Chapter III

Introduction to Python Coding

Chapter III Topics

3.1 Introduction

Some students are somewhat impatient in general, but when it comes to technology, many students are very impatient. Lots of students march in a computer science class with prior knowledge of using a computer. Such students have played fancy video games, emailed or set up video chats with their friends from all over town, researched on the Internet, and used computers for word processing and many other applications.

Computer science is not very different from many other professions in the introductory stages. Few medical students perform open-heart surgery during the first week of medical school. The average law student is not asked to argue a case before the Supreme Court after studying one court case. Navy pilot candidates are not asked to land an airplane on an aircraft carrier at night, during a storm, with one engine shut down, during week-one of flight training. In other words, if you expect to start creating programs that will guarantee you ten-thousand dollars a week, for a part-time job, then you will be disappointed.

This chapter will start by showing you the necessary tools to write simple programs. Keep in mind that you will be creating your programs in a high-level language, Python. This means you need some type of editing environment in which to write your programs. After the program is written you need to translate and execute your program. For now, learning those fundamental program-writing skills is your primary concern.

There are two primary output modes that are available with Python: *text* and *graphics*. Text is both pretty easy and pretty boring. Graphics is exciting, interesting and quite a bit more challenging. The focus in this chapter is to understand first how to execute small, simple text programs. Exciting video games, snazzy graphics based interactive programs are all possible with Python. However, that is not a practical starting point.

It is possible that you already know some other programming language, such as C++ or Visual BASIC. If such is the case, you will learn Python much faster and you may find the topic descriptions in *Exposure Computer Science* rather slow moving. That is fine, but keep in mind that this text book is written for the student who has no prior knowledge of any programming language or any exposure to computer science concepts. If you have such prior knowledge, you are lucky, but do be careful. Python is similar to some other program languages, but there are also major differences.

3.2 Using jGRASP for Python

In Chapter II you installed the ***J**ava* ***D**evelopment* ***K**it* (JDK), **jGRASP** and **Python** in order to write and execute Python programs. You may wonder why you had to install the Java Development Kit to run Python programs. The answer has nothing to do with Python. The **jGRASP** software was written in Java. Without Java software installed, **jGRASP** cannot execute.

jGRASP is a versatile little piece of software that can actually work with several different languages. One of them is Python. Before we start learning Python, we need to understand the general tools necessary to write a Python program. Basically, you need a text editor, translator and an output window. In the past, these separate features were handled with separate pieces of software which was quite tedious. Today, thing are much simpler because we have IDEs. *IDE* stands for ***I**ntegrated* ***D**evelopment* ***E**nvironment*. An IDE is an "environment" that contains everything you need to "develop" programs all "integrated" together.

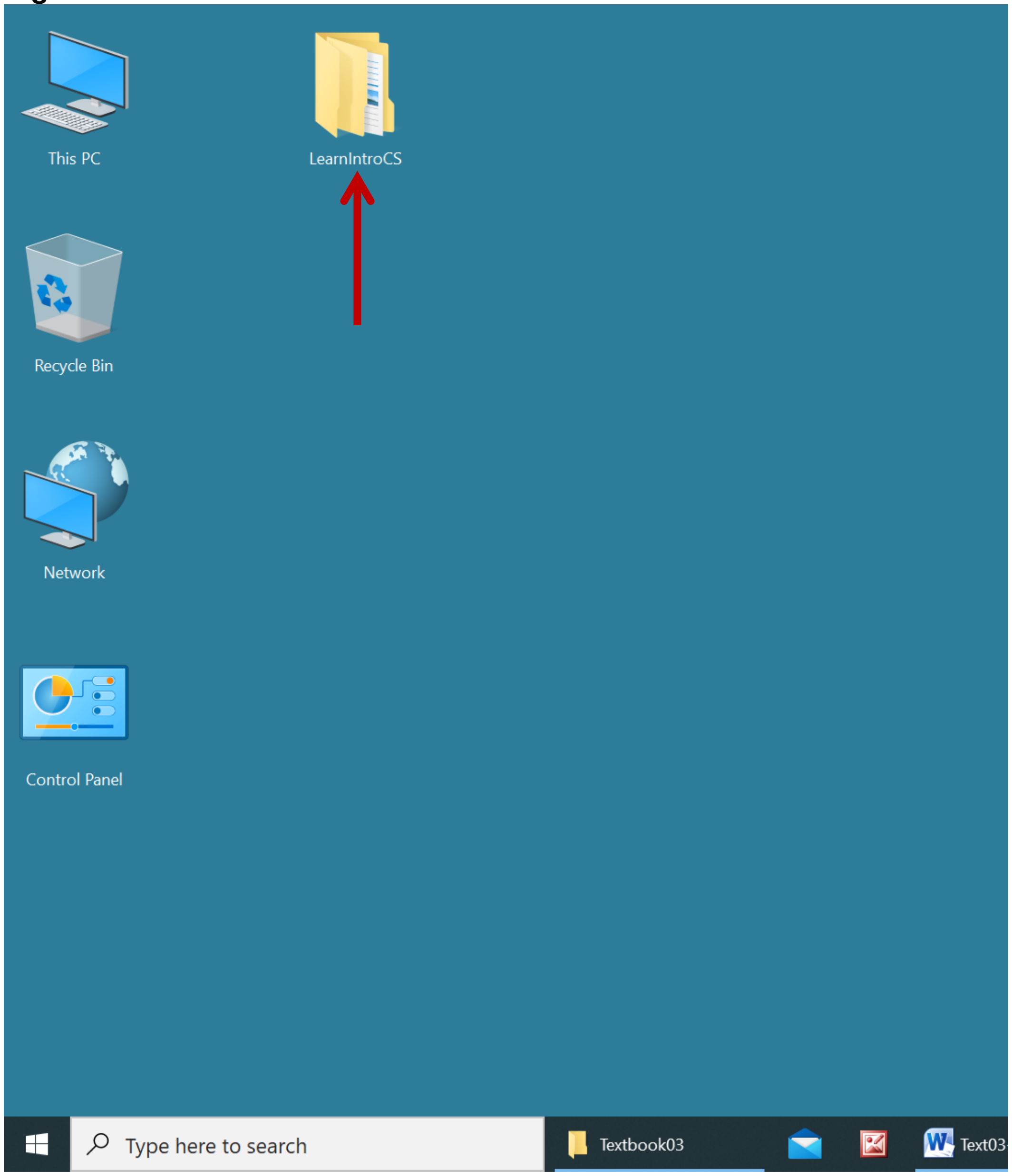
Features of an IDE	
Text Editor	A full *screen editor* for writing your programs
	A *file manager* to save programs and load them again later
Translator	Used to translate your source code into machine code. Most languages use a compiler. Python uses an interpreter.
Output/Message Window	Show the output of your programs or displays error messages

The IDE that we will be using is **jGRASP**. One of the main reasons that we are using **jGRASP** is that it is *freeware*. **jGRASP** also has other features that make it work well for our lab lectures and lab assignments.

The LearnIntroCS Folder

Before we can continue, I need to mention something about the **LearnIntroCS** folder. This chapter assumes that the **LearnIntroCS** folder has been been copied to your **Desktop** as shown in Figure 3.1. If you have this folder in a different location, (**Documents** for example) then you will need to go to that location anytime the directions say to go to the **Desktop**. If you do not have this folder at all, you will need to get a copy of it from your teacher before you can do anything else.

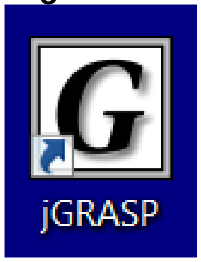
Figure 3.1



Loading jGRASP

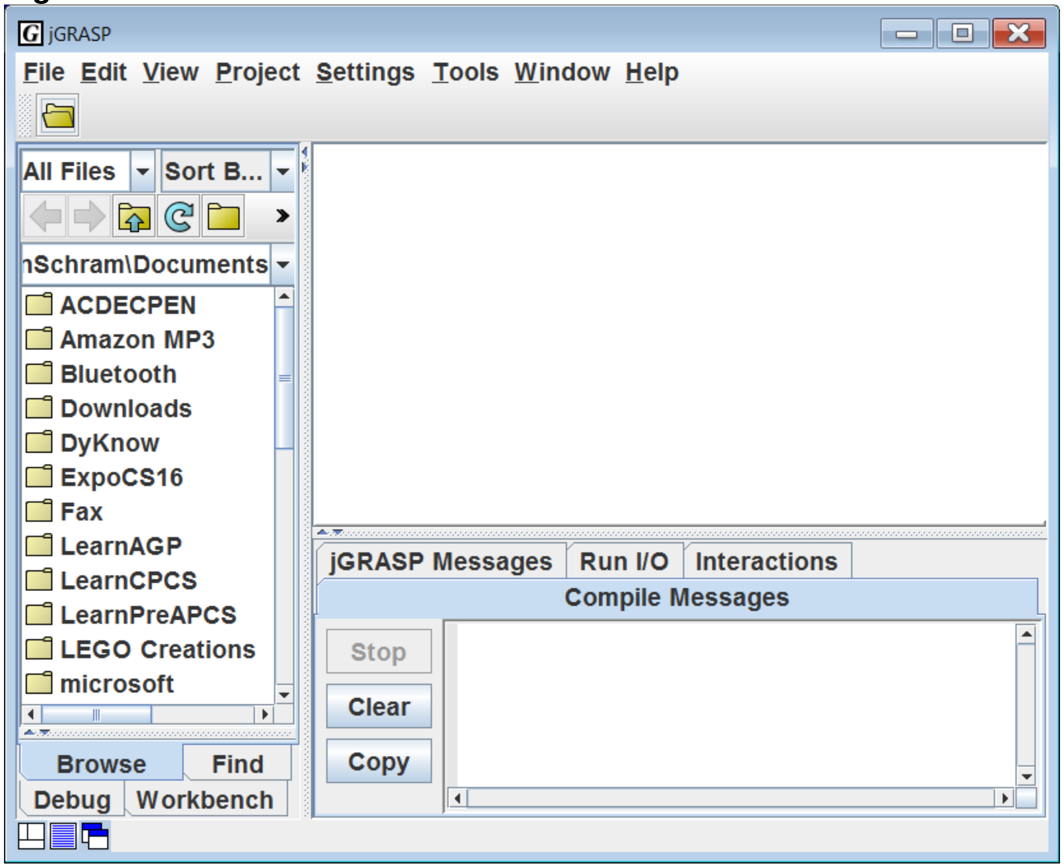
Figure 3.2 shows the icon for **jGRASP**. Depending on your computer, you should have this icon on your desktop, taskbar, Start Menu, dock, or in nthe list of applications that you see when you click the **Launcher** icon.

Figure 3.2



When you load **jGRASP**, it should resemble Figure 3.3. You will notice 3 distinct windows in the **jGRASP** software. First, the window on the left is the *Browse Window*. Do not be concerned that your *Browse Window* does not match mine right now. It will match later. Next, the window on the upper-right is the *Edit Window*. This is where you will do most of your work writing your programs. Finally, the window on the bottom-right is the *Output/Message Window*. This is where you will see your program's output. If your program has errors, this is also where the error messages will be displayed.

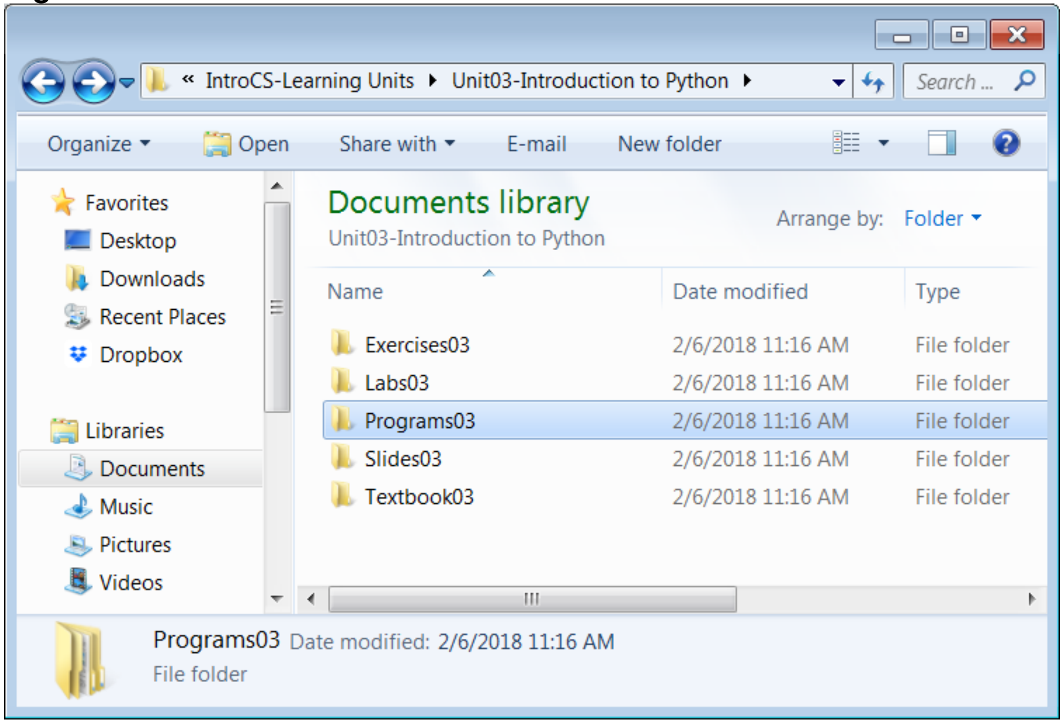
Figure 3.3



Organization of Program Examples in LearnIntroCS

All of the program examples used throughout this textbook were written as actual programs first. We will load and manipulate these examples during our Lab/Lectures. You already know that in the **Learning Units** folder, there is a separate folder for each chapter or "unit". The Unit 3 subfolder is shown in Figure 3.4. You will notice a new subfolder that was not present in Units 1 or 2. This is the **Programs** subfolder. This contains all of the program examples for that particular unit. When we have a Lab/Lecture, students will go to this folder to load the program examples. In many cases, the **Programs** subfolder has subfolders of its own. Open up **Programs03** and you see it contains 3 subfolders as shown in Figure 3.5. If you look at the names of these subfolders, you should see that they are similar to some of the sections of this chapter.

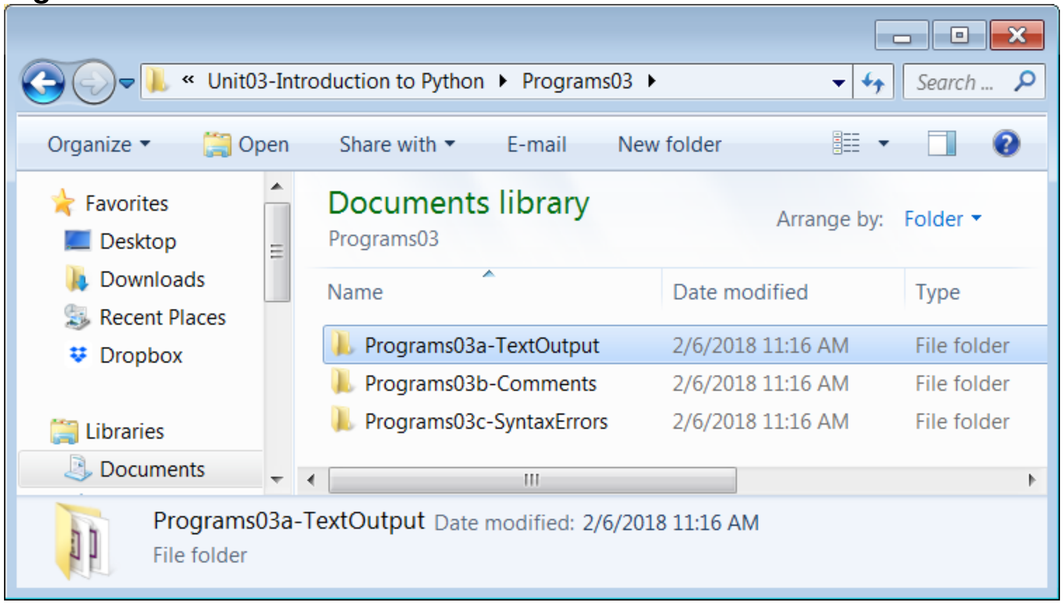
Figure 3.4



Name	Date modified	Type
Exercises03	2/6/2018 11:16 AM	File folder
Labs03	2/6/2018 11:16 AM	File folder
Programs03	2/6/2018 11:16 AM	File folder
Slides03	2/6/2018 11:16 AM	File folder
Textbook03	2/6/2018 11:16 AM	File folder

Programs03 Date modified: 2/6/2018 11:16 AM
File folder

Figure 3.5



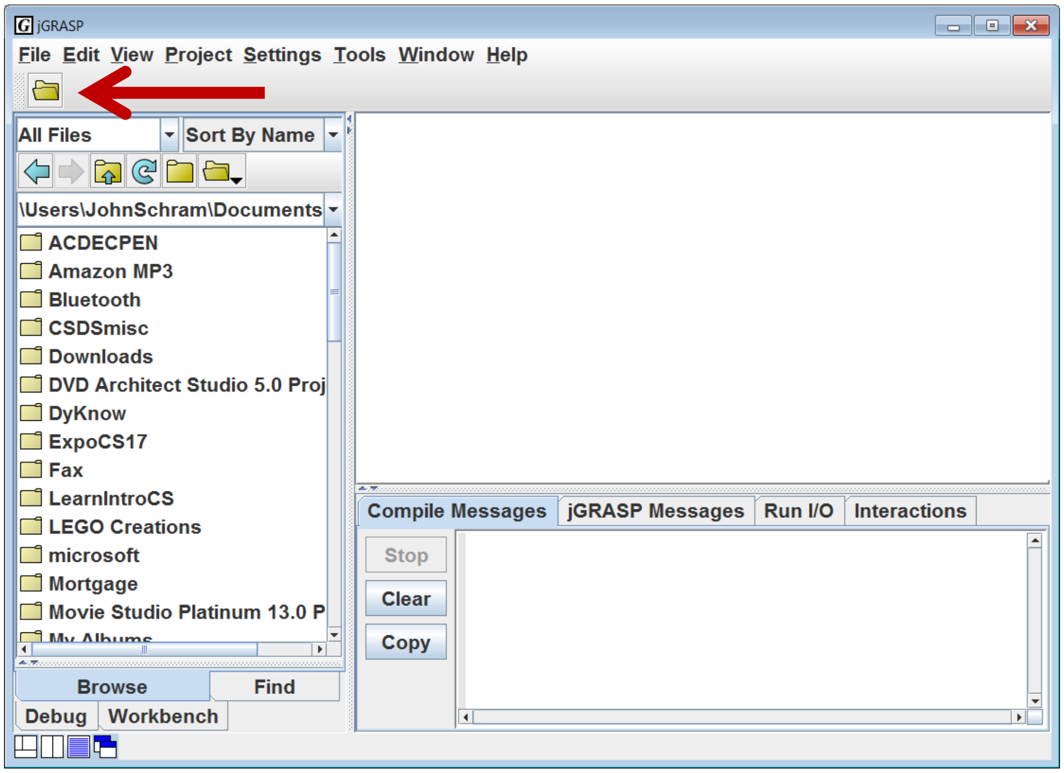
Name	Date modified	Type
Programs03a-TextOutput	2/6/2018 11:16 AM	File folder
Programs03b-Comments	2/6/2018 11:16 AM	File folder
Programs03c-SyntaxErrors	2/6/2018 11:16 AM	File folder

Programs03a-TextOutput Date modified: 2/6/2018 11:16 AM
File folder

Loading a Python File in jGRASP

Click the **Open File** icon near the top-left corner of the window as shown in Figure 3.6.

Figure 3.6

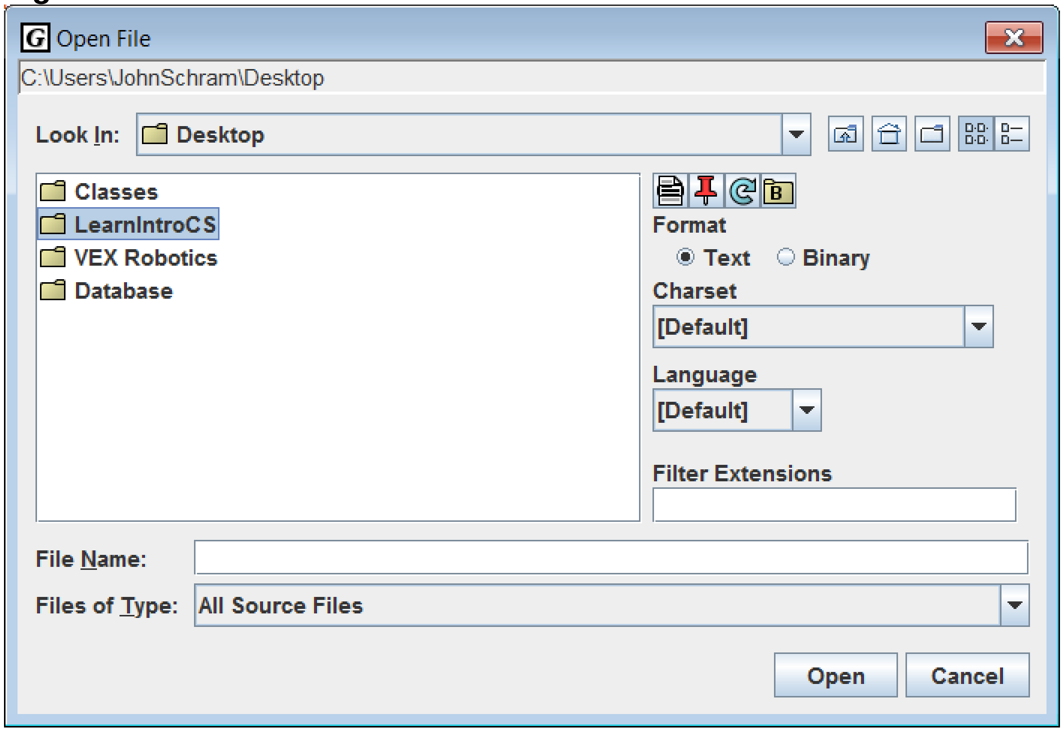


This will open the **Open File** window shown in Figure 3.7. Before we can browse to a specific program example, we need to find the **LearnIntroCS** folder. For most of you, the folder should on your **Desktop**. This is what is shown in Figure 3.7.

If you do not see the **Desktop** folder, you may need to click the *Parent Folder Icon* (looks like a folder with an up arrow) multiple times until you see a folder called **Users**. If you go to the **Users** folder, you should find a folder with your name, and then you will find the **Desktop** folder.

If you have your **LearnIntroCS** folder in a different location (like **Documents**) then you need to browse to that location instead.

Figure 3.7



Double-click **LearnIntroCS** (or whatever your folder is called).

Then double-click the following:

1. **Learning Units**
2. **Unit03-Introduction to Python**
3. **Programs03**
4. **Programs03a-TextOutput**.

Your window should now match Figure 3.8. You can now finally select the file that we want to load. Double-click **TextOutput01.py**

NOTE: Python filenames have a **.py** extension.

Figure 3.8

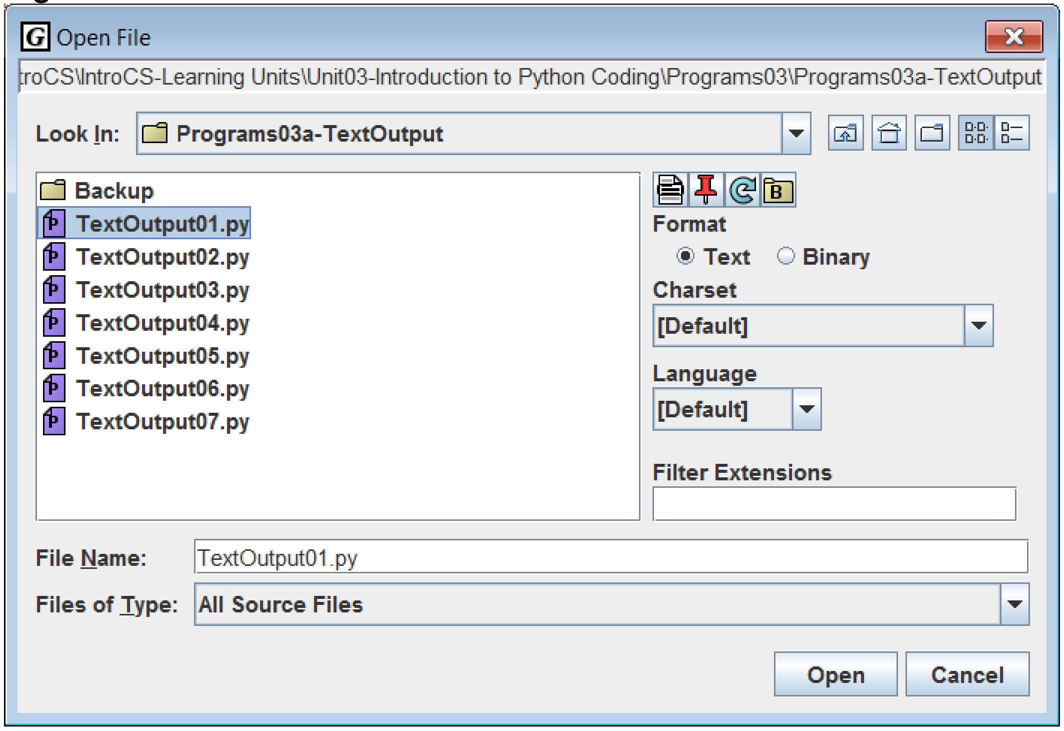
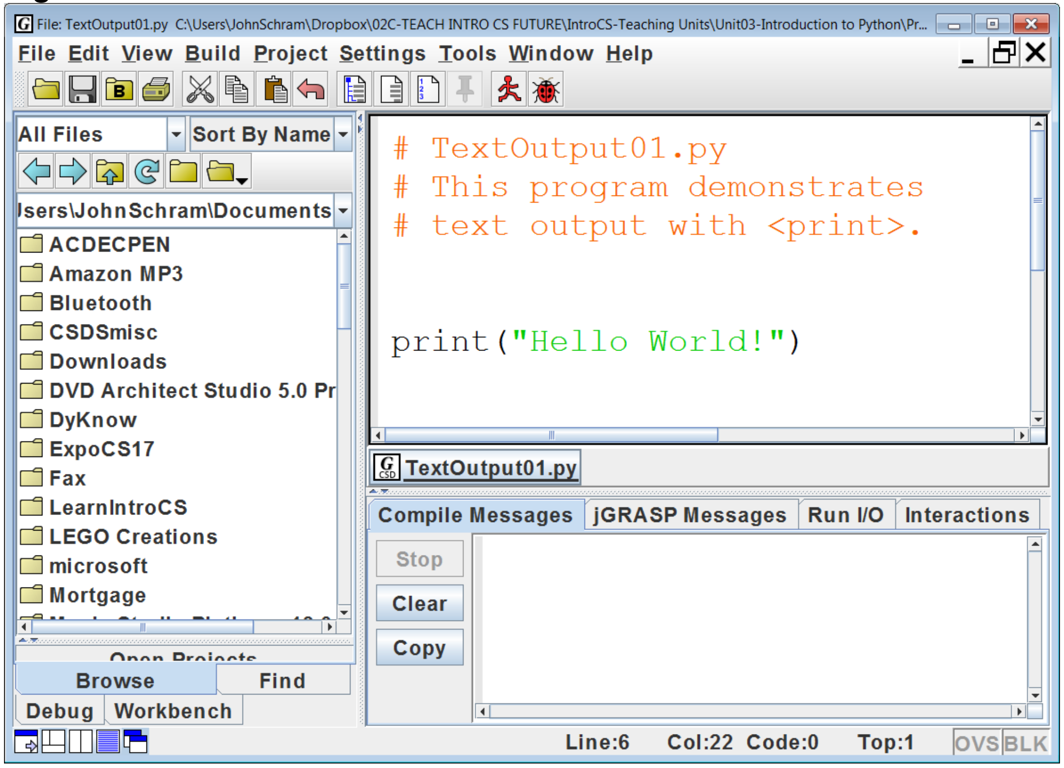


Figure 3.9 shows the file **TextOutput01.py** loaded in **jGRASP**.

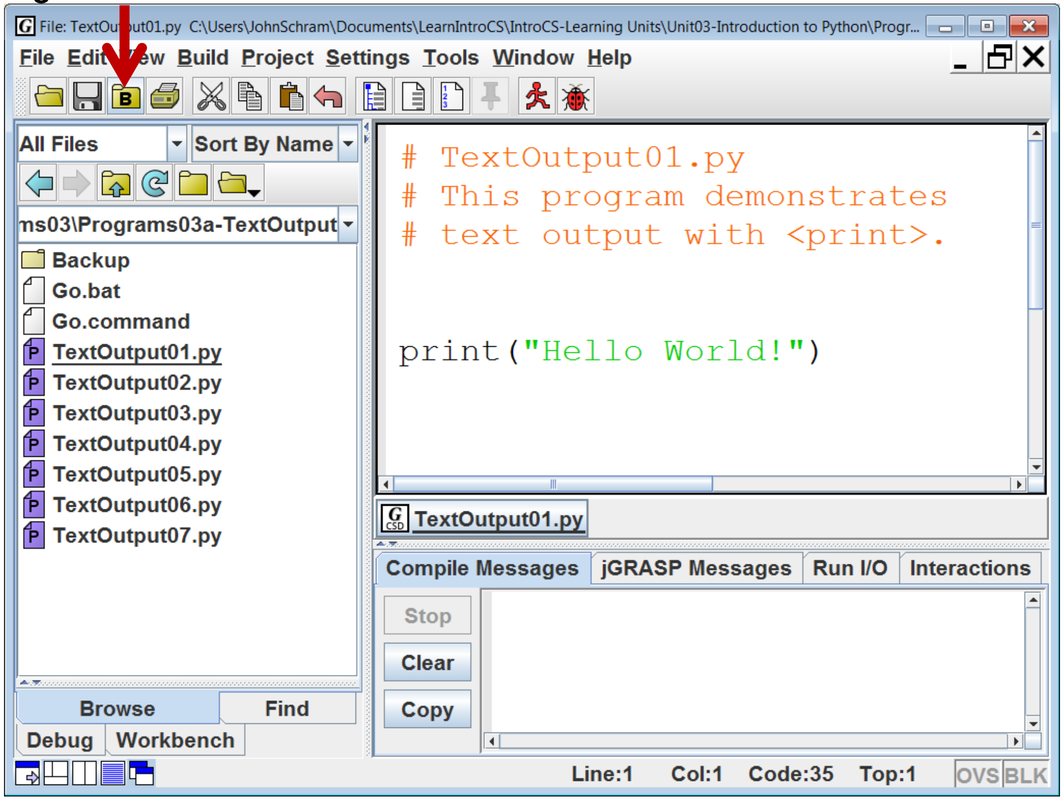
Figure 3.9


File: TextOutput01.py C:\Users\JohnSchram\Dropbox\02C-TEACH INTRO CS FUTURE\IntroCS-Teaching Units\Unit03-Introduction to Python\Pr...
File Edit View Build Project Settings Tools Window Help
All Files
Sort By Name
Isers\JohnSchram\Documents
ACDECPEN
Amazon MP3
Bluetooth
CSDSmisc
Downloads
DVD Architect Studio 5.0 Pr
DyKnow
ExpoCS17
Fax
LearnIntroCS
LEGO Creations
microsoft
Mortgage
Open Projects
Browse
Find
Debug
Workbench
TextOutput01.py
Compile Messages
jGRASP Messages
Run I/O
Interactions
Stop
Clear
Copy
Line:6 Col:22 Code:0 Top:1
OVS BLK

```
# TextOutput01.py
# This program demonstrates
# text output with <print>.


print("Hello World!")
```

We are almost ready to continue, but there are a couple more things we need to do to set up. Near the upper-left corner of the window, click the icon that of a folder with a capital **B**. This will make the *Browse Window* match the folder of the file in the *Edit Window* as shown in Figure 3.10. Now, when we want to load any other file from this folder, we can just double-click it from the *Browse Window*.

Figure 3.10



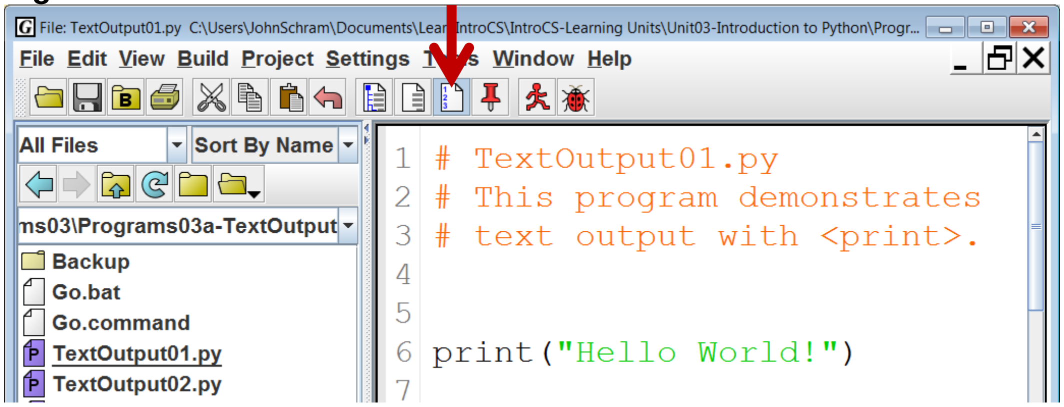
```
# TextOutput01.py
# This program demonstrates
# text output with <print>.


print("Hello World!")
```

The last thing we need to do for setup is to turn on *Line Numbers*. Line numbers are very important. When your program has errors, the computer will tell you that the error was detected on a certain line number. The ability to see these line numbers make fixing your programs much easier.

To turn on the *Line Numbers*, click the icon that is near the top-middle of the window which looks like a piece of paper with some tiny numbering. It is to the left of the pushpin icon. See Figure 3.11. Once you turn *Line Numbers* on, it should stay "on" for all of your programs. If it ever goes "off" for any reason, just click the icon again.

It needs to be understood that the line numbers are not part of the program. They are something that **jGRASP** adds for your convenience. The examples in this textbook will include the line numbers for the sake of clarity.

Figure 3.11

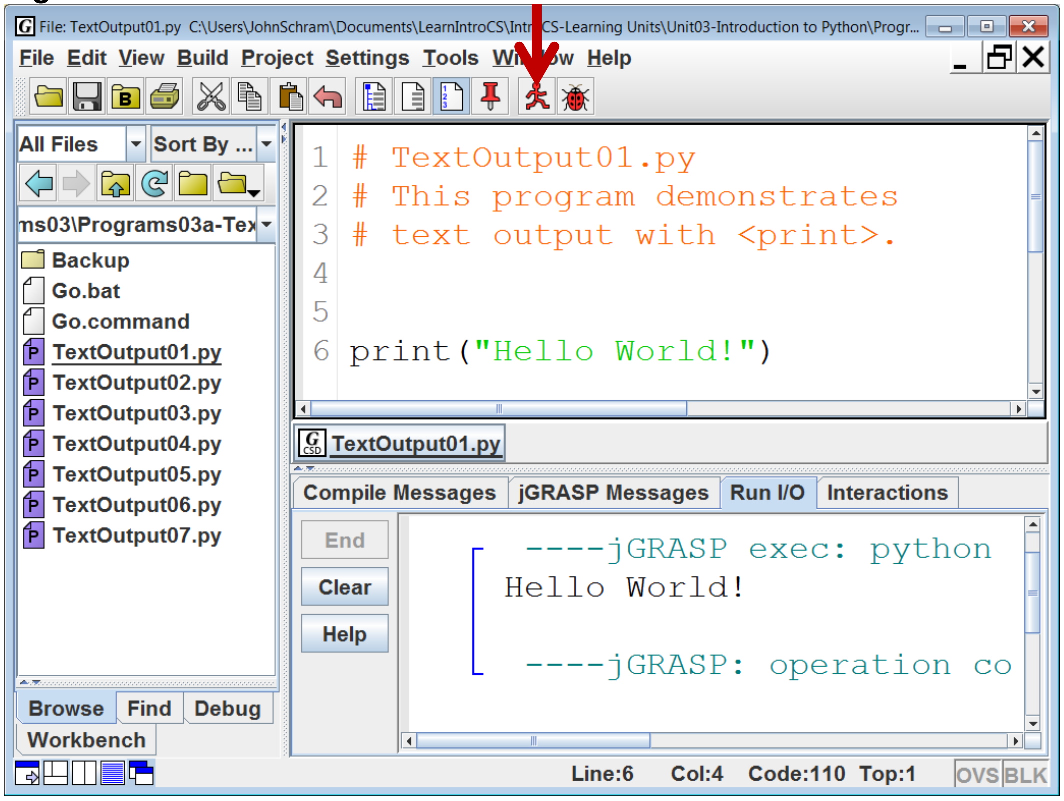

File: TextOutput01.py C:\Users\JohnSchram\Documents\LearnIntroCS\IntroCS-Learning Units\Unit03-Introduction to Python\Progr...
File Edit View Build Project Settings Tools Window Help
All Files
Sort By Name
ms03\Programs03a-TextOutput
Backup
Go.bat
Go.command
TextOutput01.py
TextOutput02.py

```
# TextOutput01.py
# This program demonstrates
# text output with <print>.


print("Hello World!")
```

Executing Python Programs in jGRASP

Another word for "executing" a program is "running" a program. The designers of **jGRASP** had this in mind. The icon to execute or "run" the program looks like a little red running man. Click it and the output will display in the *Output Window* as seen in Figure 3.12.

Figure 3.12



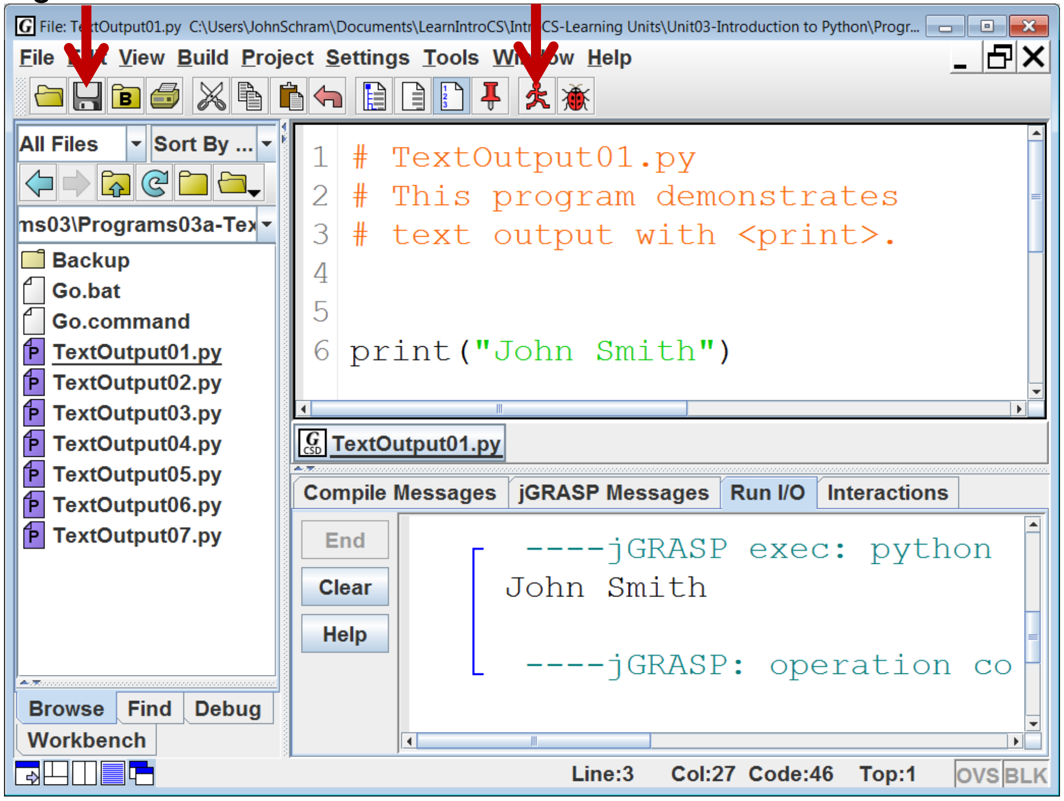
```
# TextOutput01.py
# This program demonstrates
# text output with <print>.


print("Hello World!")
```

```
 ----jGRASP exec: python
Hello World!

 ----jGRASP: operation co
```

Editing Python Programs in jGRASP

Suppose you need to edit your program. In **jGRASP**, you can take advantage of the *Screen Editor* to move about the entire program and make changes wherever you wish. Figure 3.13 shows program **TextOutput01.py** again. It still has 1 **print** command, but now it displays a name. Remember that anytime you change a program, you need to save it. To see the new output, you need to run the program again.

Figure 3.13

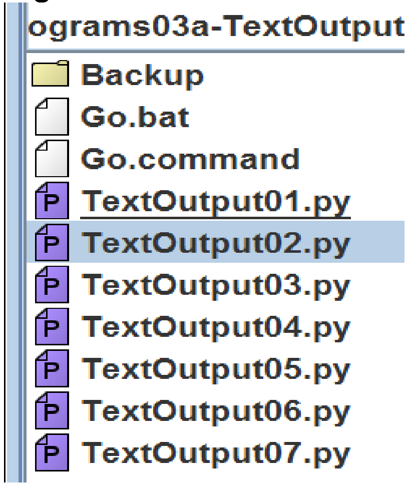


```
# TextOutput01.py
# This program demonstrates
# text output with <print>.


print("John Smith")
```

```
 ----jGRASP exec: python
John Smith

 ----jGRASP: operation co
```

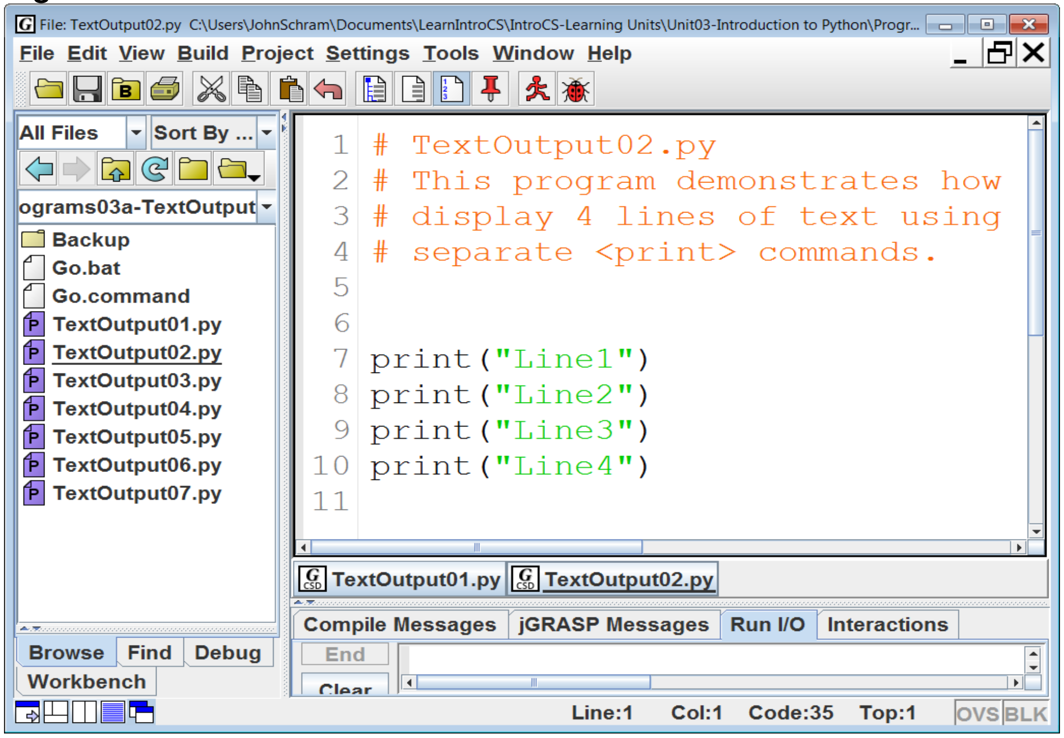
File: TextOutput01.py C:\Users\JohnSchram\Documents\LearnIntroCS\IntroCS-Learning Units\Unit03-Introduction to Python\Progr...
File Edit View Build Project Settings Tools Window Help
All Files
Sort By ...
ms03\Programs03a-Tex
Backup
Go.bat
Go.command
TextOutput01.py
TextOutput02.py
TextOutput03.py
TextOutput04.py
TextOutput05.py
TextOutput06.py
TextOutput07.py
Browse Find Debug
Workbench
TextOutput01.py
Compile Messages jGRASP Messages Run I/O Interactions
End
Clear
Help
Line:3 Col:27 Code:46 Top:1 OVS BLK

Figure 3.14



Back in Figure 3.10, I made a point to tell you to click the icon of the folder with the capital **B**. The benefit of this is now going to be demonstrated. Figure 3.14 shows a close-up view of what you should see in your **Browse** window. Suppose you want to look at program **TextOutput02.py** next. Just double-click the file in the **Browse** window and it will automatically be loaded in the **Edit** window as shown in Figure 3.15.

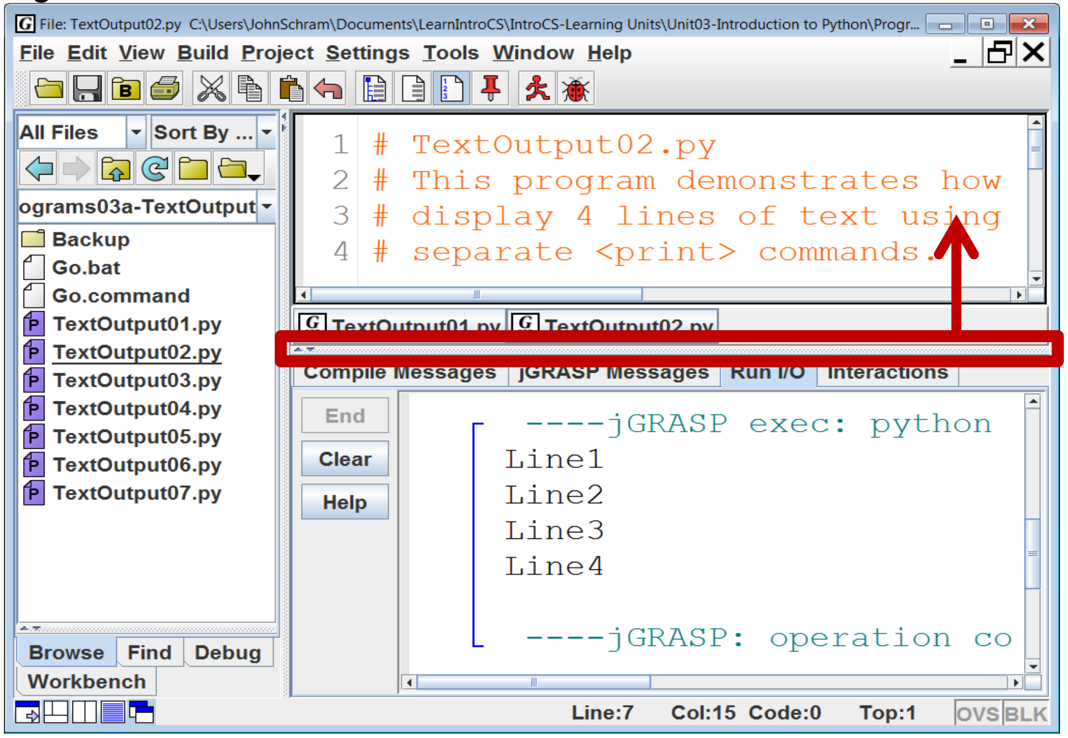
Figure 3.15



```
# TextOutput02.py
# This program demonstrates how
# display 4 lines of text using
# separate <print> commands.


print("Line1")
print("Line2")
print("Line3")
print("Line4")
```

When you execute the program, you may notice that there is not enough room in the *Output/Messages Window* to see the entire output. Moving the borders between the 3 windows will allow you to see the full output. See Figure 3.16.

Figure 3.16



```
# TextOutput02.py
# This program demonstrates how
# display 4 lines of text using
# separate <print> commands.
```

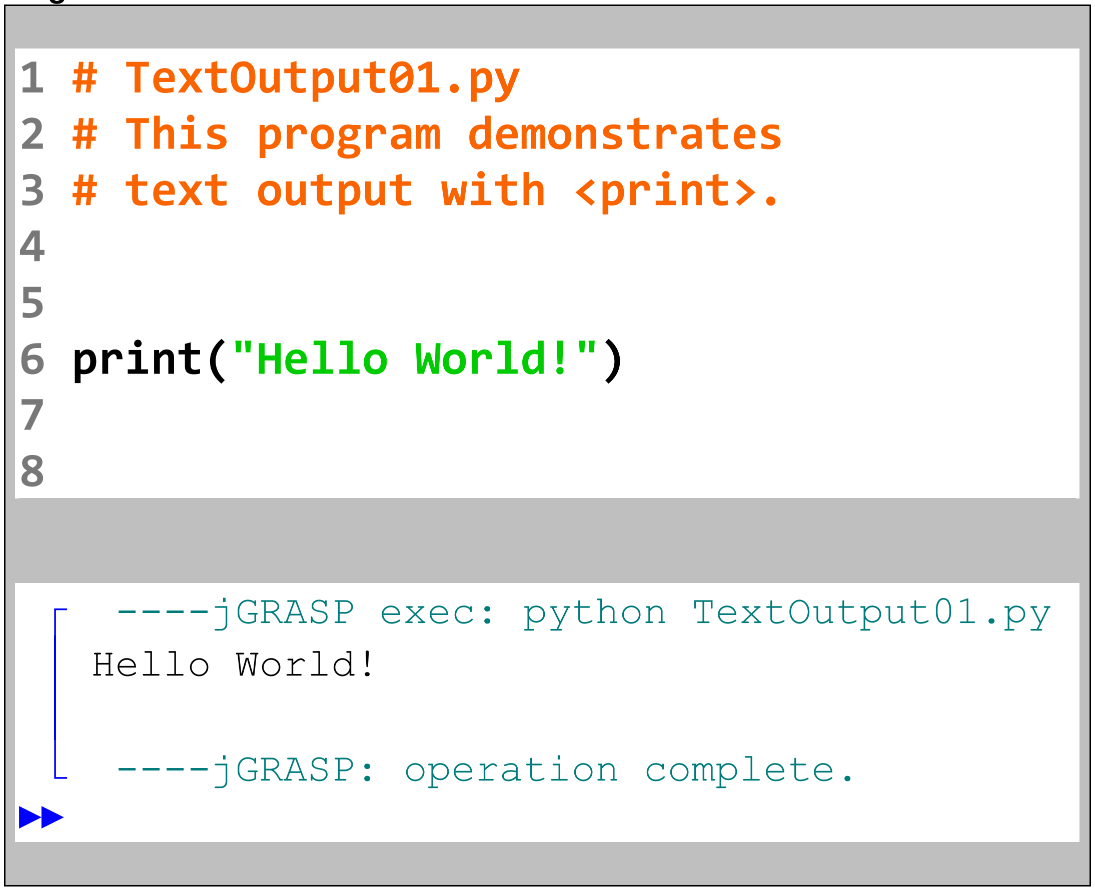
```
 ----jGRASP exec: python
Line1
Line2
Line3
Line4

 ----jGRASP: operation co
```

3.3 Text Output With print

Now that you have become proficient with loading, executing, editing and saving Python programs, we are going to look at how they work and begin the actual process of learning Python programming. The examples in this textbook were created with **jGRASP**. You may be using a different IDE. While the colors scheme may be different on your IDE, the code, the output and the logic of each program example will still be identical.

Now let us actually take a look at program **TextOutput01.py**, shown in Figure 3.17. The first 3 lines of the program, the ones that begin with hashtags (#), are *comments*. Comments do not actually do anything in a program. They do have a purpose and we will take more about them in a later section. For now, we are going to focus on the **print** command on line #6. In this command we are telling the computer to **print** the text **"Hello World"**, which is exactly what it does. See the output, which is shown right after the program.

Figure 3.17



```
1 # TextOutput01.py
2 # This program demonstrates
3 # text output with <print>.
4
5
6 print("Hello World!")
7
8
```

```
 ----jGRASP exec: python TextOutput01.py
Hello World!

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

In addition to **“Hello World”**, the output also shows the text **“jGRASP exec: python TextOutput01.py”** and **“jGRASP: operation complete.**” This addition text is something **jGRASP** adds to mark the beginning and ending of the program’s execution. It is not actually part of the program’s output. The only *output* for this program are the words, **“Hello World”**.

A computer cannot be programming using a human language like English. Human language is idiomatic, ambiguous, and the meanings of words change. When I first learned English, being “woke” is what happened when your parents thought you were sleeping too much and to “ship a couple” meant you were mailing 2 things. A program written with those words fifty years ago would not do very well today.

Writing programs has close similarities to writing an essay. There is an important creative part that revolves around telling a story, debating a point or explaining a concept. An essay is made up of individual sentences that follow proper sentence structure, and each sentence is made up of individual words. It is not possible to create an effective sentence without knowing the meaning of the words that make up the sentence. Likewise in a program language you need to know the meaning of the special words that perform some specific purpose. Right now we shall call all the special words used by Python, *keywords*.

Python Keywords and Program Statements

A Python *keyword* is a word that has a special meaning in a program or performs a special function.

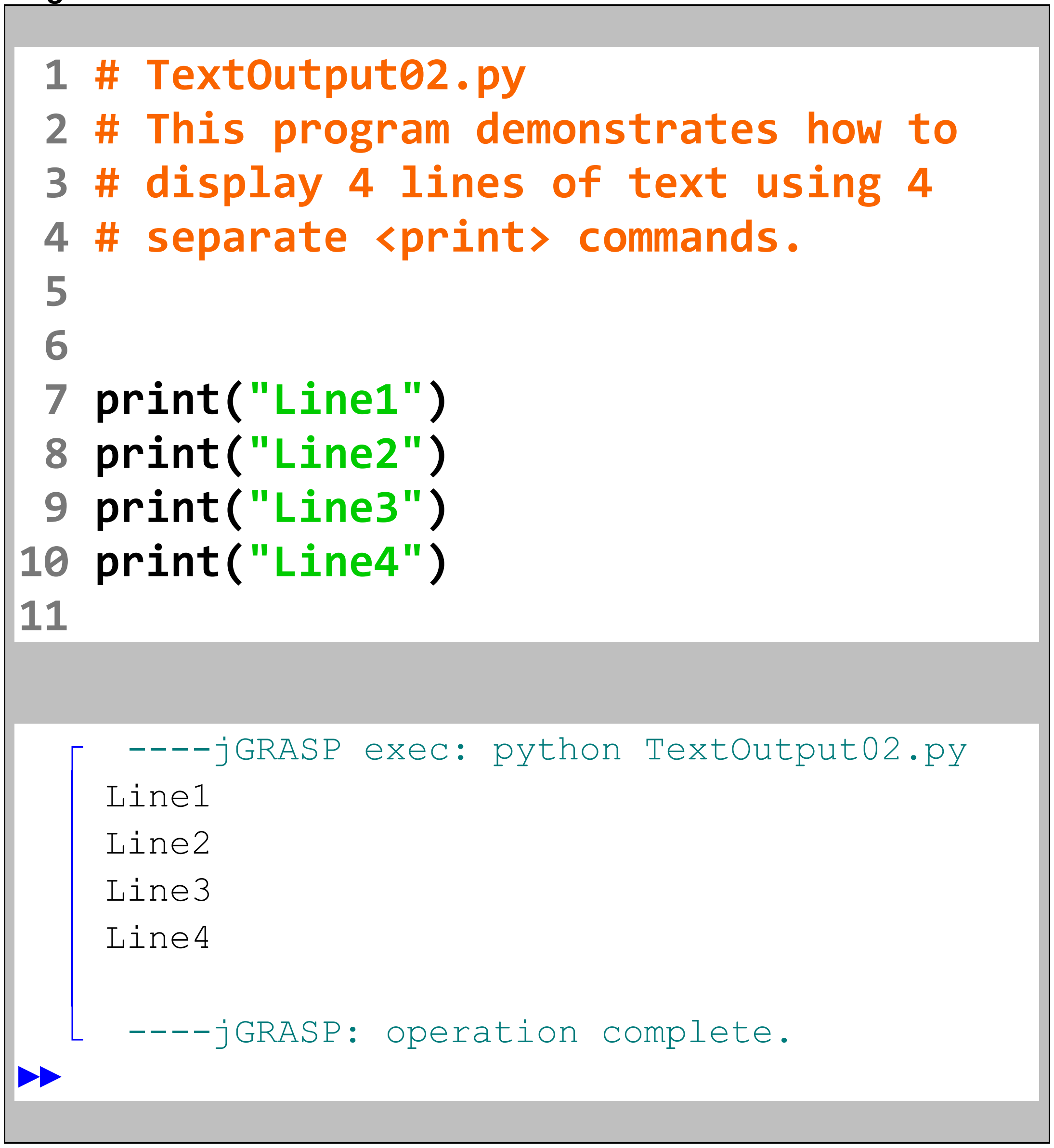
A *program statement* usually contains one or more *keywords*.

Keywords in Python are case-sensitive.
For example:
print is a Python keyword, while **PRINT** is not.

So far, the only *keyword* that you have learned is **print**, and you have only really seen one example. You will learn <u>many</u> keywords over the course of this class. Think of it this way. When you learn more English words, it is said you are “improving your vocabulary” and the result is that you are better able to communicate. When you learn more Python keywords, you are better able to program in Python. OK, back to **print**. You may be under the impression that **print** simply “prints” whatever you put inside the quotes, but there is more to it than that, and the next few program examples will demonstrates.

Program **TextOutput02.py**, shown in Figure 3.18, has 4 **print** commands. The program output shows that each **print** statement generates output on its own line. You might mistakenly think that this is caused by the fact that there are four **print** statements on four separate lines. Such an assumption is logical, but incorrect. You may not realize it, but the **print** command actually performs two operations. The first is to display what is in the quotes. The second is to generate a ***c**arriage-**r**eturn/**l**ine-**f**eed* (crlf). This is a fancy expression that means "the next output will be displayed at the beginning of the next line".

Figure 3.18



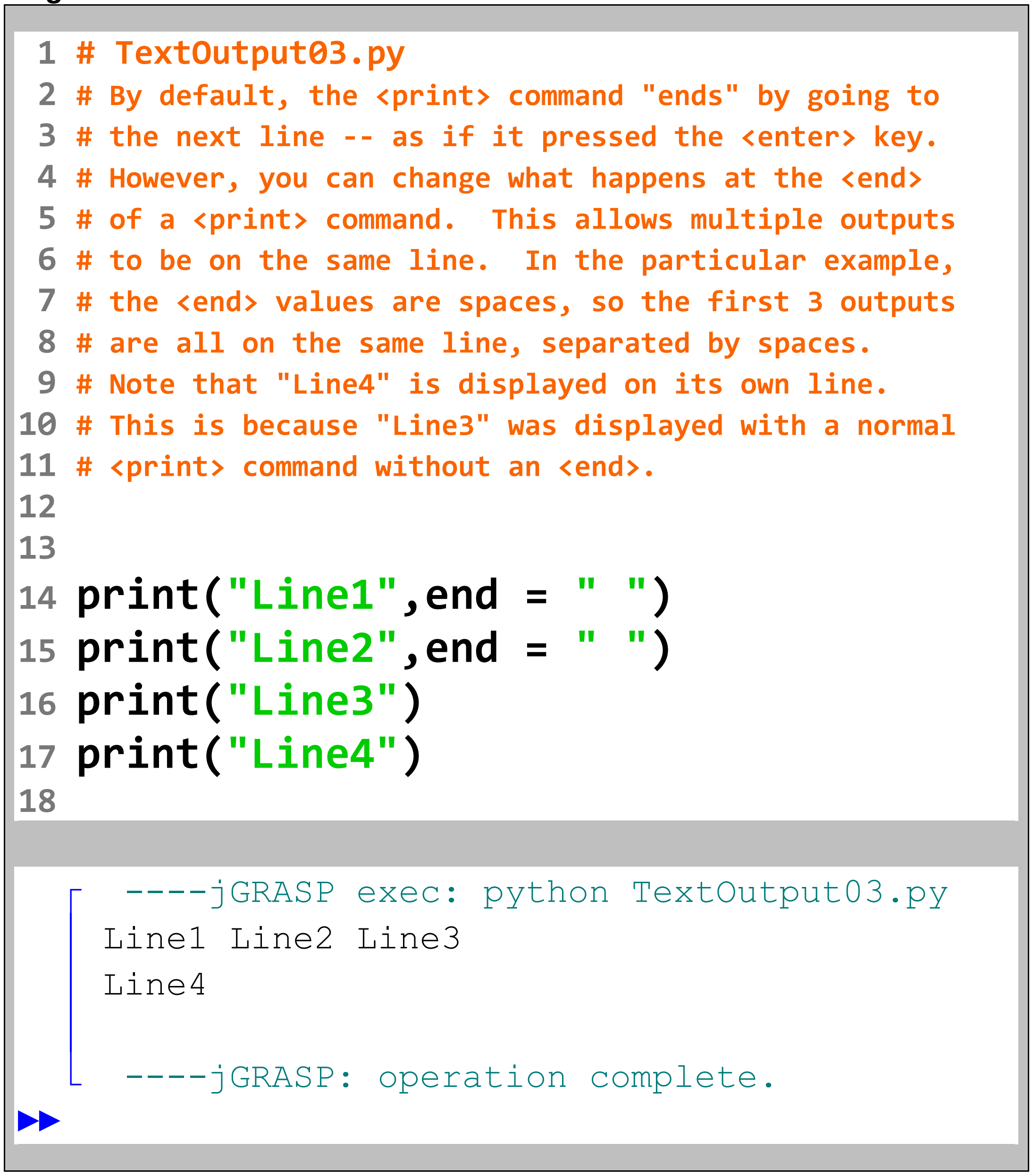
```
# TextOutput02.py
# This program demonstrates how to
# display 4 lines of text using 4
# separate <print> commands.


print("Line1")
print("Line2")
print("Line3")
print("Line4")

```

```
 ----jGRASP exec: python TextOutput02.py
Line1
Line2
Line3
Line4

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

I already mentioned that the **print** command performs two operations, the second being to generate a crlf. However, it does not actually have to be that way. You can change what the **print** command does at the end by adding the **end** keyword. This is demonstrated with program **TextOutput03.py**, shown in Figure 3.19. As with the previous program, there are four "lines" of output.

The first two **print** statements finish with **end = " "**. This means, instead of generating a crlf, this **print** command will *end* with a space. The result of this is that **"Line1"**, **"Line2"**, and **"Line3"** are all displayed on the same line, separated with spaces. Note that the third **print** command does not have an **end** keyword, so it generates a crlf. This is why **"Line4"** is on the next line.

Figure 3.19

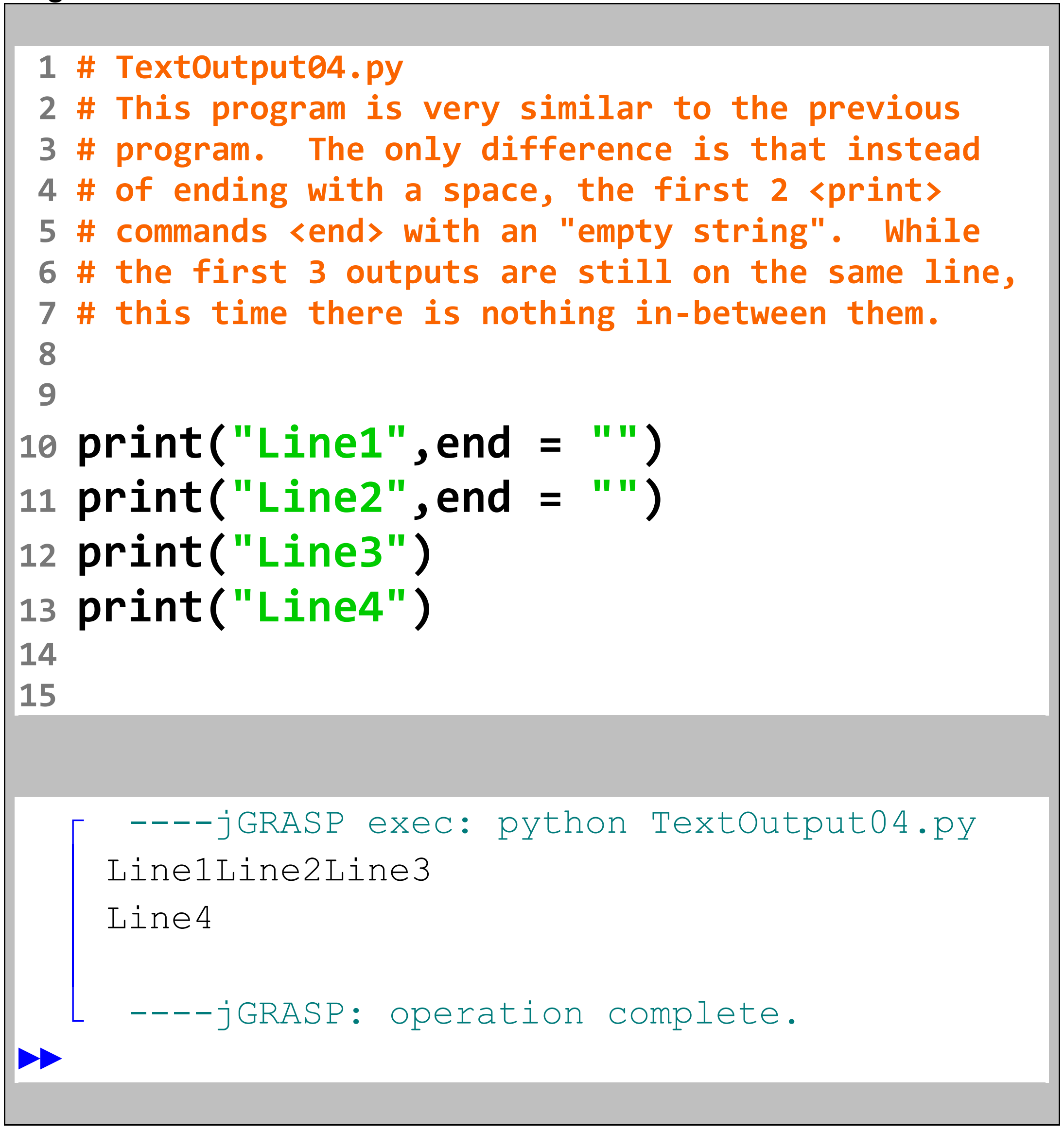


```
# TextOutput03.py
# By default, the <print> command "ends" by going to
# the next line -- as if it pressed the <enter> key.
# However, you can change what happens at the <end>
# of a <print> command.  This allows multiple outputs
# to be on the same line.  In the particular example,
# the <end> values are spaces, so the first 3 outputs
# are all on the same line, separated by spaces.
# Note that "Line4" is displayed on its own line.
# This is because "Line3" was displayed with a normal
# <print> command without an <end>.


print("Line1",end = " ")
print("Line2",end = " ")
print("Line3")
print("Line4")

```

```
 ----jGRASP exec: python TextOutput03.py
Line1 Line2 Line3
Line4

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

What if you want to display multiple things on the same line, but you do not want anything separating them? All you need to do is remove the space between the quotes in the **end** keyword. This is demonstrated with program **TextOutput04.py**, shown in Figure 3.20. By using **end** = "" instead of **end** = " " all of the outputs that are on the same line (**"Line1"**, **"Line2"**, and **"Line3"**) will have nothing displayed in-between them.

Figure 3.20



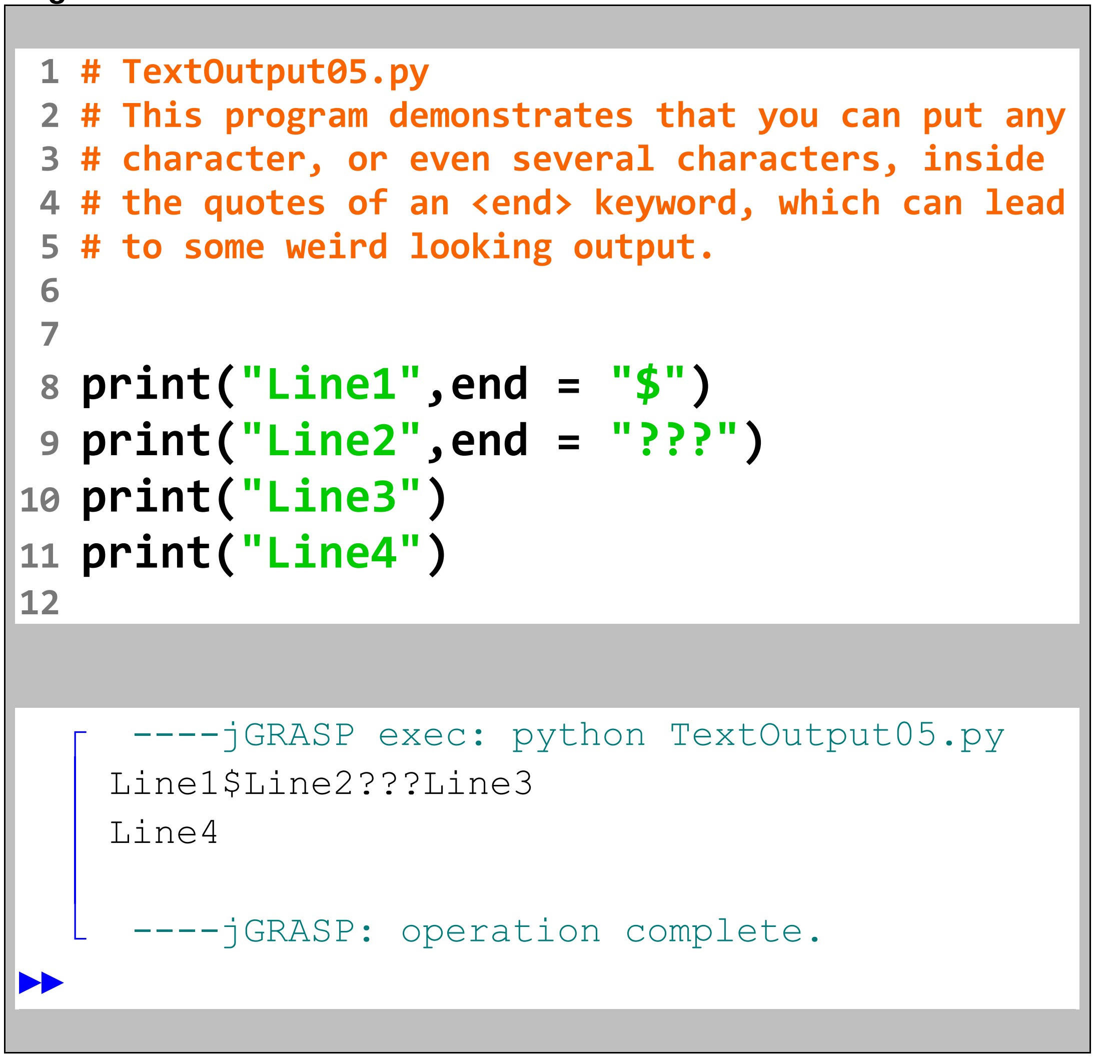
```
# TextOutput04.py
# This program is very similar to the previous
# program.  The only difference is that instead
# of ending with a space, the first 2 <print>
# commands <end> with an "empty string".  While
# the first 3 outputs are still on the same line,
# this time there is nothing in-between them.


print("Line1",end = "")
print("Line2",end = "")
print("Line3")
print("Line4")


```

```
 ----jGRASP exec: python TextOutput04.py
Line1Line2Line3
Line4

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

NOTE: The term *empty string* is used to describe a string that contains absolutely no text. This is generated when you put nothing between the quotes, as in "".

Technically, you can put anything between the quote of the **end** keyword. It does not even have to be just one single character. It can be several. Program **TextOutput05.py**, shown in Figure 3.21, uses a dollar sign (**$**) in the **end** of the first **print** command and three question marks (**???**) in the **end** of the second. The resulting output may look a little weird, but hopefully it makes sense.

Figure 3.21

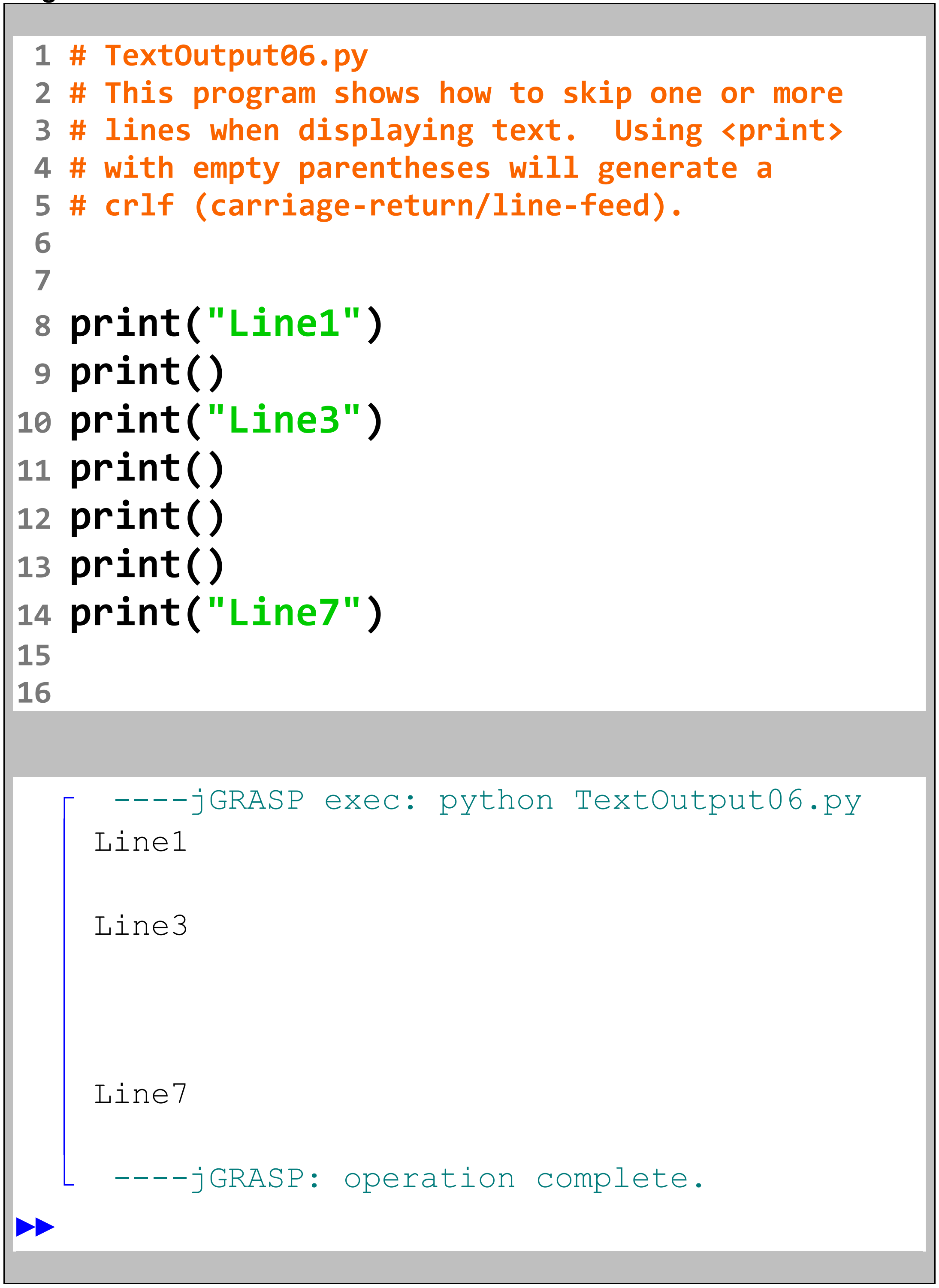


```
# TextOutput05.py
# This program demonstrates that you can put any
# character, or even several characters, inside
# the quotes of an <end> keyword, which can lead
# to some weird looking output.


print("Line1",end = "$")
print("Line2",end = "???")
print("Line3")
print("Line4")

```

```
 ----jGRASP exec: python TextOutput05.py
Line1$Line2???Line3
Line4

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

OK, you have seen several programs with **print** commands, and each and every one of these **print** commands had quotes with some text inside the quotes. You probably noticed in each program's output that the actual text that is displayed is the same text inside the quotes from the **print** statements. Based on these examples, you may conclude that this quoted text is necessary for **print**. This is not always a requirement. Program example **TextOutput06.py**, in Figure 3.22, includes several **print** statements with empty parentheses. Based on the output, it appears these "empty" **print** statements still have some output. They actually generates a *crlf*, resulting is a “skipped line”.

Figure 3.22



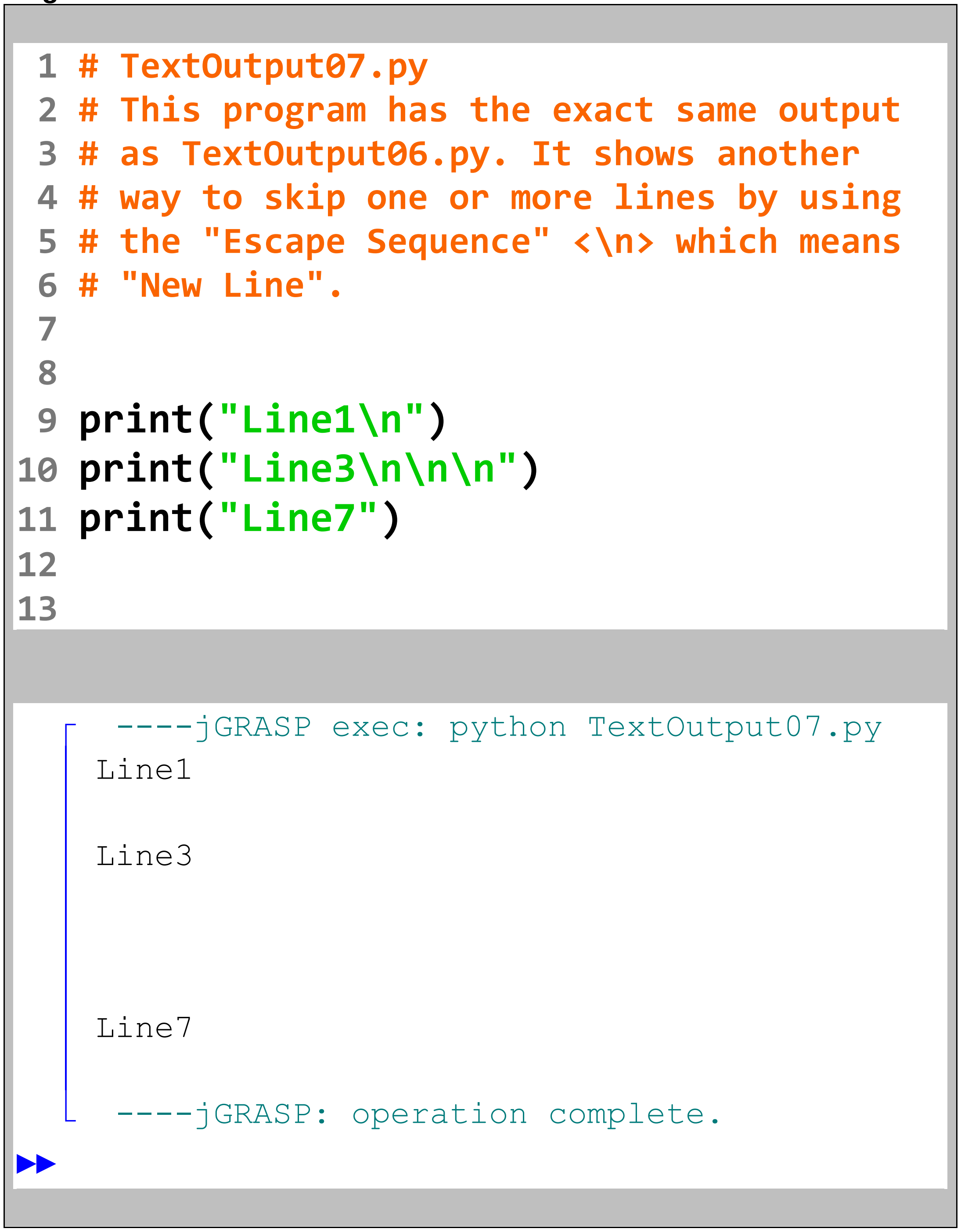
```
# TextOutput06.py
# This program shows how to skip one or more
# lines when displaying text.  Using <print>
# with empty parentheses will generate a
# crlf (carriage-return/line-feed).


print("Line1")
print()
print("Line3")
print()
print()
print()
print("Line7")


```

```
 ----jGRASP exec: python TextOutput06.py
Line1

Line3



Line7

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

Note that there is one line skipped between **"Line1"** and **"Line3"**. Note also that there are 3 lines skipped between **"Line3"** and **"Line7"**. This is because each individual empty **print** command generates a separate *crlf*.

Program example **TextOutput07.py**, in Figure 3.23, has the exact same output as the previous program, but demonstrates a shorter way to skip multiple lines in your program output. Instead of adding extra **print** statements, you can add one or more **\n** *Escape Sequence* to your existing **print** statements.

Figure 3.23



```python
# TextOutput07.py
# This program has the exact same output
# as TextOutput06.py. It shows another
# way to skip one or more lines by using
# the "Escape Sequence" <\n> which means
# "New Line".


print("Line1\n")
print("Line3\n\n\n")
print("Line7")


```

```
 ----jGRASP exec: python TextOutput07.py
Line1

Line3



Line7

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

NOTE: **\n** means "New Line".

print & end

The **print** command generates an output display of the characters contained between double quotes.

If the **end** keyword is not used, **print** will also generate a *carriage-return/line-feed* (crlf).

If the **end** keyword is used, **print** will not generate a crlf and instead display the specified text in the second set of quotes.

Examples:

`print("Hello")`	**`print("Hello", end = " ")`**
`print("World")`	**`print("World")`**
will display:	will display:
`Hello`	`Hello World`
`World`	

Creating a blank line of output can be done 2 different ways. One is to use **print()** with absolutely nothing in the parentheses. The other is to add the **\n** *Escape Sequence* to an existing **print** statement:

Examples:

`print("Hello")`	**`print("Hello\n")`**
`print()`	**`print("World")`**
`print("World")`	
will display:	will also display:
`Hello`	`Hello`
`World`	`World`

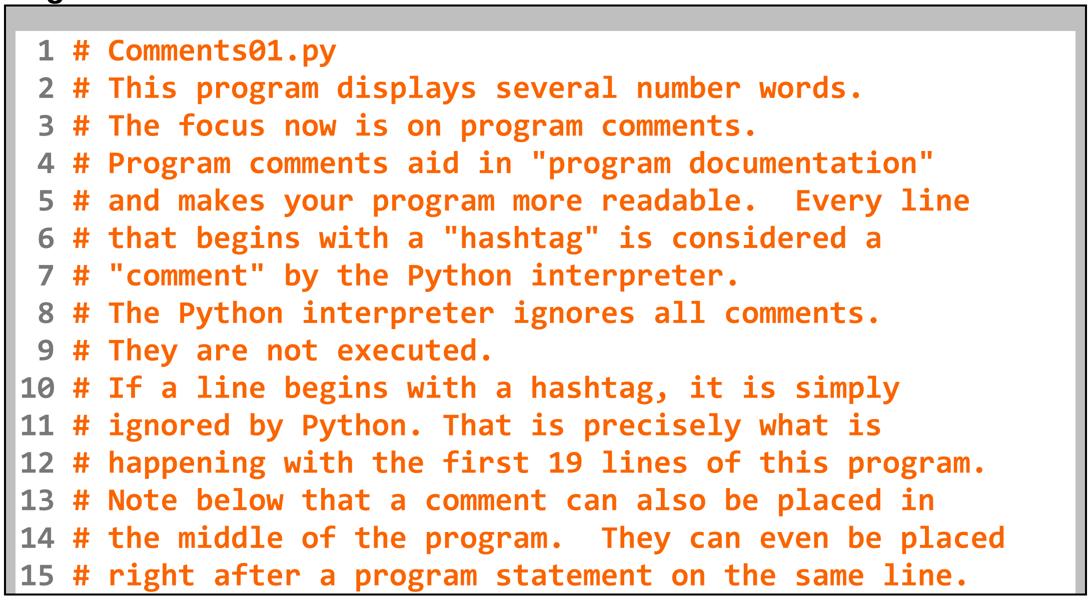
3.4 Program Comments

All of the programs that we have looked at have begun in the same way. They all begin with a few lines of text that briefly explain the program. In the first program example, it was mentioned that these lines of text were called *comments*. What exactly are *comments*? What purpose do they serve? Well for one thing, they are totally ignored by the computer. They are not translated into machine code. They are not executed. So what good are they? Comments are very useful because they aid in *Program Documentation*. A well-documented program is easier to read, debug, update and enhance. The proper use of comments is actually only the first part program documentation. You will learn more about this in later chapters.

Many programming languages have 2 kinds of comments. The first is what we have been using all along. These are *Single-Line Comments*. In Python, when you begin a line with a *hashtag* (**#**) the entire line is ignored by the computer. This is demonstrated in program **Comments01.py** in Figure 3.24. As with the earlier programs, there are several single-line comments at the beginning of the program. This is not the only place comments can be used. They can be used anywhere in the program. You should see that a comment can be placed right after a program statement. At the end of the program you see that an entire program statement can be *commented-out*. This is useful if a program statement has an error that you cannot fix right now. The statement can be commented-out and you can check the rest of the program.

NOTE: In **jGRASP**, comments are displayed in **red**.

Figure 3.24



```
# Comments01.py
# This program displays several number words.
# The focus now is on program comments.
# Program comments aid in "program documentation"
# and makes your program more readable.  Every line
# that begins with a "hashtag" is considered a
# "comment" by the Python interpreter.
# The Python interpreter ignores all comments.
# They are not executed.
# If a line begins with a hashtag, it is simply
# ignored by Python. That is precisely what is
# happening with the first 19 lines of this program.
# Note below that a comment can also be placed in
# the middle of the program.  They can even be placed
# right after a program statement on the same line.
```

```python
# You will also see that the word "Thirteen" is not
# displayed because it has been "commented-out"
# possibly by someone suffering from "Triskaidekaphobia"
# (the fear of the number 13).


print()
print("One")
print("Two")
print("Three")
print("Four")
print("Five")
print("Six")         # half a dozen
print("Seven")
print("Eight")
print("Nine")
print("Ten")
print("Eleven")
print("Twelve")      # one dozen
#print("Thirteen") # one baker's dozen

```

```
 ----jGRASP exec: python Comments01.py


One
Two
Three
Four
Five
Six
Seven
Eight
Nine
Ten
Eleven
Twelve


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

The second kind of comment is the *Multi-Line Comment*. There are times that you may need to temporarily “comment out” a large section of code, and you do not wish to go through tedious process of placing a hashtag at the beginning of every single line, only to have to remove them later. It would be simpler if you could just mark the beginning and ending of this multi-line comment. You can accomplish this in Python by using “Triple-Double-Quotes” at both end of the comment. This is demonstrated in program **Comments02.py**, shown in Figure 3.25.

Figure 3.25

```python
# Comments02.py
# This program demonstrates that a section of
# code can essentially be "commented-out" by
# using triple-double-quotes.


print()
print("One")
print("Two")
"""
print("Three")
print("Four")
print("Five")
print("Six")        # half a dozen
print("Seven")
print("Eight")
print("Nine")
print("Ten")
print("Eleven")
"""
print("Twelve")     # one dozen
print("Thirteen")   # one baker's dozen

```

```
 ----jGRASP exec: python Comments02.py

One
Two
Twelve
Thirteen

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

NOTE: Be careful not to forget to *close* your multi-line comment.
If you do, the remainder of your program becomes one big comment.

This may seem like I am changing the subject, but the term “String Literal” is about to be used multiple times in this chapter, so I am going to go ahead and introduce it now.

String Literal Definition

A *string literal* is any text in-between a set of quotes.

The “text” can be a name, letter, group of words, or basically anything that you can type inside a set of quotes.

Examples:
"computer"
"John Smith"
"Hello all you happy people."
"Q"
"?"
"811 Fleming Trail"
"Richardson, Texas"
"75081"

Most **print** statements contain a *string literal* in their parentheses.

Python Multi-Line Comment Disclaimer

Technically, Python does not have *Multi-Line Comments*.

The “Triple-Double-Quote” is actually used to create a multi-line *string literal*.

This is why they are **green** and not **red** like the *Single-Line Comments*.

However, multi-line string literals which are not part of a **print** statement are simply ignored by the Python interpreter.

This essentially makes them multi-line comments.

Program **Comments03.py** in Figure 3.26 shows another use for comments. They can also be used to create a *heading* in your program. Right now we tend to grade student program on the monitor, but in the past when we required students to print their program, the students would need to include a heading using this technique.

Figure 3.26

```python
#################################
#                               #
#         Comments03.py         #
#       Numbers from 1-13       #
#        By: John Schram        #
#            11/8/17            #
#                               #
#  This program is similar to   #
#  the previous two and shows   #
#  that a comment can be used   #
#  to create a heading.         #
#                               #
#################################


print()
print("One")
print("Two")
print("Three")
print("Four")
print("Five")
print("Six")       # half a dozen
print("Seven")
print("Eight")
print("Nine")
print("Ten")
print("Eleven")
print("Twelve")    # one dozen
print("Thirteen")  # one baker's dozen

```

Note that the comment box is a proper rectangle. This works with fonts like **`Courier`** or `Consolas`. A proportional font, like **Arial**, will not line up quite so neatly.

3.5 Syntax Errors

One of the most difficult aspects of computer programming for beginning students is handling error messages. You will see seven program examples to give you an idea how the Python interpreter responds when a program has some type of mistake.

While there are some things about which Python is very “relaxed”, there are others where Python is very “uptight”. For example, Python is very relaxed about *vertical spacing*. You may have notice that I skip a couple lines between the comments and the program code in my examples. When the programs get bigger, I will also skip lines between different parts of the program. This is fine, but when it comes to *horizontal spacing* Python becomes very uptight. Program **SyntaxError01.py**, shown in Figure 3.27, demonstrates what happens when I indent one of my **print** statement one single space. This tiny, subtle little error prevents the entire program from executing. Now, there are times when program statements are required to be indented, but we will not get to that until Chapter VII. For now, all program statements need to line up properly on the left side with absolutely no indenting.

Figure 3.27

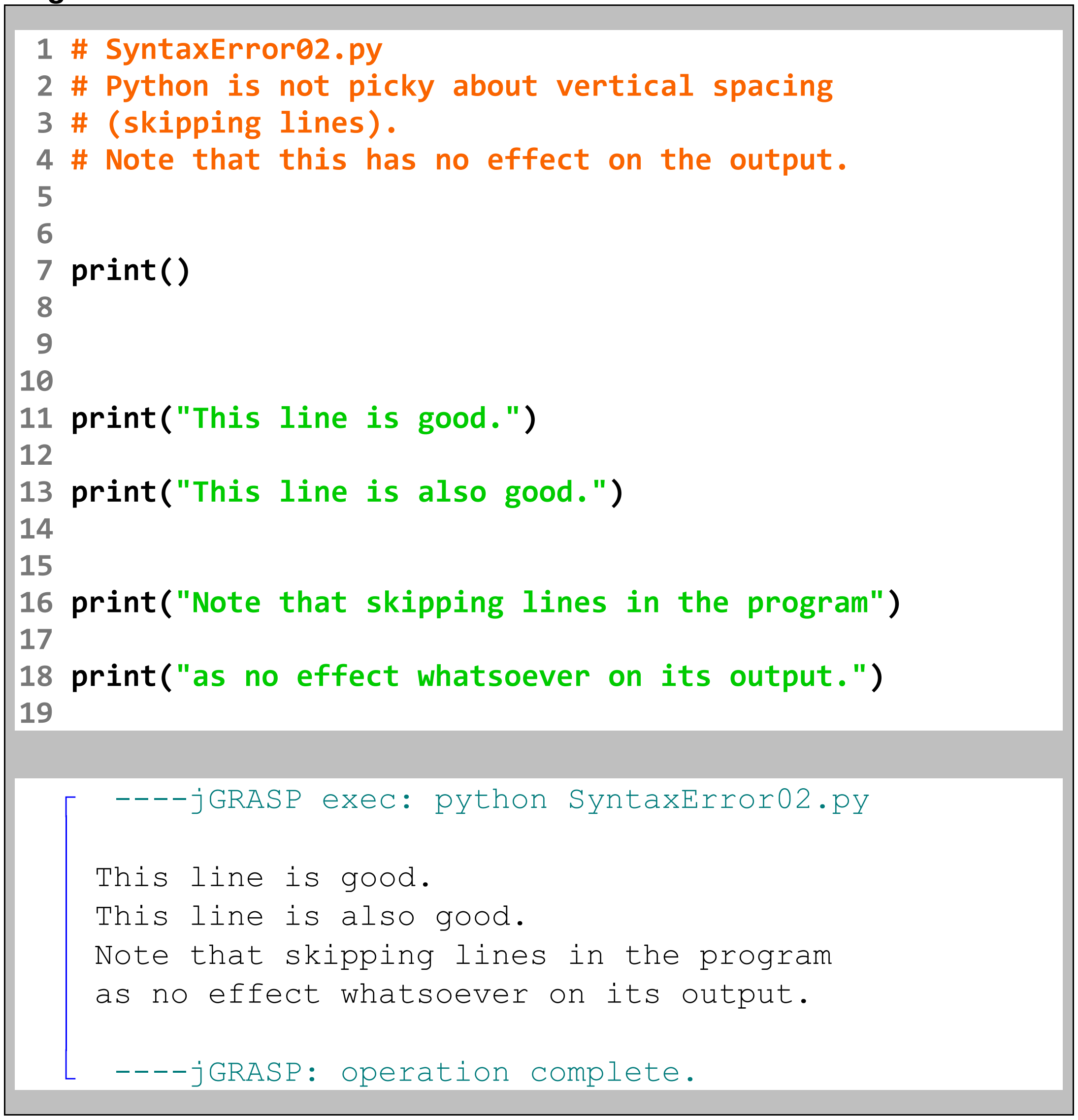
```
 1 # SyntaxError01.py
 2 # Python is very picky about horizontal spacing
 3 # (indenting). For now, all Python command need
 4 # to start on the left side of the screen with
 5 # no indenting at all.
 6
 7
 8 print("This line is good.")
 9  print("This line is bad.")
10
```

```
 ----jGRASP exec: python SyntaxError01.py
  File "SyntaxError01.py", line 9
    print("This line is bad.")
    ^
IndentationError: unexpected indent

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

Note that the error message indicates that the error occurred on "**line 9**". This is another reason why having the line numbers *turned on* is a good thing. Finding errors like this is made much easier with visible line numbers, especially when your programs become large.

Program **SyntaxError02.py**, shown in Figure 3.28, reaffirms what I said earlier about Python not having a problem with *vertical spacing*. The program contains five **print** commands, which are separated with one, two or three blank lines. The output shows two things. First, these blank lines do not prevent the program from executing. Second, these blank lines have absolutely no effect on the program's output. Note how the extra vertical spacing in the program code does not cause extra vertical spacing in the output. If we wanted that, we would need to add some **print( )** commands or add some **\n** Escape Sequences to the existing **print** commands.

Figure 3.28

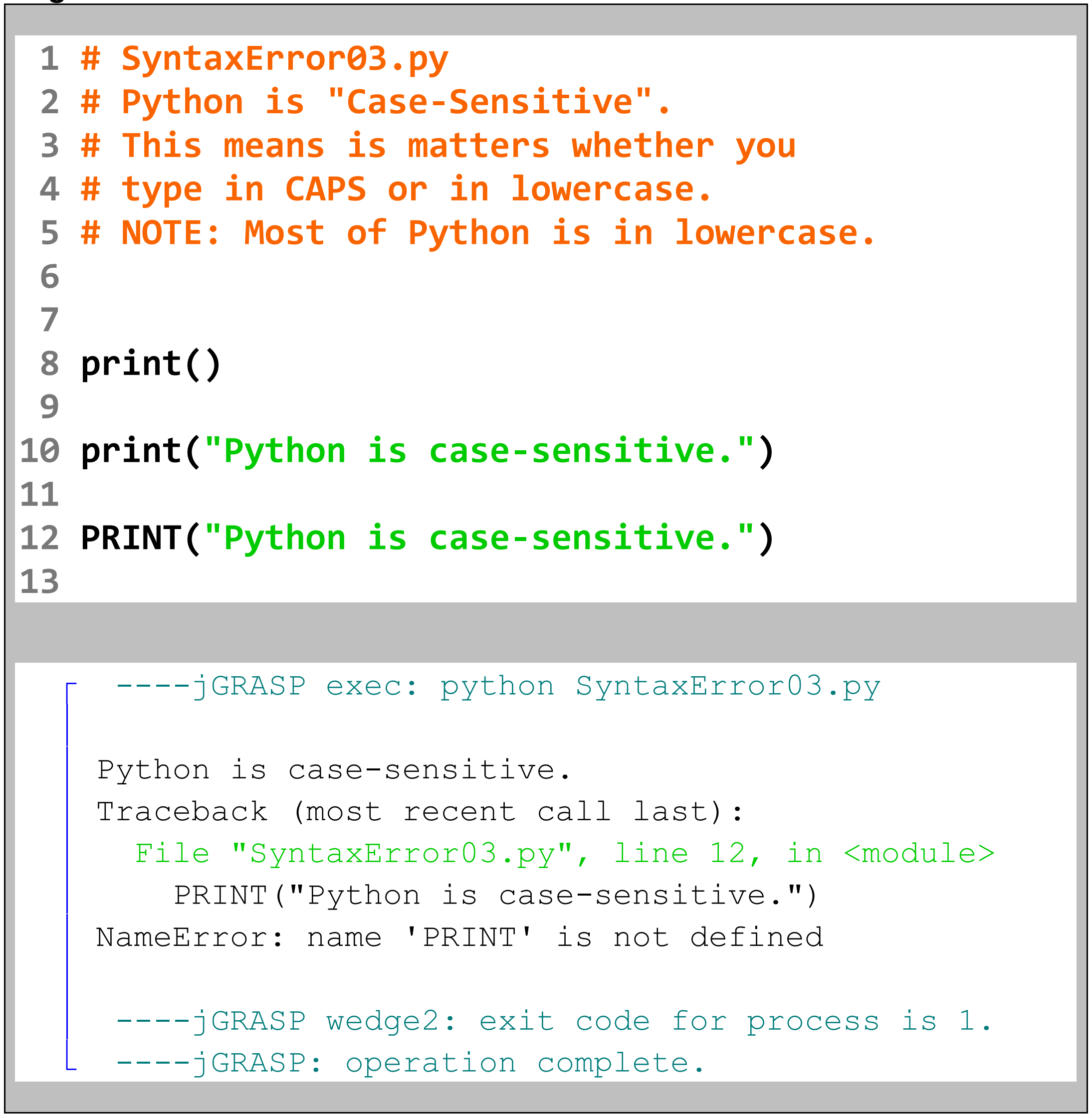


```python
# SyntaxError02.py
# Python is not picky about vertical spacing
# (skipping lines).
# Note that this has no effect on the output.


print()



print("This line is good.")

print("This line is also good.")


print("Note that skipping lines in the program")

print("as no effect whatsoever on its output.")

```

```
 ----jGRASP exec: python SyntaxError02.py

This line is good.
This line is also good.
Note that skipping lines in the program
as no effect whatsoever on its output.

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

Another way Python is picky is the fact that it is *Case Sensitive*. What is “case sensitivity?” It means it matters whether you type with CAPITAL or lowercase letters. Most passwords are “case sensitive”. If your password is “**Hello123**” and you type either “**hello123**” or “**HELLO123**” it will not be accepted. The same is true for Python, as is demonstrated by program **SyntaxError03.py**, shown in Figure 3.29. In this program, the third **print** command is in all CAPS. While we can still read it, Python cannot and gives an error message stating that "**name 'PRINT' is not defined**".

Figure 3.29



```
 1 # SyntaxError03.py
 2 # Python is "Case-Sensitive".
 3 # This means is matters whether you
 4 # type in CAPS or in lowercase.
 5 # NOTE: Most of Python is in lowercase.
 6
 7
 8 print()
 9
10 print("Python is case-sensitive.")
11
12 PRINT("Python is case-sensitive.")
13
```

```
 ----jGRASP exec: python SyntaxError03.py

Python is case-sensitive.
Traceback (most recent call last):
  File "SyntaxError03.py", line 12, in <module>
    PRINT("Python is case-sensitive.")
NameError: name 'PRINT' is not defined

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

If you look closely at the output, you should notice that the program partially executed. It did display the output of the first two **print** commands (the first one just skipped a line). The execution halted when it reached the **NameError** on **line 12**.

Program **SyntaxError04.py**, shown in Figure 3.30, demonstrates one of the most common syntax errors, a simple *typo*. In this program, the third **print** statement is misspelled. As with the previous program, while we can understand what was meant, the computer cannot and we get the same **NameError** message displayed. Also, just like the previous program, there is a partial execution as the first two **print** commands do execute. It is not until the third, misspelled **prnt** command that the execution is halted.

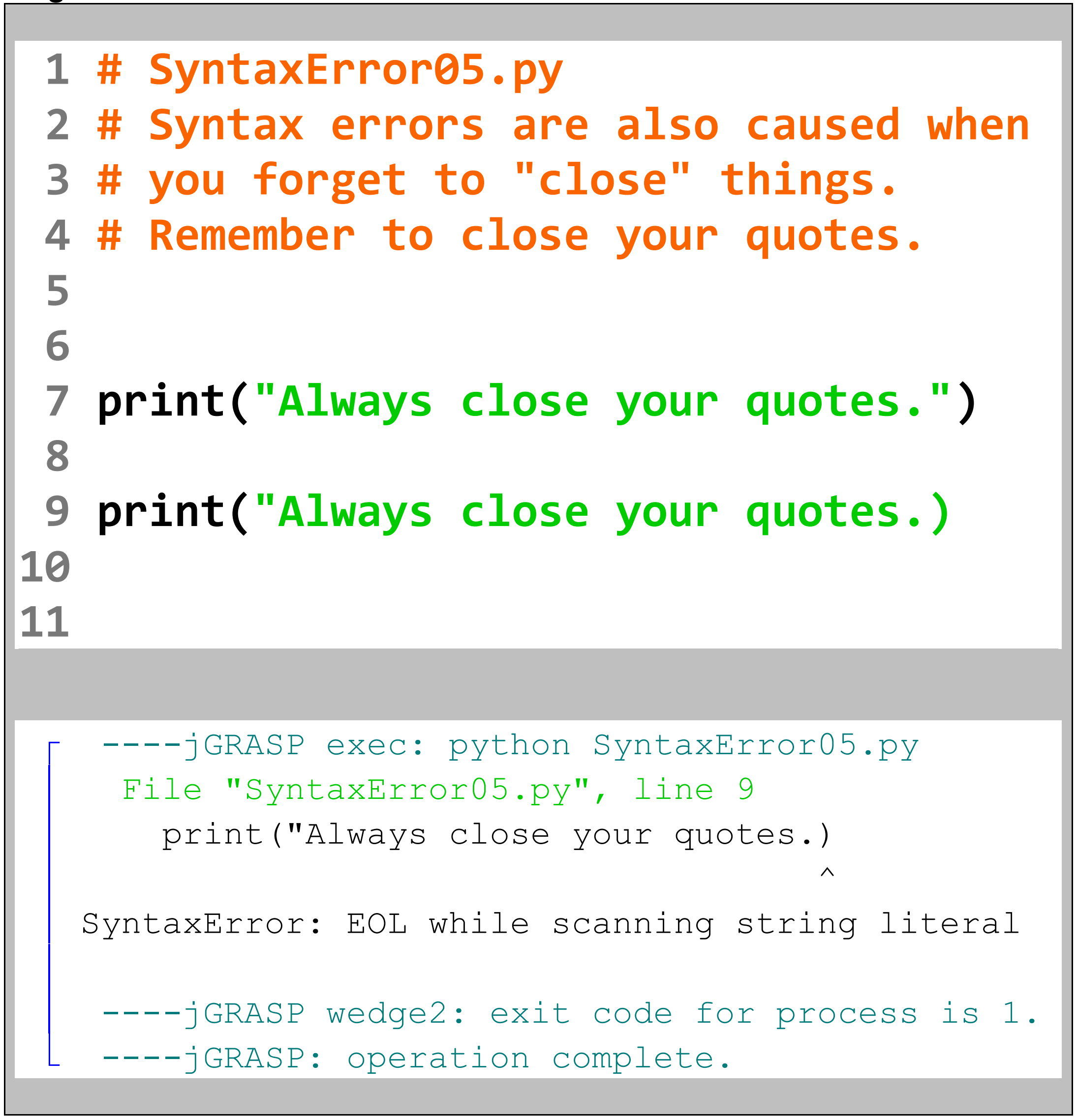
Figure 3.30

```python
# SyntaxError04.py
# Typos, like misspelling commands, is one
# of the main sources of Syntax Errors.


print()

print("Type your commands carefully.")

prnt("Type your commands carefully.")


```

```
 ----jGRASP exec: python SyntaxError04.py

Type your commands carefully.
Traceback (most recent call last):
  File "SyntaxError04.py", line 10, in <module>
    prnt("Type your commands carefully.")
NameError: name 'prnt' is not defined

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

Program **SyntaxError05.py**, shown in Figure 3.31, demonstrates what happens when you forget the closing quote in a **print** statement. The error message says "**EOL while scanning string literal**". We already know that a *string literal* is some text inside quotes, but the closing quote is missing. Let us look at the very first word, "EOL". What is that? EOL stands for **E**nd **O**f **L**ine. The error message is basically saying that while the computer was reading the string literal, the line suddenly ended, and this happened before it detected the end of the string literal.

Figure 3.31



```
# SyntaxError05.py
# Syntax errors are also caused when
# you forget to "close" things.
# Remember to close your quotes.


print("Always close your quotes.")

print("Always close your quotes.)


```

```
 ----jGRASP exec: python SyntaxError05.py
  File "SyntaxError05.py", line 9
    print("Always close your quotes.)
                                    ^
SyntaxError: EOL while scanning string literal

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

When programming, string literals are not the only things that need to be "closed". In the realm of computer science, there is a certain rule that I would almost call a "commandment"

If you open it, you must close it.

Program **SyntaxError06.py**, shown in Figure 3.32, demonstrates what happens when you forget the closing parenthesis in a **print** statement. You get a simple error message which states "**invalid syntax**", but something it weird. It is line 10 that is missing a parenthesis, but the error is reported to have occurred on **line 12**. What is going on here? Well, here is what is happening. Without the closing parenthesis, the **print** command on Line 10 never properly finished. This is not detected until we attempt to start a new **print** command in Line 12. Now the computer is like, "Whoa, hold on there. You cannot start a new **print** command here. You never finished the previous one!" As a result, this program also demonstrates something very important. Error messages do not always accurately indicate the location of the actual errors in a program.

Figure 3.32

```
 1 # SyntaxError06.py
 2 # Remember to close parentheses too.
 3 # The program also shows that the error message
 4 # does not always accurately indicate the location
 5 # of the error. In this case, the line 10 was not
 6 # finished properly. As a result, line 12 is not
 7 # able to begin properly.
 8
 9
10 print("Close your parentheses too."
11
12 print("Close your parentheses too.")
13
```

```
 ----jGRASP exec: python SyntaxError06.py
  File "SyntaxError06.py", line 12
    print("Close your parentheses too.")
        ^
SyntaxError: invalid syntax

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

We have one final program in this chapter. As with the previous two programs, program **SyntaxError07.py**, shown in Figure 3.33, echoes the idea of always closing what you open. In this case, it is a multi-line comment. What happens if you forget to close a multi-line comment? Well, if your multi-line comment never ends, it means the entire rest of your program is now one big comment. The error message you get is "**EOF while scanning triple-quoted string literal**". Hmm, EOF? That seams similar to EOL. It is in fact. EOF means **E**nd **O***f* **F**ile. The error message is saying that the file suddenly ended in the middle of this multi-line comment (or "triple-quoted string literal").

Figure 3.33

```python
# SyntaxError07.py
# This program demonstrates what happens if you
# forget to close your triple-double-quote.


print("One")
print("Two")
"""
print("Three")
print("Four")
print("Five")
print("Six")        # half a dozen
print("Seven")
print("Eight")
print("Nine")
print("Ten")
print("Eleven")
print("Twelve")     # one dozen
print("Thirteen")   # one baker's dozen

```

```
 ----jGRASP exec: python SyntaxError07.py
  File "SyntaxError07.py", line 21

      ^
SyntaxError: EOF while scanning triple-quoted string literal

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

3.6 The Issue with Quotes

We just finished looking at several programs that had various issues. We are actually going to look at some more. This "issue" with these program all revolve around the quotation marks used in a **print** statement. You may wonder how these can cause an issue. I mean all you need to do is place the text that you wish to print inside double quotes? What could be simpler? Well, what if you wish to display the following sentence which includes some quoted text?

John said, "Time to go!" and then he left.

Program **Quotes01.py**, shown in Figure 3.34, demonstrates what happens when you try to simply put the about text inside double quotes. The computer is not happy with this and we get yet another *syntax error*. What is happening is the interpreter is getting confused between the double quotes that we are trying to print (the ones around "Time to go!") and the double quotes used at the beginning and ending of the **print** statement.

Figure 3.34

```python
# Quotes01.py
# This program demonstrates the "Issue with Quotes".
# The interpreter gets confused because we are trying
# to display double quotes, but everything we print
# goes inside double quotes.


print()
print("John said, "Time to go!" and then he left.")

```

```
 ----jGRASP exec: python Quotes01.py
  File "Quotes01.py", line 9
    print("John said, "Time to go!" and then he left.")
                          ^
SyntaxError: invalid syntax

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

Program **Quotes02.py**, shown in Figure 3.35, demonstrates a very simple solution to the problem of the previous program. We can make the program work by using single quotes with our **print** statement instead of double quotes.

Figure 3.35

```python
# Quotes02.py
# This program demonstrates a fix for the quote issue.
# Python allows you to print with single quotes as
# well as double quotes.


print()
print('John said, "Time to go!" and then he left.')

```

```
 ----jGRASP exec: python Quotes02.py

John said, "Time to go!" and then he left.

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

Well, that seemed simple enough. So all we have to do to prevent the "issue with quotes" is use single quotes instead of double quotes. Actually, there is more to it than that. What if you wish to display this sentence which included a possessive apostrophe?

That is Tom's car.

Program **Quotes03.py**, shown in Figure 3.36, tries to display the above sentence use a **print** statement with single quotes. The result is the same syntax error that we had with program **Quotes01.py**. We have another example where the interpreter is getting confused between a quotation mark that is being displayed (the apostrophe in Tom's) and those at both ends of the **print** statement. The only difference now is that the confusion is between single quotes instead of double quotes.

Figure 3.36

```
1 # Quotes03.py
2 # This program demonstrates a similar quote issue.
3 # Using single quotes causes an issue when trying
4 # to print a possessive apostrophe (').
5
6
7 print()
8 print('That is Tom's car.')
9
```

```
 ----jGRASP exec: python Quotes03.py
  File "Quotes03.py", line 8
    print('That is Tom's car.')
                       ^
SyntaxError: invalid syntax

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

Program **Quotes04.py**, shown in Figure 3.37, demonstrates another very simple solution. This time, we can fix the problem of the previous program by using double quotes instead of single quotes.

Figure 3.37

```
# Quotes04.py
# This program demonstrates a simple fix
# for the previous program by putting the
# text inside double quotes.


print()
print("That is Tom's car.")

```

```
 ----jGRASP exec: python Quotes04.py

That is Tom's car.

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

So, if we want to print something that contains double quotes (like a literal quotation) we use single quotes with our **print** statement, and if we want to print something that contains single quotes (like a possessive apostrophe) we use double quotes with our **print** statement. Is that it? Well, not completely. It can actually get more complicated. What if I wish to this this sentence which contains both single and double quotes?

Bill said, "That is Tom's car."

Programs **Quotes05.py**, shown in Figure 3.38, and **Quotes06.py**, shown in Figure 3.39, demonstrates that this is not something that can be fixed by simply using either single or double quotes at both ends of the **print** statement. What we need is a way to explicitly tell the computer if a single or double quote is meant to be displayed as opposed to being part of the syntax of **print** statement itself.

Figure 3.38

```
# Quotes05.py
# This demonstrates a more complicated quote issue.


print()
print("Bill said, "That is Tom's car."")
```

```
 ----jGRASP exec: python Quotes05.py
  File "Quotes05.py", line 6
    print("Bill said, "That is Tom's car."")
                          ^
SyntaxError: invalid syntax

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

Figure 3.39

```python
# Quotes06.py
# This program demonstrates that switching to
# single quotes does not fix the issue of the
# previous program.


print()
print('Bill said, "That is Tom's car."')

```

```
 ----jGRASP exec: python Quotes06.py
  File "Quotes06.py", line 8
    print("Bill said, "That is Tom's car."")
                                    ^
SyntaxError: invalid syntax

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

More Escape Sequences

You have already seen the **\n** which is the escape sequence for a "New Line". There are actually several escape sequences. Program **Quotes07.py**, shown in Figure 3.40, demonstrates 2 more. The \' escape sequence can be used to display a single quote and the \" escape sequence can be used to display a double quote. These always work, regardless of what type of quotes was used at the beginning and ending of the **print** statement. Note only can it display the complicated sentence from the previous 2 programs. It can even display more complicated sentences, like those that have a quotation within another quotation.

Figure 3.40

```python
# Quotes07.py
# This program demonstrates a couple more "escape sequences".
# By using \" and \', you can print single and double quotes
# and it does not matter which type of quotes were used at
# both ends of the <print> statement.

```

```python

print()
print('Bill said, "That is Tom\'s car."')
print()
print("Joe said, \"Bill said, \'That is Tom\'s car.\'\"")

```

```
 ----jGRASP exec: python Quotes07.py

Bill said, "That is Tom's car."

Joe said, "Bill said, 'That is Tom's car.'"

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

You have probably noticed that all escape sequences start with a backslash (\ ) character. The backslash is what lets the interpreter know that this is a special escape sequence. One question I sometimes get is, "Mr. Schram, what if you just wanted to display a backslash?" Good question. Program **Quotes08.py**, shown in Figure 3.41, demonstrates what happens when you put a backslash in a **print** statement with nothing else. The result is the same syntax error that we got back in program **SyntaxError05.py**. That was the program that was missing a closing quote. If you look closely at this program, you will see it is also missing a closing quote.

Figure 3.41

```python
# Quotes08.py
# This program demonstrates that printing a backslash (\)
# is not as easy as it would seem to be.  This is because
# a backslash is the special character that is used to
# create an "Escape Sequence".


print()
print("\")

```

```
 ----jGRASP exec: python Quotes08.py
  File "Quotes08.py", line 9
    print("\")
             ^
SyntaxError: EOL while scanning string literal

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

If you wish to display a backslash, you need to use the special escape sequence (yes, another one) for a backslash. The escape sequence for a backslash is actually a double-backslash (\\ ). You may wonder why you would ever need to print a backslash. Well, look at program **Quotes08.py**, shown in Figure 3.42. It introduces one more escape sequence: **\t** which is for a *tab*. If you look at the output you see that the **\t** did in fact generate a *tab*. You also see that even though line 10 starts by printing a double-backslash, only one was actually displayed.

Figure 3.42

```
 1 # Quotes09.py
 2 # This program demonstrates that to
 3 # display a backslash, you need to
 4 # use a double backslash (\\).
 5 # It also shows how to generate a
 6 # "tab" with the \t Escape Sequence.
 7
 8
 9 print()
10 print("\\t is used for \tTab")
```

```
 ----jGRASP exec: python Quotes09.py

\t is used for    Tab

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

We have seen a number of program examples where single quotes are used with **print** statements instead of double quotes. While I personally prefer to use double quotes, it does not matter either way to Python. Program **Quotes08.py**, shown in Figure 3.43, demonstrates that this also applies to multi-line comments. This program is almost identical to program **Comments02.py** from earlier in the chapter. The only difference is the *triple-double-quotes* have been replaced with *triple-single-quotes*. The result is the exact same output as before.

Figure 3.43

```
# Quotes10.py
# This program is almost identical to Comments02.py
# and demonstrates that a multi-line comment can
# also be created with triple-single-quotes.


print()
print("One")
print("Two")
'''
print("Three")
print("Four")
print("Five")
print("Six")       # half a dozen
print("Seven")
print("Eight")
print("Nine")
print("Ten")
print("Eleven")
'''
print("Twelve")    # one dozen
print("Thirteen")  # one baker's dozen

```

```
 ----jGRASP exec: python Quotes10.py

One
Two
Twelve
Thirteen

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

3.7 The Responsible Use of Computers

Computers have been made simpler in so many ways. Tedious computations are now performed by computers. Creating any type of writing is so much simpler with a word processor. International travel reservations can be made in minutes, rather than in days. Communication with anybody in the world is now instantly possible and it is very cheap. Information about any conceivable topic can be found on the Internet. And yes, let us not forget that there is now an explosion of electronic games that can be played on computers.

Unfortunately, computers can be used incorrectly as well. Electronic software is easy to copy and many people are tempted to help themselves to a free copy of software. Electronic transfer of money can be abused and computers have created a level of white-collar crimes that were unimaginable before. The Internet provides connections for millions of people, but it also makes it possible to collect private information from people. A whole new criminal area of identity theft has evolved. Computers are marvelous tools, but at the start of this course when you are about to learn some serious computer information, you need to stop and think about the responsibility and consequences of using computers.

This chapter section is designed to make you think and discuss some important computer issues.

DO NOT TAKE THIS TOPIC LIGHTLY!

I have known many students who have lost hundreds of hours of computer work, because they neglected to back up their efforts. I also know several students, who have felony records because they made some seriously bad decisions with their computer knowledge. The evening news also reports cases regularly of naive young teenagers who put their private information on the Internet and find themselves getting involved with adults who take advantage of children who are clueless about the consequences of providing private information to an unknown audience.

Using computers responsibly involves two major categories. What can happen to your computer and you personally and what can you do to other computers and other people. Hopefully, this section will make you realize that you have a tremendous amount of control over using computers properly, and you will neither be a victim nor cause somebody else to become a victim.

Protect Your Computer From the Environment

Computers and computer information are vulnerable. Get used to it. Any electronic device, computers included, does poorly when it is mishandled. Computers should not get too hot or too cold. They should not be dropped down a stair case. You think I am kidding? I have observed three occasions where a student dropped a laptop computer down a staircase. I have seen a student lift up a laptop by the power cord, only to watch it become unplugged and obey the laws of gravity. I have seen a student walk on top of a computer. I have seen a student walk off with a computer connected to a power cord and watch the computer slip from the holding hands and fall down. I have observed multiple cases of drinks that spilled on computer keyboards, making the keyboard very sticky at best and totally non-functional at worst.

Be aware. For starters do not place a computer where it can fall from a desk or table. Place cords in such a way that people will neither trip nor pull computers down. Do not keep drinks of any kind anywhere near a computer.

Physical damage to a computer happens, but it is not the biggest problem. The number one source of grief for many people, especially students, is loss of data. All computer data is stored in some electronic fashion. For starters the essay you are writing is only stored in the Random Access Memory (RAM) of the computer. The RAM holds information while the computer is powered up. If there is a power outage, no matter how brief, you information will be gone. If you cleverly forget to save your work then it will also be gone. Everything you create needs some type of permanent storage. Your data can be stored on a hard drive, CD, jump drive, network or a second computer. All these devices store the information even when the computer is turned off, but information is still vulnerable. Hard drives do "crash" and CDs can become dirty or become lost. It is wise to have a second backup for anything important.

Please realize the following: it is very human to think that problems only happen to other people. Other people have accidents. Other people drop their computers. Other people have bad hard drives and lose data. I have actually seen a person get ready to pump gas with a burning cigarette in the same hand as the gas hose. He told me it was fine, because he did it before and nothing happened. The poor kid was not familiar with the "fat chance" theory.

Aside from saving programs regularly and then backing up all your data to some secondary backup device, is there something else you can do? Yes, there is. You can use a "battery backup" system. Such a system is much more than an extension cord with some power surge protection. The battery backup system provides power surge protection and keeps the computer turned on when there is power failure. This allows you to safely store your data. Many of these systems also have **A***utomatic* **V***oltage* **R***egulation* (AVR), which protects your computer from "dirty" electricity that has over-voltage and under-voltage periods.

Protect Your Computer From Viruses

Computer viruses are a fact of life. Do you actually know what a computer virus is? It is a special program that has two qualities. First, there is the duplication part. A computer virus can attach itself to your computer, search for email addresses and happily spread to computers on your email address book. The virus can also travel to other computers connected to your computer. So far the virus has only traveled and duplicated. The second part of a virus is the payload. The payload is the actual part of the virus that executes a program and does potential harm. There are mild viruses, which only display some silly message. There are also nasty viruses, which delete files on a hard drive, shut down the computer, alter information and cause major headaches to people who rely on computers.

Why do viruses exist? Well there are people with too much time on their hands who apparently have reasons - in their minds - to give major grief to other people. Sometimes computer programmers, who were fired, sabotage computers with a time-delay virus. This happened to Eastern Airlines in the Eighties. Six months after a major computer programmer was fired, the airline reservation computers lost all their data. Not a single passenger was confirmed on any flight. There are also military reasons. During the Cold War, there were hundreds of computer scientists employed by the USSR to create viruses that would handicap computer use in the West. During the first Iraq war, the United States sold computer equipment to France and France in turn sold this equipment to Iraq. The equipment was sold with a time-delay virus. The day that the U.S. attacked Iraq, the virus disabled the anti-aircraft computer programs in Baghdad. U.S. planes could fly over Baghdad with anti-aircraft missiles paralyzed by ineffective software.

Today almost all computers come equipped with some type of virus software. Frequently, people think that they have proper protection. Here is the problem. All anti-virus software comes with a *virus definitions* file. This file is pretty much guaranteed to be outdated by the time the software arrives in the store and is sold. All anti-virus software alert computer users to get virus updates, but many people ignore these warnings. Do keep in mind that bad things only happen to other people. The better anti-viruses software automatically updates your computer at regular intervals. New viruses are created at an alarming rate.

There are still other access problems invading your computer. Special software, called *spyware*, snoops around your computer and tries to record information. The spyware may be used for various purposes, but one immediate effect of this extra software is a computer slowdown. There also is your good friend SPAM. SPAM are annoying, unwanted email messages and other miscellaneous pop-up windows, that can slow your computer down tremendously. Most modern email programs include protection against SPAM and spyware.

Protect Your Computer From Improper Access

In the business world and at school, computers are frequently turned on, logged in and then computer users runs off to go to the bathroom, make come copies, get a drink, get a donut, meet with other people or any other reason that removes them from their computers. While they are gone, their computers can be accessed. In this day, it is easy to slip in a "jump drive", which is a miniature memory stick that plugs into a **U***niversal* **S***erial* **B***us* (USB) port. These memory sticks are small, but they can hold up to 2 GBs of memory or more and that is plenty of space to store all kinds of data from your computer.

People with some sophisticated knowledge also know how to make your computer remotely accessible. Later, in their own environment, your computer is accessed and you will not even realize what is happening.

In this day-and-age of laptop computers, access becomes simpler. It is easy to steal a laptop computer. During the last school year, on the last day of school, in my own classroom, a computer was stolen. All laptops look pretty much the same. A few have some personal stickers on the laptop, but the reality is that the action of placing a laptop computer in a backpack is a normal action. It does not look suspicious. In this case, a student left the classroom to talk to a student in an adjacent class, to put together an end-of-year team project. When the student returned at the end of class, his laptop computer was gone.

In a school and business environment, technological devices look very similar. The laptop charger and power cords look all alike. There are different brands of jump drives, but the same brands look identical. Frequently, people use CDs without a label on them. What does this all mean? You can easily lose your stuff, because perfectly honest people confuse your property with their property. In other words put labels on your CD, mark your computers, mark your chargers, mark your jump drives.

The Ethical Use of Computer Software

It is easy to be outraged when somebody steals your property, copies your files, sends you annoying email and invades your computer with nasty viruses. Is proper behavior different in the other direction? It used to amaze me when my wife had child care that parents had no difficulty dropping off sick children. *He is only a little sick with a small fever. I do not think he will infect anybody,* was a common statement heard in the morning at drop off time. Yet if the child would become sick, there would follow a major *crime scene investigation* to determine which offending child has infected their child.

What happens if your friend has a new game, a new word processing program or any other new type of software? You sure like that new software, but you cannot afford it. At this point the brain kicks in with its brilliant rationalization skills and minutes later you find it completely justified to convince your friend into a copy. "What hurt is there? It is only one copy and the company won't miss that single copy." Another popular rationalization is to claim that the company charges too much for the software. You, the noble person you are, will fight this tyranny by acquiring the software for free. That will teach the company a lesson.

There is an irony here. Department store merchandise costs more because the cost of shoplifting is added the merchandise cost. Likewise, software would actually be quite a bit cheaper if everybody purchased an honest copy for themselves. Is it surprising that so many people think that this is just fine? Perhaps not, if you consider the "what is a conscience" story.

When I was a child, my great-grandfather used to tell me stories of timeless wisdom. These stories were based on Native American Indians. Most of the stories involve conversations between a young impulsive Indian boy named *Curious Beaver*, and his grandfather, the wise *Aging Bear*. In one of these stories Curious Beaver asks Aging Bear what a *conscience* is. The grandfather replies that a conscience helps you to tell right from wrong. The grandson then asks how does a conscience manage to help a person to know what is right. The grandfather replies:

> *My dear grandson, every person is born with a conscience. The conscience is a triangle inside your body, located close to your heart. The three edges of the triangle are very sharp. Anytime that you do something wrong, the triangle turns. As the triangle turns, the sharp edges cause pain. You feel this pain when you behave incorrectly. The worse your deed and the faster the triangle turns, causing greater pain.*

The son appears puzzled and tells his dad that he understands how the conscience works, but he is confused. The boy says that he has seen several people who have performed major misdeeds and they did not seem bothered by any pain. The father looks solemn and becomes very serious.

> *You are very observant and very correct for one so young. Yes there are people who commit great sins frequently and yet, they seem unconcerned about the gravity of their acts. You see, the conscience turns a lot when people sin, but the frequent turning makes the sharp edges dull. Soon the triangle can turn freely inside the person and there is no longer pain. It is a great tragedy when somebody has done so much wrong that the conscience is no longer able to help people realize how wrong their actions are.*

Software is sold with simple information about licensing. I have purchased a copy of Microsoft Office Teacher Edition. It states directly on the front of the box, inside a conspicuous green star, that the *product is licensed for noncommercial use on up to 3 home PCs*. Pretty simple to understand.

Hacking and Other Misdeeds

There are people who think that visiting other networks and other computers is quite innocent. The innocence is based on the fact that no attempt is made to transfer money illegally. No attempt is made to acquire any private information to assume some else's identity to acquire credit cards and other such evils. No these folks are very nice people - so they claim - who happen to do a little bit of innocent hacking into other systems. Besides, if somebody is dumb enough to allow easy access to their computer then they deserve this hacking business.

You may or may not accept this type of explanation, but there is a fundamental problem. There are plenty of hackers who have sufficient knowledge to break into computer systems. Frequently, this knowledge is based on hacking software that is downloaded from the Internet. You now have somebody who is smart enough to figure out how to break into a system with some special program, but in reality these hackers have no clue how their tools work. You can assist somebody else to do considerable damage to computer systems by releasing viruses you do not even realize are attached to the free hacking programs.

There are other problems, which are not hacking. Consider the following. There was a student several years ago who decided to send an email message to all the teachers in the school. This was not hacking. The email addresses of teachers are advertised on the school's web site. His message was the following:

I know where you are and I know what you are doing.

Many teachers were relaxed and responded to come over and help cut the grass or grade papers. Other teachers were scared and went to the police feeling they were stalked. A day later the young man, who was having some fun, found himself arrested and he was in a legal nightmare for the next three months. As a minor everything was eventually dismissed in some fashion. He thought that his clever prank had gone away. A year later he was a senior student and suddenly found out that not a single teacher in his school was willing to write a positive college recommendation for him.

Computer Vandalism

I will finish this section with vandalism deeds that are quite common. Stealing computers is not such a common problem. It is much more common that keys are altered on a keyboard. It is also common that goofy, unwanted programs appear on computers. Sometimes students alter window titles and replace them with some clever message, usually with an improper choice of vocabulary. This type of misbehavior is minor in its effect, but it is annoying and time-consuming to clean up the mess.