25

```

```

----jGRASP exec: python TextFiles03.py

TextFiles01.txt stores 27 bytes of data.

qwerty.txt does not exist.

----jGRASP: operation complete.

```

## Multi-Line Files

Files rarely contain just a single line of text. Multi-line files are much more common. Program **TextFiles04.py**, in Figure 17.4, attempts to create a file with 5 lines of text. Note that 5 separate **write** commands are used.

**Figure 17.4**

```

1 # TextFiles04.py
2 # This program attempts to send multiple lines
3 # of output to a text file. After you run this
4 # program, load the file TextFiles04.txt and
5 # you will see this did not work properly.
6
7
8 file = open("TextFiles04.txt", 'w')
9
10 file.write("Hello")
11 file.write("all")

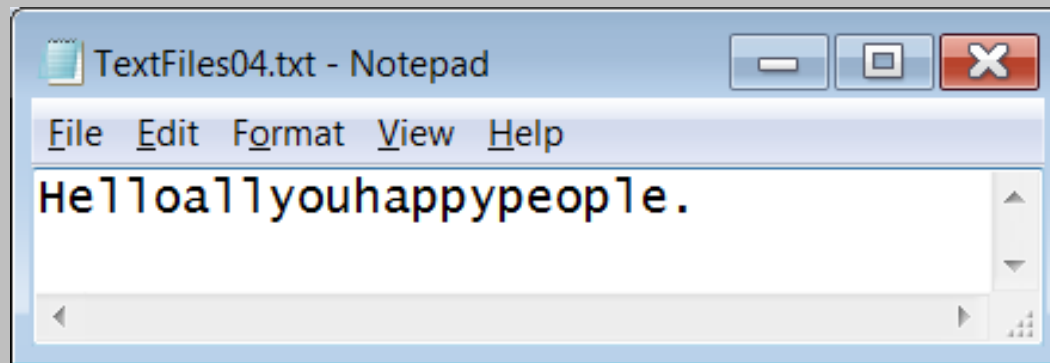
```

```
12 file.write("you")
13 file.write("happy")
14 file.write("people.")
15
16 file.close()
```

Monitor Output: *None*

```
----jGRASP exec: python TextFiles01.py
----jGRASP: operation complete.
```

File Output:



When you look at the “File Output: of this program, you can see that this did not work. One big difference between **print** and **write**, other than the monitor/file output destination, is the fact that **print** generates a `\n` (carriage-return/line-feed) and **write** does not. So, what is supposed to be five separate lines of text all got concatenated together into one.

Program **TextFiles05.py**, in Figure 17.5, fixed the issue of the previous program by explicitly generating the `\n` (new line) escape sequence at the end of every **write** command.

**Figure 17.5**

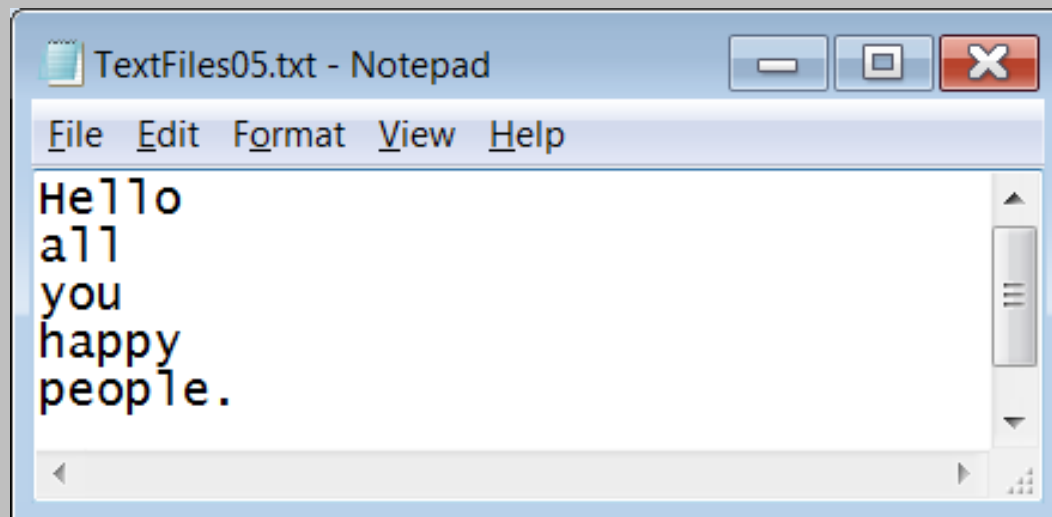
```
1 # TextFiles05.py
2 # This program fixes the issue of the
3 # previous program by adding the
4 # "new line escape sequence" <\n>.
```

```
5
6
7 file = open("TextFiles05.txt", 'w')
8
9 file.write("Hello\n")
10 file.write("all\n")
11 file.write("you\n")
12 file.write("happy\n")
13 file.write("people.\n")
14
15 file.close()
16
```

Monitor Output: *None*

```
----jGRASP exec: python TextFiles05.py
----jGRASP: operation complete.
```

File Output:



Program **TextFiles06.py**, in Figure 17.6, attempts to read the file created by the previous program. Please note that if you have not executed the previous

program, this program, and the next several programs, will crash. When you look at the output, it will seem like the execution is incomplete as only the first line of text from the file is displayed.

**Figure 17.6**

```
1 # TextFiles06.py
2 # This program tries to read the file created
3 # by the previous program. The problem is only
4 # one <readline> command is executed, so only
5 # one line of text is read and displayed.
6 # NOTE: If the previous program, TextFiles05.py,
7 #       has not been executed, this program, and
8 #       the next several programs, will crash.
9
10
11 file = open("TextFiles05.txt", 'r')
12
13 lineOfText = file.readline()
14
15 print()
16 print(lineOfText)
17
18 file.close()
19
```

```
----jGRASP exec: python TextFiles06.py
Hello
----jGRASP: operation complete.
```

The problem with this program is the **readline** command, which reads one line of text, is only used once. This is why only one line of text is read and displayed.

Program **TextFiles07.py**, in Figure 17.7, fixes the issue of the previous program by using a **for** loop to make the computer read in and display all 5 lines of text.

**Figure 17.7**

```
1 # TextFiles07.py
2 # This program uses a <for> loop to read
3 # all 5 lines of text from the file.
4 # There are a couple problems. First, there
5 # are extra lines skipped in the output, caused
6 # by the <\n> characters used to write the
7 # data when the file was created.
8 # Second, this program only works because we
9 # know, ahead of time, how many lines of text
10 # are in the file, which is not realistic.
11
12
13 file = open("TextFiles05.txt",'r')
14
15 print()
16
17 for k in range(5):
18     lineOfText = file.readline()
19     print(lineOfText)
20
21 file.close()
```

```
----jGRASP exec: python TextFiles07.py
```

```
Hello
```

```
all
```

```
you
```

```
happy
```

```
people.
```

```
----jGRASP: operation complete.
```

There are 2 new issues with this program. First, there are extra lines skipped in the output. These are caused by the `\n` characters used to write the data to the file a couple programs ago. Second, the only reason this program works at all is that we know, ahead of time, that there are exactly 5 lines of text in the file. That is not realistic.

Program **TextFiles08.py**, in Figure 17.8, fixes the first issue of the previous program. It does this by using the **strip** function which “strips” away any *invisible characters* at the beginning or ending of each line of text. “Invisible characters” can include things like spaces, tabs or `\n` escape sequences.

**Figure 17.8**

```
1 # TextFiles08.py
2 # This program fixes the first issue of the
3 # previous program by using the <strip> command
4 # to "strip" away any invisible characters
5 # before and after each line of text.
6 # This includes </n> characters.
7
8
9 file = open("TextFiles05.txt",'r')
10
11 print()
12
13 for k in range(5):
14     lineOfText = file.readline().strip()
15     print(lineOfText)
16
17 file.close()
```

```
----jGRASP exec: python TextFiles08.py

Hello
all
you
happy
people.

----jGRASP: operation complete.
```



We still have the second issue from program **TextFile07.py** to fix. Remember, the only reason the past couple programs work is because we know the exact number of items in the file ahead of time, which is not realistic. You may wonder, “Why is this a big deal?” Well, if we simply guess the size of the file, it is almost certain that our guess will be wrong. It will either be too small or too large. The problem with guessing too small is that we will not retrieve the entire file. What if we guess too large? Program **TextFiles09.py**, in Figure 17.9, explores this question by attempting to read 8 lines of text from a file that only has 5.

**Figure 17.9**

```
1 # TextFiles09.py
2 # This program demonstrates what happens when you
3 # attempt to read past the end of a text file.
4 # In most languages, this would crash the program.
5 # In Python, you just get extra blank lines.
6
7
8 file = open("TextFiles05.txt", 'r')
9
10 print()
11
12 for k in range(8):
13     lineOfText = file.readline().strip()
14     print(lineOfText)
15
16 file.close()
17
```

```
----jGRASP exec: python TextFiles09.py

Hello
all
you
happy
people.

----jGRASP: operation complete.
```

In some program languages, this will crash the program. Reading past the end of the file is like using an index that is too large in an array. Python actually does not crash or display any error message. Anytime you read past the end of the file, you just get a blank or empty line of text. If you look closely at the output, you should see extra blank lines after the word **people**.

Program **TextFiles10.py**, in Figure 17.10, finally fixes the second issues of program **TextFile07.py**. On line 11 you see we are using the same **for..each** loop that we used for arrays. This is also the preferred way to read all of the information from a file.

**Figure 17.10**

```
1 # TextFiles10.py
2 # This program fixes the second issue from
3 # program TextFiles07.py by using the same
4 # <for..each> loop that works with arrays.
5
6
7 file = open("TextFiles05.txt",'r')
8
9 print()
10
11 for lineOfText in file:
12     lineOfText = lineOfText.strip()
13     print(lineOfText)
14
15 file.close()
```

```
----jGRASP exec: python TextFiles10.py

Hello
all
you
happy
people.

----jGRASP: operation complete.
```

We have seen that Python is language that is full of shortcuts, some of which are very powerful. Program **TextFiles11.py**, in Figure 17.11, demonstrates the **readlines** command. Note the plural **readlines**. You have already seen the **readline** (singular) command and how it is frequently placed in some kind of loop to read all of the information from a file. The **readlines** (plural) command will read the entire file and store everything in an array of strings.

When you look at the program, you see that all of the file processing is done in 3 lines of code. Line 6 open the file. Line 7 reads in all of the lines of text. Line 8 closes the file. The **for..each** loop on lines 11-13 is used to traverse and display the array of strings. Note that we still need to use the **strip** function to get rid of the extra blank lines.

**Figure 17.11**

```
1 # TextFiles11.py
2 # This program demonstrates the <readlines> command which
3 # reads in the ENTIRE file and creates an array of strings.
4
5
6 file = open("TextFiles05.txt",'r')
7 allLinesOfText = file.readlines()
8 file.close()
9
10 print()
11 for lineOfText in allLinesOfText:
12     lineOfText = lineOfText.strip()
13     print(lineOfText)
14
```

```
----jGRASP exec: python TextFiles12.py

Hello
all
you
happy
people.

----jGRASP: operation complete.
```

Suppose we want to make this program more flexible. Instead of simply displaying the contents of one specific file with a fixed name, what if the program could display the contents of any file? This would require some *keyboard input* so the user can enter the name of the file he/she wishes to display. This is precisely what is done in program **TextFiles12.py**, shown in Figure 17.12. You will see 4 different outputs. The first is for the file **"TextFiles01.txt"**. The second is for the file **"TextFiles05.txt"**, which is the same file displayed by the last couple programs. The third is for the file **"qwerty.txt"**, which does not exist. Note how the program can deal with this situation without crashing because the **os.path.exists** function is used. The fourth and final output may surprise you as it first is for the file **"TextFiles12.py"**.

**Figure 17.12**

```
1 # TextFiles12.py
2 # This program allows the user to enter a file name.
3 # Then it displays the contents of that file,
4 # provided it exists.
5
6
7 import os
8
9
10 print()
11 fileName = input("Enter the name of a file. --> ")
12 print()
13
14 if os.path.exists(fileName):
15     file = open(fileName, 'r')
16     for text in file:
17         text = text.strip()
18         print(text)
19     file.close()
20 else:
21     print(fileName, "does not exist.")
```

```
----jGRASP exec: python TextFiles12.py
▶ Enter the name of a file. --> TextFiles01.txt
Hello all you happy people.
----jGRASP: operation complete.
```

```
----jGRASP exec: python TextFiles12.py
▶ Enter the name of a file. --> TextFiles05.txt

Hello
all
you
happy
people.

----jGRASP: operation complete.
```

```
----jGRASP exec: python TextFiles12.py
▶ Enter the name of a file. --> qwerty.txt

qwerty.txt does not exist.

----jGRASP: operation complete.
```

```
----jGRASP exec: python TextFiles12.py
▶ Enter the name of a file. --> TextFiles12.py

# TextFiles12.py
# This program allows the user to enter a file name.
# Then it displays the contents of that file,
# provided it exists.

import os

print()
fileName = input("Enter the name of a file. --> ")
print()

if os.path.exists(fileName):
    file = open(fileName, 'r')
    for text in file:
        text = text.strip()
        print(text)
    file.close()
else:
    print(fileName, "does not exist.")

----jGRASP: operation complete.
```

So if we look at the 4 outputs, we see that in the first 2, we enter the name of an existing text file, and its contents are displayed. In the third output, we enter the name of a non-existent file and the program tells us the file does not exist. In the fourth output we enter the name of the program itself. How does this work? Keep in mind that this is a program that can display the contents of any text file and Python programs are text files. So yes, the program basically displays itself. You may be wondering what happened to the indenting in the output. Remember, the **strip** command removes all invisible characters from both ends of the string. While this includes the `\n` character at the end, it also includes the spaces at the beginning.

## 17.4 Files of Numbers

We have spent the first half of the chapter becoming really good at storing strings and retrieving strings from text files. What if I want to store something that is not a string? What if I want to store integers, real numbers or Boolean values in a text file? Program **TextFiles13.py**, in Figure 17.13, will experiment to see if this is possible. The program contains a loop that will repeat 10 times. Each time through the loop the program will generate a random 4-digit integer and then write it to the "**TextFiles13.txt**" text file... at least, it is going to try.

**Figure 17.13**

```
1 # TextFiles13.py
2 # This program attempts to create a file of 10 random
3 # integers. The program has a syntax error because
4 # only string values can be written to a text file.
5
6
7 from random import *
8
9 seed(1234)
10
11 file = open("TextFiles13.txt", 'w')
12
```

```

13 for k in range(10):
14     number = randint(1000,9999)
15     file.write(number)
16
17 file.close()
18

```

```

----jGRASP exec: python TextFiles13.py
Traceback (most recent call last):
  File "TextFiles13.py", line 15, in <module>
    file.write(number)
TypeError: write() argument must be str, not int

----jGRASP wedge2: exit code for process is 1.
----jGRASP: operation complete.

```

So this did not work for one simple reason. Text files can only store “text” (hence the name). You will note the error message says, **TypeError: write() argument must be str, not int**.

Now we could create a special *binary file* to store this information, but that requires the sort of complexity that we want to avoid in this first year class. Text files are nice and simple. We like text files. Is there any way we can store things like numbers in text files? Well yes we can... indirectly. Remember, back in chapter XIV, you learned commands that can convert other data types to strings and vice versa. So if we want to store numbers in a text file, we can do it, provided we first convert the numbers to strings.

Program **TextFiles14.py**, in Figure 17.14, fixes the problem of the previous program by using the **str** command to convert each number to a string before it is written to the text file. While this program works, it does have another issue.

**Figure 17.14**

```

1 # TextFiles14.py
2 # This program fixes the issue of the previous program
3 # by using the <str> command to convert the numbers to
4 # strings. However, a new issue is created; which can
5 # be seen by loading the file TextFiles15.txt
6

```

```

7
8 from random import *
9
10 seed(1234)
11
12 file = open("TextFiles14.txt", 'w')
13
14 for k in range(10):
15     number = randint(1000, 9999)
16     file.write(str(number))
17
18 file.close()
19

```

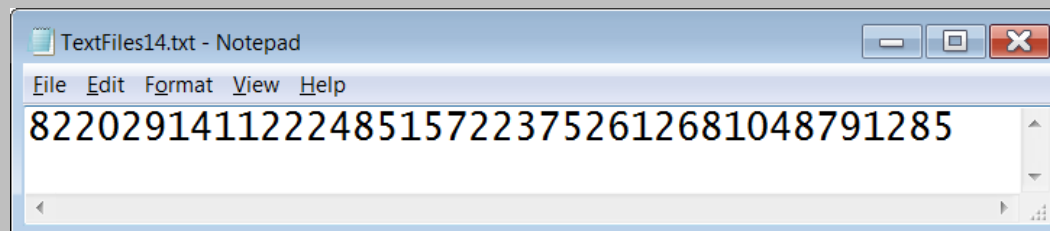
Monitor Output: *None*

```

----jGRASP exec: python TextFiles14.py
----jGRASP: operation complete.

```

File Output:



When we look at the “file output” of this program, we see that all of the numbers were sent to the file; however, instead of storing ten 4-digit numbers, the file actually is storing one 40-digit number. While it may be true that Python is one of the few languages that can handle numbers this big, we never wanted a number this big. When sending multiple strings to a file, you have the option to put them on separate lines using the “\n” *escape sequence*. When working with numbers, this is not really an option. It is required to prevent all of your numbers from concatenating together.

Program **TextFiles15.py**, in Figure 17.15, fixes the problem of the previous program by writing a “\n” character after each converted number. The “File Output” (and later, program **TextFiles17.py**) shows that this works properly now.



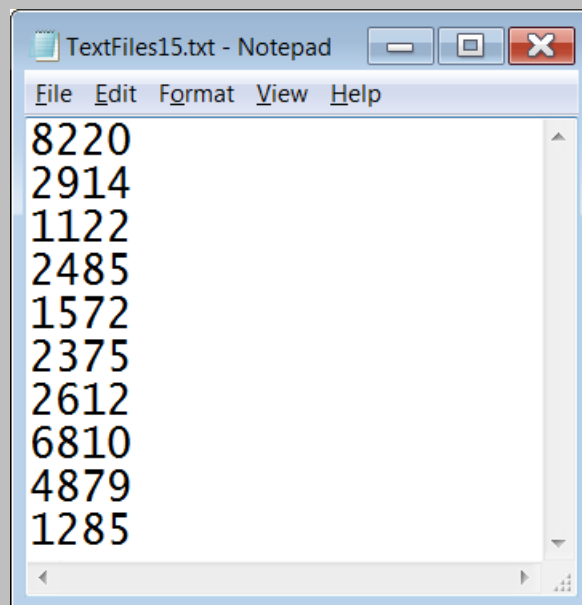
Figure 17.15

```
1 # TextFiles15.py
2 # This program fixes the issue of the previous
3 # program by adding a "new line escape sequence" <\n>
4 # as was done in earlier program examples.
5
6
7 from random import *
8
9 seed(1234)
10
11 file = open("TextFiles15.txt", 'w')
12
13 for k in range(10):
14     number = randint(1000,9999)
15     file.write(str(number)+"\n")
16
17 file.close()
```

Monitor Output: *None*

```
----jGRASP exec: python TextFiles15.py
----jGRASP: operation complete.
```

File Output:



OK, we converted 10 random 4-digit numbers to text and sent them to a text file. Now we want to retrieve them. Program **TextFiles16.py**, in Figure 17.16, is supposed to retrieve and display the numbers. Question: How do I know they are in fact numbers? Remember, the numbers were converted to strings when they were sent to the text file in the previous program. To find out – one way or the other – if these are in fact numbers, the program is going to do something mathematical with them; specifically, it will compute their average.

**Figure 17.16**

```
1 # TextFiles16.py
2 # This program attempts to read in the numbers from the
3 # file created by the previous program and average them.
4 # The problem is the numbers are stored as text and need to
5 # be converted back to numbers before they can be averaged.
6
7
8 file = open("TextFiles15.txt",'r')
9
10 print()
11
12 total = 0
13 count = 0
14
15 for number in file:
16     print(number)
17     total += number
18     count += 1
19
20 file.close()
21
22 average = total / count
23
24 print()
25 print("Total  =",total)
26 print()
27 print("Average =",average)
28
```

```

----jGRASP exec: python TextFiles16.py

8220

Traceback (most recent call last):
  File "TextFiles16.py", line 17, in <module>
    total += number
TypeError: unsupported operand type(s) for +=: 'int' and 'str'

----jGRASP wedge2: exit code for process is 1.
----jGRASP: operation complete.

```

OK, when we look at the output we see that only the first number is displayed. Then the program crashes when we try to add the “**number**” to the **total**. The problem is **number** is still a string. Before we can do anything mathematical with any of the numbers from the text file, we need to convert them back to integers.

Program **TextFiles17.py**, in Figure 17.17, fixes the issue of the previous program by using the **int** command to convert each **lineOfText** that is read from the file back into an integer. Now, these numbers can be properly totaled and averaged as the output demonstrates.

**Figure 17.17**

```

1 # TextFiles17.py
2 # This program fixes the issue of the previous
3 # program by using the <int> command to convert
4 # the text numbers back into integer values.
5 # NOTE: This process also eliminates the <\n>
6 #       escape sequence character.
7
8
9 file = open("TextFiles15.txt", 'r')
10 print()
11 total = 0
12 count = 0
13

```

```

14 for lineOfText in file:
15     number = int(lineOfText)
16     print(number)
17     total += number
18     count += 1
19
20 file.close()
21
22 average = total / count
23
24 print()
25 print("Total    =",total)
26 print()
27 print("Average =",average)
28

```

```

----jGRASP exec: python TextFiles17.py

```

```

8220
2914
1122
2485
1572
2375
2612
6810
4879
1285

```

```

Total    = 34274

```

```

Average = 3427.4

```

```

----jGRASP: operation complete.

```

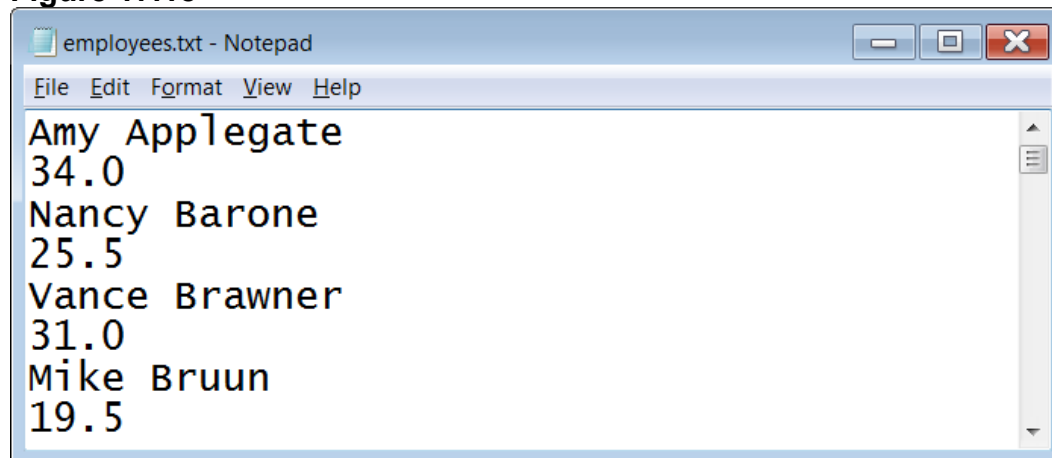
## 17.5 Files of Multiple Data Types

In real life, when a business stores a file of information it is not just a file of names or a file of numbers. Many times the file contains *records* of information. When I say “record” I am referring to the term mentioned back in chapter XIII:

**A record consists of a set of *fields* for a specific purpose. These *fields* can be of the same or different data types.**

So this means the files do not just contain strings or numbers. They can contain any combination of both. Look at the first 4 records from the **employees.txt** file in Figure 17.18 for an example.

**Figure 17.18**



The **employees.txt** file contains 2 pieces of information for each employee: their name (a string) and the number of hours they worked (a real number). Program **TextFiles18.py**, in Figure 17.19, shows one way to read such a file. We start with 2 empty parallel arrays, **names** and **hourlyRates**, and the same **for..each** loop that we have been using. When we look at the text file (see figure 17.18) we see the file alternates between names and hourly rates. I need to make the program alternate with it. That is the purpose of the **count** variable. The **count** starts at **0**. Every time through the loop **1** is added to count. If the **count** is even we **append** the string to the **names** array. If the **count** is odd we **append** the number to the **hourlyRates** array. This does require the use of the **float** command to convert the string into a real number (a.k.a. a *floating point number*). As the names and hourly rates are displayed for each employee, the hourly rates are accumulated and counted so that the average hourly rate can be displayed at the end of the output.

Figure 17.19

```
1 # TextFiles18.py
2 # This program demonstrates how to read information
3 # from a file containing different data types.
4 # In this case, the "employees.txt" file contains the
5 # names (string values) and hourly rates (real# values)
6 # for several employees.
7 # The file information is read into 2 "parallel arrays".
8 # After that, the information in both arrays is displayed,
9 # and the average hourly rate is computed and displayed.
10
11
12 file = open("employees.txt",'r')
13 names = []
14 hourlyRates = []
15 count = 0
16 for lineOfText in file:
17     if (count % 2 == 0):
18         names.append(lineOfText.strip())
19     else:
20         hourlyRates.append(float(lineOfText))
21     count += 1
22 file.close()
23
24 count = 0
25 total = 0
26 for k in range(len(names)):
27     print()
28     print("Employee Name: ",names[k])
29     print("Hourly Rate:  ", "${:.2f}".format(hourlyRates[k]))
30     total += hourlyRates[k]
31     count += 1
32
33 average = total / count
34
35 print("\n")
36 print("Average Hourly Rate:", "${:.2f}".format(average))
37
```

NOTE: There are 36 names and GPAs in this file which makes the entire output very long. For this reason, only the first 5 and the last 5 records will be shown.

```
----jGRASP exec: python TextFiles18.py
```

```
Employee Name:  Amy Applegate  
Hourly Rate:    $34.00
```

```
Employee Name:  Nancy Barone  
Hourly Rate:    $25.50
```

```
Employee Name:  Vance Brawner  
Hourly Rate:    $31.00
```

```
Employee Name:  Mike Bruun  
Hourly Rate:    $19.50
```

```
Employee Name:  Gordon Collins  
Hourly Rate:    $25.00
```

```
:      :      :      :
```

```
Employee Name:  Chris Stark  
Hourly Rate:    $80.00
```

```
Employee Name:  Tom Tooch  
Hourly Rate:    $26.50
```

```
Employee Name:  Michael Ward  
Hourly Rate:    $19.00
```

```
Employee Name:  Cheryl Willis  
Hourly Rate:    $37.00
```

```
Employee Name:  Ziggy Zighlander  
Hourly Rate:    $77.50
```

```
Average Hourly Rate: $34.43
```

```
----jGRASP: operation complete.
```

## Parallel Array Example:

names	
0	Amy Applegate
1	Nancy Barone
2	Vance Brawner
3	Mike Bruun
4	Gordon Collions
:	:
35	Ziggy Zighlander

hourlyRates	
0	34.0
1	25.5
2	31.0
3	19.5
4	25.0
:	:
35	77.5

### Parallel Arrays Disclaimer

When working with multiple records of information, the use of *Parallel Arrays* is not the only option, nor is it the best option.

It is much more organized, efficient, and even intuitive to use an *array of records* to store all of the information.

However, this requires a complexity in coding that we wish to avoid in a first year computer science class.

*Arrays of Records* will be discussed along with *Object Oriented Programming* in AP<sup>®</sup> Computer Science-A.



## 17.6 Reading and Writing Simultaneously

Suppose you want the ability to create a *backup copy* of a file. This is actually fairly simple. A program can have 2 file data structures. One is open for input and is used to read the information one line at a time from the original file. The other is open for output and is used to write each line of information to the backup copy file. This is exactly what is done in program **TextFiles19.py**, shown in Figure 17.21. You will see that we have 2 file data structure variable. **inFile** is used to read the information from the "**original.txt**" file, shown in Figure 17.20. **outFile** is used to write the information to the "**backup.txt**" file. When you look at the file output for this program, you will see the newly created "**backup.txt**" file is identical to the "**original.txt**" file.

Figure 17.20

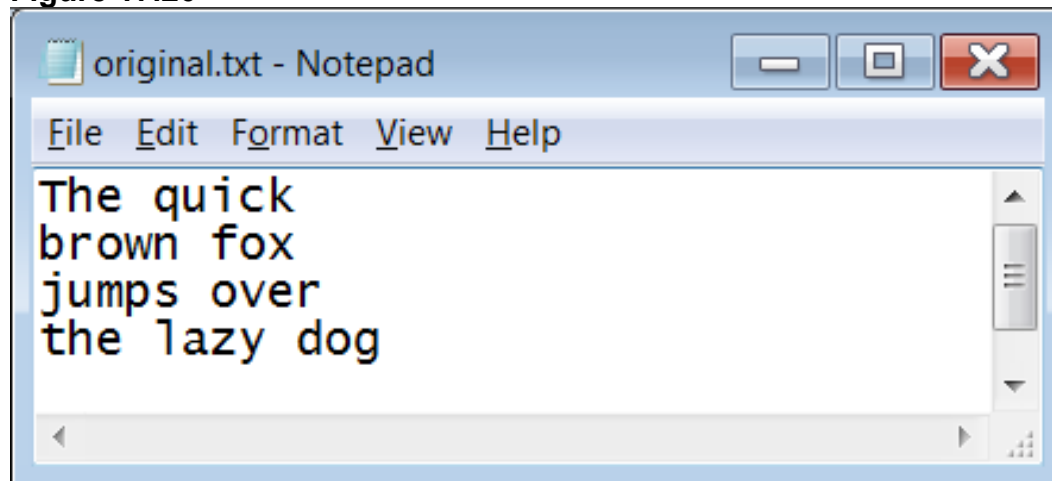


Figure 17.21

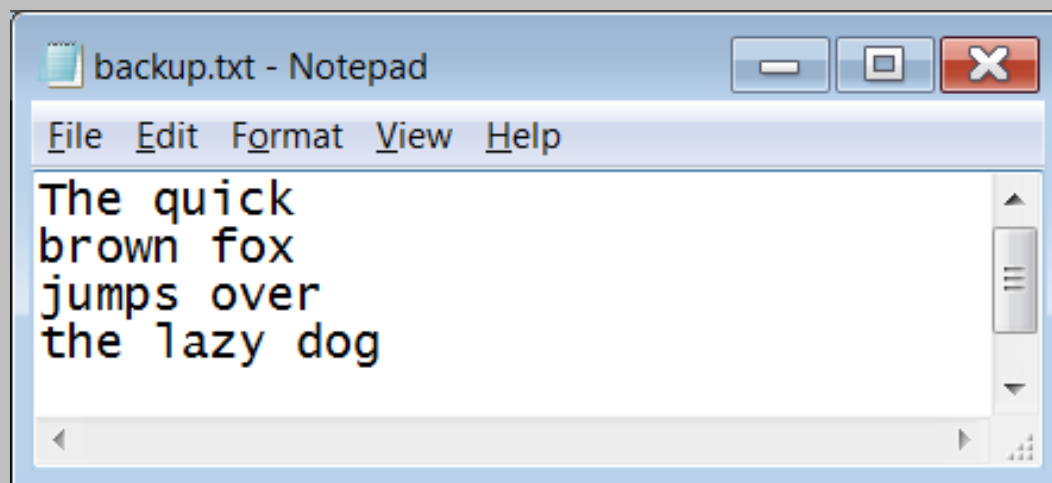
```
1 # TextFiles19.py
2 # This program reads an original file and writes
3 # a backup file with the exact same contents.
4 # It demonstrates that a program can read from
5 # and/or write to multiple files simultaneously.
6 # After you run the program load both original.txt
7 # and backup.txt and you should see the two files
8 # are identical.
9
```

```
10
11 inFile = open("original.txt",'r')
12 outFile = open("backup.txt",'w')
13
14 print()
15
16 for lineOfText in inFile:
17     outFile.write(lineOfText)
18
19 inFile.close()
20 outFile.close()
21
```

Monitor Output: *None*

```
----jGRASP exec: python TextFiles19.py
----jGRASP: operation complete.
```

File Output:



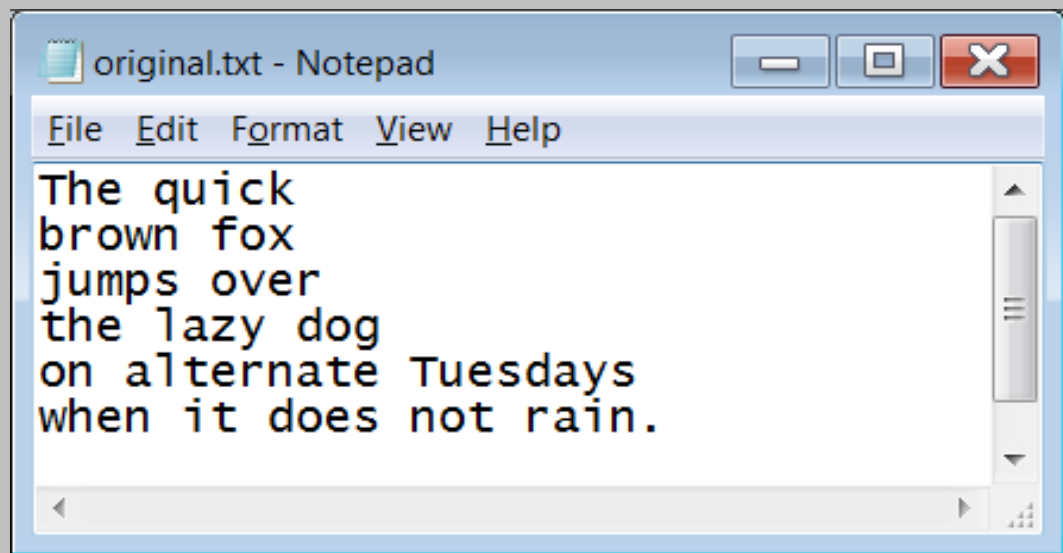
## 17.7 Appending to an Existing File

When you work with files, you hear the term “append” from time to time. What does “append” mean. Well, you have probably seen books with an “appendix”. This is an extra chapter that was added to the end of the book. To *append* means to *add something to the end*. In the case of files, this means you are writing new information to the end of an existing file. How does this work? Opening a file for *writing* erases and replaces any existing file. The trick is to open the file for *appending* instead of writing. Note the 'a' used in the **open** function in program **TextFiles20.py**, shown in Figure 17.22. The program accesses the **"original.txt"** file and opens it for *appending*. Then it writes 2 additional lines of text to the end of the file. Load the **"original.txt"** file to see how it has changed.

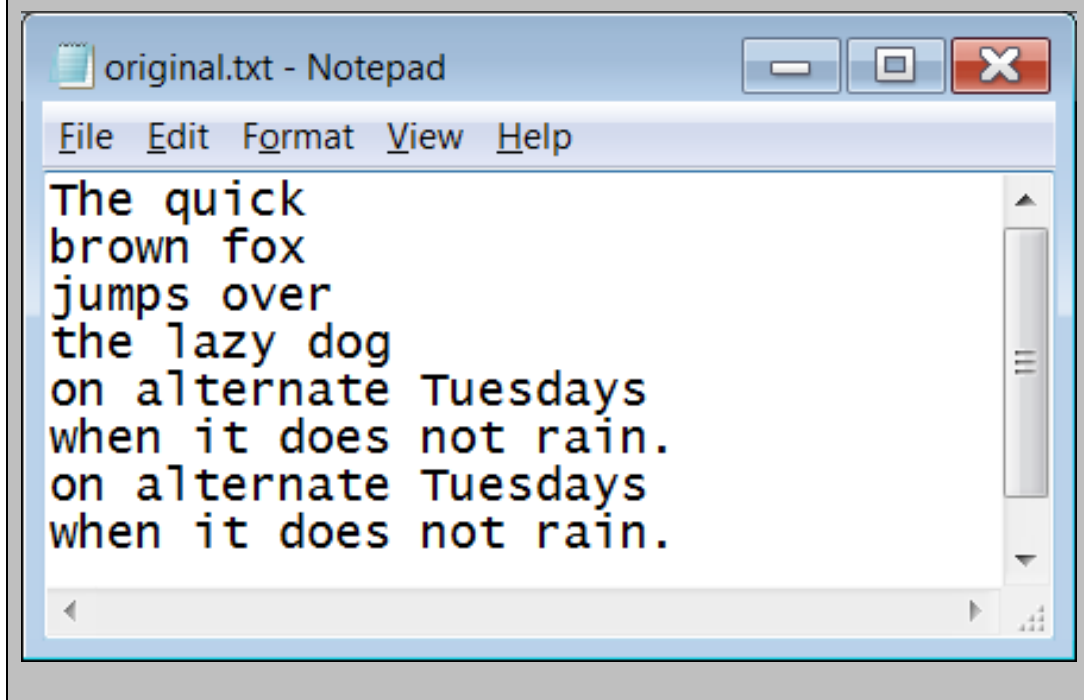
Figure 17.22

```
1 # TextFiles20.py
2 # This program demonstrates how to "append"
3 # data to the end of an existing file.
4
5
6 file = open("original.txt",'a')
7
8 file.write("on alternate Tuesdays\n")
9 file.write("when it does not rain.\n")
10
11 file.close()
```

File Output After First Execution:



### File Output After Second Execution:



Notice what happens when you run the program a second time. Does this make sense? This program will take whatever is in the file "**original.txt**" and add 2 new lines of text to the end of the file. Those 2 new lines of text become part of the file. When executed a second time, the program adds the same 2 lines of text to the end of the file "**original.txt**" a second time. Run this a few more times and see what happens.

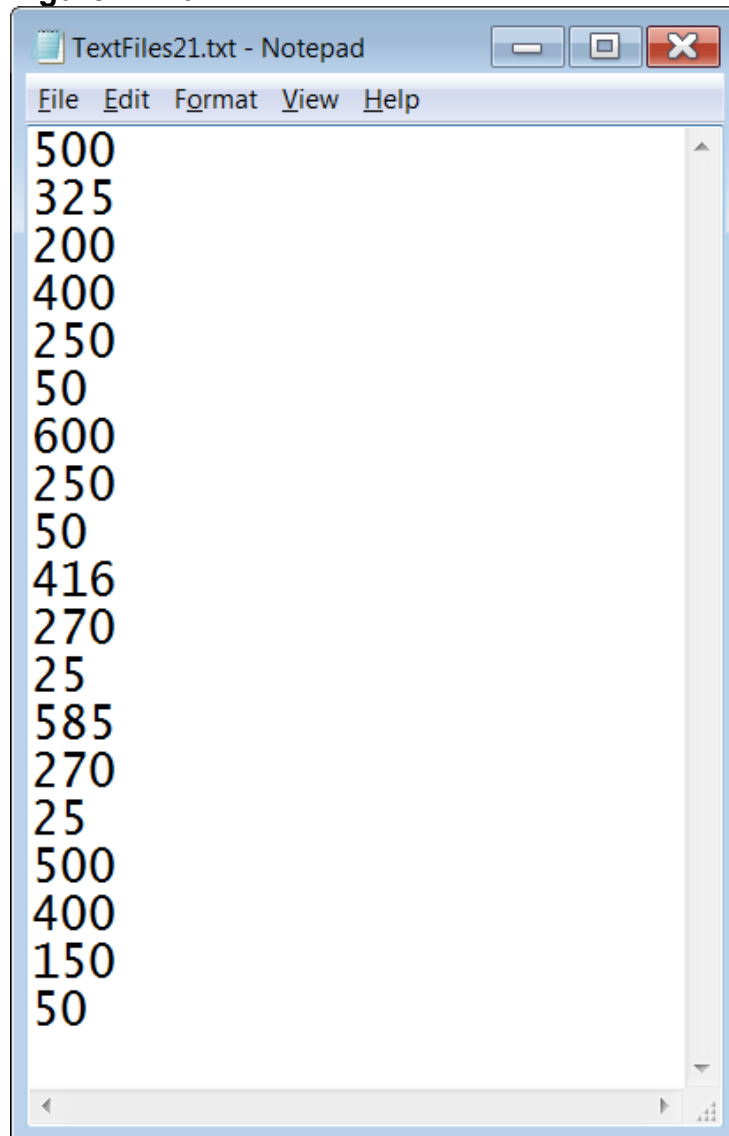


## 17.8 Using Text File Data in Graphics Programs

Imagine that you need to store an image. Depending on the image this can take up quite a bit of memory when information for every single pixel is stored. Sometimes, you can store the information for graphics images simply by storing handful of numbers or some characters. We can actually use text files to efficiently store graphical information. This is what we are going to explore in this final section of the chapter.

First, let us look at the contents of the file **TextFiles21.txt**, shown in Figure 17.23. This file contains 19 integers. In this form, these numbers do not look like they represent anything.

**Figure 17.23**

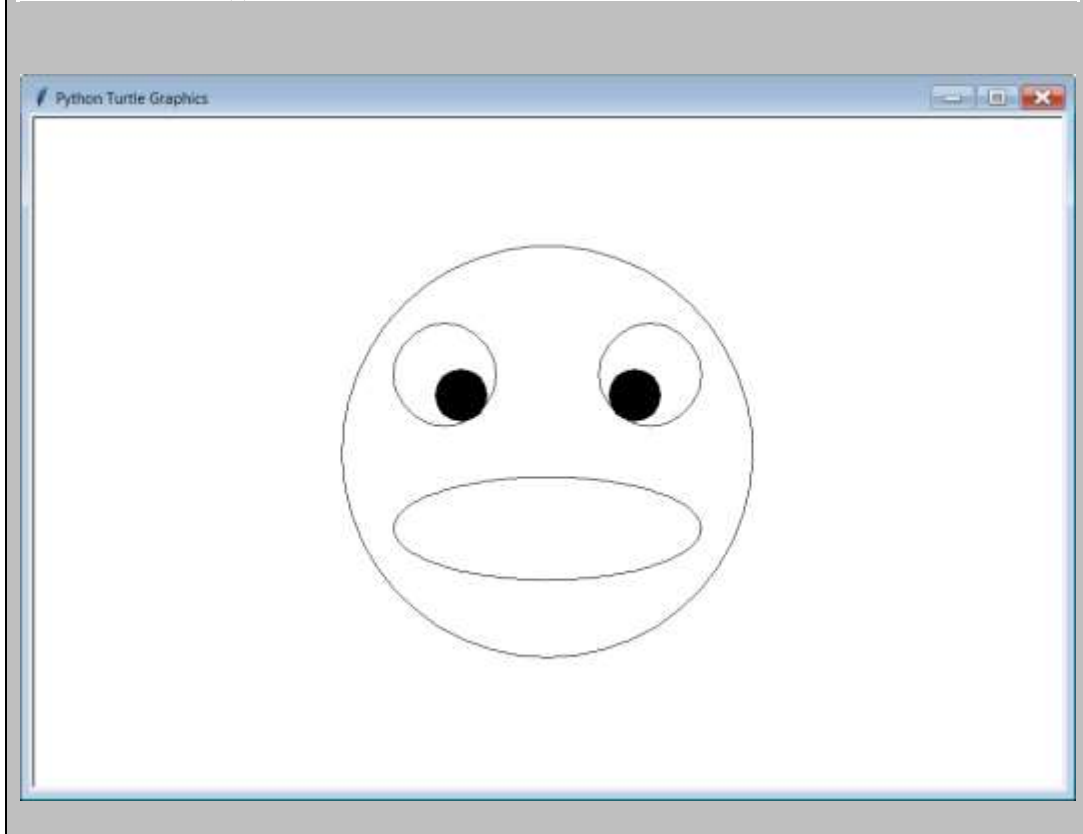


Now let us look at program **TextFiles21.py**, shown in Figure 17.24. This program reads in each of the 19 integers and stores them in 19 different variables. These variables are then used in various **Graphics** commands like **drawCircle**, **fillCircle** and **drawOval**. The result is shown in the output. If this image were stored pixel by pixel, it would have required a lot more than 19 integers.

**Figure 17.24**

```
1 # TextFiles21.py
2 # This program demonstrates that data from a
3 # text file can be used in a graphics program.
4
5
6 from Graphics import *
7
8
9 file = open("TextFiles12.txt",'r')
10
11 x1 = int(file.readline())
12 y1 = int(file.readline())
13 r1 = int(file.readline())
14 x2 = int(file.readline())
15 y2 = int(file.readline())
16 r2 = int(file.readline())
17 x3 = int(file.readline())
18 y3 = int(file.readline())
19 r3 = int(file.readline())
20 x4 = int(file.readline())
21 y4 = int(file.readline())
22 r4 = int(file.readline())
23 x5 = int(file.readline())
24 y5 = int(file.readline())
25 r5 = int(file.readline())
26 x6 = int(file.readline())
27 y6 = int(file.readline())
28 r6 = int(file.readline())
29 r7 = int(file.readline())
30
31 file.close()
32
33 beginGrafX(1000,650)
34
35 drawCircle(x1,y1,r1)
36 drawCircle(x2,y2,r2)
37 drawCircle(x3,y3,r3)
```

```
38 fillCircle(x4,y4,r4)
39 fillCircle(x5,y5,r5)
40 drawOval(x6,y6,r6,r7)
41
42 endGrfx()
```



Text files can be quite useful in graphics programs. We just saw the *face* example which showed that we can store the image as 19 integers instead of many, many pixels. This is far more memory efficient. Graphics backgrounds can take up even more memory. If the graphics background is stored as text and then converted to graphics, it will save memory as well.

Before we worry about drawing an entire background, let us focus on a single line. Program **TextFiles22.py**, in Figure 17.25, reads in one line of text from the file **TextFiles13.txt**, shown in Figure 17.26. There are 3 possible characters in this file. These are CAPITAL letters 'C', 'S' and 'T'. The file stores just one line of text which is **STSC TCSC SCTCSSCTSTTCSSCTC**. The computer will go through this string character by character, traversing it like an array (which it is). Every time it finds a 'C' it will draw a red circle; every time it finds a 'S' it will draw a blue square and every time it finds a 'T' it will draw a green triangle. Compare the output to the text file and see how they correspond. Try altering this text file and then see what happens when you re-execute the program.

Figure 17.25

```
1 # TextFiles22.py
2 # This program demonstrates that non-numerical
3 # data can also be used in a graphics program.
4
5
6 from Graphics import *
7
8
9 def drawBlueSquare(x):
10     setColor("blue")
11     x = x * 50 + 25
12     fillRegularPolygon(x,175,30,4)
13
14
15 def drawRedCircle(x):
16     setColor("red")
17     x = x * 50 + 25
18     fillCircle(x,175,25)
19
20
21 def drawGreenTriangle(x):
22     setColor("green")
23     x = x * 50 + 25
24     fillRegularPolygon(x,183,27,3)
25
26
27
28 #####
29 #  MAIN  #
30 #####
31
32 file = open("TextFiles22.txt",'r')
33 lineOfText = file.readline()
34 file.close()
35
36 beginGrafX(1300,350)
37
38 for x in range(len(lineOfText)):
39     if lineOfText[x] == "S":
40         drawBlueSquare(x)
41     elif lineOfText[x] == "C":
42         drawRedCircle(x)
43     elif lineOfText[x] == "T":
```



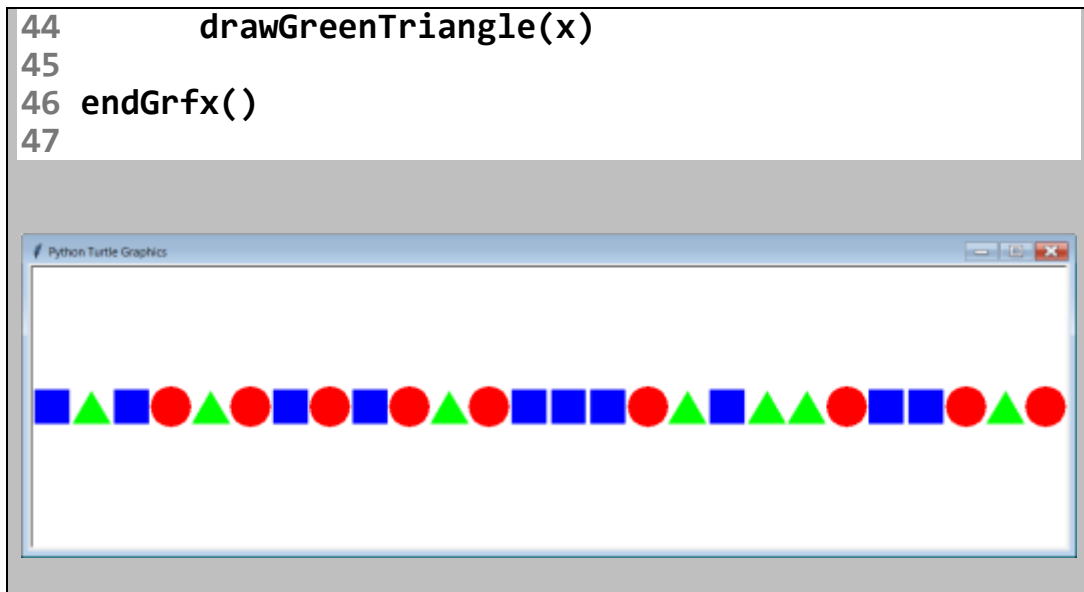
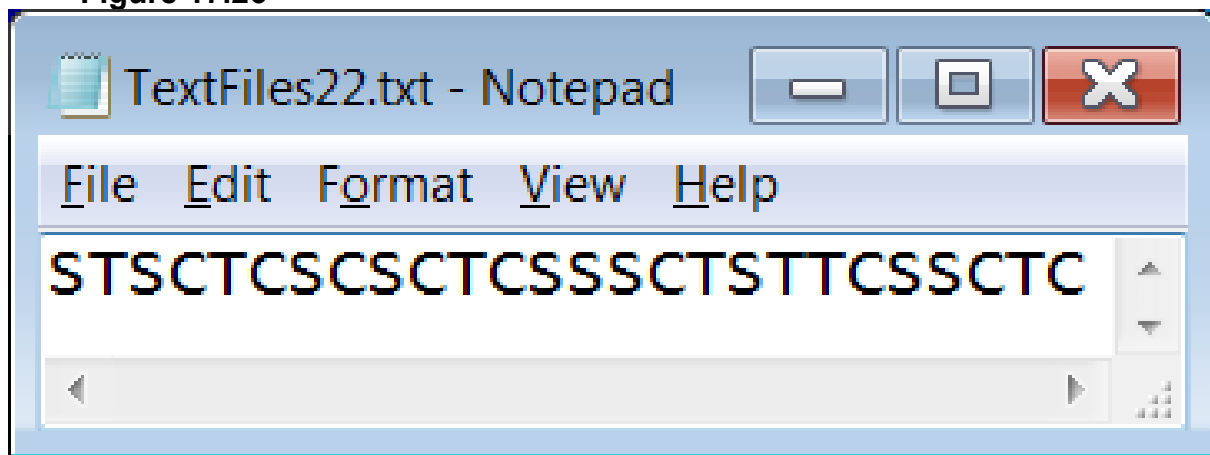


Figure 17.26

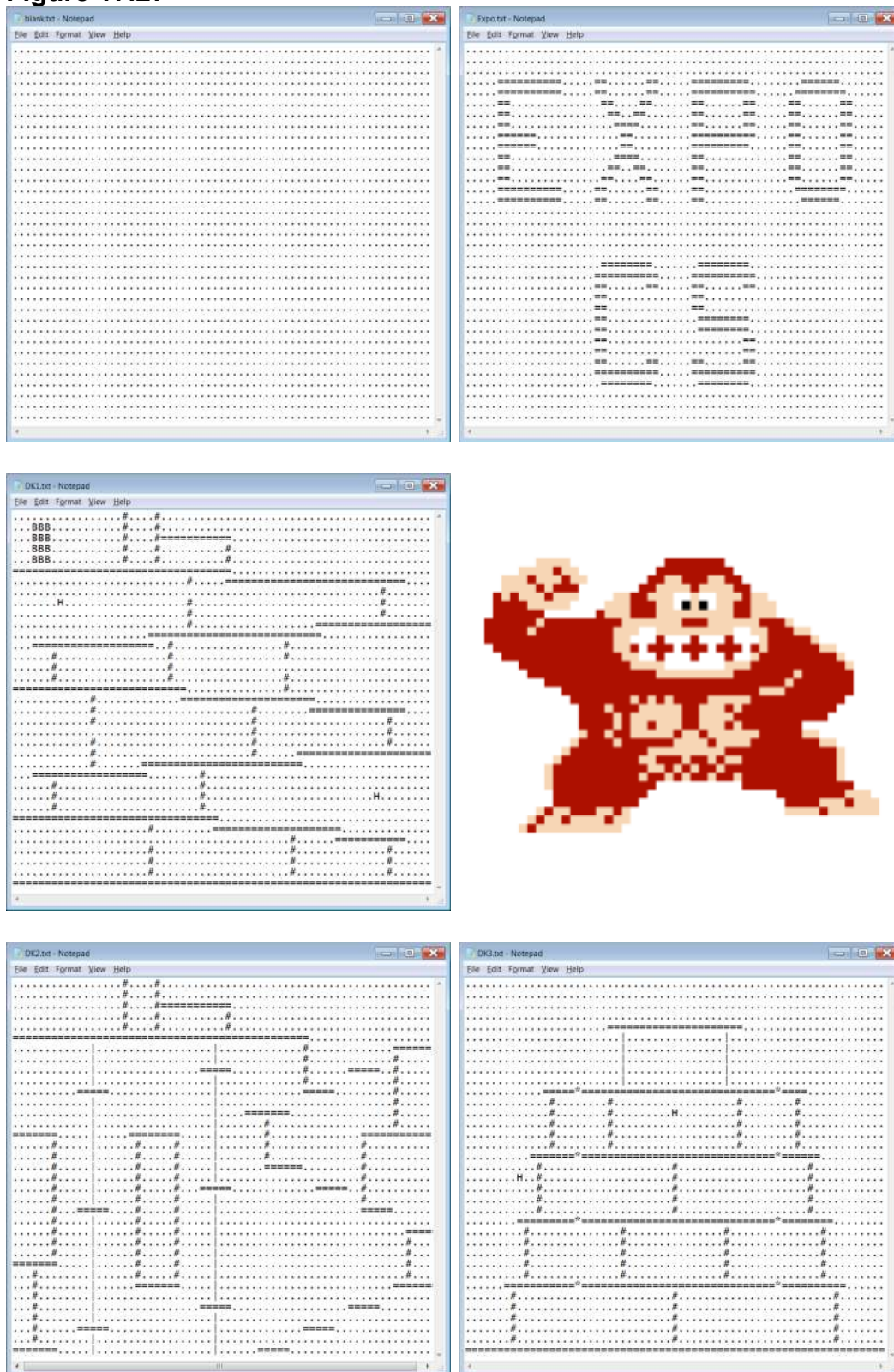


The previous example worked for a single row of output, but the same process can actually apply to an entire background. In the examples that follow, the data files store 35 rows of 65-character strings. This is a total of 2275 characters which is just over 2K of memory. In addition to requiring considerably less memory than what would be required if we stored every pixel, there is also another benefit. We have an easy way to manipulate backgrounds and create new ones.

The final program in this chapter can work with a variety of text files. 5 such text files are provided and shown in Figure 17.27. The names of the 5 files are **blank.txt**, **Expo.txt**, **DK1.txt**, **DK2.txt**, and **DK3.txt**.

**NOTE:** While it is a convention, text files are not required to have a **.txt** extension.

Figure 17.27



Program **TextFiles23.py**, shown in Figure 17.28, is similar to the previous program, except now it reads multiple lines of text to draw a background for a **1300** by **700** graphics window. The program starts by asking you for a file name.

**Figure 17.28**

```
1 # TextFiles23.py
2 # This program will ask the user to enter
3 # the name of a text file, and then use that
4 # text file to display a graphics background.
5 # Students are provided with 5 example files:
6 # blank.txt, Expo.txt, DK1.txt, DK2.txt, and
7 # DK3.txt -- the latter 3 resemble stages from
8 # Nintendo's Donkey Kong arcade game.
9 # NOTE: This same program is used for Lab 16B.
10
11
12 from Graphics import *
13 import os
14
15
16 def convert(q):
17     return q * 20
18
19
20 def drawSpace(r,c):
21     x = convert(c)
22     y = convert(r)
23     setColor("black")
24     fillRect(x,y,20,20)
25
26
27 def drawGirder(r,c):
28     x = convert(c)
29     y = convert(r)
30     setColor("red");
31     fillRect(x,y,20,20)
32     setColor("black")
33     fillCircle(x+10,y+9,6)
34
```

```

35
36 def drawLadder(r,c):
37     x = convert(c)
38     y = convert(r)
39     setColor("black")
40     fillRect(x,y,20,20)
41     setColor("white")
42     fillRect(x,y,3,20)
43     fillRect(x+16,y,4,20)
44     fillRect(x,y+8,20,4)
45
46
47 def drawHammer(r,c):
48     x = convert(c)
49     y = convert(r)
50     setColor("black")
51     fillRect(x,y,20,20)
52     setColor(150,100,15)
53     fillRect(x,y,20,10)
54     setColor("yellow");
55     fillRect(x+8,y+10,4,10)
56
57
58 def drawBarrel(r,c):
59     x = convert(c)
60     y = convert(r)
61     setColor("black")
62     fillRect(x,y,20,20)
63     setColor(150,100,15)
64     fillRect(x+5,y,10,20)
65     fillArc(x+5,y+10,5,10,180,360)
66     fillArc(x+15,y+10,5,10,0,180)
67     setColor("black")
68     drawLine(x+16,y+9,x+16,y+13)
69     drawLine(x+15,y+6,x+15,y+16)
70     setColor("white")
71     drawLine(x+5,y+4,x+5,y+14)
72     drawLine(x+4,y+7,x+4,y+10)
73     setColor(211,211,211)
74     drawLine(x+5,y,x+15,y)
75

```

```

76
77 def drawLock(r,c):
78     x = convert(c)
79     y = convert(r)
80     setColor("cyan")
81     fillRect(x,y,20,5)
82     setColor("yellow")
83     fillRect(x,y+5,20,15)
84
85
86 def drawPole(r,c):
87     x = convert(c)
88     y = convert(r)
89     setColor("black")
90     fillRect(x,y,20,20)
91     setColor("cyan")
92     fillRect(x+7,y,6,20)
93
94
95 def drawUnknown(r,c):
96     x = convert(c)
97     y = convert(r)
98     setColor("pink")
99     fillRect(x,y,20,20)
100    setColor("black")
101    drawString("?",x+3,y+25,"Courier",16,"bold")
102
103
104
105 #####
106 #  MAIN  #
107 #####
108
109 numRows = 35
110 numCols = 65
111 fileName = textinput("Graphics Background",
112 "Enter the name of the text file.")
113 while not os.path.exists(fileName):
114     fileName = textinput("File does not exist.",
115 "Enter the name of the text file.")

```

```

114 file = open(fileName,"r")
115 background = file.readlines()
116 file.close()
117
118 beginGrfx(1300,700)
119
120 for r in range(numRows):
121     for c in range(numCols):
122         if background[r][c] == '.':
123             drawSpace(r,c)
124         elif background[r][c] == '=':
125             drawGirder(r,c)
126         elif background[r][c] == '#':
127             drawLadder(r,c)
128         elif background[r][c] == 'H':
129             drawHammer(r,c)
130         elif background[r][c] == 'B':
131             drawBarrel(r,c)
132         elif background[r][c] == '*':
133             drawLock(r,c)
134         elif background[r][c] == '|':
135             drawPole(r,c)
136         else:
137             drawUnknown(r,c)
138     update()
139
140 endGrfx()
141

```

Figures 17.29 through 17.33 show 5 different outputs of this program. Each output is accompanied by the text file that was used to create it.

Figure 17.29

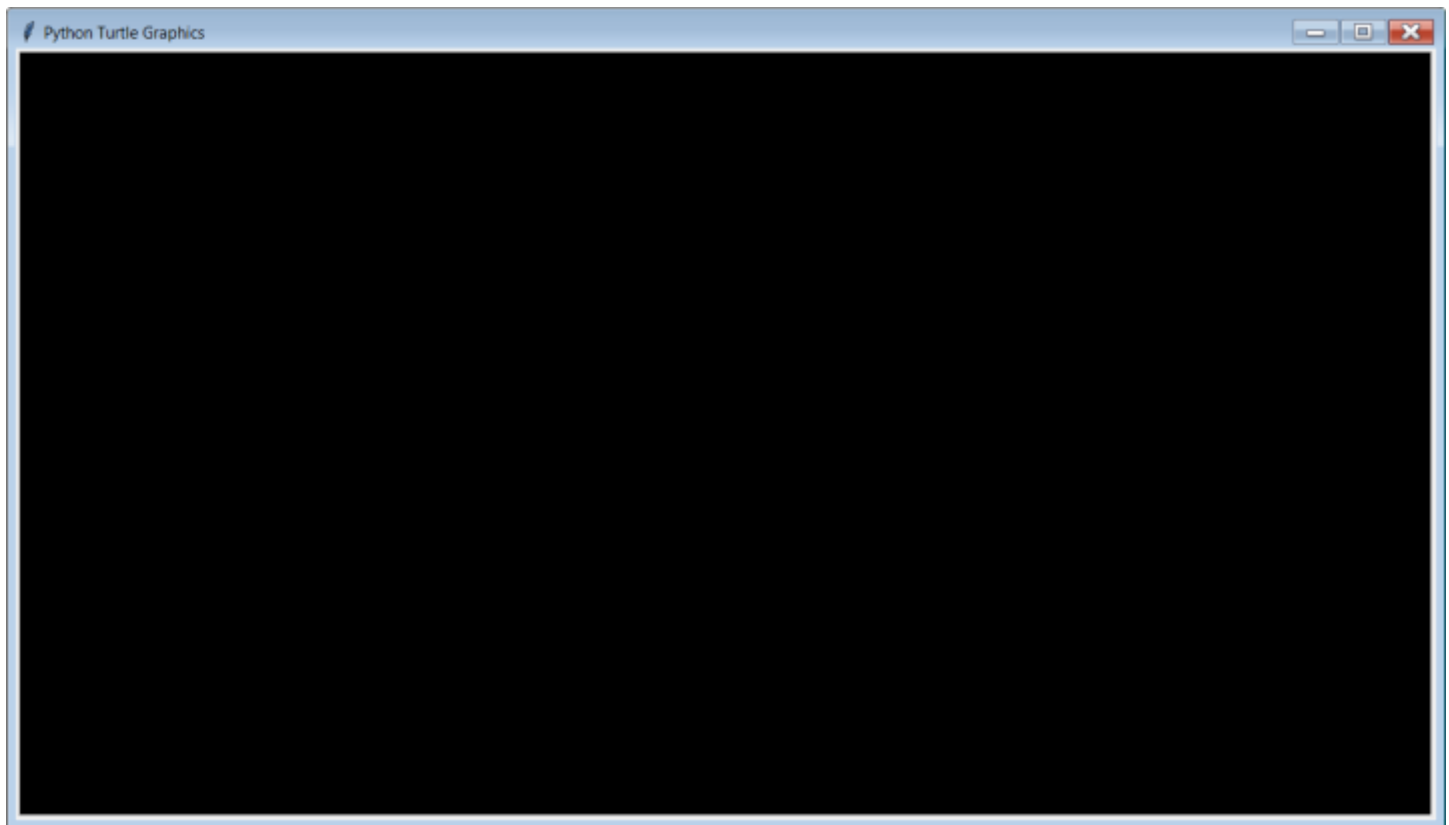
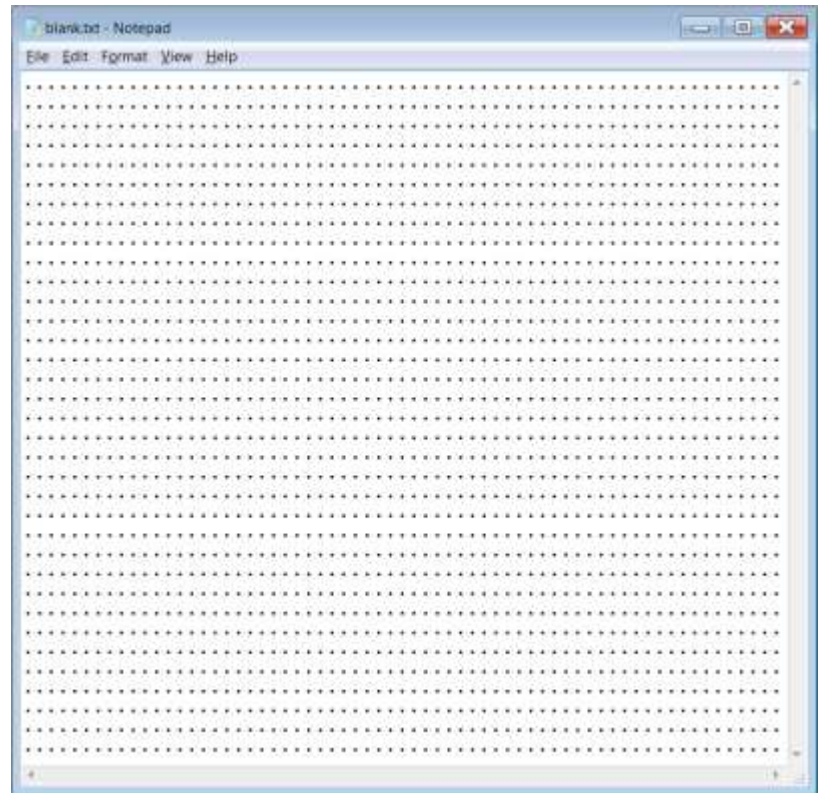
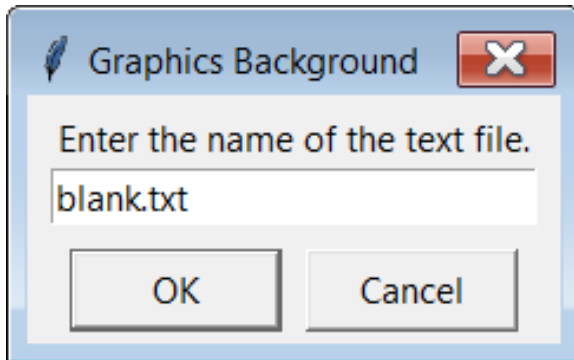
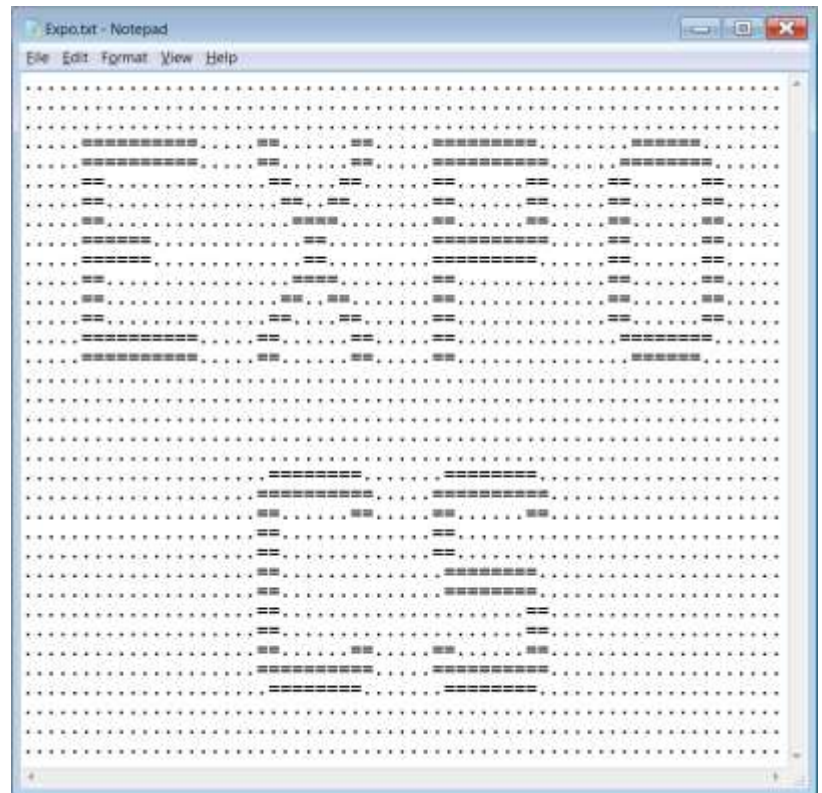
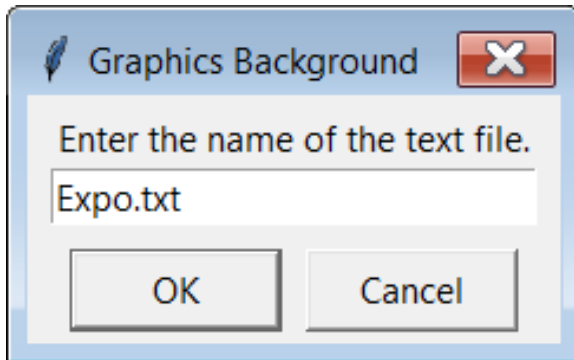




Figure 17.30





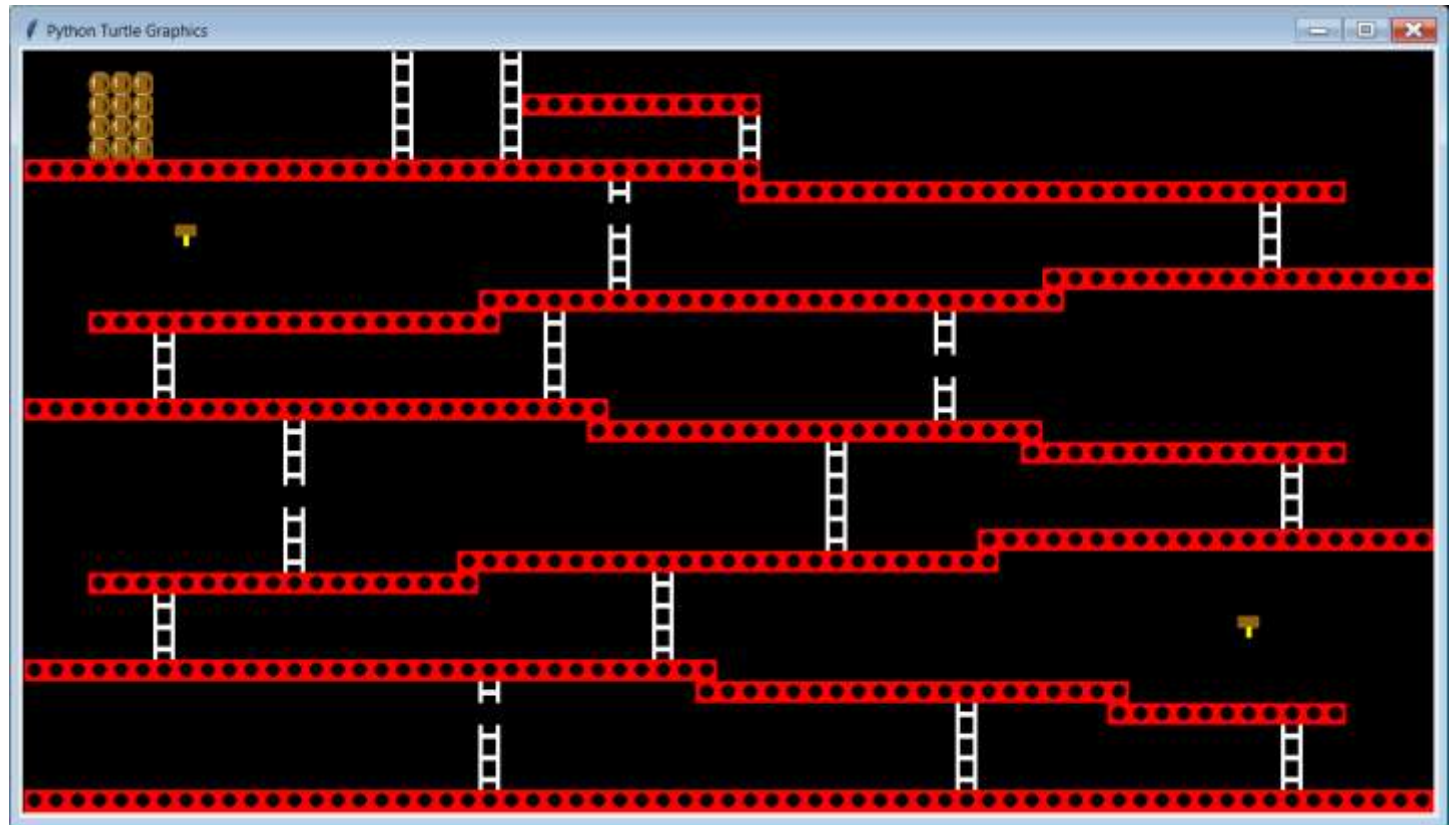
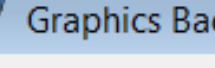


Figure 17.32

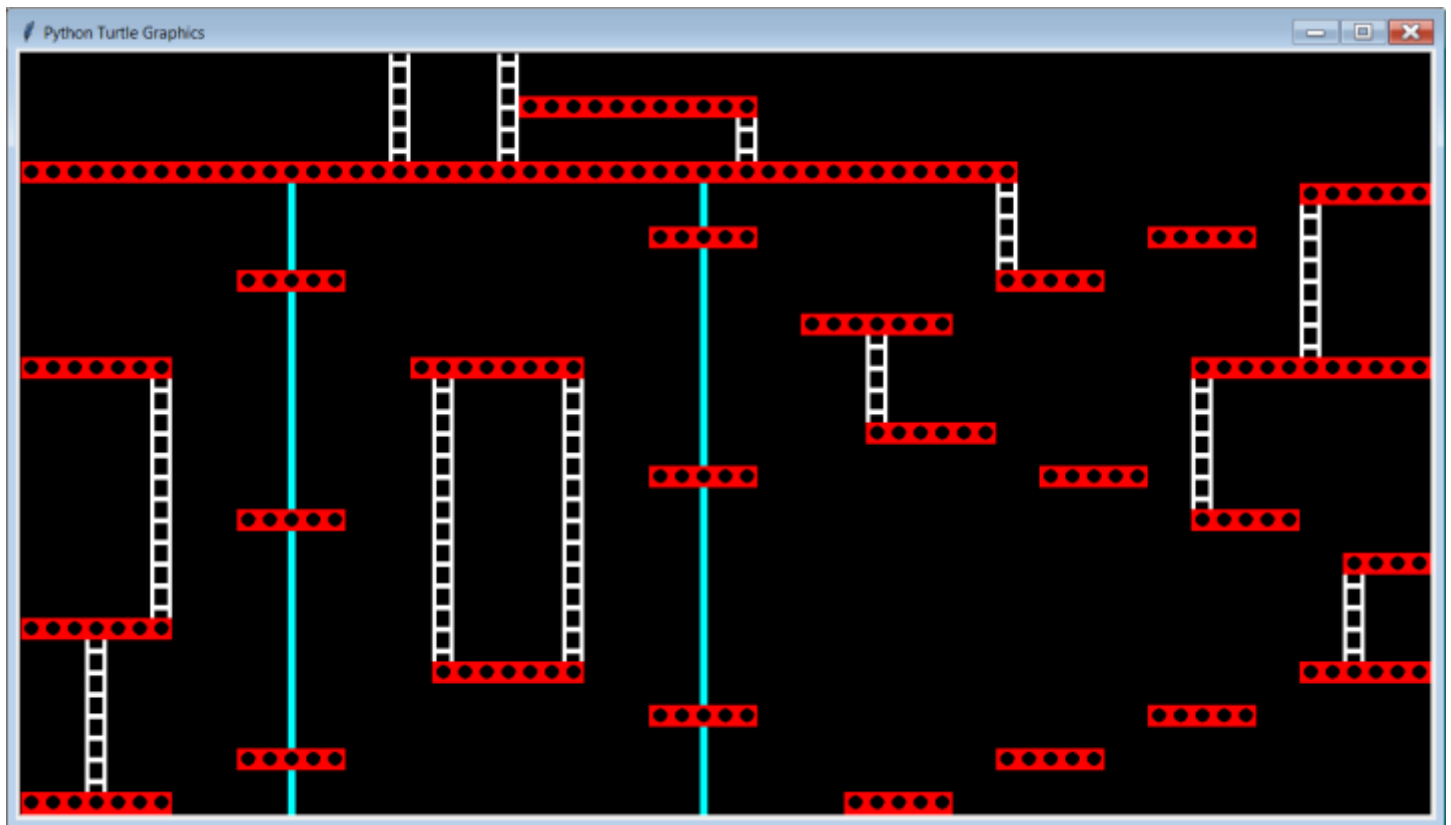
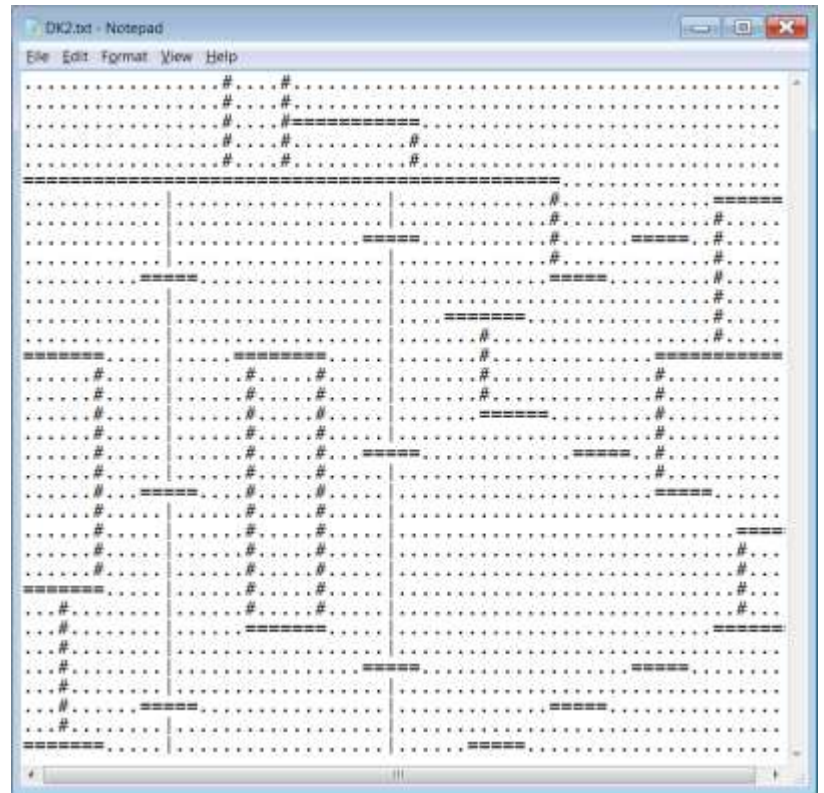
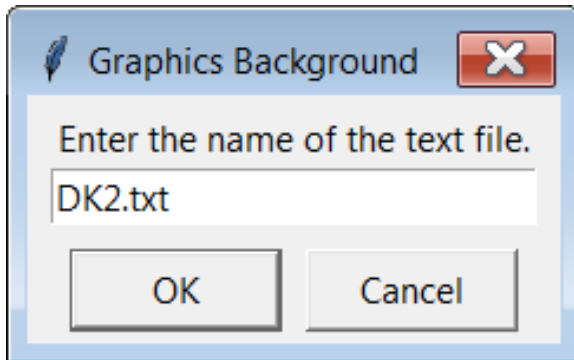


Figure 17.33

