CS 1110 Fall 2013: Walker White

- **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/cs1110/2013fa/

Class Structure

- **Lectures.** Every Tuesday/Thursday
  - Not just slides; interactive demos almost every lecture
  - Please stay in your lecture (no room to move between)
  - **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, 7:30
  - Contact Molly (mjt264@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 **next Tuesday**
  - Will periodically ask questions during lecture
  - Used to judge class understanding
  - Will get credit for answering – even if wrong
- **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
   - Quiz: About the Course
   - Complete Survey 0
4. Read the textbook
   - Chapter 1 (browse)
   - Chapter 2 (in detail)

- Everything is on website!
  - Piazza instructions
  - Class announcements
  - Consultant calendar
  - Reading schedule
  - Lecture slides
  - Exam dates
- Check it regularly:
  - www.cs.cornell.edu/courses/cs1110/2013fa/

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** (integer):
  - values: …, –3, –2, –1, 0, 1, 2, 3, 4, 5, …
    - "Whole" numbers w/o decimals
  - operations: +, –, *, /, **, unary –
    - multiply
    - to power of
- **Principal**: 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: Set of values and the operations on them

- Type **float** (floating point):
  - values: (approximations of) real numbers
    - In Python a number with a "." is a **float** literal (e.g. 2.0)
    - Without a decimal a number is an **int** literal (e.g. 2)
  - operations: +, –, *, /, **, unary –
    - But meaning is different for floats
    - **Example**: 1.0/2.0 evaluates to 0.5
- **Exponent notation** is useful for large (or small) values
  - `-22.51e6` is $-22.51 * 10^6$ or –22510000
  - `22.51e-6` is $22.51 * 10^{-6}$ or 0.00002251
    - A second kind of **float** literal

Representation Error

- Python stores floats as **binary fractions**
  - Integer mantissa times a power of 2
  - Example: 12.5 is $10 * 2^{-3}$
    - mantissa
    - exponent
- Impossible to write every number 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 boolean or **bool**:
  - values: **True**, **False**
    - Boolean literals are just `True` and `False` (have to be capitalized)
  - operations: not, and, or
    - not b: **True** if b is false and **False** if b is true
    - b and c: **True** if both b and c are true; **False** otherwise
    - b || c: **True** if b is true or c is true; **False** otherwise
- Often come from comparing **int** or **float** values
  - Order comparison: i < j i <= j i >= j i > j
  - Equality, inequality: i == j i != j
    - = means something else!

Type: Set of values and the operations on them

- Type String or **str**:
  - values: any sequence of characters
  - operation(s): + (catenation, or concatenation)
- **String literal**: sequence of chars in quotes
  - Double quotes: `" abcex3$g<&e"` 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: *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 ⇒ int ⇒ 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