Nested Loops

- At the top level, we will use a file-processing loop that computes a running sum and count.

```python
for i in range(11):
    for j in range(11):
        print i*j,
    print
```
Nested Loops

- We get the following:

```
1 2 3 4 5 6 7 8 9 10
2 4 6 8 10 12 14 16 18 20
3 6 9 12 15 18 21 24 27 30
...```

A better version

```python
for i in range(1,11):
    for j in range(1,11):
        print "%d\%%d"%(i,j),
    print
```
**Nested Loops**

- **Final Program**

```python
def main():
    print "\%s\%(5* " ",
    for i in range(1,11):
        print "\%5d"%(i),
        print "\%4d\%(4*" " ")
    for i in range(1,11):
        print "\%4d\%(i, "|"),
        for j in range(1,11):
            print "\%5d"%(i*j),
```

- The loop that processes the numbers in each line is indented inside of the file processing loop.
- The outer loop iterates once for row of the table.
- For each iteration of the outer loop, the inner loop iterates as many times as there are columns in the table.
- When the inner loop finishes, the next row of the table is processed.
Nested Loops

- Designing nested loops –
  - Design the outer loop without worrying about what goes inside
  - Design what goes inside, ignoring the outer loop.
  - Put the pieces together, preserving the nesting.

Computing with Booleans

- `if` and `while` both use Boolean expressions.
- Boolean expressions evaluate to `True` or `False`.
- So far we’ve used Boolean expressions to compare two values, e.g. `(while x >= 0)`
### Boolean Operators

- Sometimes our simple expressions do not seem expressive enough.
- Suppose you need to determine whether two points are in the same position – their x coordinates are equal and their y coordinates are equal.

```python
if p1.getX() == p2.getX():
    if p1.getY() == p2.getY():
        # points are the same
    else:
        # points are different
else:
    # points are different
```

- It’s easy to see that this is an awkward way to evaluate multiple Boolean expressions!
- Let’s check out the three Boolean operators and, or, and not.
Boolean Operators

- The Boolean operators `and` and `or` are used to combine two Boolean expressions and produce a Boolean result.
- `<expr> and <expr>`
- `<expr> or <expr>`

Boolean Operators

- The `and` of two expressions is true exactly when both of the expressions are true.
- We can represent this in a truth table.

<table>
<thead>
<tr>
<th>P</th>
<th>Q</th>
<th>P and Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>
Boolean Expressions

- In the truth table, P and Q represent smaller Boolean expressions.
- Since each expression has two possible values, there are four possible combinations of values.
- The last column gives the value of P and Q.

<table>
<thead>
<tr>
<th>P</th>
<th>Q</th>
<th>P or Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>
Boolean Expressions

- The only time \( \lor \) is false is when both expressions are false.
- Also, note that \( \lor \) is true when both expressions are true. This isn’t how we normally use “or” in language.

Boolean Operators

- The \( \text{not} \) operator computes the opposite of a Boolean expression.
- \( \text{not} \) is a unary operator, meaning it operates on a single expression.

<table>
<thead>
<tr>
<th>P</th>
<th>not P</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
</tr>
</tbody>
</table>
Boolean Operators

- We can put these operators together to make arbitrarily complex Boolean expressions.
- The interpretation of the expressions relies on the precedence rules for the operators.

Boolean Operators

- Consider \( a \text{ or } \text{not } b \text{ and } c \)
- How should this be evaluated?
- The order of precedence, from high to low, is not, and, or.
- This statement is equivalent to \((a \text{ or } ((\text{not } b) \text{ and } c))\)
- Since most people don’t memorize the the Boolean precedence rules, use parentheses to prevent confusion.
Boolean Operators

- To test for the co-location of two points, we could use an `and`.
  ```python
  if p1.getX() == p2.getX() and p2.getY() == p1.getY():
    # points are the same
  else:
    # points are different
  ```

- The entire condition will be true only when both of the simpler conditions are true.

Boolean Operators

- Say you’re writing a racquetball simulation. The game is over as soon as either player has scored 15 points.
  ```python
  scoreA == 15 or scoreB == 15
  ```

- How can you represent that in a Boolean expression?
- When either of the conditions becomes true, the entire expression is true. If neither condition is true, the expression is false.
Boolean Operators

- We want to construct a loop that continues as long as the game is not over.
- You can do this by taking the negation of the game-over condition as your loop condition!
  ```python
  while not(scoreA == 15 or scoreB == 15):
    # continue playing
  ```

Boolean Operators

- Some racquetball players also use a shutout condition to end the game, where if one player has scored 7 points and the other person hasn’t scored yet, the game is over.
  ```python
  while not(scoreA == 15 or scoreB == 15 or \
    (scoreA == 7 and scoreB == 0) or (scoreB == 7 and scoreA == 0):
    # continue playing
  ```
Boolean Operators

- Let’s look at volleyball scoring. To win, a volleyball team needs to win by at least two points.
- In volleyball, a team wins at 15 points.
- If the score is 15 – 14, play continues, just as it does for 21 – 20.
  - \((a \geq 15 \text{ and } a - b \geq 2) \text{ or } (b \geq 15 \text{ and } b - a \geq 2)\)
  - \((a \geq 15 \text{ or } b \geq 15) \text{ and } \text{abs}(a - b) \geq 2\)

Boolean Algebra

- The ability to formulate, manipulate, and reason with Boolean expressions is an important skill.
- Boolean expressions obey certain algebraic laws called Boolean logic or Boolean algebra.
Boolean Algebra

<table>
<thead>
<tr>
<th>Algebra</th>
<th>Boolean algebra</th>
</tr>
</thead>
<tbody>
<tr>
<td>a * 0 = 0</td>
<td>a and false == false</td>
</tr>
<tr>
<td>a * 1 = a</td>
<td>a and true == a</td>
</tr>
<tr>
<td>a + 0 = a</td>
<td>a or false == a</td>
</tr>
</tbody>
</table>

- and has properties similar to multiplication
- or has properties similar to addition
- 0 and 1 correspond to false and true, respectively.

Boolean Algebra

- Anything or ed with true is true:
  
  a or true == true

- Both and and or distribute:
  
  a or (b and c) == (a or b) and (a or c)
  a and (b or c) == (a and b) or (a and c)

- Double negatives cancel out:
  
  not(not a) == a

- DeMorgan’s laws:
  
  not(a or b) == (not a) and (not b)
  not(a and b) == (not a) or (not b)
Boolean Algebra

- We can use these rules to simplify our Boolean expressions.

  ```python
  while not(scoreA == 15 or scoreB == 15):
      # continue playing
  ```

- This is saying something like “While it is not the case that player A has 15 or player B has 15, continue playing.”

- Applying DeMorgan’s law:

  ```python
  while (not scoreA == 15) and (not scoreB == 15):
      # continue playing
  ```

  This becomes:

  ```python
  while scoreA != 15 and scoreB != 15
      # continue playing
  ```

- Isn’t this easier to understand? “While player A has not reached 15 and player B has not reached 15, continue playing.”
Boolean Algebra

- Sometimes it’s easier to figure out when a loop should stop, rather than when the loop should continue.
- In this case, write the loop termination condition and put a `not` in front of it. After a couple applications of DeMorgan’s law you are ready to go with a simpler but equivalent expression.