Conditionals: if/elif/else and while

Goals for this Section: "Conditionals"

- **Employ** *conditional (if/else) logic* in our code
- **Make use of** *boolean values* and **True**/**False** tests
- **Identify** code block syntax
- **Use** *while loops* to produce repetitive logic

'if' statement

The **if** statement executes code in its *block* only if the *test* is **True**.

```
aa = raw_input('please enter a positive integer: ')
int_aa = int(aa)

if int_aa < 0:                    # test:  is this a True statement?

    print 'error:  input invalid'  # block (2 lines) -- lines are
    exit(1)                        # executed only if test is True

d_int_aa = int_aa * 2              # double the value
print 'your value doubled is ' + str(d_int_aa)
```

The two components of an if statement are the *test* and the *block*. The test determines whether the block will be executed.

'else' statement

An **else** statement will execute its block if the **if** test before it was *not* **True**.

```
xx = raw_input('enter an even or odd number:  ')
yy = int(xx)

if yy % 2 == 0:                # can 2 divide into yy evenly?
    print xx + ' is even'
    print 'congratulations.'

else:
    print xx + ' is odd'
    print 'you are odd too.'
```

Therefore we can say that only one block of an if/else statement will execute.

'elif' statement

elif is also used with **if** (and optionally **else**): you can chain additional conditions for other behavior:

```
zz = raw_input('type an integer and I will tell you its sign:  ')
zyz = int(zz)

if zyz > 0:
    print 'that number is positive'
    print 'we should all be positive'

elif zyz < 0:
    print 'that number is negative'
    print "please don't be negative"

else:
    print '0 is neutral'
    print "there's no neutral come November."
```

if can be used alone with **elif** alone, with **else** alone, together with both, or by itself. **elif** and **else** obviously require a prior **if** statement.

The Python code block

A *code block* is marked by indented lines. The end of the block is marked by a line that returns to the prior indent.

```
xx = raw_input('enter an even or odd number:  ')  # not in any block
yy = int(xx)                                      # ditto


if yy % 2 == 0:                  # the start of the 'if' block
    print 'your number is even'
    print 'even is cool'         # last line of the 'if' block


else:                            # the start of the 'else' block
    print 'your number is odd'
    print 'you are cool'         # last line of the 'else' block


print 'thanks for playing "even/odd number"'      # not in any block
```

Note also that a block is *preceded by an unindented line that ends in a colon.*

The blocks that we'll look at in this unit are:

```
  - if       conditional
  - elif     conditional
  - else     conditional
  - while    conditional loop
```

Blocks we'll see in upcoming lessons:

```
  - for      sequence loop
  - def      function definition
  - class    class definition
  - try      exception handling block
  - except   exception handling block
```

Nested blocks increase indent

Blocks can be nested within one another. A nested block (a "block within a block") simply moves the code block further to the right.

```
var_a = int(raw_input('enter a number: '))
var_b = int(raw_input('enter another number:  '))

if var_b >= var_a:                      # compare int values for truth
    print "the test was true"           #
    print "var b is at least as large"

    if var_a == var_b:                  # if the two values are equivalen
        print 'the two values are equivalent'

    print "now we're in the outer block but not in the inner block"

print 'this gets printed in any case (i.e., not part of either block)'
```

Complex decision trees using 'if' and 'else' is the basis for most programs.

'and' and 'or' for "compound" tests

Python uses the operators **and** and **or** to allow *compound tests*, meaning tests that depend on multiple conditions.

'and' compound statement: both tests must be True

```
xx = raw_input('what is your ID?  ')
yy = raw_input('what is your pin?  ')

if xx == 'dbb212' and yy == '3859':
    print 'you are a validated user'
else:
    print 'you are not validated'
```

'or' compound statement: one test must be True

```
aa = raw_input('please enter "q" or "quit" to quit: ')
if aa == 'q' or aa == 'quit':
    exit()
print 'continuing...'
```

Note the lack of parentheses around the tests -- if the syntax is unambiguous, Python will understand. We can use parentheses to clarify compound statements like these, but they often aren't necessary.

negating an 'if' test with 'not'

You can negate a test with the **not** keyword:

```
var_a = 5
var_b = 10

if not var_a > var_b:
    print "var_a is not larger than var_b (well - it isn't)."
```

Of course this particular test can also be expressed by replacing the comparison operator **>** with **<=**, but when we learn about new True/False condition types we'll see how this operator can come in handy.

The meaning of True and False

True and **False** are *boolean* values, and are produced by expressions that can be seen as True or False.

```
aa = 3
bb = 5

if aa < bb:
    print "that is true"


var = 10
var2 = 10.0

if var == var2:
    print "those two are equal"
```

The *if test* is actually different from the *value being tested*. **aa < bb** and **var == var2** look like tests but are actually expressions that resolve to **True** or **False**, which are values of *boolean type*:

```
xx = (5 < 3)
print xx             # True
print type(xx)       # <type 'bool'>

yy = (var == var2)
print yy             # False
print type(yy)       # <type 'bool'>
```

Note that we would almost never assign comparisons like these to variables, but we are doing so here to illustrate that they resolve to boolean values.

while loops

A **while** test causes Python to loop through a block repetitively, as long as the test is True.

This program prints each number between 0 and 4

```
cc = 0              # initialize a counter

while cc < 5:       # "if test is True, enter the block"
    print cc
    cc = cc + 1     # "increment" cc:  add 1 to its current value
                    # WHEN WE REACH THE END OF THE BLOCK,
                    # JUMP BACK TO THE while TEST

print 'done'
```

This means that the block is executing the **print** and **cc = cc + 1** lines multiple times - again and again until the test becomes False.

Here's the execution order of lines, spelled out as if it were successive statements.

```
cc = 0
while cc < 5:    # testing 0 < 5:  True
    print cc         # 0
    cc = cc + 1      # 0 becomes 1
                 # block ends, Python returns to top of loop

while cc < 5:    # testing 1 < 5:  True
    print cc         # 1
    cc = cc + 1      # 1 becomes 2
                 # block ends, Python returns to top of loop

while cc < 5:    # testing 2 < 5:  True
    print cc         # 2
    cc = cc + 1      # 2 becomes 3
                 # block ends, Python returns to top of loop

while cc < 5:    # testing 3 < 5:  True
    print cc         # 3
    cc = cc + 1      # 3 becomes 4
                 # block ends, Python returns to top of loop

while cc < 5:    # testing 4 < 5:  True
    print cc         # 4
    cc = cc + 1      # 4 becomes 5
                 # block ends, Python returns to top of loop

while cc < 5:    # testing 5 < 5:  False.
                 # Python drops to below the loop.

 print 'done'
```

Of course, the value being tested must change as the loop progresses - otherwise the loop will cycle indefinitely (endless loop).

Understanding while loops

while loops have 3 components: the *test*, the *block*, and the *automatic return*.

```
cc = 10

while cc > 0:     # THE TEST (if True, enter the block)

    print cc      # THE BLOCK (execute as regular Python statements)
    cc = cc - 1

    [invisible!]  # THE AUTOMATIC RETURN
                  # (at end of block, go back to the test)

print 'done'
```

Software's *repetitive processing* technique requires that we be able to say "do this thing over and over until a condition is met, or until I tell you to stop".

Can you tell just from reading what this code prints? You'll need to keep track of the value of **cc** and calculate its changes as the code block is executed repetitively.

In order to use **while** loops you must be able to model code execution in your head. This takes some practice but isn't complicated. (See the fully modeled explanation in the previous slide for an example of modeling.)

Loop control: "break"

break is used to exit a loop regardless of the test condition.

```
xx = 0
while xx < 10:
    answer = raw_input("do you want loop to break? ")
    if answer == 'y':
        break             # drop down below the block
    print 'Hello, User'
    xx = xx + 1
    print 'I have now greeted you ', xx, ' times'

print "ok, I'm done"
```

Loop control: "continue"

The **continue** statement jumps program flow to next loop iteration.

```
x = 0
while x < 10:
    x = x + 1
    if x % 2 != 0:   # will be True if x is odd
        continue     # jump back up to the test and test again
    print x
```

Note that **print x** will not be executed if the **continue** statement comes first. Can you figure out what this program prints?

The "while True" loop

while with **True** and **break** provide us with a handy way to keep looping until we feel like stopping.

```
while True:
    var = raw_input('please enter a positive integer:  ')
    if int(var) > 0:
        break
    else:
        print 'sorry, try again'

print 'thanks for the integer!'
```

Note the use of **True** in a **while** expression: since **True** is always **True** this test will be false. Therefore the **break** statement is essential to keep this loop from looping indefinitely.

Debugging loops: the "fog of code"

The challenge in working with code that contains loops and conditional statements is that it's sometimes hard to tell what the program did. I call this lack of visibility the "fog of code".

Consider this code, which attempts to add all the integers from a range (i.e., 1 + 2 + 3 + 4, etc.). *It has a major bug*. Can you spot it?

```
revcounter = 0
while revcounter < 10:

    varsum = 0                      # set varsum to 0
    revcounter = revcounter + 1     # increment value of revcounter by 1
    varsum = varsum + revcounter    # add value of revcounter to varsum

print varsum                        # prints 11 (should be 55)
```

It's quite easy to run code like this and not understand the outcome. After all, we are adding each value of **revcounter** to **varsum** and increasing **revcounter** by one until it becomes **10**. At the end, why is the value of **varsum** only 10? What happened to the other values? Why weren't they added to **varsum**?

Now, it is possible to read the code, think it through and model it in your head, and figure out what has gone wrong. But this modeling doesn't always come easily, and it's easy to make a mistake in your head.

Some students approach this problem by tinkering with the code, trying to get it to output the right values. But this is a very time-wasteful way to work, not to mention that it *allows the code to remain mysterious*. In other words, it is not a way of working that enhances understanding, and so it is practically useless for attaining our objectives in this course.

What we need to do is bring some visibility to the loop, and begin to ask questions about what happened. The loop is iterating multiple times, but how many times? What is happening to the variables **varsum** and **revcounter** to produce this outcome? Rather than guessing semi-randomly at a solution, *which can lead you to hours of experimentation*, we should ask questions of our code. And we can do this with **print** statements and the use of **raw_input()**.

```
revcounter = 0
while revcounter < 10:

    varsum = 0
    revcounter = revcounter + 1
    varsum = varsum + revcounter

    print "loop iteration complete"
    print "revcounter value: ", revcounter
    print "varsum value: ", varsum
    raw_input('pausing...')
    print
    print

print varsum                        # 10
```

I've added quite a few statements, but if you run this example you will be able to get a hint as to what is happening:

```
loop iteration complete
revcounter value:  1
varsum value:  0
pausing...                           # here I hit [Return] to continue


loop iteration complete
revcounter value:  2
varsum value:  1
pausing...                           # [Return]


loop iteration complete
revcounter value:  3
varsum value:  2
pausing...


loop iteration complete
revcounter value:  4
varsum value:  3
pausing...
```

Just looking at the first iteration, we can see that the values are as we might expect: it seems that **revcounter** was 0, it added its value to **varsum**, and then had its value incremented by 1. So **revcounter** is now **1** and **varsum** is now 0.

So far so good. The code has paused, we hit **[Return]** to continue, and the loop iterates again. We can see that **revcounter** has increased by 1, as we expected: it is now **2**. But... why is **varsum** only 9? How has its value *decreased*? We are continually *adding* to **varsum**. Why isn't its value increasing?

And if we continue hitting **[Return]**, we'll start to see a pattern - **varsum** is always one higher than **revcounter** for some reason. It doesn't make sense given that we're adding each value of **revcounter** to **varsum**.

Until we look at the code again, and see that we are also *initializing* **varsum** to 0 with every iteration of the loop. Which now makes it clear why we're seeing what we're seeing, and in fact why the final value of **varsum** is 1.

So the solution is to initialize **varsum** *before* the loop and not inside of it:

```
revcounter = 0
varsum = 0
while revcounter < 10:

    revcounter = revcounter + 1
    varsum = varsum + revcounter

print varsum
```

This outcome makes more sense. We might want to check the total to be sure, but it looks right.

The hardest part of learning how to code is in designing a solution. This is also the hardest part to teach! But the last thing you want to do in response is to guess repeatedly. Instead, please examine the outcome of your code through print statements, see what's happening in each step, then compare this to what you think should be happening. Eventually you'll start to see what you need to do. Step-by-baby-step!