Welcome to CS61A!

This week

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

• Please see the course web site (http://cs61a.org), esp. the announcements and Course Info link. (Bear with us: the web site is under construction.)

• If you did not complete the lab this week, you should try to do it offline (see http://cs61a.org/lab/lab00).

• Next week, labs (between Monday and Wednesday lecture) and discussions meet according to the published schedule.

• Try to find a lab and discussion section with the same TA, if possible. If enrolled, don’t worry about changing things on CalCentral. If waitlisted, choose some sections that your partner submits a discussion/proficiency check for.

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

• Use the discussion board (Piazza) for news, advice, etc.

• 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 with your professor’s permission, but you must attribute it!

• Otherwise, copying is against the Code of Conduct, and generally results in penalties, a grade of 20% or 0 on the project.

• If you’re having trouble, that’s what the TA and office hours are for. Just ask! We’re here to help you understand the material.

What’s In A Programming Language?

• Values: the things programs fiddle with.

• Definitional mechanisms, which allow one to introduce symbols for values.

• Combining mechanisms, which glue operations together.

• Primitive operations (on values):

  • Values: The things programs fiddle with:

  • What’s in a Programming Language?
Python Values (I)

- **Type**: Python has a rich set of values, including:
  - **Integers**: 0, -1, 116, 13, 36893488147419103232, 0x20000000000000000
  - **Boolean (truth) values**: true, false
  - **None**: None
  - **Functions**: operator.add, operator.mul, operator.lt, operator.eq

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)

- **Strings**: pear, "pear", "I ♥ NY", "I ♥ NY"
- **Tuples**: (), (1, "Hello", (3, 5))
- **Ranges**: 0–10, 1–5
- **Lists**: [], [1, "Hello", (3, 5)]
- **Dictionaries**: { "Paul" : 60, "Ann" : 59, "John" : 56 }
- **Sets**: {}, {1, 2}, set([]), {1, 2}, {x | 0 ≤ x < 20} { x for x in range(20) if prime(x) }

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:
  - 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

Combining and Defining

- Certain primitives are needed to allow conditional execution:
  - `print(1 if x > 0 else -1 if x < 0 else 0)`
  - If `x` is greater than 0, print 1; if less than 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, in fact we already have enough to implement all the computable functions on the integers.

Evaluating a Function Call

- Consider `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:
  - sub(truediv(mul(add(add(3, 7), 10), sub(1000, 8)), 992), 17)
  - sub(truediv(mul(10, 10), 992), 17)
  - sub(truediv(19840, 992), 17)
  - sub(20, 17)
  - 3

Combining and Defining

- Certain primitives are needed to allow conditional execution:
  - `print(1 if x > 0 else -1 if x < 0 else 0)`
  - If `x` is greater than 0, print 1; if less than 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, in fact we already have enough to implement all the computable functions on the integers.

Evaluating a Function Call

- Consider `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:
  - sub(truediv(mul(add(add(3, 7), 10), sub(1000, 8)), 992), 17)
  - sub(truediv(mul(10, 10), 992), 17)
  - sub(truediv(19840, 992), 17)
  - sub(20, 17)
  - 3
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 \( x + x^2 + x^3 + \ldots + x^N \) for any \( N \).

- A few general rules:
  - Whatever the assignment, start now.
  - Practice is important. Don't just assume you can do it.
  - Practice is important. Don't just assume you can do it.
  - Practice is important. Don't just assume you can do it.
  - Practice is important. Don't just assume you can do it.
  - Practice is important. Don't just assume you can do it.

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

- 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
```