Outline

1. **Boolean Algebra**
   - logical expressions
   - pseudocode and flowcharts

2. **Conditional Constructs**
   - conditional operators
   - if, else, elif

3. **Logic in Sage**
   - computing truth tables with Sage

4. **Summary + Assignments**

MCS 260 Lecture 8
Introduction to Computer Science
Jan Verschelde, 12 September 2008
Boolean Algebra, Flowcharts
Conditional Expressions

1. Boolean Algebra
   logical expressions
   pseudocode and flowcharts

2. Conditional Constructs
   conditional operators
   if, else, elif

3. Logic in Sage
   computing truth tables with Sage

4. Summary + Assignments
Boolean algebra is the calculation with True and False (often having values 1 and 0). The operators are and, or, and not. Truth tables define the outcome for all values:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>x and y</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>x or y</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>
Boolean Algebra

computing with logical expressions

Boolean algebra is the calculation with True and False (often having values 1 and 0). The operators are and, or, and not. Truth tables define the outcome for all values:

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>x and y</th>
</tr>
</thead>
<tbody>
<tr>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>x or y</th>
</tr>
</thead>
<tbody>
<tr>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
</tbody>
</table>
Boolean Algebra
computing with logical expressions

Boolean algebra is the calculation with True and False (often having values 1 and 0). The operators are and, or, and not. Truth tables define the outcome for all values:

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>x and y</th>
</tr>
</thead>
<tbody>
<tr>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>x or y</th>
</tr>
</thead>
<tbody>
<tr>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
</tbody>
</table>
Evaluation Laws

law and order in the Boolean algebra

When not, and, or occur in an expression, not is first evaluated, before and, and finally or.

De Morgan’s laws for simplifying expressions:

- not (( not x ) or ( not y )) = x and y
  Negating not being alive or not being well means being alive and being well.

- not (( not x ) and ( not y )) = x or y
  Negating not going to school and not going to work means going to school or going to work.

We prove these laws by truth tables.
Application: realization of electronic circuits.
Evaluation Laws

law and order in the Boolean algebra

When not, and, or occur in an expression, not is first evaluated, before and, and finally or.

De Morgan’s laws for simplifying expressions:

- not (( not x ) or ( not y )) = x and y
  Negating not being alive or not being well means being alive and being well.

- not (( not x ) and ( not y )) = x or y
  Negating not going to school and not going to work means going to school or going to work.

We prove these laws by truth tables.
Application: realization of electronic circuits.
Evaluation Laws
law and order in the Boolean algebra

When `not`, `and`, `or` occur in an expression, `not` is first evaluated, before `and`, and finally `or`.

De Morgan’s laws for simplifying expressions:

- `not (( not x ) or ( not y )) = x and y`
  Negating not being alive or not being well means being alive and being well.

- `not (( not x ) and ( not y )) = x or y`
  Negating not going to school and not going to work means going to school or going to work.

We prove these laws by truth tables.

Application: realization of electronic circuits.
Evaluation Laws
law and order in the Boolean algebra

When not, and, or occur in an expression, not is first evaluated, before and, and finally or.

De Morgan’s laws for simplifying expressions:

- $\neg (\neg x \lor \neg y) = x \land y$  
  Negating not being alive or not being well means being alive and being well.

- $\neg (\neg x \land \neg y) = x \lor y$  
  Negating not going to school and not going to work means going to school or going to work.

We prove these laws by truth tables. Application: realization of electronic circuits.
Evaluation Laws

time and order in the Boolean algebra

When not, and, or occur in an expression, not is first evaluated, before and, and finally or.

De Morgan’s laws for simplifying expressions:

- not (( not x ) or ( not y )) = x and y
  Negating not being alive or not being well means being alive and being well.

- not (( not x ) and ( not y )) = x or y
  Negating not going to school and not going to work means going to school or going to work.

We prove these laws by truth tables.
Application: realization of electronic circuits.
Evaluation Laws
law and order in the Boolean algebra

When \texttt{not}, \texttt{and}, \texttt{or} occur in an expression, \texttt{not} is first evaluated, before \texttt{and}, and finally \texttt{or}.

De Morgan’s laws for simplifying expressions:

- \[\text{not } (( \text{not } x ) \text{ or } ( \text{not } y )) = x \text{ and } y\]
  Negating not being alive or not being well means being alive and being well.

- \[\text{not } (( \text{not } x ) \text{ and } ( \text{not } y )) = x \text{ or } y\]
  Negating not going to school and not going to work means going to school or going to work.

We prove these laws by truth tables.
Application: realization of electronic circuits.
Boolean Algebra, Flowcharts
Conditional Expressions

1. Boolean Algebra
   logical expressions
   pseudocode and flowcharts

2. Conditional Constructs
   conditional operators
   if, else, elif

3. Logic in Sage
   computing truth tables with Sage

4. Summary + Assignments
The function `abs` is available in Python:

```python
>>> abs(-3.5)
3.5
```

```python
>>> abs(3.5)
3.5
```

The mathematical definition of `abs(x)` as $y = |x|:

\[
|x| = \begin{cases} 
  x & \text{if } x \geq 0 \\
  -x & \text{if } x < 0
\end{cases}
\]
The Absolute Value

an example of an \texttt{if} statement

The function \texttt{abs} is available in Python:

\begin{verbatim}
>>> abs(-3.5)
3.5
>>> abs(3.5)
3.5
\end{verbatim}

The mathematical definition of \texttt{abs}(x) as \( y = |x| \):

\[
|x| = \begin{cases} 
  x & \text{if } x \geq 0 \\
  -x & \text{if } x < 0 
\end{cases}
\]
The Absolute Value
an example of an `if` statement

The function `abs` is available in Python:

```python
>>> abs(-3.5)
3.5
>>> abs(3.5)
3.5
```

The mathematical definition of `abs(x)` as $y = |x|:$

$$|x| = \begin{cases} 
  x & \text{if } x \geq 0 \\
  -x & \text{if } x < 0 
\end{cases}$$
Pseudocode

to formally describe algorithms

To develop and define an algorithm, we use *pseudocode*. Pseudocode is not real code, but to the reader it has the same properties as a formal language.

Example: print the absolute value of a number.
The number is given by the user.

In words, we could describe the program as:

ask the user for a number;
if the number is less than zero,
then print – before the number;
else print the number.

Mix of formal *if*, *then*, and *else* with English.
To develop and define an algorithm, we use *pseudocode*. Pseudocode is not real code, but to the reader it has the same properties as a formal language.

Example: print the absolute value of a number.
The number is given by the user.

In words, we could describe the program as:

- ask the user for a number;
  - *if* the number is less than zero,
    - *then* print – before the number;
  - *else* print the number.

Mix of formal *if*, *then*, and *else* with English.
To develop and define an algorithm, we use pseudocode. Pseudocode is not real code, but to the reader it has the same properties as a formal language.

Example: print the absolute value of a number. The number is given by the user.

In words, we could describe the program as:

ask the user for a number;
if the number is less than zero, then print – before the number;
else print the number.

Mix of formal if, then, and else with English.
Printing the absolute value of a number:

```
x = input()
x < 0?
    True
    print -x
    False
    print x
```

Flowcharts schematically represent the logical flow.
Flowcharts pictures of algorithms

Printing the absolute value of a number:

Flowcharts schematically represent the logical flow.
Flowcharts
pictures of algorithms

Printing the absolute value of a number:

\[ x = \text{input()} \]

\[ x < 0? \]

\[ \text{False} \quad \text{print } x \]

\[ \text{True} \]

\[ \text{print } -x \]

Flowcharts schematically represent the logical flow.
Printing the absolute value of a number:

```
x = input()

if x < 0:
    print -x
else:
    print x
```

Flowcharts schematically represent the logical flow.
Boolean Algebra, Flowcharts
Conditional Expressions

1. Boolean Algebra
   logical expressions
   pseudocode and flowcharts

2. Conditional Constructs
   conditional operators
   if, else, elif

3. Logic in Sage
   computing truth tables with Sage

4. Summary + Assignments
Comparison Operators
comparing values

The outcome of a comparison is True or False:

>>> 1 < 7
True

>>> 1 >= 7
False

The comparison operators:

= x == y is equal?
!= or <> x != y not equal?
< x < y less than?
> x > y greater than?
<= x <= y less or equal?
>= x >= y greater or equal?
Comparison Operators

comparing values

The outcome of a comparison is True or False:

```python
>>> 1 < 7
True
>>> 1 >= 7
False
```

The comparison operators:

```plaintext
==     x == y       is equal?
!= or <> x != y     not equal?
<      x < y       less than?
>      x > y       greater than?
<=     x <= y      less or equal?
>=     x >= y      greater or equal?
```
Boolean Operators
combining results of logical expressions

```python
>>> x = 3
>>> (x > 0) and (x < 10)
True
>>> (x < 0) or (x < 5)
True
>>> not (x < 0)
True
```

The brackets are not needed.

```
and  x and y  both True?
or   x or y   is one True?
not  not x   is False?
```
Boolean Operators

combining results of logical expressions

```python
>>> x = 3
>>> (x > 0) and (x < 10)
True
>>> (x < 0) or (x < 5)
True
>>> not (x < 0)
True
```

The brackets are not needed.

```plaintext
and x and y   both True?
or x or y     is one True?
not not x     is False?
```
Type `bool` is another elementary data type:

```python
>>> type(True)
<type 'bool'>
```

Although `True` is 1 and `False` is 0:

```python
>>> '%d' % True
'1'

>>> '%d' % False
'0'
```

Printing booleans as strings:

```python
>>> str(True)
'True'

>>> '%s' % True
'True'
```
Printing Booleans as numbers or strings

Type `bool` is another elementary data type:

```python
>>> type(True)
<type 'bool'>
```

Although `True` is 1 and `False` is 0:

```python
>>> '%d' % True
'1'
>>> '%d' % False
'0'
```

Printing booleans as strings:

```python
>>> str(True)
'True'
>>> '%s' % True
'True'
```
Type `bool` is another elementary data type:

```python
>>> type(True)
<type 'bool'>
```

Although `True` is 1 and `False` is 0:

```python
>>> '%d' % True
'1'
>>> '%d' % False
'0'
```

Printing booleans as strings:

```python
>>> str(True)
'True'
>>> '%s' % True
'True'
```
Boolean Algebra, Flowcharts
Conditional Expressions

1. Boolean Algebra
   - logical expressions
   - pseudocode and flowcharts

2. Conditional Constructs
   - conditional operators
     - if, else, elif

3. Logic in Sage
   - computing truth tables with Sage

4. Summary + Assignments
The if Statement
conditional execution of code

The syntax of the if:

```python
if < condition >:
    < statements when condition is True >
```

All statements to be executed only if the condition is true must be preceded by the right intendations!

Suppose we want to print the ’+’ for positive numbers.

With an if we could do it as follows:

```python
if x > 0:
    print ’+’
    print x
if x > 0:
    print ’+’
    print x
```

Only the second one works correctly for all x.
The if Statement
conditional execution of code

The syntax of the if:

```python
if < condition >:
    < statements when condition is True >
```

All statements to be executed only if the condition is true **must** be preceded by the right intendations!

Suppose we want to print the ’+’ for positive numbers.

With an if we could do it as follows:

```python
if x > 0:
    print ’+’
    print x
```

Only the second one works correctly for all x.
The if Statement
conditional execution of code

The syntax of the if:

```
if < condition >:
    < statements when condition is True >
```

All statements to be executed only if the condition is true
must be preceded by the right intendations!

Suppose we want to print the ’+’ for positive numbers.
With an if we could do it as follows:

```
if x > 0:
    print ’+’
    print x
```

Only the second one works correctly for all x.
The if Statement
conditional execution of code

The syntax of the if:

```python
if < condition >:
    < statements when condition is True >
```

All statements to be executed only if the condition is true
must be preceded by the right intendations!

Suppose we want to print the ’+’ for positive numbers.

With an if we could do it as follows:

```python
if x > 0:
    print ’+’
    print x
```

Only the second one works correctly for all x.
The if else Statement
choosing between two alternatives

The syntax of the if else:

if < condition >:
    < statements when condition is True >
else:
    < statements when condition is False >

Printing the absolute value of a number:

if x < 0:
    print -x
else:
    print x
The if else Statement
choosing between two alternatives

The syntax of the if else:

```
if < condition >:
    < statements when condition is True >
else:
    < statements when condition is False >
```

Printing the absolute value of a number:

```
if x < 0:
    print -x
else:
    print x
```
# L-8 MCS 260 Fri 12 Sep 2008  pass or fail
#
# This program asks to enter a number.
# If the input is larger than or equal to 80,
# then the user is congratulated,
# else we are sorry and ask to retry.
#

n = input('Enter your number : ')
if n >= 80:
    print('Congratulations. You passed!')
else:
    print('Sorry. Please try again...')
pass or fail

This program asks to enter a number. If the input is larger than or equal to 80, then the user is congratulated, else we are sorry and ask to retry.

n = input('Enter your number : ')
if n >= 80:
    print('Congratulations. You passed!')
else:
    print('Sorry. Please try again...')
Flowchart of Grade Scale

```python
n = input('Enter your number :')

if n >= 90:
    grade = 'A'
elif n >= 80:
    grade = 'B'
elif n >= 70:
    grade = 'C'
elif n >= 60:
    grade = 'D'
else:
    grade = 'F'

print(grade)
```
The if elif else Statement
choosing between multiple alternatives

The syntax of the if elif else:

if < condition 1 >:
    < statements when condition 1 is True >
elif < condition 2 >:
    < statements when condition 2 is True >
...
elif < condition n >:
    < statements when condition n is True >
else:
    < statements when everything is False >

The conditions are evaluated in the order as they appear.
The if elif else Statement
choosing between multiple alternatives

The syntax of the if elif else:

if < condition 1 >:
    < statements when condition 1 is True >
elif < condition 2 >:
    < statements when condition 2 is True >
...
elif < condition n >:
    < statements when condition n is True >
else:
    < statements when everything is False >

The conditions are evaluated in the order as they appear.
The grade for a course is represented by a letter. We compute the grade along a scale.

```python
n = input('Enter your number : ')
if n >= 90:
    grade = 'A'
elif n >= 80:
    grade = 'B'
elif n >= 70:
    grade = 'C'
elif n >= 60:
    grade = 'D'
else:
    grade = 'F'
print 'Your grade is ' + grade + '.'
```
The grade for a course is represented by a letter. We compute the grade along a scale.

```python
n = input('Enter your number : ')
if n >= 90:
    grade = 'A'
elif n >= 80:
    grade = 'B'
elif n >= 70:
    grade = 'C'
elif n >= 60:
    grade = 'D'
else:
    grade = 'F'
print 'Your grade is ' + grade + '.'
```
Nested if else statements
follow up questions

Statements following `if` or `else` can again be conditional. Nested if statements are good for dialogues with a user, when the outcome cannot be anticipated:

```python
ans = raw_input('happy ? (y/n) ')
if ans == 'n':
    ans = raw_input('bored ? (y/n) ')
    if ans == 'y':
        print 'class is soon over'
    else:
        print 'but it is Friday'
else:
    print 'keep up the good work'
```

Python gives an error when `=` is used instead of `==`. 
Nested if else statements
follow up questions

Statements following if or else can again be conditional. Nested if statements are good for dialogues with a user, when the outcome cannot be anticipated:

```python
ans = raw_input('happy ? (y/n) ')  
if ans == 'n':  
    ans = raw_input('bored ? (y/n) ')  
    if ans == 'y':  
        print 'class is soon over'  
    else:  
        print 'but it is Friday'  
else:  
    print 'keep up the good work'
```

Python gives an error when = is used instead of ==.
Nested if else statements
follow up questions

Statements following if or else can again be conditional. Nested if statements are good for dialogues with a user, when the outcome cannot be anticipated:

```python
ans = raw_input('happy ? (y/n) ')
if ans == 'n':
    ans = raw_input('bored ? (y/n) ')
    if ans == 'y':
        print 'class is soon over'
    else:
        print 'but it is Friday'
else:
    print 'keep up the good work'
```

Python gives an error when = is used instead of ==.
Boolean Algebra, Flowcharts
Conditional Expressions

1. Boolean Algebra
   logical expressions
   pseudocode and flowcharts

2. Conditional Constructs
   conditional operators
   if, else, elif

3. Logic in Sage
   computing truth tables with Sage

4. Summary + Assignments
Truth Tables in Sage
using SymbolicLogic

```
sage: logic = SymbolicLogic()
sage: s = logic.statement("a&b")

Instead of `and`, use the `&` operator.

sage: t = logic.truthtable(s)
sage: logic.print_table(t)
```

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
</tbody>
</table>
sage: logic = SymbolicLogic()
sage: s = logic.statement("a&b")

Instead of and, use the & operator.

sage: t = logic.truthtable(s)
sage: logic.print_table(t)

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
</tbody>
</table>
Truth Tables in Sage
using SymbolicLogic

```
sage: logic = SymbolicLogic()
sage: s = logic.statement("a&b")

Instead of and, use the & operator.

sage: t = logic.truthtable(s)
sage: logic.print_table(t)
```

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
</tbody>
</table>
Truth Tables continued

the or operation

Sage session continued...

```
sage: s = logic.statement("a|b")

Instead of or, use the | operator.

sage: logic.print_table(logic.truthtable(s))

a | b | value |
---------------
False | False | False |
False | True | True |
True | False | True |
True | True | True |
```
Truth Tables continued
the or operation

Sage session continued...

```
sage: s = logic.statement("a\|b")
Instead of or, use the | operator.

sage: logic.print_table(logic.truthtable(s))
```

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
</tbody>
</table>
Sage session continued...

```
sage: s = logic.statement("!a")
Instead of not, use the ! operator.

sage: t = logic.truthtable(s)
sage: logic.print_table(t)
a | value |
----------------
False | True |
True  | False |
```
The first law of De Morgan:

\[
\text{not } (( \text{not } x) \text{ or } (\text{not } y)) = x \text{ and } y
\]

Sage session continued ...

```
sage: law = '!((!x) | (!y))'
sage: s = logic.statement(law)
sage: logic.print_table(t)
```

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
</tbody>
</table>

We recognize the truth table for \(x \text{ and } y\).
Proving De Morgan’s Law

with truth tables

The first law of De Morgan:

\[
\text{not (( not } x \text{ ) or ( not } y \text{ )) } = x \text{ and } y
\]

Sage session continued ...

```
sage: law = '!((!x)|(!y))'
sage: s = logic.statement(law)
sage: logic.print_table(t)

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
</tbody>
</table>
```

We recognize the truth table for \( x \) and \( y \).
Summary + Assignments

In this lecture we covered

- section 1.1 in *Computer Science. An Overview*
- start of chapter 5 of *Python Power!*

Assignments:

1. Omit the brackets in De Morgan’s Laws and create a truth table to evaluate the expressions.
2. Write pseudocode and draw of flowchart for a program that reads in a positive number and prints out whether the number is divisible by 2, 3, 5, or not.
3. Give Python code for the program in assignment 2.
4. Define a Python dictionary for the truth table of $x$ and $y$. 