Chapter VII

Keyboard Input and
Selection Control Structures

Chapter VII Topics

7.1 Keyboard Input

Program input has seemed less than impressive so far. Frequently, you have executed programs multiple times with different values hard-coded in various program statements. Such an approach is hardly user-friendly and will sell little software. At the same time, programs without input are quite tedious. This is especially true when you work with control structures that behave differently with different values.

Program **KeyboardInput01.py**, in Figure 7.1, starts by showing an example of what <u>not</u> to do. The program does properly use the **input** command which pauses the execution until the user types something and then presses the <enter> key. The problem is, the user will have absolutely no idea what to enter. For that matter, the user may not even know that he/she is supposed to enter anything. The output just seems to have frozen. If the user does have knowledge of the inner workings of the program (something that cannot be assumed) he/she will enter his/her name and then the program execution can continue. To prove that the name was entered, it is displayed in the form of a greeting. For example, if "**John**" is entered, the computer displays, "**Hello John**".

Figure 7.1

```
 1 # KeyboardInput01.py
 2 # This program demonstrates a very user-UNfriendly
 3 # way to do keyboard input.
 4 # The user has no idea what he/she is entering.
 5
 6
 7 print()
 8
 9 name = input()
10
11 print("Hello",name)
12
```

Initial Output:

```
    ----jGRASP exec: python KeyboardInput01.py
```

```
    ----jGRASP exec: python KeyboardInput01.py


▶▶  John
    Hello John


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

Program **KeyboardInput02.py**, in Figure 7.2, improves upon the previous program by adding a user-friendly *prompt* to instruct the user to enter his/her name. Note that the prompt is a *string literal* argument of the **input** command.

Figure 7.2

```
 1 # KeyboardInput02.py
 2 # This program improves the keyboard input
 3 # by adding a user-friendly "prompt".
 4
 5
 6 print()
 7
 8 name = input("Please enter your name.  -->  ")
 9
10 print("Hello",name)
```

```
    ----jGRASP exec: python KeyboardInput02.py


▶▶  Please enter your name.  -->  John
    Hello John


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

```
    ----jGRASP exec: python KeyboardInput02.py


▶▶  Please enter your name.  -->  Diana
    Hello Diana


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

Program **KeyboardInput03.py**, in Figure 7.3, demonstrates how to write a program with multiple lines of input entered from the keyboard during program execution. The **input** command is used three times, with three different prompts and three different variables. The 3 string variables: **firstName**, **middleName** and **lastName** all get concatenated together with spaces to make the **fullName** which is displayed at the end.

Figure 7.3

```
 1 # KeyboardInput03.py
 2 # This program enters 3 different names: <firstName>,
 3 # <middleName> and <lastName>; on 3 different lines
 4 # and then combines them all into a full name.
 5 # NOTE: You will not see the second prompt until you
 6 # finish entering the information from the first and
 7 # press <enter>.
 8
 9
10 print()
11
12 firstName  = input("Please enter your first name.   -->  ")
13 middleName = input("Please enter your middle name.  -->  ")
14 lastName   = input("Please enter your last name.    -->  ")
15
16 fullName = firstName + " " + middleName + " " + lastName
17
18 print()
19 print("Your full name is",fullName)
20
```

```
     ----jGRASP exec: python KeyboardInput03.py


  Please enter your first name.    -->   John
  Please enter your middle name.   -->   Quincy
  Please enter your last name.     -->   Public


  Your full name is John Quincy Public


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

There is no limit to how much data a program can enter. You just need to make sure you have a separate variable for each piece of data, and that the data is the correct type.

Speaking of *types*, it appears that keyboard input during program input happens with strings only. At least that has been the evidence during the last two program examples. Is it possible to enter numbers during program execution? Program **KeyboardInput04.py**, in Figure 7.4, enters two integers and tries to display their sum. Unfortunately, the program performs *concatenation* instead of addition.

Figure 7.4

```
 1 # KeyboardInput04.py
 2 # This program attempts to compute the sum of two
 3 # numbers entered by the user.  The problem is the
 4 # numbers are being entered as strings instead of
 5 # numbers.
 6
 7
 8 print()
 9
10 num1 = input("Please enter the 1st number.  -->  ")
11 num2 = input("Please enter the 2nd number.  -->  ")
12
13 sum = num1 + num2
14
15 print()
16 print("The sum of",num1,"and",num2,"is",sum)
17
```

```
    ----jGRASP exec: python KeyboardInput04.py

   Please enter the 1st number.  -->  100
   Please enter the 2nd number.  -->  200


   The sum of 100 and 200 is 100200


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

If you want keyboard input to be <u>eval</u>uated as a number, you need to use the **eval** command. This command essentially wraps around the **input** command, using it as **eval**'s argument. Program **KeyboardInput05.py**, in Figure 7.5, demonstrates **eval** being used for the input of both numbers. Now we get the desired addition.

Figure 7.5

```
1 # KeyboardInput05.py
2 # This program demonstrates the proper way to enter
3 # numerical input by using the <eval> function.
4 # The function makes the computer EVALuate the
5 # input to see what kind of information it is
6 # and properly identify numerical input.
7
8
9 print()
10
11 num1 = eval(input("Please enter the 1st number.  -->  "))
12 num2 = eval(input("Please enter the 2nd number.  -->  "))
13
14 sum = num1 + num2
15
16 print()
17 print("The sum of",num1,"and",num2,"is",sum)
18
```

```
    ----jGRASP exec: python KeyboardInput05.py

  Please enter the 1st number.  -->  100
  Please enter the 2nd number.  -->  200

  The sum of 100 and 200 is 300

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

Program **KeyboardInput06.py**, in Figure 7.6, shows that **eval** works for real numbers as well as integers. In this case, the program enters 3 numbers, which can be integers or real numbers, and then computes and displays their average.

Figure 7.6

```
 1 # KeyboardInput06.py
 2 # This program computes the average of 3 numbers
 3 # entered by the user.  Note that this works for
 4 # both integers and real numbers.
 5
 6
 7 print()
 8
 9 num1 = eval(input("Please enter the 1st number.  -->  "))
10 num2 = eval(input("Please enter the 2nd number.  -->  "))
11 num3 = eval(input("Please enter the 3rd number.  -->  "))
12
13 average = (num1 + num2 + num3) / 3
14
15 print()
16 print("The average of",num1,"and",num2,"and",num3,"is",
average)
17
```

```
    ----jGRASP exec: python KeyboardInput06.py

  Please enter the 1st number.  -->  22.22
  Please enter the 2nd number.  -->  33.33
  Please enter the 3rd number.  -->  77.77


  The average of 22.22 and 33.33 and 77.77 is 44.44


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

What if you want to have program input in a graphics program? For example, a graphics program might display a circle, but you want to allow the user to specify the color and size. Program **KeyboardInput07.py**, in Figure 7.7, attempts to do this very thing. The output shows that there are definitely a couple issues.

The first issue becomes apparent immediately after you run the program. The graphics window covers most if not all of **jGRASP** and its output window which prevents you from entering the necessary input. Now, after you move, minimize or even close the graphics window, we come to the second, bigger issue that occurs with certain versions of Python. While it looks like we can enter the color name, note what happens after we press <enter>. It does not move on to the radius input. Instead, the program is now permanently stuck in the first input. No matter what you enter, you cannot get out. You have to click the **End** button to force the program to terminate.

Figure 7.7

```
 1 # KeyboardInput07.py
 2 # This program demonstrates the issues that arise
 3 # when you try to use <input> in a graphics program.
 4 # One issue is the graphics window covers the input.
 5 # A bigger issue is that the input may be unending
 6 # depending on your version of Python.
 7
 8
 9 from Graphics import *
10
11 beginGrfx(1300,700)
12
13 myColor = input("Enter any color name.  -->  ")
14 myRadius = eval(input("Enter a radius value from 1-300.  -->  "))
15 setColor(myColor)
16 fillCircle(650,350,myRadius)
17
18 endGrfx()
```

```
    ----jGRASP exec: python KeyboardInput08.py
▶▶  Enter any color name.  -->  orange
▶▶  orange
▶▶  orange
▶▶  2
▶▶  3
▶▶  L
```

Program **KeyboardInput08.py**, in Figure 7.8, demonstrates two new commands which are better suited for graphics programs.   These are **textInput** and **numInput** and they solve both of the issues of the previous program. **textInput** is for entering strings.   You probably guessed that **numInput** is for entering numbers. Unlike the text-based **input** command, these graphics-based commands open a small, "pop-up", input window.   Both of these commands have 2 arguments. The first specifies the title/header of the window, the second is the prompt. After you type the information, you need to click the [**OK**] button. Now, when you run the program, these is no fumbling between windows and – more importantly – no never-ending input. First you see a small window to enter the color, then you see a small window to enter the radius, and finally you see the output. Again, 2 outputs will be shown.

Figure 7.8

```python
1  # KeyboardInput08.py
2  # This program demonstrates <textinput> and <numinput>
3  # which are better suited for graphics programs.
4  # Note that <numinput> does not require the <eval> function.
5
6
7  from Graphics import *
8
9  beginGrfx(1300,700)
10
11 myColor = textinput("Color Input","Enter any color name.")
12 myRadius = numinput("Radius Input","Enter a # from 1-300.")
13 setColor(myColor)
14 fillCircle(650,350,myRadius)
15
16 endGrfx()
```
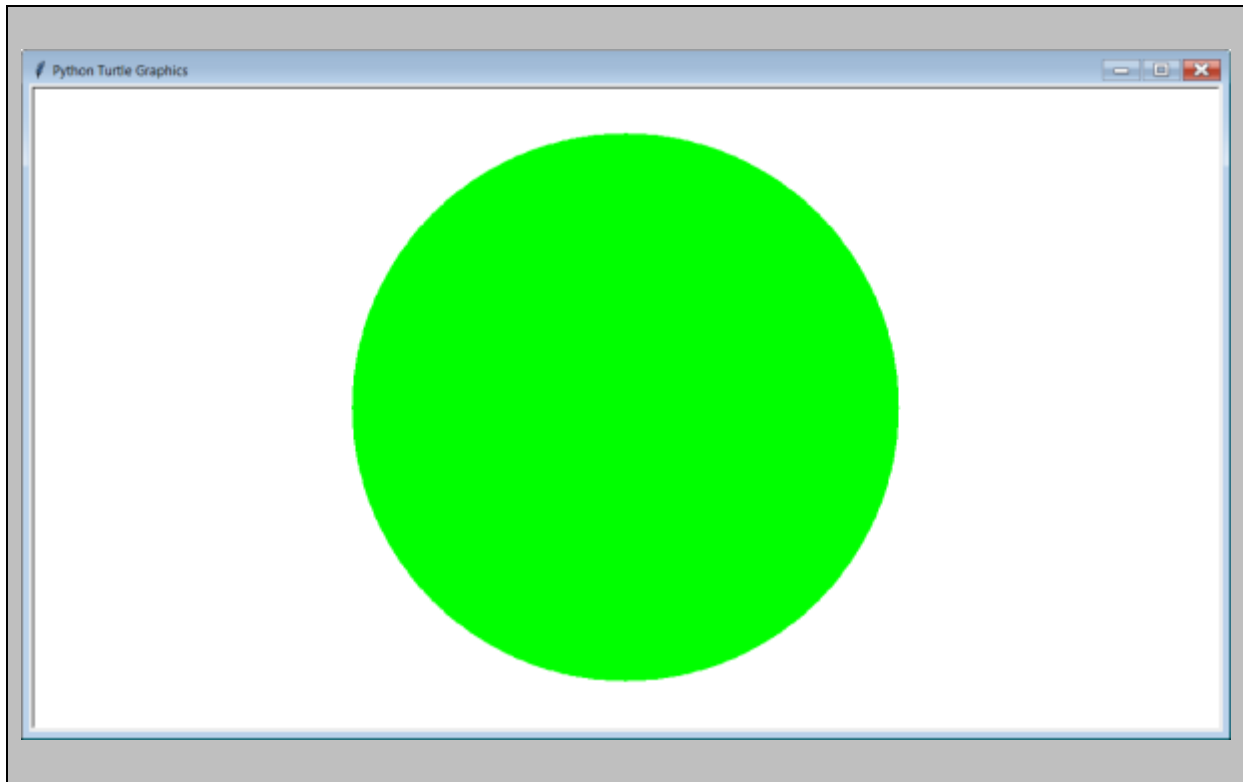
Input Functions

input(_prompt_**)** is used for <u>string</u> input on the <u>text</u> screen.

eval(input(_prompt_**))** is used for <u>number</u> input on the <u>text</u> screen.

textinput(_title,prompt_**)** is used for <u>string</u> input on the <u>graphics</u> window.

numinput(_title,prompt_**)** is used for <u>number</u> input on the <u>graphics</u> window.

7.2 Introduction to Control Structures

The previous chapter focused on using *functions*, *procedures* and *arguments*, which are an integral part of Python program organization. The intention in *Exposure Computer Science* is to present program organization in stages. A thorough understanding of program organization and program design is vital in modern computer science. At the same time, you also need to learn the syntax of a programming language. You cannot write proper Python programs unless you know the Python sentence structure or syntax. A prerequisite for a creative writing course is knowledge of grammar.

Frequently, it is mentioned that programming language syntax is trivial. *The essence of programming is design, data structures and algorithms.* This is very true, and very lovely, but language syntax tends to be trivial only if you know syntax. You will get very frustrated with your programs when they do not execute because of syntax problems.

In an earlier chapter we mentioned that program execution follows the exact sequence of program statements. That was a true, but also a rather incomplete explanation. There is a lot more to the program flow picture. We can expand on the exact program sequence by stating that program flow follows the sequence of program statements, unless directed otherwise by a Python control structure.

Program Flow

Program Flow follows the exact sequence of listed program statements, unless directed otherwise by a Python control structure.

Programs in any computer language require control structures. It may appear that all our program examples were written without the use of any control structures, but the control was subtle. As mentioned earlier, control was provided by a sequence of program statements. That type of control is called *simple sequence*.

Simple sequence alone is not very satisfactory for programs of any type of consequence. Programs constantly make decisions. A payroll program needs to change the amount paid, if the hours exceed 40 per week. The same payroll program can have many variations of tax deductions based on the number of dependents claimed. A payroll program also needs to have the ability to repeat the same actions for additional employees. In other words, a program needs to have the ability to repeat itself, or repeat certain program segments. The language features that allow that type of control will be introduced in this chapter.

7.3 Types of Control Structures

Program-execution-flow is controlled by three general types of control structures. They are *simple sequence*, *selection*, and *repetition*. Python provides syntax, and special keywords for each of these three control structures. Before we look at actual Python source code required to implement control, let us first take a look at diagrams that explain each control structure.

Simple Sequence
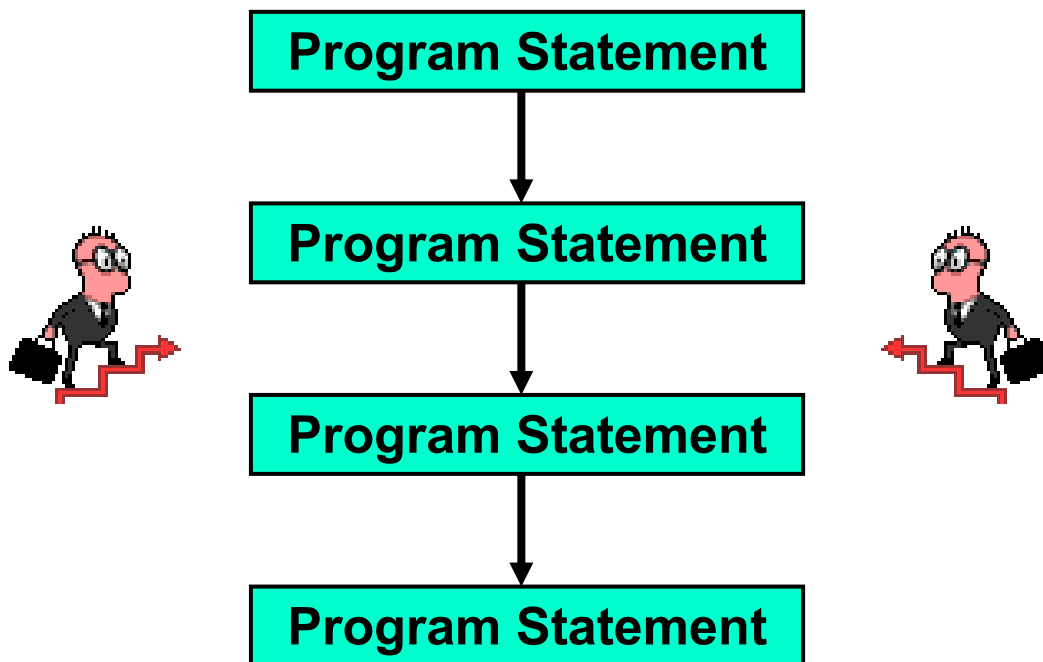
Simple sequence holds no surprises. A series of program statements are executed in the exact sequence that they are written. Altering the program execution logic requires altering the sequence of the program statements.

Selection

Frequently, programs cannot follow a single, simple sequence, path. Decisions need to be made like should the applicant be hired or not? Does the employee get overtime pay? Which tax bracket is the deduction to be computed from?

Selection is also called *conditional branching* or *decision making*. The program flow encounters a special condition. The value of the condition determines if the program flow will "branch off" from the main program sequence. There are 3 types of selection: *one-way*, *two-way* and *multiple-way*. Three diagrams, one for each type of selection, will be shown.

One-Way Selection
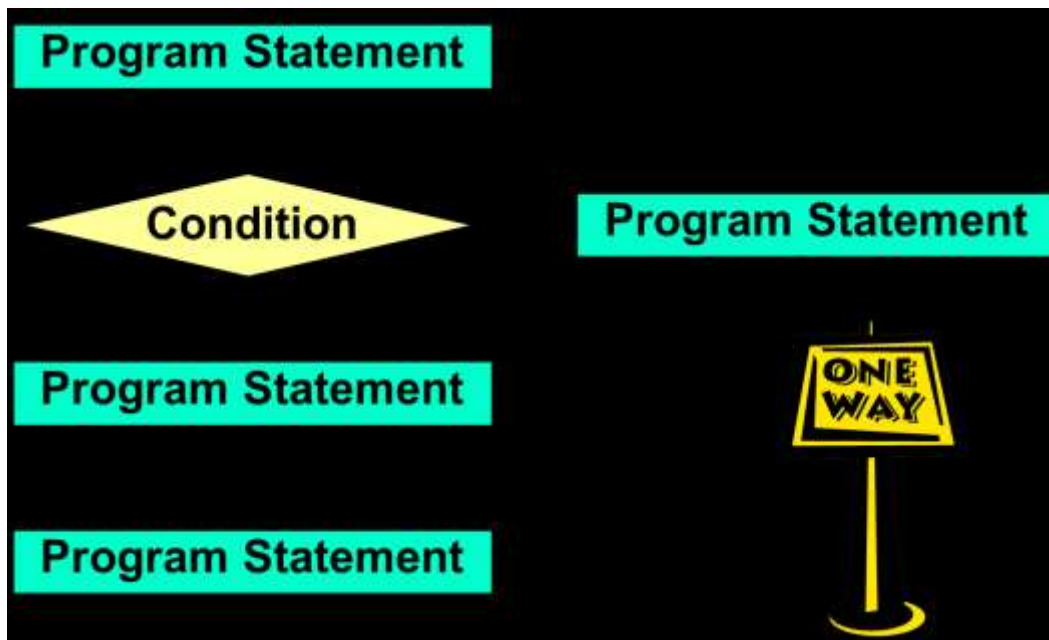
Selection control structures use a special conditional statement. If the condition is **True**, some action is performed, such as branching off to another sequence of program statements. In the case of *one-way selection*, the **True** condition branches off. If the condition is **False**, the program flow continues without change in program sequence.

Consider the analogy of driving South from Dallas to Austin. Along the way you check if your gas tank is low. If the tank is low, you stop for gas, and then continue to Austin. If the tank is not low you continue to drive south. Keep in mind that regardless of the tank condition, you are heading to Austin.

Two-Way Selection
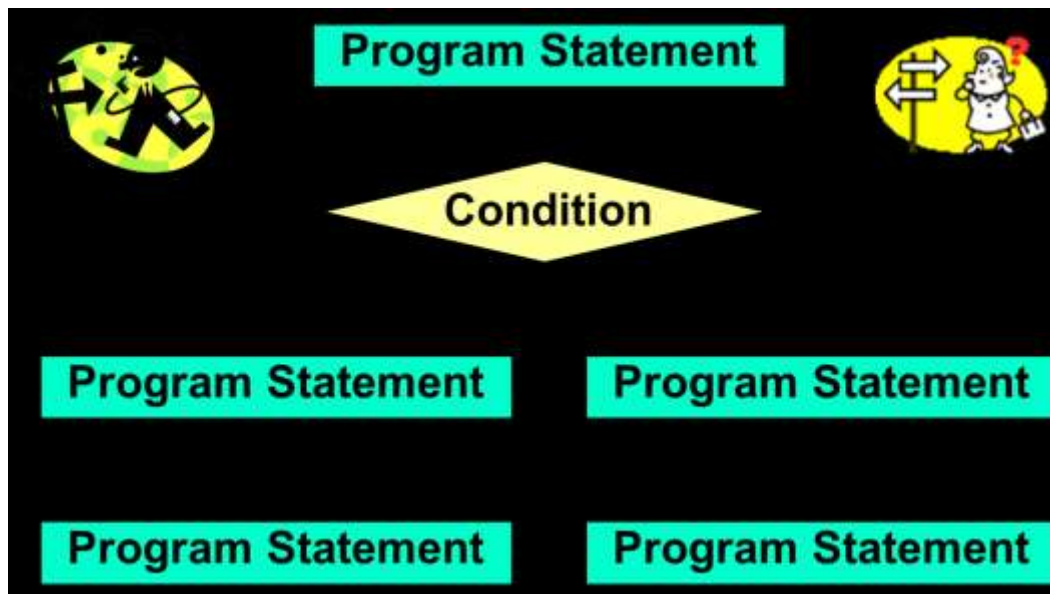
The *two-way selection* control structure also checks to see if some special condition is **True**. But there is a significant difference in how the program flow is handled. With *one-way* selection, a **True** condition means to do something, like get off the road and get gas, before continuing. The *two-way selection* structure selects one direction, or the other direction, but never both.

The *one-way* analogy describes a trip traveling south from Dallas to Austin. Regardless of the gas tank situation, the trip travels to Austin in the same car. Now consider an analogy for *two-way selection*. You are now driving from Austin back to Dallas. The highway you would take is I35 (Interstate 35). When you are about 70 miles from Dallas, shortly after you pass the town of Hillsboro the highway *forks*. It splits in two. You need to decide between going left which means you will take I35W (Interstate 35 West) to Fort Worth or going right which means you will take I35E (Interstate 35 East) to Dallas.

Multiple-Way Selection
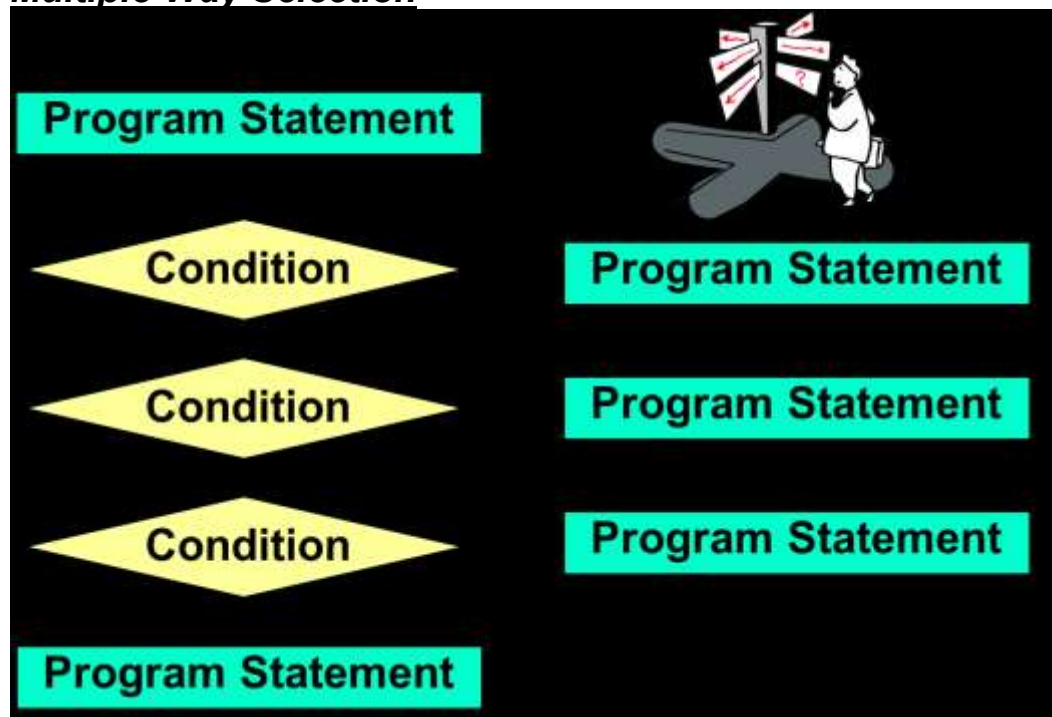
Multiple-way (or *Multi-way*) selection describes a situation where there are more than 2 possible *paths* for the program to follow. Different languages implement multi-way selection in different ways. Some languages, like Python, use an approach that looks like there is just a lot of two-way selection. You can argue it either way. Each individual condition has 2 possible paths to follow. When all of the conditions are combined, there are multiple paths the program can follow.

Multi-way selection is a commonly used control structure that simulates many situations in real life. Many times there are more than 2 things to choose from. For example, do you want chocolate, vanilla, strawberry, pistachio, rocky road, cookies & cream, mint chocolate chip or moo-llennium crunch ice cream for dessert? Do you plan to go to Harvard, Yale, Stanford, Princeton, Texas Tech, UT, OU or some other university? In fact, any time you take a multiple choice test, you are experiencing real life multi-way selection.

Repetition

Another common application occurs when repetition is required. A grade book program needs to average grades for every student in a class of twenty-five students. A payroll program needs to process paychecks for many employees. Practically everything you can imagine is done multiple times. Nobody is interested in repeating program source code 500 times for some task that is to be performed 500 times. We want to create one program segment, and place this segment in some type of loop control structure that repeats 500 times.
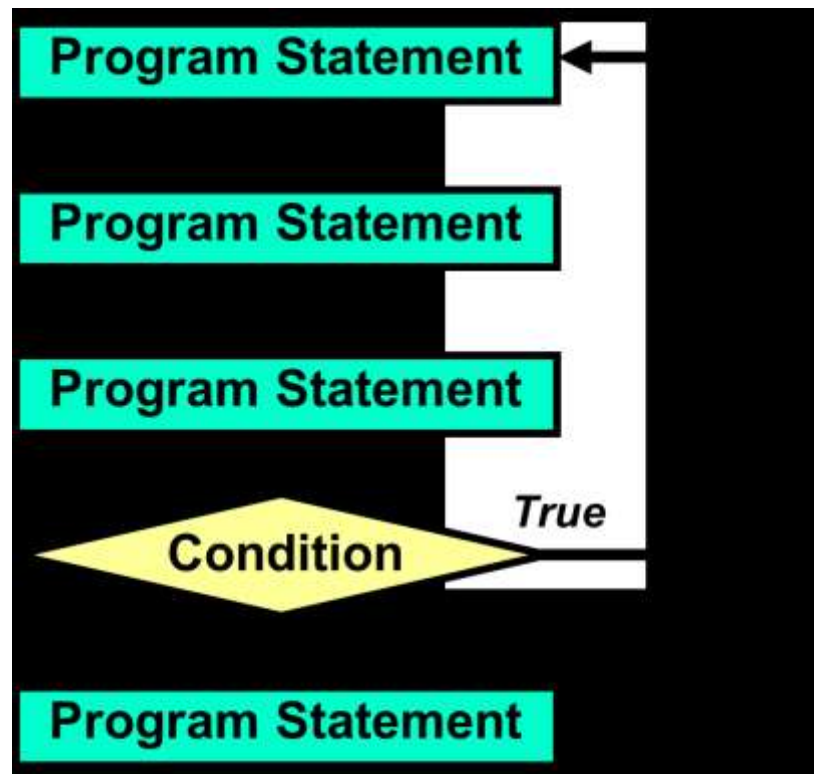
NOTE: This chapter will focus mainly on *Selection*. The next chapter will focus on *Repetition*.

7.4 Relational Operators

All of the control structure diagrams (both selection and repetition) indicate a change of program flow occurring after some *condition*.   Understanding conditional statements is the essence of understanding, and using, control structures. However, before we plunge into the syntax of the various conditional statements, you need to understand the *relational operators* that are used by Python in the conditional statements.

Conditional Statement Definition

A *conditional statement* is a program expression, which evaluates to **True** or **False**.

Most conditional statements require a relational operator.

All conditions end with a colon (**:**).

Consider an expression, such as 5 + 4. Does that expression evaluate to **True** or **False**? Neither, it evaluates to **9**. A *relational operator* is required to make an expression evaluate to **True** or **False**.   Python has six relational operators: *Equals*, *Not equals*, *Greater than*, *Less than*, *Greater than or equal to*, and *Less than or equal to*.

The idea of conditional statements that are based on a relational operator can be considered in regular English statements:

If we save more than $200.00 a month, we can go on a vacation

If your SAT score is high enough you will be admitted to college, otherwise you will not be able to go to college

Repeat calling established customers until you have 25 surveys

Python Relational Operators

Name	Operator	Expression	Evaluates
Equals	==	5 == 5 5 == 10	True False
Not equals	!=	50 != 25 100 != 100	True False
Less than	<	100 < 200 200 < 100	True False
Greater than	>	200 > 100 200 > 200	True False
Less than or equals	<=	100 <= 200 200 <= 200 200 <= 100	True True False
Greater than or equals	>=	100 >= 200 200 >= 200 200 >= 100	False True True

The relational operators shown in this diagram will be used in the Python example programs that demonstrate the different control structures. Be careful not to confuse the *equality operator* (==) with the *assignment operator* (=).

7.5 One-Way Selection

The simplest control structure is *one-way selection*. Basically, *one-way selection* says to perform some indicated action if a condition is **True**. If the condition is **False**, continue to march as if the control structure did not exist.

Program **Selection01.py**, in Figure 7.9, computes the Christmas bonus for employees of some imaginary company. The company is generous, but only for those who work very hard. Employees will get a $1000 Christmas bonus only if they sell at least $500,000 in merchandise. You will see two outputs for this program, as will be the case with future program examples.

Figure 7.9

```
 1 # Selection01.py
 2 # This program demonstrates one-way selection
 3 # with <if>.  Run the program twice.
 4 # First with <sales> equal to 300,000 and a
 5 # second time with <sales> equal to 500,000.
 6
 7
 8 print()
 9 sales = eval(input("Enter Sales  -->  "))
10 bonus = 0
11
12 if sales >= 500000:
13     bonus = 1000
14
15 print()
16 print("Christmas bonus:",bonus)
```

```
    ----jGRASP exec: python Selection01.py

▶▶  Enter Sales  -->  300000

    Christmas bonus: 0

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

```
     ----jGRASP exec: python Selection01.py

  Enter Sales  -->  500000


  Christmas bonus: 1000


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

If you look at line 13 of program **Selection01.py**, you see that the statement
bonus = 1000 is indented. In most programming languages, it is <u>encouraged</u> that
you indent the program statements that are being "controlled" by a *control
structure*. While this is "encouraged" it is not enforced, at least not in most
programming languages.

Program **Selection02.py**, in Figure 7.10, demonstrates what happens if you do not
use proper indenting in Python.

Figure 7.10

```
 1 # Selection02.py
 2 # This program demonstrates the Syntax Error
 3 # you receive when you do not properly indent
 4 # the programming statement(s) being controlled
 5 # by a control structure.
 6
 7 # NOTE: In most languages, indentation is recommended.
 8 #       In Python, indentation is required.
 9
10
11 print()
12 sales = eval(input("Enter Sales  -->  "))
13 bonus = 0
14
15 if sales >= 500000:
16 bonus = 1000
17
18 print()
19 print("Christmas bonus:",bonus)
```

```
   ----jGRASP exec: python Selection02.py
    File "Selection02.py", line 16
       bonus = 1000
            ^
   IndentationError: expected an indented block
after 'if' statement on line 15


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

Indentation Rule

In most languages, indenting the program statements that are "controlled" by control structures is <u>recommended</u>.

In Python, it is <u>required</u>.

Python programs that do not use proper and consistent indentation will not execute.

Suppose I want to improve the Christmas Bonus program so that is displays extra messages if the employee gets a bonus. This is certainly possible. Program **Selection03.py**, in Figure 7.11 demonstrates that multiple program statements can be controlled by the same control structure as long as proper, consistent indentation is used.

Figure 7.11

```
1 # Selection03.py
2 # This program demonstrates a control structure
3 # can control multiple programming commands as
4 # long as proper, consistent indentation is used.
5
6
```

```
 7 print()
 8 sales = eval(input("Enter Sales  -->  "))
 9 bonus = 0
10
11 if sales >= 500000:
12     print("\nCONGRATULATIONS!")
13     print("You sold half a million dollars in merchandise!")
14     print("You will receive a $1000 Christmas Bonus!")
15     print("Keep up the good work!")
16     bonus = 1000
17
18 print()
19 print("Christmas bonus:",bonus)
20
```

```
   ----jGRASP exec: python Selection03.py


▶▶  Enter Sales  -->  500000


   CONGRATULATIONS!
   You sold half a million dollars in merchandise!
   You will receive a $1000 Christmas Bonus!
   Keep up the good work!


   Christmas bonus: 1000


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

```
   ----jGRASP exec: python Selection03.py


▶▶  Enter Sales  -->  300000


   Christmas bonus: 0


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

One-Way Selection

General Syntax:

```
if condition is True:
    execute program statement(s)
```

Specific Examples:

```
if gpa >= 90:
    print("Honor Grad!")


if savings >= 10000:
    print("It's skiing time")
    print("Let's pack")
    print("Remember your skis")
```

You can control as many statements as you wish with a control structure, as long as you use proper, consistent indentation.

7.6 Two-Way Selection

A slight variation of *one-way selection* is *two-way selection*. With *two-way selection* there are precisely two paths. The computer will either take one path, if the condition is **True**, or take the other path if the condition is **False**. It is not necessary to check a conditional statement twice. The reserved word **else** makes sure that the second path is chosen when the conditional statement evaluates to **False**. The program example that follows is very short. The program considers the value of a student's SAT score. College admission is granted or denied based on the SAT score. Note the indentation style of the **if..else** control structure that is shown in figure by **Selection04.py**, in Figure 7.12.

Figure 7.12

```
 1 # Selection04.py
 2 # This program demonstrates two-way selection
 3 # with <if..else>.  Run the program twice:
 4 # First with 1200, then with 1000.
 5
 6
 7 print()
 8 sat = eval(input("Enter SAT score  -->  "))
 9 print()
10
11 if sat >= 1100:
12     print("You are admitted.")
13 else:
14     print("You are not admitted.")
15
```

```
   ----jGRASP exec: python Selection04.py


▶▶  Enter SAT score  -->  1200


   You are admitted.


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

```
   ----jGRASP exec: python Selection04.py


▶▶  Enter SAT score  -->  1000


   You are not admitted.


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

Program **Selection05.py**, in Figure 7.13, demonstrates that you can control multiple program statements in both parts of a two-way selection structure. As before, the key is to use proper, consistent indentation for all program statements that are being "controlled" by the control structure. The outputs of this program also make a very important point. The minimum score for admission is **1100**. The first output shows that the student with an **1100** for his SAT score is admitted. The second output shows that the student with a **1099** for his SAT score is not. So what is the "point?" The computer has absolutely <u>no mercy</u>.

Figure 7.13

```
1 # Selection05.py
2 # This program demonstrates that multiple program
3 # statements can be controlled in both parts of an
4 # <if..else> structure as long as proper, consistent
5 # indentation is used.  Run the program twice:
6 # First with 1100, then with 1099.
7
8
9 print()
10 sat = eval(input("Enter SAT score  -->  "))
11 print()
12
13 if sat >= 1100:
14     print("You are admitted.")
15     print("Orientation will start in June.")
16 else:
17     print("You are not admitted.")
18     print("Please try again when your SAT improves.")
19
```

```
    ----jGRASP exec: python Selection05.py


  Enter SAT score  -->  1100


  You are admitted.
  Orientation will start in June.


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

```
   ----jGRASP exec: python Selection05.py

 Enter SAT score  -->  1099


 You are not admitted.
 Please try again when your SAT improves.


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

Two-Way Selection

General Syntax:

```
   if condition is True:
       execute program statement(s)
   else:  # when condition is False
       execute alternate program statement(s)
```

Specific Example:

```
   if average >= 70:
      print("You passed!")
      print("Get ready for summer vacation!")
   else:
      print("You failed.")
      print("Get ready for summer school.")
```

7.7 Multi-Way Selection

Multiple-Way Selection occurs whenever you have more than 2 choices. This means the program has more than 2 possible paths to follow. We already used the analogy of a "fork in the road" for two-way selection. A similar analogy for multi-way selection would be a 4-way stop like the one shown in Figure 7.14.

Figure 7.14
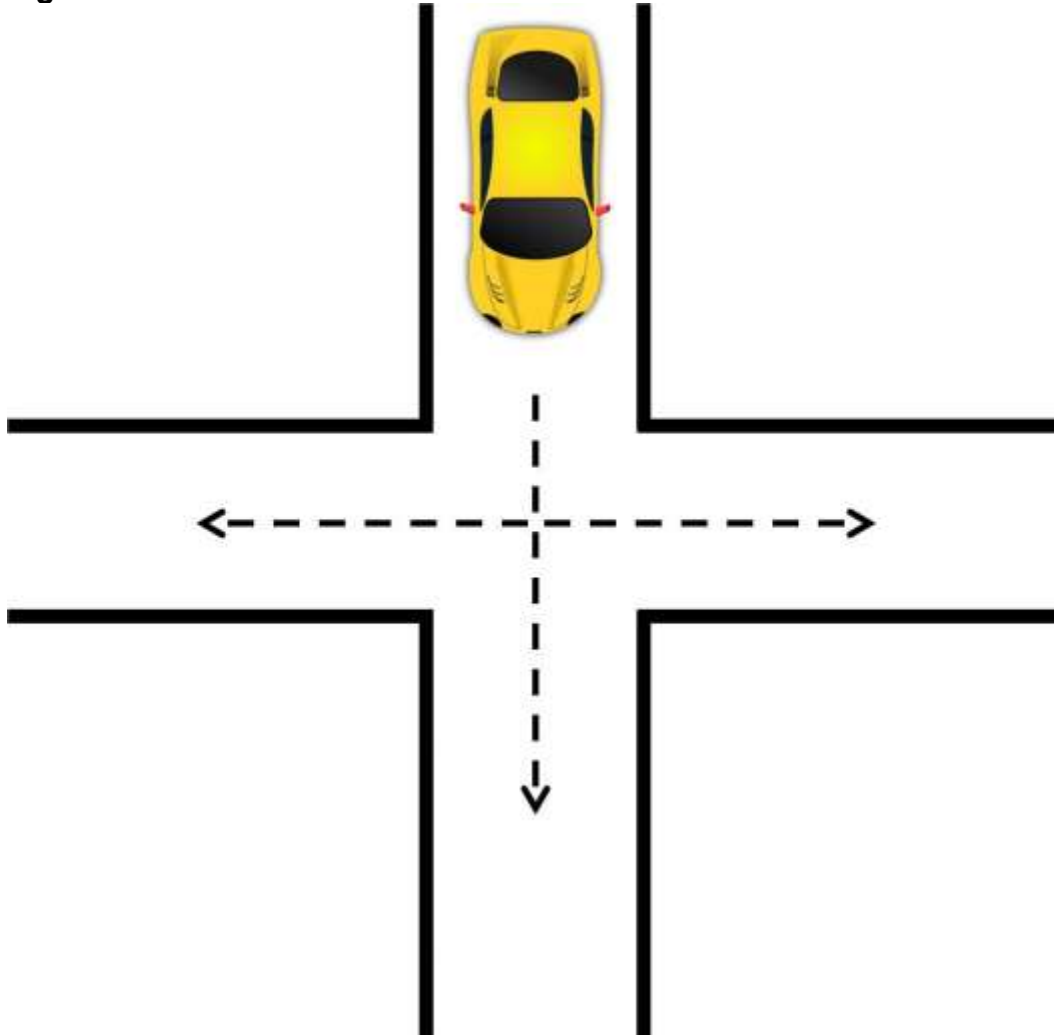
Program **Selection06.py**, in Figure 7.15, is an attempt at multi-way selection. The user enters a normal assignment **grade** which would be from **0** to **100**. Based on that number, the computer will display your letter grade which is either an **A**, **B**, **C**, **D** or **F**. So there are 5 possible outputs. To implement this, 5 different **if** statements are used. If you run the program and enter a failing grade, like **50**, the program seems to work. See what happens when you enter a passing grade, like **100**.

Figure 7.15

```python
1  # Selection06.py
2  # This program is supposed to display the letter grade
3  # earned based on the number grade entered by the user.
4  # Since there are more than 2 possible paths (A,B,C,D,F)
5  # this would be an example of "Multi-Way Selection" if
6  # the program worked.  Using 5 separate <if> statements
7  # has created a Logic Error that gives strange output.
8
9
10 print()
11 grade = eval(input("Enter Number Grade  -->  "))
12 print()
13
14 if grade >= 90:
15     print("You earned an A!")
16 if grade >= 80:
17     print("You earned a B.")
18 if grade >= 70:
19     print("You earned a C.")
20 if grade >= 60:
21     print("You earned a D.")
22 if grade >= 0:
23     print("You earned an F.")
24
25
```
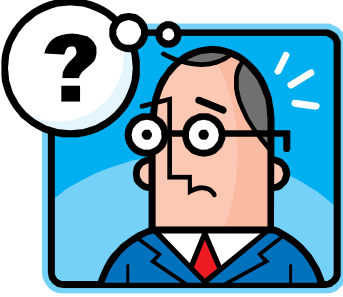
```
    ----jGRASP exec: python Selection06.py

 ►► Enter Number Grade  -->  50


    You earned an F.


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

```
    ----jGRASP exec: python Selection06.py


Enter Number Grade   -->  100


You earned an A!
You earned a B.
You earned a C.
You earned a D.
You earned an F.


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

There is a definite logic error in program **Selection06.py** which stems from the 5 **if** statements. The problem is, any grade that is >= **90** is also >= **80** and >= **70** and >= **60** and >= **0**. This is why entering **100** resulted in all 5 outputs. What we need is a way to make the outputs stop once the appropriate message is displayed.

Program **Selection07.py**, in Figure 7.16, fixes the logic issue of the previous program by adding several strategic **else** statements. Now, every **if** statement after the first is only executed if the previous **if** statement yielded a **False** result. While this works, each successive **if..else** block needs to be indented more than its predecessor. This makes the indenting of this program somewhat annoying.

Figure 7.16

```
 1 # Selection07.py
 2 # This program fixes the Logic Error of the
 3 # previous program by adding several strategic
 4 # <else> statements which will ensure that only
 5 # 1 letter grade is displayed.  While this works,
 6 # the program's indentation is somewhat annoying.
 7
 8 print()
 9 grade = eval(input("Enter Number Grade  -->  "))
10 print()
```

```
11
12 if grade >= 90:
13     print("You earned an A!")
14 else:
15     if grade >= 80:
16         print("You earned a B.")
17     else:
18         if grade >= 70:
19             print("You earned a C.")
20         else:
21             if grade >= 60:
22                 print("You earned a D.")
23             else:
24                 print("You earned an F.")
25
```

```
    ----jGRASP exec: python Selection07.py

▶▶  Enter Number Grade  -->  100

    You earned an A!

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

```
    ----jGRASP exec: python Selection07.py

▶▶  Enter Number Grade  -->  65

    You earned a D.

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

Program **Selection08.py**, in Figure 7.17, behaves in the exact same way as the previous program, but has much cleaner code. This is because the **elif** command is added. **elif** essentially combined an **else** with the next **if**. Now our nested and tediously indented **if..else** structure is replaced with a clean **if..elif..else** structure.

Figure 7.17

```
1 # Selection08.py
2 # This program shows a better way to do "Multi-Way
3 # Selection" using <if..elif..else>.  The <elif>
4 # command essentially combines the <else> with the
5 # next <if>.  Not only is this less code to type,
6 # it also has nicer indentation.
7
8
9 print()
10 grade = eval(input("Enter Number Grade  -->  "))
11 print()
12
13 if grade >= 90:
14    print("You earned an A!")
15 elif grade >= 80:
16    print("You earned a B.")
17 elif grade >= 70:
18    print("You earned a C.")
19 elif grade >= 60:
20    print("You earned a D.")
21 else:
22    print("You earned an F.")
23
```

```
    ----jGRASP exec: python Selection08.py


 ►► Enter Number Grade  -->  85


   You earned a B.


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

```
   ----jGRASP exec: python Selection08.py


▶▶   Enter Number Grade  -->  75


    You earned a C.


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

Program **Selection09.py**, in Figure 7.18, demonstrates a number of things. The first thing you will notice is that the program enters a "Letter Grade" instead of a "Number Grade". This means text values can "control" the flow of a program just like number values. Second, it shows that multi-way selection (just like one-way and two-way selection) can control multiple program statements as long as proper, consistent indentation is used. This last thing is demonstrated in the final output of the program when a grade of **Q** is entered. This is not a valid grade and the program is not able to generate any output for it.

Figure 7.18

```
 1 # Selection09.py
 2 # This program demonstrates a number of things:
 3 # 1. Selection can be based on text values also,
 4 #    not just number values.
 5 # 2. As with other selection structures, Multi-Way
 6 #    Selection can control multiple programming
 7 #    commands as long as proper, consistent
 8 #    indentation is used.
 9 # 3. The program will not work properly if the
10 #    user does not enter an A, B, C, D or F.
11
12 print()
13 grade = input("Enter Letter Grade  -->  ")
14 print()
15
16 if grade == 'A':
17     print("You grade is 90 or above.")
18     print("Excellent!")
```

```
19 elif grade == 'B':
20      print("You grade is in the 80s.")
21      print("Good")
22 elif grade == 'C':
23      print("You grade is in the 70s.")
24      print("Fair")
25 elif grade == 'D':
26      print("You grade is in the 60s.")
27      print("Poor")
28 elif grade == 'F':
29      print("You grade is below 60.")
30      print("Bad")
```

```
 ----jGRASP exec: python Selection09.py


Enter Letter Grade  -->  A


You grade is 90 or above.
Excellent!


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

```
 ----jGRASP exec: python Selection09.py


Enter Letter Grade  -->  B


You grade is in the 80s.
Good


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

```
 ----jGRASP exec: python Selection09.py


Enter Letter Grade  -->  Q


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

Program **Selection09.py** works in most respects. As long as the user enters an **A**, **B**, **C**, **D** or **F**, the program executes just fine. The only issue occurs when the user does not enter one of these 5 letters. As you can see from the last output, when the user entered **Q** the program was not able to generate any output.

Program **Selection10.py**, in Figure 7.19, fixes this issue by adding the **else** command to the **if..elif** structure. In multi-way selection, **else** will catch any values that do not match any of the above cases. Now the program can display appropriate output for inappropriate input.

Figure 7.19

```
1 # Selection10.py
2 # This program demonstrates how the <else> command
3 # is used in Multi-Way Selection to deal with the
4 # case of a value that does not match any of the
5 # cases in your <if..elif> structure.
6
7 print()
8 grade = input("Enter Letter Grade  -->  ")
9 print()
10
11 if grade == 'A':
12    print("You grade is 90 or above.")
13    print("Excellent!")
14 elif grade == 'B':
15    print("You grade is in the 80s.")
16    print("Good")
17 elif grade == 'C':
18    print("You grade is in the 70s.")
19    print("Fair")
20 elif grade == 'D':
21    print("You grade is in the 60s.")
22    print("Poor")
23 elif grade == 'F':
24    print("You grade is below 60.")
25    print("Bad")
26 else:
27    print("You did not enter an A, B, C, D or F.")
28    print("Please re-run the program and try again.")
29
```

```
   ----jGRASP exec: python Selection10.py

Enter Letter Grade  -->  Q


You did not enter an A, B, C, D or F.
Please re-run the program and try again.


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

Multiple-Way Selection

General Syntax:

```
if first condition is True:
    execute first set of program statements
elif second condition is True:
    execute second set of program statements
:     :     :     :     :     :     :
else:  # when all above conditions are False
    execute default set of program statements
```

Specific Example:

```
if grade == 'A':
    points = 4.0
elif grade == 'B':
    points = 3.0
elif grade == 'C':
    points = 2.0
elif grade == 'D':
    points = 1.0
elif grade == 'F':
    points = 0.0
else:
    print("Error.  Please try again.")
```

7.8 Nested Selection

When you think of the term *nest* several things may come to your mind. You might be thinking of a bird's home or the place on top of a ship where the lookout stands. You might be thinking of *Russian Nesting Dolls*. You might even be thinking of your thermostat. For our purposes, the *Russian Nesting Dolls* make the best analogy. An example is shown in Figure 7.20.

Figure 7.20

With *Russian Nesting Dolls*, all if the dolls, except for the smallest, are hollow. The smallest doll fits inside the second smallest doll. The second smallest doll fits inside the third smallest doll and so on all the way up to the largest doll.

In computer science, control structures, like *Selection*, can be nested as well. We can place one selection structure inside another. Why would you want to do this? Is it even practical. The next couple program will explore this issue.

Let us return to the college interview analogy and take it a little further. One issue that may come up is the possibility of *financial aid*. Getting admitted to the college is still based on the SAT score, but getting financial aid will be based on your family's income.

Program **Selection11.py**, in Figure 7.21, has 2 separate **if..else** structures. The first deals with the SAT score and college admission. The second deals with the family income and financial aid. When you look at the first output the program seems to work. When you look at the second output, something it not right.

Figure 7.21

```
 1 # Selection11.py
 2 # This program has 2 separate <if..else> structures.
 3 # The first determines if a student is admitted to
 4 # the college based on his/her SAT score.
 5 # The second determines is that student qualifies
 6 # for financial aid based on his/her family income.
 7 # The problem with this program is that even if the
 8 # student is not admitted, it still asks about family
 9 # income and has the potential of telling a student who
10 # was not admitted that he/she qualifies for financial aid.
11
12 print()
13 sat = eval(input("Enter SAT score --> "))
14 print()
15
16 if sat >= 1100:
17     print("You are admitted.")
18     print("Orientation will start in June.")
19 else:
20     print("You are not admitted.")
21     print("Please try again when your SAT improves.")
22
23 print()
24 income = eval(input("Enter your family income --> "))
25 print()
26
27 if income < 20000:
28     print("You qualify for financial aid.")
29 else:
30     print("You do not qualify for financial aid.")
31
```

```
   ----jGRASP exec: python Selection11.py

Enter SAT score  -->  1500


You are admitted.
Orientation will start in June.


Enter your family income  -->  90000


You do not qualify for financial aid.


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

```
   ----jGRASP exec: python Selection11.py

Enter SAT score  -->  1000


You are not admitted.
Please try again when your SAT improves.


Enter your family income  -->  19000


You qualify for financial aid.


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

The first output looks fine. The student has a high enough SAT score to get admitted to the college. He does not qualify for financial aid because his family's income is too high. The second output has an issue. After the student is told that he is not admitted, he is then asked about his family's income and told he qualifies for financial aid… to a college to which he was not admitted. Awkward. What should have happened is that after the student was told that he was not admitted, the interview should have ended. Essentially the question about

financial aid depends on the question about acceptance. One condition depends on another. In this situation, the second question should never have been asked. Program **Selection12.py**, in Figure 7.22, implements this idea. You will notice one **if..else** structure nested inside the **if** part of another **if..else** structure. The *inner* **if..else** deals with the issue of financial aid. The *outer* **if...else** deal with acceptance to the college itself. Since the *inner* **if..else** is inside the **if** part of the *outer* structure, it can only be accessed if that acceptance condition is **True**; otherwise the question will never arise. When you execute this program, there are 3 possible outputs, each of which is shown. The first shows a student who is admitted and qualifies for financial aid. The second shows a student who is admitted and does not qualify for financial aid. The third shows a student who is not admitted. Note that financial aid is not even an issue in this case.

Figure 7.22

```
1 # Selection12.py
2 # This program fixes the issue of the previous program
3 # by "nesting" the second <if..else> structure inside
4 # the <if> part of the first.  Now, the "family income"
5 # question is only asked to students who are admitted.
6 # NOTE: Proper indentation is VERY important here.
7
8
9 print()
10 sat = eval(input("Enter SAT score  -->  "))
11 print()
12
13 if sat >= 1100:
14     print("You are admitted.")
15     print("Orientation will start in June.")
16     print()
17     income = eval(input("Enter your family income  -->  "))
18     print()
19     if income < 20000:
20         print("You qualify for financial aid.")
21     else:
22         print("You do not qualify for financial aid.")
23 else:
24     print("You are not admitted.")
25     print("Please try again when your SAT improves.")
26
```

```
   ----jGRASP exec: python Selection12.py

▶▶ Enter SAT score  -->  1350

   You are admitted.
   Orientation will start in June.

▶▶ Enter your family income  -->  18000

   You qualify for financial aid.

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

```
   ----jGRASP exec: python Selection12.py

▶▶ Enter SAT score  -->  1500

   You are admitted.
   Orientation will start in June.

▶▶ Enter your family income  -->  90000

   You do not qualify for financial aid.

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

```
   ----jGRASP exec: python Selection12.py

▶▶ Enter SAT score  -->  700

   You are not admitted.
   Please try again when your SAT improves.

   ----jGRASP: operation complete.
```
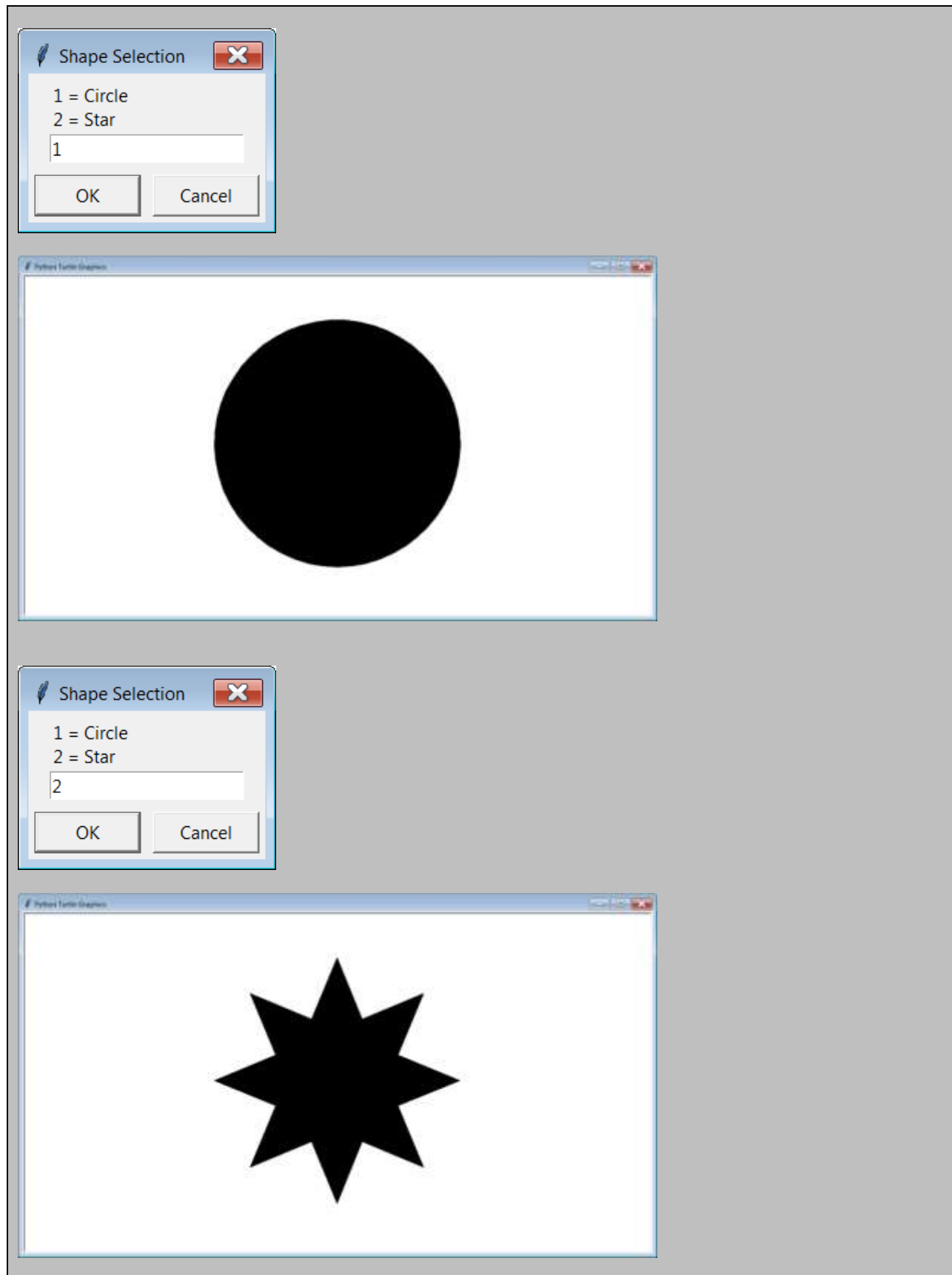
7.9 Combining Selection with Graphics

Control structures, like *selection*, are not limited to controlling text-based programs. They can "control" graphics programs as well.
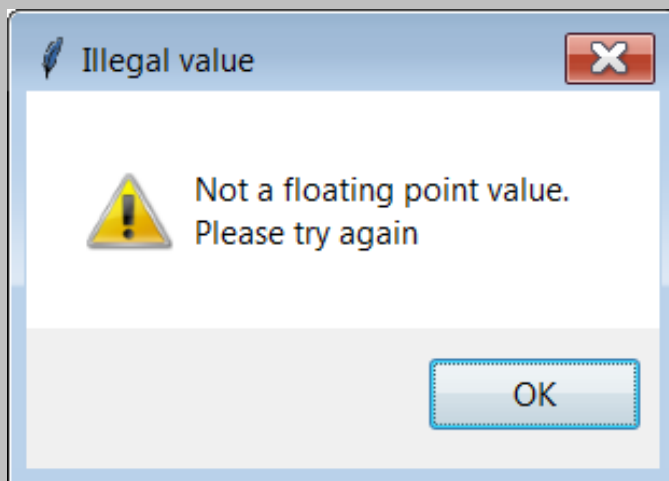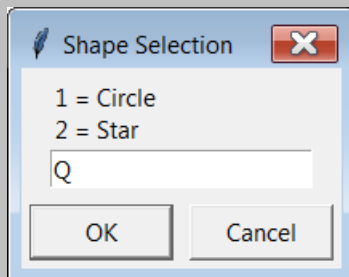
Program **Selection13.py**, in Figure 7.23, starts by asking the user to enter a **1** or a **2**. If you enter **1**, a circle is displayed. If you enter **2**, a star is displayed. If you enter anything other number, an error message is displayed.   If you enter something that is not a number, Python will display its own error message and force you to re-enter.

NOTE: This program example is a simplified version of what you will be doing in Lab 7B.

Figure 7.23

```
 1 # Selection13.py
 2 # This program demonstrates that selection can be used
 3 # to manipulate the output of a graphics program.
 4 # This is the very thing you will be doing in Lab 7B.
 5
 6
 7 from Graphics import *
 8
 9 beginGrfx(1300,700)
10
11 shapeNum = numinput("Shape Selection","1 = Circle \n2 = Star")
12
13 if shapeNum == 1:
14     fillCircle(650,350,250)
15 elif shapeNum == 2:
16     fillStar(650,350,250,8)
17 else:
18     drawString("You did not enter a 1 or a 2.",100,315,
"Arial",48,"bold")
19     drawString("Please try again.",100,465,"Arial",48,"bold")
20
21 endGrfx()
22
```

Shape Selection

1 = Circle
2 = Star

3

OK   Cancel

You did not enter a 1 or a 2.

Please try again.

Shape Selection

1 = Circle
2 = Star

Q

OK   Cancel

Illegal value

Not a floating point value.
Please try again

OK
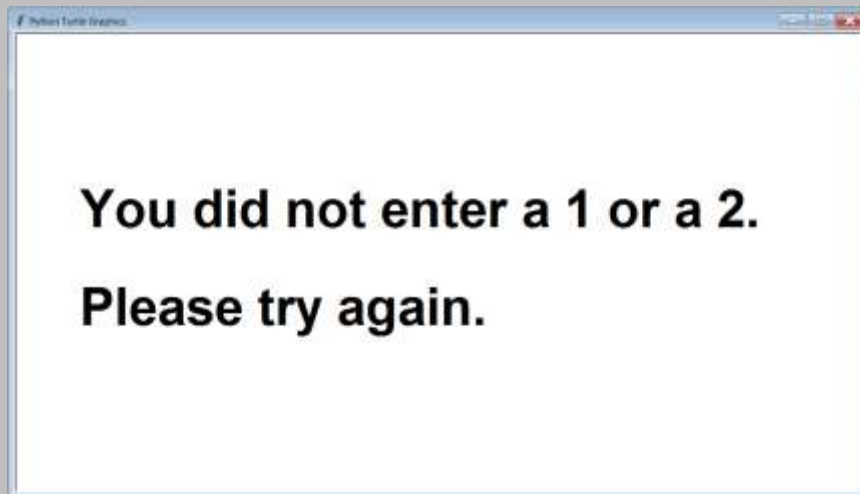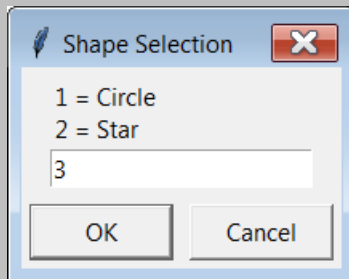
7.10 Formatting Numerical Output

The final section of this chapter will actually involve something completely different. It describes how to format the output of numbers. This will allow us to line up our numbers properly, add commas and even specify the number of places after the decimal point.

Program **NumberFormat01.py**, in Figure 7.24, displays 5 integers: **1**, **12**, **123**, **1234** and **12345**. When you look at the output, pay particular attention as to how the numbers line up. By default, anything displayed by **print** will be all the way on the left side of the screen. This means our numbers are *left-justified*.

Figure 7.24

```
 1 # NumberFormat01.py
 2 # This program demonstrates that by default
 3 # numbers are displayed "left-justified"
 4 # which means they do not line up by their
 5 # place value.
 6
 7
 8 print()
 9 print(1)
10 print(12)
11 print(123)
12 print(1234)
13 print(12345)
```

```
    ----jGRASP exec: python NumberFormat01.py


  1
  12
  123
  1234
  12345


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

But what if we want to line up the numbers differently, like by place value? What if we want the ones column and the tens column and the hundreds column to all line up? Isn't this exactly what you need to do if you want to add several numbers on paper?

Python does have a special **format** command. This command needs to be preceded with a specific string literal and a period. A variety of useful formatting features are available, depending on the string literal used. One of these features allows you to essentially *right-justify* your numbers. Program **NumberFormat02.py**, in Figure 7.25, displays the same 5 numbers as before, but this time each number is displayed with a **format** command. Each of the **format** commands is preceded with the string literal **"{:05}".** The **5** in this string literal specifies that the number will occupy 5 digits. If the number has less than 5 digits, the appropriate number of leading zeros will be concatenated to the beginning of the number to force it to be a 5-digit number.

Figure 7.25

```
1 # NumberFormat02.py
2 # This program demonstrates one way to properly line
3 # numbers up by place value using the <format> command.
4 # The 05 inside "{:05}" means each number will have
5 # enough 0s placed at the front of the number to force
6 # it to be displayed as a 5 digit number.
7
8
9 print()
10 print("{:05}".format(1))
11 print("{:05}".format(12))
12 print("{:05}".format(123))
13 print("{:05}".format(1234))
14 print("{:05}".format(12345))
```

```
    ----jGRASP exec: python NumberFormat02.py

 00001
 00012
 00123
 01234
 12345


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

What if you want to right-justify your numbers, but you do not want leading zeros? Well, leading zeros are not required. Just take the **0** out of the string literal and you will get leading spaces instead. This is demonstrated in program **NumberFormat03.py**, shown in Figure 7.26.

Figure 7.26

```
 1 # NumberFormat03.py
 2 # This program demonstrates that leading 0s are
 3 # not required to line up numbers by place value.
 4 # Just leave out the 0, and the number will be
 5 # displayed with leading spaces instead.
 6
 7
 8 print()
 9 print("{:5}".format(1))
10 print("{:5}".format(12))
11 print("{:5}".format(123))
12 print("{:5}".format(1234))
13 print("{:5}".format(12345))
14
```

```
    ----jGRASP exec: python NumberFormat03.py


        1
       12
      123
     1234
    12345


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

OK, so if the number does not have enough digits, the **format** will add enough leading zeroes or spaces to make things line up properly. Fine, but what if you have too many digits? Suppose I try to use the same "5-digit" format to display numbers greater than **99999**. Program **NumberFormat04.py**, in Figure 7.27, investigates this by adding 4 more lines to the previous program. Now we will

attempt to display numbers with 6, 7, 8 and 9 digits using with the same 5-digit format. So what will happen? Will the program crash? If it does not crash, how will it display the numbers? Will some of the digits be missing?

Figure 7.26

```
 1 # NumberFormat04.py
 2 # This program demonstrates what happens when the
 3 # format size for the number is not large enough.
 4 # If the format is not possible, it is simply
 5 # ignored.  There is no error message.
 6
 7
 8 print()
 9 print("{:5}".format(1))
10 print("{:5}".format(12))
11 print("{:5}".format(123))
12 print("{:5}".format(1234))
13 print("{:5}".format(12345))
14 print("{:5}".format(123456))
15 print("{:5}".format(1234567))
16 print("{:5}".format(12345678))
17 print("{:5}".format(123456789))
18
```

```
    ----jGRASP exec: python NumberFormat04.py


        1
       12
      123
     1234
    12345
    123456
    1234567
    12345678
    123456789


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

The result is somewhat anti-climactic. The program executes without any errors. All numbers are displayed in their entirety. For the last 4 numbers however, the 5-digit format was not possible, so it was simply ignored.

<div style="background:black;color:white">

format Command Reality

</div>

<div style="background:yellow">

While the **format** command can change the <u>appearance</u> of a number, it cannot change the <u>value</u> of a number. This is why it is OK to add leading zeros or spaces when a number does not have enough digits. It is also why it is not OK to remove digits when a number has too many.

</div>

Program **NumberFormat05.py**, in Figure 7.27, demonstrates that it is possible to display large numbers with the traditional commas displayed every third digit. For some reason, the justification seems to stop working for numbers with more than 7 digits, even though **9** was specified.

Figure 7.27

```
 1 # NumberFormat05.py
 2 # This program demonstrates how to add commas (,)
 3 # as a "thousand separator".  It may seem like
 4 # it stops working at 10 million.  The problem is
 5 # the commas count as digits.
 6
 7
 8 print()
 9 print("{:9,}".format(1))
10 print("{:9,}".format(12))
11 print("{:9,}".format(123))
12 print("{:9,}".format(1234))
13 print("{:9,}".format(12345))
14 print("{:9,}".format(123456))
15 print("{:9,}".format(1234567))
16 print("{:9,}".format(12345678))
17 print("{:9,}".format(123456789))
18
```

```
    ----jGRASP exec: python NumberFormat05.py


             1
            12
           123
         1,234
        12,345
       123,456
     1,234,567
    12,345,678
   123,456,789


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

The problem with program **NumberFormat05.py**, is demonstrated in Figure 7.28. What needs to be understood is that the number in **format**'s string literal does not specify the total number of digits, it specifies the total number of <u>characters</u> which includes digits, commas, and even the decimal point.

Figure 7.28

Number	1	2	3	,	4	5	6	,	7	8	9
Count	1	2	3	4	5	6	7	8	9	10	11

Using the information from Figure 7.28, program **NumberFormat06.py**, in Figure 7.29, fixes the problem of the previous program by changing the **9** in all of the **format** command's string literals to **11**.

Figure 7.29

```
1 # NumberFormat06.py
2 # This program fixes the problem of the previous
3 # program by increasing the total character count
4 # to accommodate the commas.
```

```
 5
 6
 7 print()
 8 print("{:11,}".format(1))
 9 print("{:11,}".format(12))
10 print("{:11,}".format(123))
11 print("{:11,}".format(1234))
12 print("{:11,}".format(12345))
13 print("{:11,}".format(123456))
14 print("{:11,}".format(1234567))
15 print("{:11,}".format(12345678))
16 print("{:11,}".format(123456789))
17
```

```
    ----jGRASP exec: python NumberFormat06.py


              1
             12
            123
          1,234
         12,345
        123,456
      1,234,567
     12,345,678
    123,456,789


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

Suppose you want commas, but you do not want leading spaces or leading zeros. That is also possible. Just do not put any number in the **format** string literal. This is demonstrated in program **NumberFormat07.py**, shown in Figure 7.30.

Figure 7.30

```
 1 # NumberFormat07.py
 2 # This program demonstrates that commas can
 3 # be used without leading zeros or spaces.
 4
 5
 6 print()
 7 print("{:,}".format(1))
 8 print("{:,}".format(12))
 9 print("{:,}".format(123))
10 print("{:,}".format(1234))
11 print("{:,}".format(12345))
12 print("{:,}".format(123456))
13 print("{:,}".format(1234567))
14 print("{:,}".format(12345678))
15 print("{:,}".format(123456789))
16
```

```
    ----jGRASP exec: python NumberFormat07.py


  1
  12
  123
  1,234
  12,345
  123,456
  1,234,567
  12,345,678
  123,456,789


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

Formatting Real Number Output

We have looked at several example of formatting integer output. Now we will see some examples that involve real numbers. To start off, we are going to repeat a program from several chapters ago. Program **NumberFormat08.py**, in Figure 7.31, repeats program **Documentation02.py** from Chapter IV. This is a program that computes and displays gross pay, deductions and net pay. While the program works, the values displayed, which are supposed to represent money, are not rounded to the nearest penny.

Figure 7.31

```
 1 # NumberFormat08.py
 2 # This program repeats Documentation02.py from Chapter 4.
 3 # One problem with the program still has is that the
 4 # numbers which are displayed represent money, but they
 5 # are not rounded to 2 decimal places.
 6
 7
 8 hoursWorked = 35
 9 hourlyRate = 8.75
10 grossPay = hoursWorked * hourlyRate
11 deductions = grossPay * 0.29
12 netPay = grossPay - deductions
13
14 print()
15 print("Hours Worked: ",hoursWorked)
16 print("Hourly Rate:  ",hourlyRate)
17 print("Gross Pay:    ",grossPay)
18 print("Deductions:   ",deductions)
19 print("Net Pay:      ",netPay)
```

```
    ----jGRASP exec: python NumberFormat08.py

  Hours Worked:  35
  Hourly Rate:   8.75
  Gross Pay:     306.25
  Deductions:    88.8125
  Net Pay:       217.4375

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

Program **NumberFormat09.py**, in Figure 7.32, demonstrates how to format real numbers. Remember that "real numbers" are also called *floating point numbers*, hence the **f** in the string literal. The string literal now also contains an additional number. The first number still specifies the total number of characters for the number. The second number – which is after a period – specifies the total number of digits that follow the decimal point. In the output you will see that trailing zeros are added when a number has less than 2 fractional digits.

Figure 7.32

```
1 # NumberFormat09.py
2 # This program demonstrates how to format real number
3 # output with the <format> command.  The f inside
4 # "{:6.2f}" means this is a "floating-point number"
5 # which is the same thing as a "real number".
6 # The 6 indicates the entire number will be 6 characters
7 # long, including the decimal point.  The 2 indicates
8 # there will be 2 digits after the decimal point.
9
10
11 hoursWorked = 35
12 hourlyRate = 8.75
13 grossPay = hoursWorked * hourlyRate
14 deductions = grossPay * 0.29
15 netPay = grossPay - deductions
16
17 print()
18 print("Hours Worked:","{:6.2f}".format(hoursWorked))
19 print("Hourly Rate: ","{:6.2f}".format(hourlyRate))
20 print("Gross Pay:   ","{:6.2f}".format(grossPay))
21 print("Deductions:  ","{:6.2f}".format(deductions))
22 print("Net Pay:     ","{:6.2f}".format(netPay))
```

```
    ----jGRASP exec: python NumberFormat09.py

  Hours Worked:  35.00
  Hourly Rate:    8.75
  Gross Pay:    306.25
  Deductions:    88.81
  Net Pay:      217.44

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

We saw earlier that you do not need leading spaces or zeros when formatting commas. The same thing is true for formatting digits past the decimal point. Leading spaces or leading zeros are never actually required. If you do not want them, just leave the first number out of the **format** string literal. This is demonstrated in program **NumberFormat10.py**, shown in Figure 7.33.

Figure 7.33

```
 1 # NumberFormat10.py
 2 # This program demonstrates how to format
 3 # real numbers without leading spaces.
 4
 5
 6 hoursWorked = 35
 7 hourlyRate = 8.75
 8 grossPay = hoursWorked * hourlyRate
 9 deductions = grossPay * 0.29
10 netPay = grossPay - deductions
11
12 print()
13 print("Hours Worked:","{:.2f}".format(hoursWorked))
14 print("Hourly Rate: ","{:.2f}".format(hourlyRate))
15 print("Gross Pay:   ","{:.2f}".format(grossPay))
16 print("Deductions:  ","{:.2f}".format(deductions))
17 print("Net Pay:     ","{:.2f}".format(netPay))
18
```

```
    ----jGRASP exec: python NumberFormat10.py

  Hours Worked: 35.00
  Hourly Rate:  8.75
  Gross Pay:    306.25
  Deductions:   88.81
  Net Pay:      217.44

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

The final program of this chapter combines all of the different formatting features in one program. There are leading spaces, rounded digits past the decimal point, commas and even dollar signs. This is all demonstrated in program **NumberFormat11.py**, shown in Figure 7.34.

Figure 7.34

```
 1 # NumberFormat11.py
 2 # This program demonstrates that real numbers can be
 3 # formatted with leading spaces, rounded digits past
 4 # the decimal point, commas, and even a dollar $ign.
 5
 6 hoursWorked = 50
 7 hourlyRate = 199.98
 8 grossPay = hoursWorked * hourlyRate
 9 deductions = grossPay * 0.29
10 netPay = grossPay - deductions
11
12 print()
13 print("Hours Worked:",hoursWorked)
14 print("Hourly Rate: ","${:8,.2f}".format(hourlyRate))
15 print("Gross Pay:   ","${:8,.2f}".format(grossPay))
16 print("Deductions:  ","${:8,.2f}".format(deductions))
17 print("Net Pay:     ","${:8,.2f}".format(netPay))
18
```

```
    ----jGRASP exec: python NumberFormat11.py

  Hours Worked: 50
  Hourly Rate:  $  199.98
  Gross Pay:    $9,999.00
  Deductions:   $2,899.71
  Net Pay:      $7,099.29

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