Outcomes:
- Fluency in (Python) procedural programming
  - Usage of assignments, conditionals, and loops
  - Ability to design Python modules and programs
- Competency in object-oriented programming
  - Ability to write programs using objects and classes.
- Knowledge of searching and sorting algorithms
  - Knowledge of basics of vector computation

Website:
- [www.cs.cornell.edu/courses/cs1110/2015fa/](http://www.cs.cornell.edu/courses/cs1110/2015fa/)

Class Structure

- **Lectures.** Every Tuesday/Thursday
  - Not just slides; interactive demos almost every lecture
  - Because of enrollment, please stay with your section
  - Semi-Mandatory. 1% Participation grade from iClickers
- **Section/labs.** ACCEL Lab, Carpenter 2nd floor
  - The “overflow sections” are in Phillips 318
  - Guided exercises with TAs and consultants helping out
    - Tuesday: 12:20, 1:25, 2:30, 3:35
    - Wednesday: 10:10, 11:15, 12:20, 1:25, 2:30, 3:35, 7:20
  - Contact Jessica (jd648@cornell.edu) for section conflicts
  - Mandatory. Missing more than 2 lowers your final grade

Class Materials

- **Textbook.** *Think Python* by Allen Downey
  - Supplemental text; does not replace lecture
  - Hardbound copies for sale in Campus Store
  - Book available for free as PDF or eBook
- **iClicker.** Acquire one by Thursday
  - Will periodically ask questions during lecture
  - Will get credit for answering – even if wrong
  - iClicker App for smartphone is not acceptable
- **Python.** Necessary if you want to use own computer
  - See course website for how to install the software

Things to Do Before Next Class

1. Register your iClicker
   - Does not count for grade if not registered
2. Enroll in Piazza
3. Sign into CMS
   - Complete the Quiz
   - Complete Survey 0
4. Read the textbook
   - Chapter 1 (browse)
   - Chapter 2 (in detail)

Getting Started with Python

- Designed to be used from the “command line”
  - OS X/Linux: Terminal
  - Windows: Command Prompt
  - Purpose of the first lab
- Once installed type “python”
  - Starts an interactive shell
  - Type commands at >>>
  - Shell responds to commands
  - Can use it like a calculator
  - Use to evaluate expressions

Python and Expressions

- An expression represents something
  - Python evaluates it (turns it into a value)
  - Similar to what a calculator does
- Examples:
  - `2.3`
    - Literal (evaluates to self)
  - `(3 * 7 + 2) * 0.1`
    - An expression with four literals and some operators
Type: Set of values and the operations on them

- **Type int** represents integers
  - values: \( ..., -3, -2, -1, 0, 1, 2, 3, 4, 5, ... \)
  - Integer literals look like this: 1, 48, 4508450 (no commas or periods)
  - operations: +, -, *, /, **, unary -

- **Principle:** operations on int values must yield an int
  - Example: 1 / 2 rounds result down to 0
  - Companion operation: \( \% \) (remainder)
  - \( 7 \% 3 \) evaluates to 1, remainder when dividing 7 by 3
  - Operator / is not an int operation in Python 3 (use // instead)

- **Type float** (floating point) represents real numbers
  - values: distinguished from integers by decimal points
    - In Python a number with a "." is a float literal (e.g. 12.0)
    - Without a decimal a number is an int literal (e.g. 12)
  - operations: +, -, *, /, **, unary -
    - The meaning for floats differs from that for ints
  - Example: 1/0.0 evaluates to 0.5

- **Exponent notation** is useful for large (or small) values
  - \( -2.51e6 \) is \( -2.51 \times 10^6 \) or \( -2510000 \)
  - \( 2.51e-6 \) is \( 2.51 \times 10^{-6} \) or \( 0.0000251 \)

Floats Have Finite Precision

- Python stores floats as binary fractions
  - Integer mantissa times a power of 2
  - Example: 1.25 is \( 5 \times 2^{-2} \)

- Impossible to write most real numbers this way exactly
  - Similar to problem of writing \( 1/3 \) with decimals
  - Python chooses the closest binary fraction it can

- This approximation results in **representation error**
  - When combined in expressions, the error can get worse
  - Example: type \( 0.1 + 0.2 \) at the prompt >>>

Type: Set of values and the operations on them

- **Type String or str** represents text
  - values: any sequence of characters
  - operation(s): + (concatenation, or concatenation)

- **String literal**:
  - sequence of characters in quotes
    - Double quotes: " abox$%$d" or "Hello World!"
    - Single quotes: 'Hello World!'
  - Concatenation can only apply to strings.
    - 'ab' + 'cd' evaluates to 'abcd'
    - 'ab' + 2 produces an error

Converting Values Between Types

- **Basic form:** \( \text{type}(value) \)
  - float(2) converts value 2 to type float (value now 2.0)
  - int(2.6) converts value 2.6 to type int (value now 2)
  - Explicit conversion is also called "casting"

- **Narrow to wide:** bool \( \Rightarrow \) int \( \Rightarrow \) float

- **Widening**: Python does automatically if needed
  - Example: 1/2.0 evaluates to 0.5 (casts 1 to float)

- **Narrowing**: Python **never** does this automatically
  - Narrowing conversions cause information to be lost
  - Example: float(int(2.6)) evaluates to 2.0