Chapter XI

Boolean Logic

Chapter XI Topics

11.1 Introduction

This chapter will be a departure from the usual format. While there are several Python programs in the second half of this chapter, the first half barely mentions Python. Why is this? Isn't Python what we learn in computer science? Well, yes and no. In an introductory, first year course like Computer Science 1 or Computer Science 1 – Honors, what you learn, first and foremost, is computer science. There are several components in a computer science class. One is *The History of Computers*. Another is *How Computers Work*. The largest component, by far is *Programming*. While you may learn a little HTML, the primary language used to teach the programming component is Python.

Some time ago you learned about different kinds of errors. There are *Compile Errors*, *Run-time Errors* and *Logic Errors*. Of the three, *Logic Errors* are the most frustrating. Writing programs in Python, or in any other language, is much more than just learning several commands. You need to be able to combine the right commands in the right sequence to create a program that works. This requires logical thinking. For this reason, another major component of introductory computer science is to learn the logic and problem solving skills that are part of programming.

In this chapter we will investigate *Boolean Logic* concepts. The information in this chapter will be very beneficial in understanding control structures. In an earlier chapter you were introduced to control structures. In this chapter you will look at control structures again, and discover that there are many complex situations that can easily cause confusion. A good understanding of Boolean Logic will help tremendously in writing program code that is logically correct. On the other hand, a weak or non-existent understanding about Boolean principles can cause program problems without a clue why the program executed incorrectly.

More than 160 years ago, there was a mathematician, *George Boole*, who founded a branch of mathematics called *Boolean Algebra*. In *Boolean Algebra*, everything is either **True** or **False**. Statements that are either **True** or **False** are called *Boolean Statements*. The conditions you used with selection and repetition back in chapters VII and VIII were all *Boolean Statements*. This chapter could very well have been called *Boolean Algebra*. The biggest reason it is not is that many concepts that students have learned in Algebra do not apply to Boolean Algebra. At any rate, the title matters very little. Let us get down to business and learn some serious Boolean Logic. If you like mathematics, you will be happy to increase your mathematical horizon. If you do not like mathematics, you should be happy that you are learning something that will be very beneficial in understanding computer science.

11.2 What is a Boolean Statement?

A good starting point is to look at a variety of English sentences and determine if these sentences are Boolean statements or not. So, what are the criteria for a Boolean statement? The sentence, statement, condition, whatever, must be **True** or **False**. Questions, ambiguities, opinions and arguments are not Boolean statements. You can see why this branch of mathematics has a major impact on computer science. The basis of processing data is the binary system of *on* and *off,* which certainly sounds a bunch like **True** or **False**. Each one of the following five statements is a Boolean statement.

A mile is longer than a kilometer.
July and August both have the same number of days.
A pound of feathers is lighter than a pound of lead.
The Moon is larger than the Sun.
New York City has more people than Baltimore.

The five sentences may not have seemed very Boolean to you. Let us look at the sentences again, translate them into brief logic statements, and indicate whether the statements are true of false. Some Python relational operators are used for the Boolean statements to help clarify the meaning.

English Sentence	Boolean Statement	T/F
A mile is longer than a kilometer.	`Mile > Kilometer`	True
July and August have the same days.	`JulDays == AugDays`	True
A pound of feathers is lighter than a pound of lead.	`PoundF < PoundL`	False
The Moon is larger than the Sun.	`MoonSize > SunSize`	False
New York City has more people than Baltimore.	`NYPop > BaltPop`	True

Sentences are not always so short and straightforward. Frequently there are multiple conditions in one statement. Special rules need to be followed to determine if the entire statement is true or false. Consider the following sentences with compound conditions.

She is a computer science teacher and she is a math teacher.

The number is odd or the number is even.

Enter again if gender is not male or gender is not female.

Employment requires a CPA and five years experience.

The same sentences converted into Boolean statements are:

```
(She == CSTeacher) and (She == MathTeacher)
(Number % 2 == 1) or (Number % 2 == 0)
(Gender != Male) or (Gender != Female)
(CPA == 'Y') and (YrExp >= 5)
```

These statements are certainly more complex than the earlier examples. Keep in mind that Boolean statements can be made considerably more complicated. It is the intention of this chapter to get a firm grip on Boolean logic.

11.3 Boolean Operators

In this section we will look at three different Boolean operators. You know how to handle arithmetic operators (+, -, *, /, //, %, and **). Addition, subtraction, multiplication, 3 kinds of division and exponents. There are also a set of Boolean operators with their own set of rules. The rules of Boolean operators can be conveniently displayed in a *Truth Table*. This is a table, which shows all of the possible combinations of a Boolean statement and indicates the value (true or false) of each statement.

In the truth tables that follow, a single letter indicates a single, simple Boolean condition. Such a condition is either **True** or **False**. Boolean statement **A** is true or false. Likewise Boolean statement **B** is true or false. The truth tables will show the results of Boolean statements that use both **A** and **B** with a variety of Boolean operators. Employment requirements will be used to explain the logic of each truth table. In each case imagine that an accountant needs to be hired. Condition **A** determines if the applicant has a **Degree** and condition **B** determines if the applicant has at least five years' experience.

Boolean OR

The **or** Operator		
A	B	A **or** B
T	T	T
T	F	T
F	T	T
F	F	F

Notice that two conditions have four possible combinations. It is important that you know the results for each type of combination.

In this case the employment analogy requires a **Degree OR Experience**. This requirement is quite relaxed. You have a Degree, fine. You have Experience, that's also fine. You have both, definitely fine. You have neither, that's a problem.

Boolean AND

The **and** Operator		
A	B	A **and** B
T	T	T
T	F	F
F	T	F
F	F	F

Now employment requires a **Degree AND Experience**. This requirement is much more demanding than the **or** operator. You have a Degree, fine, provided you also have Experience. If you have only one qualification, or the other qualification, that is not good enough. If you have neither qualification, forget showing up.

Boolean NOT

The **not** Operator	
A	**not** A
T	F
F	T

This section will finish with the simplest Boolean operator, **not**. This operator takes the condition that follows and changes **True** to **False** or **False** to **True**. There are special rules that need to be followed when a complex Boolean statement is used with **not**. Such rules will be explained later in the chapter. Right now we want to understand the simple truth table shown earlier. In English we need to use "double negative" sentences to create an appropriate analogy. I can say "It is **not True** that Tom Smith is valedictorian." This statement results in Tom not being Valedictorian. On the other hand, if I say "It is **not True** that Tom Smith is **not** the Valedictorian," then Tom is the Valedictorian.

11.4 Venn Diagrams and Boolean Algebra

Venn diagrams are useful tools for teaching *Set Theory*. With a rectangle to represent the *universal set* and a group of circles, representing individual sets within the rectangle, it is possible to visually demonstrate many Set Theory concepts. You probably learned about Venn diagrams in one or more mathematics classes. Boolean Algebra would not have been mentioned in any of these classes and you learned terms like *union* and *intersection*. The relationship between Boolean Algebra and Set Theory is very close, and the same visual benefits of Venn diagrams can apply to Boolean Algebra, as well as Set Theory.

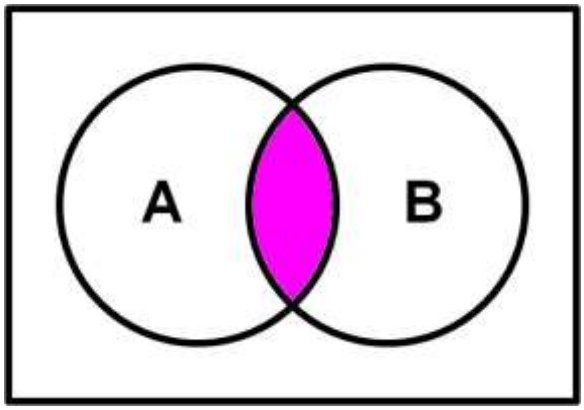
Boolean Algebra Logical Operators and Venn Diagrams

The Boolean Algebra logical operators (**and**, **or**, **not**) can be visually demonstrated with Venn Diagrams.

Venn Diagram 1 shows the intersection of sets A and B. Consider one of the Boolean examples of the past. You have an employment requirement that applicants must have a college degree and they must also have five years of work experience. Now imagine that **Set A** represents people with a college degree, and **Set B** represents people with at least five years of work experience. In this case the shaded section is the result of saying **A and B** in Boolean Algebra or the *intersection* of **Set A** and **Set B** in Set Theory.

Venn Diagram 1

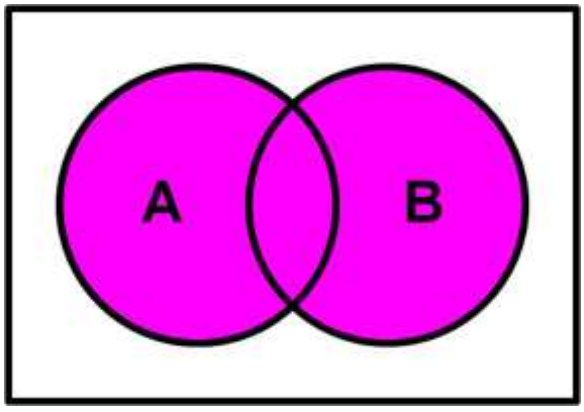
A and B
A * B



This time our business will hire people that have either a college degree or at least five years of experience. With the logical **or** there are now many more people qualified to apply for the job. We now can interview everybody in **Set A**, which are the college degree people as well as everybody in **Set B**, which are the experience people. In Boolean Algebra this is stated with the expression **A or B**. In set theory this is called the *union* of **Set A** and **Set B**, which is illustrated with Venn Diagram 2.

Venn Diagram 2

A or B
A + B

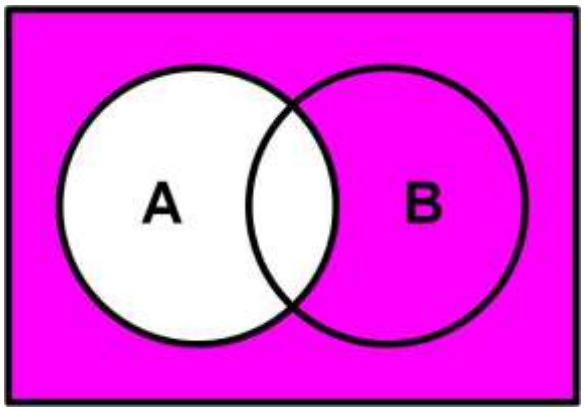


The reason for using Venn diagrams is to help illustrate Boolean Algebra concepts. With small expressions like **A and B** and **A or B** you may not see much need for using Venn diagrams. However, more complex expressions can be assisted with the help of set theory. Venn diagrams are particularly helpful to demonstrate that two different expressions are equivalent.

Venn diagrams 3 and 4 illustrate the **not** operator, which is identical in Boolean Algebra, as well as in Set Theory. Expressions **not(A)** and **not(B)** are shown in the next two diagrams.

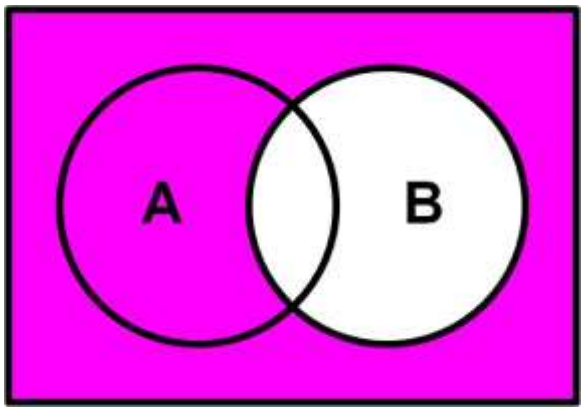
Venn Diagram 3

not(A)
~A



Venn Diagram 4

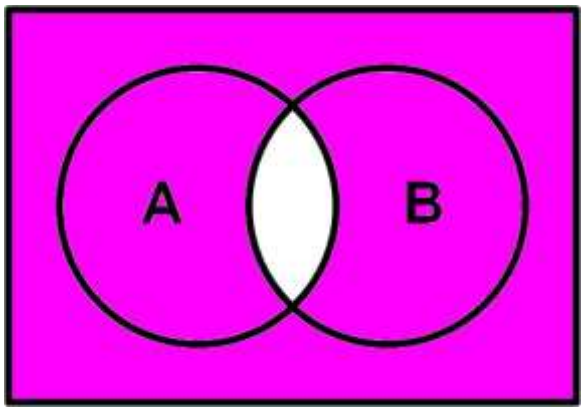
not(B)
~B



The remaining Venn diagrams will show a variety of compound Boolean expressions. Venn diagram 5 illustrates the negation of the *intersection* of **A** and **B**, which is **not(A and B)** in Boolean Algebra.

Venn Diagram 5

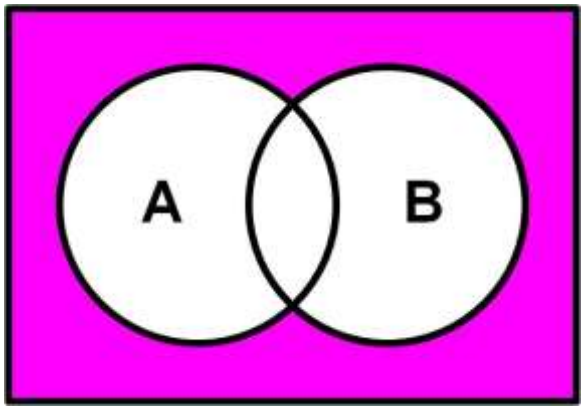
not(A and B)
~(A * B)



Venn diagram 6 is very similar to the previous example, but this illustrates the negation of the *union* of **Set A** and **Set B**. You can compare diagrams 6 and 7 with the earlier diagrams 1 and 2 to notice the pattern of negation.

Venn Diagram 6

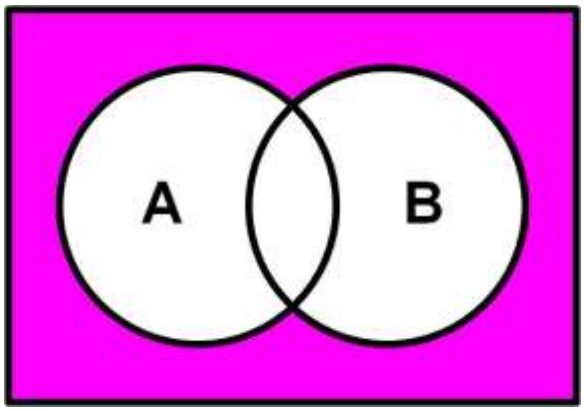
not(A or B)
~(A + B)



One of the Boolean Algebra Laws is *DeMorgan's Law*, which states that the expression **not(A or B)** is equivalent to the expression **not(A) and not(B)**. Proving this law can be done with Venn diagrams. Note that Venn diagram 6 shows **not(A or B)** and Venn diagram 7 shows **not(A) and not(B)**. More importantly, note that both Venn diagrams 6 and 7 illustrate the exact same result.

Venn Diagram 7

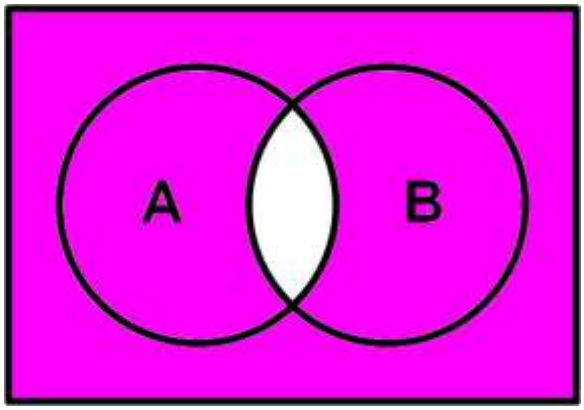
not(A) and not(B)
~A * ~B



The second part of *DeMorgan's Law* states that **not(A and B)** is equivalent to **not(A) or not(B)**. This law is proven by Venn diagrams 5 and 8. If any Boolean Algebra laws confuse you, create Venn Diagrams.

Venn Diagram 8

not(A) or not(B)
~A + ~B



DeMorgan's Law

not(A and B) = not A **or** not B

not(A or B) = not A **and** not B

11.5 Boolean Values and Variables

In Chapter VII you saw how *selection control structures* assisted in controlling program flow. The reserved word **if**, combined with a variety of conditions, impacts the outcome of a program. You were told that these conditions were **True** or **False**. Everything that you have learned about computers drums the recurring theme that data processing involves two states. There is *on* and *off*, *1* and *0*, and there is **True** and **False**. Consider program **Boolean01.py**, in Figure 11.1.

Figure 11.1

```
 1 # Boolean01.py
 2 # This program demonstrates a Boolean expression
 3 # being used in an <if> statement.
 4
 5
 6
 7 x = 10
 8
 9 print()
10 if x == 10:
11    print("True")
12 else:
13    print("False")
14
```

```
15 print()
16 if x == 5:
17    print("True")
18 else:
19    print("False")
```

```
   ----jGRASP exec: python Boolean01.py

  True

  False

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

Program **Boolean01.py** does not show any new information. It does show something that is a bit redundant. The program will display **True** if it is "true" that **x == 10**. The program will display **False** if it is "false" that **x == 10**. Program **Boolean02.py** in Figure 11.2 does the exact same thing, but is much shorter. The program simply displays the *value* of the Boolean expressions **x == 10** and **x == 5**.

Figure 11.2

```
 1 # Boolean02.py
 2 # This program demonstrates that conditional
 3 # statements have a Boolean value which is
 4 # either <True> or <False> and these values
 5 # can be displayed.
 6
 7
 8 x = 10
 9
10 print()
11 print(x == 10)
12 print()
13 print(x == 5)
```

```
----jGRASP exec: python Boolean02.py

True

False

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

It is helpful in programming logic to test if something is true. You test for correct password entries, objects that are found, values that are equivalent and so on. Whenever a **True** condition exists, a value of **1** can be assigned in the program to some integer variable. However, programmers prefer greater readability and want to assign **True** and F**alse**. Modern program languages handle this requirement with a special Boolean data type that only has two possible values: **True** or **False**.

If you are like most students, you will find a data type with two possible values less than handy. That is OK, a Boolean data type will grow on you and there are many applications where both program logic and program readability are served well with such a simple data type.

The next 3 programs all do the same thing. We will look at the output first in Figure 11.3 and then look at the code for each program:

Figure 11.3

```
----jGRASP exec: python Boolean03.py

What is the GCF of 120 and 108?  -->  1

What is the GCF of 120 and 108?  -->  2

What is the GCF of 120 and 108?  -->  3

What is the GCF of 120 and 108?  -->  12

You answered it correctly after 4 attempts.

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

The program asks a question, and keeps repeating the question until the user enters the correct answer. In this case, the correct answer is **12**.

In program **Boolean03.py**, shown in Figure 11.4, this is accomplished with the *repetition control structure*: **while gcf != 12:**. The between the **while** and the colon (**gcf != 12**) is a Boolean statement which evaluates to **True** or **False**.

Figure 11.4

```
 1 # Boolean03.py
 2 # This program shows a Boolean expression
 3 # being used in a <while> loop.
 4
 5
 6 gcf = 0
 7 attempt = 0
 8
 9 while gcf != 12:
10    attempt += 1
11    gcf = eval(input("\nWhat is the GCF of 120 and 108?  -->  "))
12
13 if attempt == 1:
14    print("\nYou got it on the first try!")
15 else:
16    print("\nYou answered it correctly after",attempt,"attempts.")
17
```

Program **Boolean04.py**, in Figure 11.5, is made a little more readable by adding a Boolean variable. The Boolean variable **correct** is used to assign **True** or **False** based on the entry of the answer. Yes the same result can be achieved without the Boolean variable, but the program has gained readability and the intention of the source is clearer.

Figure 11.5

```
 1 # Boolean04.py
 2 # This program demonstrates a practical use of the <bool>
 3 # data type, which only has two values, <True> and <False>.
 4 # NOTE: Boolean variables add readability to programs.
 5
```

```
 6
 7 gcf = 0
 8 attempt = 0
 9 correct = False
10
11 while not correct:
12    attempt += 1
13    gcf = eval(input("\nWhat is the GCF of 120 and 108?  -->  "))
14    if gcf == 12:
15       correct = True
16    else:
17       correct = False
18
19 if attempt == 1:
20    print("\nYou got it on the first try!")
21 else:
22    print("\nYou answered it correctly after",attempt,"attempts.")
```

Program **Boolean05.py**, in Figure 11.6 accomplishes the exact same task as the previous program with some simplified code. Note that the **if ... else** statement is gone. In its place is the rather "bizarre" looking

```
correct = gcf == 12
```

This statement is both proper Python syntax and the preferred way to handle two possible outcomes. Yes, the **if...else** works just fine, but the shorter approach can make excellent sense. For starters, **correct** is a Boolean data type. This means it can only take on the value of **True** or **False**. You found out earlier that conditional statements have a value, which is **True** or **False**. This value can be assigned to a Boolean variable. The program output is exactly identical to the output of the previous program.

Figure 11.6

```
1 # Boolean05.py
2 # This program executes in the same manner as Boolean04.py.
3 # The abbreviated Boolean assignment statement is used in
4 # place of the longer <if...else> expression.
5
```

```
 6
 7
 8 gcf = 0
 9 attempt = 0
10 correct = False
11
12 while not correct:
13    attempt += 1
14    gcf = eval(input("\nWhat is the GCF of 120 and 108?  -->  "))
15    correct = gcf == 12
16
17 if attempt == 1:
18    print("\nYou got it on the first try!")
19 else:
20    print("\nYou answered it correctly after",attempt,"attempts.")
21
```

If you still find program **Boolean05.py** confusing, maybe the next program will help you. Program **Boolean06.py**, in Figure 11.7 stores the values of 4 different expressions in 4 different variables. On line 9, we see the integer expression **23 + 45** being stored in the integer variable **x**. On line 11, we see the real number expression **6.33 * 6.31 * 6.23** being stored in the real number variable **y**. On line 13, we see the string expression "**John** " + "**Smith**" being stored in the string variable **name**. There is nothing new in these 3 assignment statements. The point being made here is what is done on line 16, when the Boolean expression **average >= 70** is stored in the Boolean variable **passing**, is absolutely no different.

Figure 11.7

```
 1 # Boolean06.py
 2 # This program assigns the values of 4 different
 3 # expressions to 4 different variables.
 4 # The intent is to show that storing the values
 5 # of Boolean expressions is no different than
 6 # storing the values of any other expressions.
 7
 8
 9 x = 23 + 45                  # Integer expression
10
11 y = 6.33 * 6.31 * 6.23       # Real Number expression
12
```

```
13 name = "John " + "Smith"    # String expression
14
15 average = 85
16 passing = average >= 70     # Boolean expression
17
18 print()
19 print("Integer value:    ",x)
20 print("Real Number value:",y)
21 print("String value:     ",name)
22 print("Boolean value:    ",passing)
23
```

```
----jGRASP exec: python Boolean06.py

Integer value:      68
Real Number value: 248.840529
String value:       John Smith
Boolean value:      True

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

11.6 Compound Conditions

You are about to use *Boolean Logic* in actual computer programs. Many situations in life present compound conditions and we will start by looking at three programs that decide if a person should be hired. The hiring-criteria are based on years of **education** as well as years of work **experience**.

The "Nice Boss" OR Example

Program **CompoundCondition01.py**, in Figure 11.8, uses a logical **or** to decide if somebody should be hired. In this case somebody is qualified if they have the required education *OR* they have the required work experience. We call this boss the *Nice Boss* because he is flexible. The user enters the number of years of **education** and work **experience**. Then there is an **if** statement with a *Compound Condition*. It says

```
if education >= 16 or experience >= 5:
```

As we saw in the earlier *Truth Table*, **or** will give a **True** result if either or both conditions is/are **True**. The only time **or** gives a **False** result is when both conditions are **False**.

Figure 11.8

```
 1 # CompoundCondition01.py
 2 # This program demonstrates compound decisions
 3 # with the logical <or> operator.
 4
 5
 6 print()
 7 education  = eval(input("Enter years of education   -->  "))
 8 experience = eval(input("Enter years of experience  -->  "))
 9 print()
10
11 if education >= 16 or experience >= 5:
12    print("You are hired!")
13 else:
14    print("You are not qualified.")
15
```

```
   ----jGRASP exec: python CompoundCondition01.py

▶▶ Enter years of education   -->  16
▶▶ Enter years of experience  -->  0

   You are hired!

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

```
 ----jGRASP exec: python CompoundCondition01.py

Enter years of education   -->  13
Enter years of experience  -->  7

You are hired!

 ----jGRASP: operation complete.

 ----jGRASP exec: python CompoundCondition01.py

Enter years of education   -->  18
Enter years of experience  -->  10

You are hired!

 ----jGRASP: operation complete.

 ----jGRASP exec: python CompoundCondition01.py

Enter years of education   -->  12
Enter years of experience  -->  3

You are not qualified.

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

The "Picky Boss" AND Example

Some bosses are not as flexible as the *Nice Boss.* Some are very picky. Program **CompoundCondition02.py**, in Figure 11.9**,** demonstrates the *Picky Boss*. The *Picky Boss* only wants the best people working for him. He only will hire people with BOTH **education and experience**. As with the previous program, the user enters the number of years of **education** and work **experience**. There is another **if** statement with a *Compound Condition* that is only slightly different from the previous program:

```
if education >= 16 and experience >= 5:
```

As we saw in the earlier *Truth Table*, **and** will only give a **True** result if both conditions are **True**. If either condition is **False**, **and** will give a **False** result.

Figure 11.9

```
 1 # CompoundCondition02.py
 2 # This program demonstrates compound decisions
 3 # with the logical <and> operator.
 4
 5
 6 print()
 7 education  = eval(input("Enter years of education   -->  "))
 8 experience = eval(input("Enter years of experience  -->  "))
 9 print()
10
11 if education >= 16 and experience >= 5:
12    print("You are hired!")
13 else:
14    print("You are not qualified.")
15
```

```
   ----jGRASP exec: python CompoundCondition02.py

>> Enter years of education   -->  16
>> Enter years of experience  -->  0

   You are not qualified.

   ----jGRASP: operation complete.

   ----jGRASP exec: python CompoundCondition02.py

>> Enter years of education   -->  13
>> Enter years of experience  -->  7

   You are not qualified.

   ----jGRASP: operation complete.

   ----jGRASP exec: python CompoundCondition02.py

>> Enter years of education   -->  18
```

```
Enter years of experience  -->  10

You are hired!

 ----jGRASP: operation complete.

 ----jGRASP exec: python CompoundCondition02.py

Enter years of education   -->  12
Enter years of experience  -->  3

You are not qualified.

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

The "Crazy Boss" NOT Example

We now come to the *Crazy Boss*. The *Crazy Boss* is completely certifiable because he will only hire people who have no qualifications what so ever. In other words, he will only hire people with *neither* **education** *nor* **experience**. This is the exact opposite of the *Nice Boss*. If you do not understand why the *Crazy Boss* is the opposite of the *Nice Boss* then compare the outputs of their 2 programs. You will see that everyone who the *Nice Boss* will hire is considered "not qualified" by the *Crazy Boss* and vice versa.

The *Compound Condition* for this program will involve using **or**. We also need a **not** to change the value of the *Compound Condition* to its opposite. Program **CompoundCondition03.py**, in Figure 11.10, will demonstrate this. Note the use of extra parentheses in the *Compound Condition* statement. They are necessary because we want the **not** to apply to the entire *Compound Condition*, not just the first part.

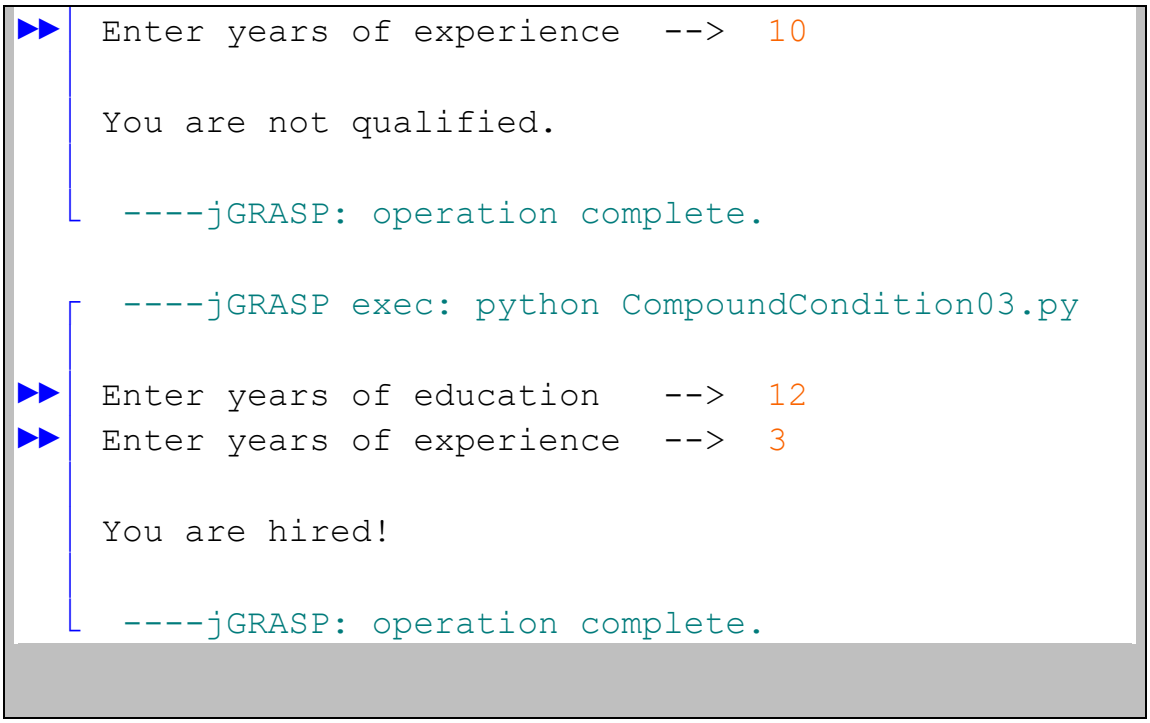
```
if not(education >= 16 or experience >= 5):
```

As we saw in the earlier *Truth Table*, **not** will simply give you the opposite of whatever you give it. The opposite of **True** is **False**. The opposite of **False** is **True**.

Figure 11.10

```
 1 # CompoundCondition03.py
 2 # This program demonstrates the logical <not> operator.
 3 # NOTE: This also requires (parentheses).
 4
 5
 6 print()
 7 education  = eval(input("Enter years of education   -->  "))
 8 experience = eval(input("Enter years of experience  -->  "))
 9 print()
10
11 if not(education >= 16 or experience >= 5):
12    print("You are hired!")
13 else:
14    print("You are not qualified.")
15
```

```
   ----jGRASP exec: python CompoundCondition03.py

▶▶ Enter years of education   -->  16
▶▶ Enter years of experience  -->  0

   You are not qualified.

   ----jGRASP: operation complete.

   ----jGRASP exec: python CompoundCondition03.py

▶▶ Enter years of education   -->  13
▶▶ Enter years of experience  -->  7

   You are not qualified.

   ----jGRASP: operation complete.

   ----jGRASP exec: python CompoundCondition03.py

▶▶ Enter years of education   -->  18
```



11.7 Ranges

There are times when you need to see if a number falls within a certain *range* of numbers. The *teen*age years, for example, are between thir*teen* and nine*teen*. In some languages we could simply specify something like **13..19** and the computer would know you mean all numbers between and including 13 and 19. Python is not one of those languages; however, if we combine the **and** Compound Condition with a couple relational operators we can specify the same *range* of numbers in this way:

```
if age >= 13 and age <= 19:
```

Program **Ranges01.py**, in Figure 11.11, will determine if someone is the proper age to drive. Assume the proper age range is from **16** through **90**. Note the use of the **if** statement:

```
if age >= 16 and age <= 90:
```

Figure 11.11

```
1  # Ranges01.py
2  # This program determines if an age is between 16 and 90.
3  # The range is defined using a compound condition with <and>.
4  # NOTE: This range logic works in any programming language.
5
6
7  print()
8  age  = eval(input("How old are you?   -->  "))
9  print()
10
11 if age >= 16 and age <= 90:
12    print("You are the proper age to drive.")
13 else:
14    print("You are not the proper age to drive.")
15
```

```
  ----jGRASP exec: python Ranges01.py

How old are you?   -->  16

You are the proper age to drive.

  ----jGRASP: operation complete.

  ----jGRASP exec: python Ranges01.py

How old are you?   -->  50

You are the proper age to drive.

  ----jGRASP: operation complete.

  ----jGRASP exec: python Ranges01.py

How old are you?   -->  13
```

```
You are not the proper age to drive.

 ----jGRASP: operation complete.

 ----jGRASP exec: python Ranges01.py

How old are you?   -->  107

You are not the proper age to drive.

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

Program **Ranges02.py**, in Figure 11.12, shows a shortcut way to do the same thing as the previous program. In your math class, if you wanted to indicate that some variable, **x**, was between **16** and **90**, you could write **16 ≤ x ≤ 90**. Python essentially allows you to write the same thing. Note the use of this **if** statement:

```
if 16 <= age <= 90:
```

This is less code to type and it will give the exact same output as if we typed out the long Compound Condition. You may wonder why I included the previous program example which required more code. Remember when you were shown how to swap the values of 2 variables? First you were shown the long way by using a temporary variable. Then you were shown the shortcut way using *Simultaneous Assignment*. Why did I show the long way back then? It was and is because *Simultaneous Assignment* only works in Python. If you are working in any other language, you need to do things the long way which works in any language. The same thing is true here. This nifty shortcut only works in Python. If you need to specify a range in any other language, you can use a Compound Condition with **and**.

Figure 11.12

```
# Ranges02.py
# This program demonstrates a shortcut way to handle ranges.
# NOTE: This range shortcut only works in Python.


print()
age  = eval(input("How old are you?   -->  "))
print()

if 16 <= age <= 90:
   print("You are the proper age to drive.")
else:
   print("You are not the proper age to drive.")
```

Program **Ranges03.py**, in Figure 11.13, shows a completely different way to do ranges using an **if..elif..else** structure.

Figure 11.13

```
# Ranges03.py
# This program defines several different ranges
# using an <if..elif..else> structure.


print()
gpa = eval(input("What is your GPA?  -->  "))
print()

if gpa >= 3.9:
   print("Summa Cum Laude")
elif gpa >= 3.75:
   print("Magna Cum Laude")
elif gpa >= 3.5:
   print("Cum Laude")
elif gpa >= 2.65:
   print("Graduate without Honors")
else:
   print("Did not graduate")
```

```
 ----jGRASP exec: python Ranges03.py
What is your GPA?  -->  4.0
Summa Cum Laude
 ----jGRASP: operation complete.

 ----jGRASP exec: python Ranges03.py
What is your GPA?  -->  3.8
Magna Cum Laude
 ----jGRASP: operation complete.

 ----jGRASP exec: python Ranges03.py
What is your GPA?  -->  3.5
Cum Laude
 ----jGRASP: operation complete.

 ----jGRASP exec: python Ranges03.py
What is your GPA?  -->  2.8
Graduate without Honors
 ----jGRASP: operation complete.

 ----jGRASP exec: python Ranges03.py
What is your GPA?  -->  2.3
Did not graduate
 ----jGRASP: operation complete.
```

11.8 Input Protection

One unfortunate reality of life is that people make mistakes. Some people make more than others. When someone uses a computer and it does not work properly, people get very frustrated. In many cases they are the cause of their own frustration. The computer is not working because the user entered the wrong information in the program. Now, if a program is written very well, it can actually *prevent* certain errors from occurring.

Program **InputProtection01.py**, in Figure 11.14, is similar to some Chapter VII programs which ask about your SAT score. This program also asks for your gender and your last name. A valid SAT score is an integer between **400** and **1600**. A valid gender is **'M'**, **'m'**, **'F'** or **'f'**. This first input protection example actually shows what happens when a program uses no input protection whatsoever.

Figure 11.14

```
# InputProtection01.py
# This program enters the SAT score, gender and last name
# of a college applicant.  Nothing prevents the user from
# entering invalid information for <sat> and <gender>.


print()
sat = eval(input("Enter SAT  {400..1600}  -->  "))
gender  =  input("Enter your gender {M/F} -->  ")
lastName = input("Enter your last name    -->  ")
print()

if gender == 'M' or gender == 'm':
   print("Mr.",lastName,end = ", ")
if gender == 'F' or gender == 'f':
   print("Ms.",lastName,end = ", ")

if sat >= 1100:
   print("you are admitted!")
else:
   print("you are not admitted.")

```

```
 ----jGRASP exec: python InputProtection01.py

Enter SAT  {400..1600}  -->  1200
Enter your gender {M/F} -->  F
Enter your last name    -->  Jones

Ms. Jones, you are admitted!

 ----jGRASP: operation complete.

 ----jGRASP exec: python InputProtection01.py

Enter SAT  {400..1600}  -->  900
Enter your gender {M/F} -->  M
Enter your last name    -->  Smith

Mr. Smith, you are not admitted.

 ----jGRASP: operation complete.

 ----jGRASP exec: python InputProtection01.py

Enter SAT  {400..1600}  -->  5000000
Enter your gender {M/F} -->  m
Enter your last name    -->  Jackson

Mr. Jackson, you are admitted!

 ----jGRASP: operation complete.

 ----jGRASP exec: python InputProtection01.py

Enter SAT  {400..1600}  -->  -100
Enter your gender {M/F} -->  z
Enter your last name    -->  Green

you are not admitted.

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

If you look only at the first 2 outputs, the program seems to work fine. The **sat** score determines admittance. The **gender** determines if the message starts with **"Mr."** or **"Ms."**. The third example begins with an SAT score of **5000000** which is considerably larger than a perfect 1600 SAT score. The program happily and merrily tells Mr. Jackson that he is "admitted" because his SAT score is *greater than or equal to 1100*. The final output has 2 problems. First there is a negative SAT score. Second, the gender is **'z'**.

Program **InputProtection02.py**, in Figure 11.15, adds a couple **while** loops to the previous program. The first of these loops forces the user to keep entering the SAT score over and over again until it is a valid score in the 400..1600 range. The second loop will repeat until a valid gender of **'M'** or **'F'** is entered. This program will not accept any invalid SAT scores or genders.

NOTE: For simplicity, we will not mess with lowercase **'m'** or **'f'** for **gender**.

If you look at the condition in each **while** loop, you will notice that we start by defining what we want. We want an SAT score between 400 and 1600. We want the gender to be male or female. So why is the whole thing placed in parentheses with a **not** in front? Consider this. If the user enters what we want, then the loop does not have to repeat. We are done. It is only when the user enters something wrong that the loop need to repeat.

Figure 11.15

```
 1 # InputProtection02.py
 2 # This program improves on the previous program by
 3 # adding 2 <while> loops that will force the user to
 4 # keep re-entering the information until it is valid.
 5
 6
 7 print()
 8
 9 sat = 0
10 while not(sat >= 400 and sat <= 1600):
11     sat = eval(input("Enter SAT  {400..1600}  -->  "))
12
13 gender = ''
14 while not(gender == 'M' or gender == 'F'):
15     gender = input("Enter your gender {M/F} -->  ")
16
17 lastName = input("Enter your last name    -->  ")
18 print()
19
```

```
20 if gender == 'M':
21    print("Mr.",lastName,end = ", ")
22 if gender == 'F':
23    print("Ms.",lastName,end = ", ")
24
25 if sat >= 1100:
26    print("you are admitted!")
27 else:
28    print("you are not admitted.")
29
```

```
   ----jGRASP exec: python InputProtection02.py

▶▶ Enter SAT  {400..1600}  -->  100
▶▶ Enter SAT  {400..1600}  -->  2000
▶▶ Enter SAT  {400..1600}  -->  1300
▶▶ Enter your gender {M/F} -->  Z
▶▶ Enter your gender {M/F} -->  1
▶▶ Enter your gender {M/F} -->  #
▶▶ Enter your gender {M/F} -->  M
▶▶ Enter your last name    -->  Smith

   Mr. Smith, you are admitted!

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

Using DeMorgan's Law in Python

Program **InputProtection03.py**, in Figure 11.16, demonstrates a practical example of DeMorgan's Law. This program does the exact same thing as the previous program, but now the **not** that was outside the parentheses in the **while** loops will be distributed through the expression.

Figure 11.16

```
 1 # InputProtection03.py
 2 # This program "distributes the not" in the
 3 # <while> loops using DeMorgan's Law.
 4
 5
 6 print()
 7
 8 sat = 0
 9 while sat < 400 or sat > 1600:
10    sat = eval(input("Enter SAT  {400..1600}  -->  "))
11
12 gender = ''
13 while gender != 'M' and gender != 'F':
14    gender = input("Enter your gender {M/F} -->  ")
15
16 lastName = input("Enter your last name    -->  ")
17 print()
18
19 if gender == 'M':
20    print("Mr.",lastName,end = ", ")
21 if gender == 'F':
22    print("Ms.",lastName,end = ", ")
23
24 if sat >= 1100:
25    print("you are admitted!")
26 else:
27    print("you are not admitted.")
28
```

Program **InputProtection04.py**, in Figure 11.17, makes the program more readable by adding Boolean variables. Additional **if** statements are also added to make the output more user-friendly. Also, if you look on line 19, you see another Python shortcut. Instead of typing **gender == 'M' or gender = 'm' or gender == 'F' or gender == 'f'**, we can just type **gender in ['M','m','F','f']**.

Figure 11.17

```
 1 # InputProtection04.py
 2 # This program uses Boolean variables to make the
 3 # program more readable.  Addition <if> statements
 4 # are also added to make the program more user-friendly.
 5
 6
 7 sat = 0
 8 satOK = False
 9 while not satOK:
10    sat = eval(input("\nEnter SAT  {400..1600}  -->  "))
11    satOK = 400 <= sat <= 1600
12    if not satOK:
13       print("\nError! Please enter a number between 400 & 1600.")
14
15 gender = ''
16 genderOK = False
17 while not genderOK:
18    gender = input("\nEnter your gender {M/F} -->  ")
19    genderOK = gender in ['M','m','F','f']
# another Python shortcut
20    if not genderOK:
21       print("\nError! Please enter either an 'M' or an 'F'.")
22
23 lastName = input("\nEnter your last name    -->  ")
24 print()
25
26 if gender == 'M' or gender == 'm':
27    print("Mr.",lastName,end = ", ")
28 if gender == 'F' or gender == 'f':
29    print("Ms.",lastName,end = ", ")
30
31 if sat >= 1100:
32    print("you are admitted!")
33 else:
34    print("you are not admitted.")
35
```

```
 ----jGRASP exec: python InputProtection04.py
Enter SAT  {400..1600}  -->  100
Error! Please enter a number between 400 & 1600.
Enter SAT  {400..1600}  -->  2000
Error! Please enter a number between 400 & 1600.
Enter SAT  {400..1600}  -->  -100
Error! Please enter a number between 400 & 1600.
Enter SAT  {400..1600}  -->  2000000000
Error! Please enter a number between 400 & 1600.
Enter SAT  {400..1600}  -->  900
Enter your gender {M/F} -->  W
Error! Please enter either an 'M' or an 'F'.
Enter your gender {M/F} -->  7
Error! Please enter either an 'M' or an 'F'.
Enter your gender {M/F} -->  @
Error! Please enter either an 'M' or an 'F'.
Enter your gender {M/F} -->  F
Enter your last name    -->  Jones
Ms. Jones, you are not admitted.
 ----jGRASP: operation complete.
```

11.9 *Logic* Errors

You may be thinking. "Logic Errors?" Haven't we already learned about those? Well yes, but this is a chapter about *Logic*. The next few program examples will not work properly because they have *Logic* errors, literally.

Program **LogicError01.py**, in Figure 11.18, is similar to program **InputProtection02.py**. The program has 2 logic errors because the inner parentheses in the **while** statements do not contain the entire Boolean expression. This means the **not** only applies to the first part of the expression. When you run the program it may seem to work, unless you are female or enter an SAT score above **1600**.

Figure 11.18

```
1  # LogicError01.py
2  # This program is similar to program InputProtection02.py,
3  # but now there are 2 logic errors because the parentheses
4  # in the <while> statements do not contain the entire Boolean
5  # expression.  This means the <not> only applies to the first
6  # part of the expression.  When you run the program it may
7  # seem to work, unless you are female or enter an SAT score
8  # above 1600.
9
10
11 print()
12
13 sat = 0
14 while not(sat >= 400) and sat <= 1600:
15    sat = eval(input("Enter SAT  {400..1600}  -->  "))
16
17 gender = ''
18 while not(gender == 'M') or gender == 'F':
19    gender = input("Enter your gender {M/F} -->  ")
20
21 lastName = input("Enter your last name    -->  ")
22 print()
23
24 if gender == 'M':
25    print("Mr.",lastName,end = ", ")
26 if gender == 'F':
27    print("Ms.",lastName,end = ", ")
```

```
28
29 if sat >= 1100:
30    print("you are admitted!")
31 else:
32    print("you are not admitted.")
33
```

```
    ----jGRASP exec: python LogicError01.py

▶▶  Enter SAT  {400..1600}  -->  -100
▶▶  Enter SAT  {400..1600}  -->  2000
▶▶  Enter your gender {M/F} -->  #
▶▶  Enter your gender {M/F} -->  F
▶▶  Enter your gender {M/F} -->  M
▶▶  Enter your last name    -->  Smith

    Mr. Smith, you are admitted!

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

Border Cases

When testing a program, at minimum, you want to test every possible path. For example, if a program determines if you *pass* or *fail*, you should test it with one passing grade and one failing grade. That is fine, but if you want to thoroughly test your program, you should test the *Border Cases* as well. This means you also need to test the program with **69** and **70**.

Program **LogicError02.py**, in Figure 11.19, is similar to program **InputProtection03.py**. This program actually works about 99% of the time. The issue is with the condition in the first **while** loop. Remember that the original condition was:

```
while not(sat >= 400 and sat <= 1600):
```

When we use DeMorgan's Law to *distribute the not* we get:

```
while sat < 400 or sat > 1600:
```

However, this program uses:

```
while sat <= 400 or sat >= 1600:
```

Do you see the difference? The error is very common. Many people think the opposite of **>=** is **<=**. It isn't. The opposite of **>=** is **<**. The opposite of **<=** is **>**. Imagine that you have 2 numbers **A** and **B**, and you know the statement **A** >= **B** is **false**. So you know **A** is not greater than **B**. You also know **A** is not equal to **B**. What is left? **A** must be less than **B**.

So how does this affect program **LogicError02.py**? Look at the first **while** loop. The condition erroneously uses **sat** <= **400** when it should use **sat** < **400**. It also uses **sat** >= **1600** when it should use **sat** > **1600**. The result is the program works fine most of the time. SAT scores less than **400** are rejected. SAT scores greater than **1600** are rejected. This is expected. The problem is SAT score EQUAL TO **400** or **1600** are also rejected.

Figure 11.19

```
 1 # LogicError02.py
 2 # This program is similar to program InputProtection03.py,
 3 # but now there is a subtle logic error in the first loop.
 4 # The opposite is sat >=  400 is sat <  400, NOT sat <=  400.
 5 # The opposite is sat <= 1600 is sat > 1600, NOT sat >= 1600.
 6 # While the program works most of the time, it will not work
 7 # for the "border cases" when the SAT is exactly 400 or 1600.
 8
 9
10 print()
11
12 sat = 0
13 while sat <= 400 or sat >= 1600:
14    sat = eval(input("Enter SAT  {400..1600}  -->  "))
15
16 gender = ''
17 while gender != 'M' and gender != 'F':
18    gender = input("Enter your gender {M/F} -->  ")
19
20 lastName = input("Enter your last name    -->  ")
21 print()
```

```
22
23 if gender == 'M':
24    print("Mr.",lastName,end = ", ")
25 if gender == 'F':
26    print("Ms.",lastName,end = ", ")
27
28 if sat >= 1100:
29    print("you are admitted!")
30 else:
31    print("you are not admitted.")
32
```

```
----jGRASP exec: python LogicError02.py

Enter SAT  {400..1600}  -->  100
Enter SAT  {400..1600}  -->  2000
Enter SAT  {400..1600}  -->  400
Enter SAT  {400..1600}  -->  1600
Enter SAT  {400..1600}  -->  1599
Enter your gender {M/F} -->  F
Enter your last name    -->  Jones

Ms. Jones, you are admitted!

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

Don't Forget DeMorgan's Law

The final program of this chapter, **LogicError03.py**, shown in Figure 11.20, is also similar to program **CompoundCondition03.py**. This time there are 2 logic errors because the conditions in both **while** loops did not follow DeMorgan's Law properly.

Here are the original 2 conditions:

```
while not(sat >= 400 and sat <= 1600):

while not(gender == 'M' or gender == 'F'):
```

When we use DeMorgan's Law to *distribute the not* we get:

```
while sat < 400 or sat > 1600:

while gender != 'M' and gender != 'F':
```

However, this program uses:

```
while sat < 400 and sat > 1600:

while gender != 'M' or gender != 'F':
```

Remember, when you *distribute the not*, DeMorgan's Law says you either need to change the **and** to an **or**, or you need to change the **or** to an **and**. Program **LogicError03.py** fails to do either. The result is both **while** loops do not work at all. The first loop never even executes. Consider this. Can a number be both less than **400** AND greater than **1600** at the same time? No, that will never happen. As for the second loop, it will repeat forever. Consider this. If you are female, then you are not male. If you are male, then you are not female. One of these conditions is always guaranteed to be **True**.

Figure 11.20

```
1 # LogicError03.py
2 # This program is also similar to program InputProtection03.py,
3 # but now there are 2 logic errors because DeMorgan's Law was
4 # not followed when the <not> was distributed in the <while>
5 # statements.  The result is the first loop will never execute,
6 # and the second loop will repeat forever.
7
```

```
 8
 9 print()
10
11 sat = 0
12 while sat < 400 and sat > 1600:
13    sat = eval(input("Enter SAT  {400..1600}  -->  "))
14
15 gender = ''
16 while gender != 'M' or gender != 'F':
17    gender = input("Enter your gender {M/F} -->  ")
18
19 lastName = input("Enter your last name    -->  ")
20 print()
21
22 if gender == 'M':
23    print("Mr.",lastName,end = ", ")
24 if gender == 'F':
25    print("Ms.",lastName,end = ", ")
26
27 if sat >= 1100:
28    print("you are admitted!")
29 else:
30    print("you are not admitted.")
31
```

```
   ----jGRASP exec: python LogicError03.py

▶▶ Enter your gender {M/F} -->  Z
▶▶ Enter your gender {M/F} -->  #
▶▶ Enter your gender {M/F} -->  F
▶▶ Enter your gender {M/F} -->  M
▶▶ Enter your gender {M/F} -->  M
▶▶ Enter your gender {M/F} -->  M
▶▶ Enter your gender {M/F} -->
```