Welcome to CS61A!
This week

- Yes, we don’t all fit in Wheeler! Anyone who does not insist on seeing my face can use the screencasts and the posted lecture slides, and not actually come to lecture.

- However, we will have activities to perform with many lectures, just to encourage you to keep up. You can do these offline, too.

- Please see the course web site (http://cs61a.org), esp. the Course Info link. (But bear with us: the web site is under construction).

- This week, lab was optional, but you will want to complete it offline (see http://cs61a.org/lab/lab00). In particular, it contains a link you can use to get an account (which you will need) and register with our system.

- You can switch labs without using TeleBEARS after this week (via a Google form). Please go to assigned sections this week (or SOME section).

- Next week, labs (between Monday and Wednesday lecture) and discussions meet according to the published schedule.
What’s This Course About?

• This is a course about *programming*, which is the art and science of constructing artifacts (“programs”) that perform computations or interact with the physical world.

• To do this, we have to learn a *programming language* (Python in our case), but programming means a great deal more, including
  - Design of what programs do.
  - Analysis of the performance of programs.
  - Confirmation of their correct operation.
  - Management of their complexity.

• This course is about the “big ideas” of programming. We expect most of what you learn to apply to any programming language.
Course Organization

- **Readings** cover the material. Try to do them before...
- **Lectures** summarize material, or present alternative “takes” on it.
- **Laboratory exercises** are “finger exercises” designed to introduce a new topic or certain practical skills. Unlimited collaboration.
- **Homework assignments** are more involved than lab exercises and often require some thought. Plan is to have them due on Monday. Feel free to discuss the homework with other students, but turn in your own solutions.
- **Projects** are four larger multi-week assignments intended to teach you how to combine ideas from the course in interesting ways. We’ll be doing at least some of these in pairs.
- **Use the discussion board (Piazza) for news, advice, etc.**
**Mandatory Warning**

- We allow unlimited collaboration on labs.
- On homework, feel free to collaborate, but try to keep your work distinct from everyone else's.
- Likewise on projects, except that you and your partner submit a joint project.
- You can take small pieces of code within reason (ask if unsure), but you **must** attribute it!
- Otherwise, copying is against the Code of Conduct, and generally results in penalties.
- Most out-and-out copying is due to desperation and time pressure. Instead, see us if you're having trouble; that's what we're here for!
What's In A Programming Language?

- **Values**: the things programs fiddle with;
- **Primitive operations** (on values);
- **Combining mechanisms**, which glue operations together;
- **Some predefined names** (the “library”);
- **Definitional mechanisms**, which allow one to introduce symbolic names and (in effect) to extend the library.
Python Values (I)

- Python has a rich set of values, including:

<table>
<thead>
<tr>
<th>Type</th>
<th>Values</th>
<th>Literals (Denotations)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integers</td>
<td>0, -1, 16, 13</td>
<td>0, -1, 0o20, 0b1101</td>
</tr>
<tr>
<td></td>
<td>36893488147419103232</td>
<td>0x2000000000000000000000</td>
</tr>
<tr>
<td>Boolean (truth) values</td>
<td>true, false</td>
<td>True, False</td>
</tr>
<tr>
<td>“Null”</td>
<td></td>
<td>None</td>
</tr>
<tr>
<td>Functions</td>
<td></td>
<td>operator.add, operator.mul, operator.lt, operator.eq</td>
</tr>
</tbody>
</table>

- Functions take values and return values (including functions). Thus, the definition of “value” is **recursive**: definition of function refers to functions.

- They don’t look like much, perhaps, but with these values we can represent anything!
Python Values (II)

• ...but not conveniently. So now we add more complex types, including:

<table>
<thead>
<tr>
<th>Type</th>
<th>Values</th>
<th>Literals (Denotations)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strings</td>
<td>pear,</td>
<td>&quot;pear&quot;</td>
</tr>
<tr>
<td></td>
<td>I ♥ NY</td>
<td>&quot;I \u2661 NY&quot;</td>
</tr>
<tr>
<td></td>
<td>Say &quot;Hello&quot;</td>
<td>&quot;Say &quot;Hello&quot;&quot;</td>
</tr>
<tr>
<td>Tuples</td>
<td>()</td>
<td>range(10), range(1, 5)</td>
</tr>
<tr>
<td>Ranges</td>
<td>0–10, 1–5</td>
<td></td>
</tr>
<tr>
<td>Lists</td>
<td>[]</td>
<td>[1, &quot;Hello&quot;, (3, 5)]</td>
</tr>
<tr>
<td></td>
<td>[ x**3 for x in range(5) ]</td>
<td></td>
</tr>
<tr>
<td>Dictionaries</td>
<td>{ &quot;Paul&quot; : 60, &quot;Ann&quot; : 59,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&quot;John&quot; : 56 }</td>
<td></td>
</tr>
<tr>
<td>Sets</td>
<td>{}, {1,2},</td>
<td>set([]), { 1, 2 },</td>
</tr>
<tr>
<td></td>
<td>{x</td>
<td>0 ≤ x &lt; 20 ∧ x is prime}</td>
</tr>
</tbody>
</table>
What Values Can Represent

- The tuple type (as well as the list, dictionary, set class types) give Python the power to represent just about anything.

- In fact, we could get away with allowing just pairs: tuples with two elements:
  - Tuples can contain tuples (and lists can contain lists), which allows us to get as fancy as we want.
  - Instead of (1, 2, 7), could use (1, (2, (7, None))),
  - But while elegant, this would make programming tedious.
Python's Primitive Operations

- Literals are the base cases.
- Functions in particular are the starting point for creating programs:
  \[
  \text{sub(truediv(mul(add(add(3, 7), 10), sub(1000, 8)), 992), 17)}
  \]
- To evaluate a function call:
  - Evaluate the caller (left of the parentheses).
  - Evaluate the arguments (within the parentheses).
  - The caller then tells what to do and what value to produce from the operand's values.
- For the convenience of the reader, though, Python employs a great deal of "syntactic sugar" to produce familiar notation:
  \[
  (3 + 7 + 10) \times (1000 - 8) / 992 - 17
  \]
Evaluating a Function Call

- Consider

\[(3 + 7 + 10) \times (1000 - 8) / 992 - 17\]

which “sugars”

\[\text{sub(truediv(mul(add(add(3, 7), 10), sub(1000, 8)), 992), 17)}\]

- The numerals all evaluate in the obvious way.

- Then proceed from the inside out:

  \[\text{sub(truediv(mul(add(add(3, 7), 10), sub(1000, 8)), 992), 17)}\]
  \[\text{sub(truediv(mul(add(10, 10), 992), 992), 17)}\]
  \[\text{sub(truediv(mul(20, 992), 992), 17)}\]
  \[\text{sub(truediv(19840, 992), 17)}\]
  \[\text{sub(20, 17)}\]

\[3\]
Combining and Defining

• Certain primitives are needed to allow *conditional execution*:

```python
print(1 if x > 0 else -1 if x < 0 else 0)
# or equivalently
if x > 0:
    print(1)
elif x < 0:
    print(-1)
else:
    print(0)
```

• Defining a new function:

```python
def signum(x):
    return 1 if x > 0 else -1 if x < 0 else 0
```

Now `signum` denotes a function.

• Doesn't look like we have a lot, but in fact we already have enough to implement *all the computable functions on the integers!*
Getting repetition

- Haven’t explicitly mentioned any construct to “repeat $X$ until ...” or “repeat $X$ $N$ times.” Technically, none is needed.

- Suppose you’d like to compute $x + 2x^2 + 3x^3 + \ldots + Nx^N$ for any $N$:

  ```python
def series(x, N):
    if N == 1:
      return x
    else:
      return N * x**N + series(x, N-1)
  ```

- But again, we have syntactic sugar (which is the usual approach in Python):

  ```python
def series(x, N):
    S = 0
    for k in range(1, N+1):
      S += k * x**k
    return S
  ```
A Few General Rules

• Whatever the assignment, start now.
• "Yes, that’s really all there is. Don’t fight the problem."
• Practice is important. Don’t just assume you can do it; do it!
• ALWAYS feel free to ask us for help.
• DBC
• RTFM
• Have fun!