In Python an **if-statement** consists of the keyword **if**, followed by a Boolean condition, the symbol **:,** and the "body" which contains the statements executed when the condition is true (these statements are indented as done for the body of a for- or while-loop):

```
if age < 16:
        print "I can't drive"
```

The key to understanding an if-statement is to know that the body will execute only when the **if condition** evaluates to **True** and that it will not execute when the **if condition** evaluates to **False**. In the above statement, "I can't drive" will only be printed when the value of the variable **age** is less than 16. When the value of **age** is greater or equal to 16, the if-statement does nothing.

Write the following program and save it in file **cond1.py**:

```
# Lab 5: cond1.py
# If statements and your age
# Written by: Your Name
age = 0
while age < 20:
        print  "I am ", age
        if age < 16:
                print "I can't drive"
        if age < 18:
                print "I can't vote"
        print    # prints an empty line
        age = age+1

print "end of program"
```

Now run the program. What does it do?

- Change the first line to **age = 7**. Run the program again. What was different?
- Modify the program to print "Now you can drive" only when you are 16 and "Now you can vote" only when you are 18. In all other situations, nothing is printed. Remember that == is the syntax for "equal to". Save the new program in file **cond2.py**.

If-then-else and a remark about nested statements

Python (as well as other languages) provide more than the simple *"if the condition is true, then do the following"* construction described. A natural extension is *"if the condition is true, then take actions 1, else take actions 2"*. Below is an example on how to write an if-then-else statement.

```
age = 17

if age < 16:
   print "I can't drive"
else:
   print "I can drive"
```

Notice the indentation: the word **else** lines up with the **if**, indicating that statements indented under it are executed when the condition is false. The body of an **else-statement** is executed when the body of the preceding **if-statement** did not execute. The statement above will always print something; what depends on the value of age.

Before we give another if-example, a comment on nested statements. When a loop contains another loop,

we refer to it as a nested loop. Nested statememts can create interesting code*: a while-loop can contain a for-loop which can contain another while-loop which can contain an if-then-else statement and the else-part can contain a while loop*. Still with me? Not surprisingly, heavily nested code gets complex and can be challenging to debug. You will not see heavily nested code in this class, but you want to realize the potential and complexity hidden in nested code.

Load program **cond3.py** which contains a nested if-statement. It uses the age and the body mass index (bmi) to determine the heart disease risk. The risk does not only depend on the bmi, but also on the age (at age 45, the classification changes). A single if-then-else statement is not enough to determine the risk of a given age and bmi pair. Program **cond3.py** shows one of the possible solutions for identifying the correct risk.

```
# Lab 5: cond3.py
# Using if-then-else and nested if's
# Written by: Your instructor
age = 55
bmi = 33

# other age-bmi pairs to use:
# 17, 10 --- 23, 55 --- 55, 33 --- 99, 21.2

print "age =", age, "body mass index =", bmi

if age < 45:
   if bmi < 22.0:
      print "risk of heart disease is low"
   else:
      print "risk of heart disease is medium"
else:
   if bmi < 22.0:
      print "risk of heart disease is medium"
   else:
      print "risk of heart disease is high"
```

Run the program with different pairs of age and bmi. Pay attention to the indentation of the if-statement within an if-statement.

Solving a quadratic equation

Consider the problem of solving a quadratic equations of the form $ax^2+bx+c = 0$ using the formula you have been taught. The Python program we write for solving a quadratic equation first computes the value $b^2 - 4ac$. When this quantity is negative, taking the square-root would result in an error. We will use an if-statement to check for this.

Below is the first quadratic equation program. We set the values of a, b, and c in the program.

```
# Lab 5:  cond4.py
# Solving a quadratic equation – version 1
# Written by: Your Name

from math import sqrt
a=6
b=10
c=-12
```

```
# other triples a, b, c you should try: 1, 2, 3; 1, 2, 1; 100, 300, -555;

discrim = b * b - 4 * a * c
if discrim >= 0:
   discRoot = sqrt(discrim)
   sol1 = (-b + discRoot) / (2 * a)
   sol2 = (-b - discRoot) / (2 * a)
   print "The solutions are:", sol1, “and”, sol2
```

Load the program and run it. It will print
>>>
The two solutions are 0.808142966966 and -2.47480963363

This program computes the real roots, if they exist. It is very bare-bones and the following guides you through a sequence of improvements**:**

1. The program does not produce an output when there are no real roots. This is, in general, not desirable in a program. The program should print a message saying that there are no real roots (and not be silent).
    - Change program **cond4.py** so that is prints the message “The quadratic equation has no real roots” when **discrim < 0.**
    - Run your program with values 1,2,1 and then with 1,2,3

2. When the quadratic equation has a double root, we want to print the double root once and say that it is a double root.
    - What is the condition for a double root? Give the condition in terms of **discrim**.
    - Change the already updated program **cond4.py** so that it prints one value when there is a double root and it indicates that it is a double root. Something like:
      The equation has a double root at 4.

3. It is often helpful to echo the input. Add a line to **cond4.py** which prints the equation with the values of a, b, and c, in some easy to read form. What is printed should look something like:
   *The quadratic equation is 6 x**2 + 10 x + -12*

<u>If-statements and VPython</u>

If-statements will be very useful in physical simulations. In the example below, we develop a program in which a ball bounces off two opposing walls (it will do so in an infinite loop so you can watch it for as long as you want).

Take a look at program **while_loop4_modified.py:** a box named **wallR** is positioned to the right of sphere **ball** and the goal is to stop the ball’s movement once it reached **wallR**. We have changed the code a little from what you saw in Lab 3,

```
# Lab 5: while_loop4_modified.py
# sphere moving towards wall – modified from Lab 4
# Written by: Your Instructor
from __future__ import division
from visual import *

ball = sphere(pos=vector(-5,0,0), radius=0.5, color=color.red)
wallR = box(pos= vector (6,0,0), size= (2,12,8), color=color.green)
vball = vector(1,0,0) #vector vball represents the velocity of ball
```

```
t = 0       #t represents time
deltat = 1  #deltat represents the time step in one iteration

while ball.pos.x + ball.radius < wallR.x - wallR.length/2:
   rate(3)
   print "position at time", t, "is:", ball.pos

   #update the position of ball using vector operations
   ball.pos = ball.pos + vball*deltat
   #update time
   t = t + deltat
```

Load **while_loop4_modified.py** and run it.
Explain the Boolean condition used in the while loop.
This idea used in this program is now used to have a ball bounce off walls indefinetly.

In order to have a ball bounce off a wall, we need to detect when the wall has been reached (an if-statement will do this) and we need to change its velocity to move into the opposite direction. This means changing the sign of **vball.x**. How do we change the sign of a quantity? We simply negate it (to change the sign of a, write the assignment statement **a = –a**).

Load **cond5.py** and run it.
What is happening?

Complete the second if-statement in the infinite while loop so the ball also bounces off the left wall.

```
# Lab 5: cond5.py
# sphere bouncing off two walls
# Written by: Your Name
from __future__ import division
from visual import *

ball = sphere(pos=vector(0,0,0), radius=1, color=color.red)

wallR = box(pos=(8,0,0),size=(2,12,8),color=color.green)
wallL = box(pos=(-8,0,0),size=(2,12,8),color=color.green)
# define a right and a left wall

vball = vector(1,0,0) #vector vball represents the velocity of ball

while 1 == 1:
   # the infinite loop lets the ball bounce off the walls
   rate(10)

   # test of the ball is ready to bounce off the right wall
   if ball.pos.x + ball.radius >= wallR.x - wallR.length/2:
      vball.x = -vball.x

   # test of the ball is ready to bounce off the left wall
   # if ball hits the left wall
   #    change the sign of the x-value of its velocity
```

```
# upadate the postion of the ball
ball.pos = ball.pos + vball
```

Run the new program. If the ball is not bouncing the right way, take a careful look at the Boolean condition.
Wait for your instructor to check your work.

Challenge

- Change the value of **vball.y**. Start with a small value like 0.1. Describe what happens to the ball.
- To keep the ball bouncing off the green walls, change the sign of **vball.y** at the "right time." Add if-statements that keep the ball between the green walls for a non-zero value of **vball.y.**

Super Challenge

- Add a top and a bottom wall in the form of two box objects **wallT** and **wallB.** If you are really ambitious, add a back wall (you are now looking into an open box),
- Add statements to the program that make the ball not only bounce off the left and right wall, but also the top and bottom.
- Add another ball so you now have two balls bounce off the two walls.

Hint: first write a program which handles nonzero **vball.x** and **vball.y** values. Once the ball bounces correctly, you can add statements handling non-zero **vball.z** values.