Many of the slides I use were created by Dr. Chuck Severance, the author of our book.

- You’ll typically recognize them by their black backgrounds.
- They are *Copyright 2010–2014 Charles R. Severance* and are made available under a Creative Commons Attribution 4.0 License.

Many of the slides I use were created by Dr. Jeremy Abramson of USC/ISI. He simply gave them to me to use in this class.

Many of the slides I use were created by me.
Clarifying PEMDAS
Order of Precedence

- **PEMDAS**
  - Parentheses
  - Exponentiation
  - Multiplication
  - Division
  - Addition
  - Subtraction
  - Left to right

- Which operator takes precedence?
  
  $1 + 2 \times 3 - 4 \div 5 \times 6$

- So $5 \div 5 \times 2$ is 2, not 0

- When in doubt, just use parentheses!
A Few Variations on Things We’ve Seen
## Types We Have Seen

<table>
<thead>
<tr>
<th>Types</th>
<th>Type</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer Numbers</td>
<td>int</td>
<td>( x = 32 )</td>
</tr>
<tr>
<td>Floating Point Numbers</td>
<td>float</td>
<td>( x = 212.00 )</td>
</tr>
<tr>
<td>Strings</td>
<td>str</td>
<td>( x = \text{‘forgetMeNot’} )</td>
</tr>
</tbody>
</table>
# Types We Haven’t Seen

<table>
<thead>
<tr>
<th>Type</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Boolean Values</strong></td>
<td></td>
</tr>
<tr>
<td>bool</td>
<td><code>x = True</code></td>
</tr>
<tr>
<td></td>
<td><code>y = False</code></td>
</tr>
<tr>
<td><strong>List of Values</strong></td>
<td></td>
</tr>
<tr>
<td>list</td>
<td><code>x = [4, 3, 1, 2]</code></td>
</tr>
<tr>
<td><strong>Dictionary</strong></td>
<td></td>
</tr>
<tr>
<td>dict</td>
<td><code>y = {‘first’: ‘Yigal’, ‘last’: ‘Arens’}</code></td>
</tr>
</tbody>
</table>

- `x[2]` is 1
- `y[‘last’]` is ‘Arens’
More Printing

• What happens if we need to print these lines?
  I am 5’9” tall
  I want to see the 5’9\” backslash!
## Characters We Haven’t Seen

<table>
<thead>
<tr>
<th>Escape Sequence</th>
<th>What It Does</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>\</code></td>
<td>Backslash ()</td>
</tr>
<tr>
<td><code>'</code></td>
<td>Single-quote (')</td>
</tr>
<tr>
<td><code>&quot;</code></td>
<td>Double-quote (&quot;)</td>
</tr>
<tr>
<td><code>\a</code></td>
<td>ASCII bell (BEL)</td>
</tr>
<tr>
<td><code>\b</code></td>
<td>ASCII backspace (BS)</td>
</tr>
<tr>
<td><code>\n</code></td>
<td>ASCII linefeed (LF)</td>
</tr>
<tr>
<td><code>\r</code></td>
<td>Carriage Return (CR)</td>
</tr>
<tr>
<td><code>\t</code></td>
<td>Horizontal Tab (TAB)</td>
</tr>
</tbody>
</table>
print 'Take me to your leader'
print "Don’t touch that dial!"
print "That’s 2\" in diameter."

x = 3
print 'We would like', x, 'cups of tea.'
Printing We Haven’t Seen

x = 3.4
print ‘We would like %s cups of tea’ % x
print ‘We would like %i cups of tea’ % x
print ‘We would like %f cups of tea’ % x
print ‘We would like %r cups of tea’ % x

print ‘%s%s’ % (‘One’, ‘Two’)
Conditional Execution, Functions
Conditional Execution

Program:

```python
x = 5
if x < 10:
    print 'Smaller'
if x > 20:
    print 'Bigger'
print 'Finis'
```

Diagram:

```
x = 5
  ↓
x < 10?
  ↓
  No
  ↓
print 'Smaller'
  ↓
  Yes
  ↓
x > 20?
  ↓
  Yes
  ↓
print 'Bigger'
  ↓
  No
  ↓
print 'Finis'
```
Conditional Execution

- So far we’ve seen Python execute lines in order
  - One line of code after the next, whether in the interpreter or from a file
    ```
    >>> x = "10"
    >>> y = x + "1"
    >>> print y
    ```
- Conditional execution allows you to say
  - “Only if this is true, do that”
- But how do we know if something is true?
- Let’s spend some time on that question
Boolean Variables

• Boolean (bool) is another type of variable (like int, float, str) but it can only take two values:
  • True or False (note capitalization!)
  
  >>> type(True)
  <type 'bool'>

• By the way, what are the outputs of the following?
  >>> type("True")
  >>> type(true)
Comparison Operators

- **Comparison operators** operate on *operands* which are Python variables or constants. The comparison operators are:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>x == y</code></td>
<td>is equal to</td>
</tr>
<tr>
<td><code>x != y</code></td>
<td>not equal to</td>
</tr>
<tr>
<td><code>x &gt; y</code></td>
<td>greater than</td>
</tr>
<tr>
<td><code>x &lt; y</code></td>
<td>smaller than</td>
</tr>
<tr>
<td><code>x &gt;= y</code></td>
<td>greater than or equal to</td>
</tr>
<tr>
<td><code>x &lt;= y</code></td>
<td>less than or equal to</td>
</tr>
<tr>
<td><code>x is y</code></td>
<td>is the same as</td>
</tr>
<tr>
<td><code>x is not y</code></td>
<td>is not the same as</td>
</tr>
</tbody>
</table>

- Each of the above is a “Boolean expression”
- Boolean expressions ask a question, to which the answer is “yes” or “no”, i. e., True or False. They do not change values.
Equality Is Not Assignment!

• These are not the same:
  my_var = 10
  my_var == 10

• If you confuse the two, Python will either let you know (e.g., it’s syntax error), or your program will fail in another way…
Logical Operators

• *Logical operators* operate on Boolean expressions/values
  • On things like \((x == 'Bobby'), False, (y > 2)\)
• There are three logical operators:
  • `and`
  • `or`
  • `not`
• Their meaning in Python is essentially their common-sense meaning
• Are you familiar with “truth tables”?
Logical Operator Caveats

- You can plug just about anything into a logical operator
  - For example, nonzero integers will evaluate to True.
- You do not want to do this!
- Stick with things you know evaluate explicitly to Boolean variables (i.e. Boolean expressions)
Program:

```python
x = 5
if x < 10:
    print('Smaller')
if x > 20:
    print('Bigger')
print('Finis')
```

Output:

Smaller
Finis
Conditional Execution

```python
if x == 5:
    print ‘x is definitely 5’
print ‘x may or may not be 5’
```

- Note the colon!
- Note the indentation!
- Why does the last line above say what it does?
Indentation

• **Increase indent** indent after an *if* statement or *for* statement (after `:`)

• **Maintain indent** to indicate the **scope** of the block (which lines are affected by the *if*/*for*)

• **Reduce indent** *back to* the level of the *if* statement or *for* statement to indicate the end of the block

• **Blank lines** are ignored - they do not affect indentation

• **Comments** on a line by themselves are ignored with regard to indentation
Warning: Turn Off Tabs!!

• Most text editors can turn tabs into spaces - make sure to enable this feature
  > NotePad++: Settings -> Preferences -> Language Menu/Tab Settings
  > TextWrangler: TextWrangler -> Preferences -> Editor Defaults

• Python cares a *lot* about how far a line is indented. If you mix tabs and spaces, you may get “indentation errors” even if everything looks fine

Please do this now while you are thinking about it so we can all stay sane...
This will save you much unnecessary pain.
Try Some Code
Else

• With an else statement, only one block will be executed, either the if block or the else block

```python
if age >= 18:
    print ‘you can vote!’
else:
    print “you’re too young to vote!”
```

• (Note the use of double quotes. Why did I do that?)

• Same indentation rules apply to else as to if

• And note the colon again!
Two-way using else:

```python
x = 4
if x > 2:
    print 'Bigger'
else:
    print 'Smaller'
print 'All done'
```
Two-way using else:

```python
x = 4
if x > 2 :
    print 'Bigger'
else :
    print 'Smaller'
print 'All done'
```

```
x = 4
x > 2

if x > 2 :
    print 'Smaller'
else :
    print 'Bigger'
print 'All done'
```
Chained Conditionals – *elif*

- *elif* can be thought of as “else if”
- If this is true <do something>, otherwise, if this is true <do something else> otherwise, if this is true <do something else> . . .
- Note: One of an *if/else* has to execute. That is not true of *elif*
  - But an *else* at the end will catch the case where all the ones above were false
Multi-way

```python
if x < 2 :
    print 'small'
elif x < 10 :
    print 'Medium'
else :
    print 'LARGE'
print 'All done'
```
Multi-way

```python
x = 0
if x < 2 :
    print 'small'
elif x < 10 :
    print 'Medium'
else :
    print 'LARGE'
print 'All done'
```

![Flowchart diagram](image-url)
x = 5
if x < 2 :
    print 'small'
elif x < 10 :
    print 'Medium'
else :
    print 'LARGE'
print 'All done'
Multi-way

\[ x = 20 \]

if \( x < 2 \):
    print 'small'
elif \( x < 10 \):
    print 'Medium'
else:
    print 'LARGE'
print 'All done'
Can End with Else, or Not

Multi-way

# No Else
x = 5
if x < 2 :
    print 'Small'
elif x < 10 :
    print 'Medium'
print 'All done'

if x < 2 :
    print 'Small'
elif x < 10 :
    print 'Medium'
elif x < 20 :
    print 'Big'
elif x < 40 :
    print 'Large'
elif x < 100:
    print 'Huge'
else :
    print 'Ginormous'

But note that ‘All done’ will **always** print!
Multi-way Puzzles

Which will never print?

```python
if x < 2 :
    print 'Below 2'
elif x >= 2 :
    print 'Two or more'
else :
    print 'Something else'
```

```python
if x < 2 :
    print 'Below 2'
elif x < 20 :
    print 'Below 20'
elif x < 10 :
    print 'Below 10'
else :
    print 'Something else'
```

Why might you nevertheless want that last `else`?
Nested Conditionals

• The <do stuff> in an if (or else or elif) block can be another if statement!

• You can nest as many as you’d like

```python
if age > 21:
    print "You can drink if you want to."
else:
    if age > 16:
        print "At least you can drive!"
    else:
        print "You’re too young to have any fun!"
```
Don’t Forget the Logical Operators

• Logical operators: and, or, not
• They can be used to simplify conditionals
• For example,
  
  ```python
  if x > 0:
      if x < 10:
          print ‘x is not too big.’
  ```

  • (Note indentation!)

• That code can be replaced by the equivalent
  
  ```python
  if x > 0 and x < 10:
      print ‘x is not too big.’
  ```
try/except

• Sometimes things do not go according to plan

• For example, you may get input you don’t expect

# Convert Fahrenheit to Celsius

f_temp = raw_input('Temp in Fahrenheit ')
f_temp = float(f_temp)
c_temp = (f_temp - 32.0) * 5.0 / 9.0
print f_temp, 'Fahrenheit is', c_temp, 'Celsius'

• What if someone types something other than a number?

• Let’s check it out
try/except

try:
    <a block of code>
except:
    <a block of code that will be executed only if there was an error in the block above>

• try/except blocks are in some sense similar to if/else blocks

• A statement in the “try” block is said to throw the exception, and the “except” block catches it
Functions

• Often you want to repeat blocks of code in your program
  • For example, you may need to convert temperature in Fahrenheit to Celsius multiple times
  • It would be wasteful to have to include the same code multiple times
  • A function may take *input(s)* and may provide *output*
Examples of Functions

- We’ve already seen several *built-in* functions, e. g.,
  - `int('12')`, `raw_input("> ")`, `type("string")`, `float(6)`
- Do they take input(s)? Do they have output?
- There are many other built-in functions, for instance:
  - `max()`, `min()`
\[ \text{big} = \max(\text{'Hello world'}) \]

\[ \text{Argument} \]

\[ \text{Assignment} \]

\[ 'w' \]

\[ \text{Result} \]

\[
>>> \text{big} = \max(\text{'Hello world'}) \\
>>> \text{print big} \\
\text{w} \\
>>> \text{tiny} = \min(\text{'Hello world'}) \\
>>> \text{print tiny}
\]
Python Functions

• There are two kinds of functions in Python.

  › **Built-in functions** that are provided as part of Python - raw_input(), type(), float(), int() ...

  › **Functions that we define ourselves** and then use

• We treat the built-in function names as “new” **reserved words** (i.e., we avoid them as variable names)
Max Function

A function is some stored code that we use. A function takes some input and produces an output.

Guido wrote this code.

```python
>>> big = max('Hello world')
>>> print big
w
```
Max Function

A function is some stored code that we use. A function takes some input and produces an output.

```
def max(inp):
    blah
    blah
    for x in y:
        blah
        blah

>>> big = max('Hello world')
>>> print big
w

'Hello world' (a string)  ->  'w' (a string)

Guido wrote this code
Building our Own Functions

• We create a new function using the `def` keyword followed by optional parameters in parentheses

• We indent the body of the function

• This defines the function but does not execute the body of the function

```python
def print_lyrics():
    print "I'm a lumberjack, and I'm okay."
    print 'I sleep all night and I work all day.'
```
Arguments

• An argument is a value we pass into the function as its input when we call the function

• We use arguments so we can direct the function to do different kinds of work when we call it at different times

• We put the arguments in parentheses after the name of the function

\[ \text{big} = \text{max('Hello world')} \]
A parameter is a variable which we use in the function definition. It is a “handle” that allows the code in the function to access the arguments for a particular function invocation.

```python
>>> def greet(lang):
...     if lang == 'es':
...         print('Hola')
...     elif lang == 'fr':
...         print('Bonjour')
...     else:
...         print('Hello')
...
>>> greet('en')
Hello
>>> greet('es')
Hola
>>> greet('fr')
Bonjour
>>> 
```
Return Values

Often a function will take its arguments, do some computation, and return a value to be used as the value of the function call in the calling expression. The return keyword is used for this.

```python
def greet():
    return "Hello"

print greet(), "Glenn"  # Hello Glenn
print greet(), "Sally"   # Hello Sally
```
Return Value

• A “fruitful” function is one that produces a result (or return value)

• The return statement ends the function execution and “sends back” the result of the function

```python
>>> def greet(lang):
...     if lang == 'es':
...         return 'Hola'
...     elif lang == 'fr':
...         return 'Bonjour'
...     else:
...         return 'Hello'
...
>>> print greet('en'),'Glenn'
Hello Glenn
>>> print greet('es'),'Sally'
Hola Sally
>>> print greet('fr'),'Michael'
Bonjour Michael
>>> 
```
Multiple Parameters / Arguments

- We can define more than one parameter in the function definition
- We simply add more arguments when we call the function
- We match the number and order of arguments and parameters

```python
def addtwo(a, b):
    added = a + b
    return added

x = addtwo(3, 5)
print(x)
8```
Void (non-fruitful) Functions

- When a function does not return a value, we call it a “void” function
- Functions that return values are “fruitful” functions
- Void functions are “not fruitful”
Functions Must Be Defined Before Use

- Built-in functions are all predefined and may be used at any time.
- Python has many modules that contain function definitions — you must \textit{load} those modules before using their functions.
  - See section 4.5 in the book for how to get many math functions.
- You must define your own functions before you can use them.
To function or not to function...

- Organize your code into “paragraphs” - capture a complete thought and “name it”

- Don’t repeat yourself - make it work once and then reuse it

- If something gets too long or complex, break it up into logical chunks and put those chunks in functions

- Make a library of common stuff that you do over and over - perhaps share this with your friends...