Welcome to CS61A!

- 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.

Programming and Computer Science

- Programming is the main tool of computer science:

  The study of computation and its applications.

- It is one of the *Sciences of the Artificial* (H. Simon).
- There are many applications and subareas, including:
  - Systems
  - Artificial Intelligence

    Games, robotics, natural language processing...
  - Graphics
  - Security
  - Networking
  - Programming Languages
  - Theory
  - Scientific Computation

This week

- Please see the course web site, especially the course information link. (Please bear with us: the web site is under construction).
- This week, there was no lab. Discussion section will meet as usual.
- You'll get account forms next week in lab.
- Discussion sections are full: please try to find a non-full section, even if it conflicts.
- Attend any lab or section where there is some room. Those enrolled in a lab get priority, but you can get around this by bringing a laptop.

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.

Alternatives to this Course

- If you have very little exposure to programming...
- Or, after the first few sessions, feel that you really aren't ready,
- You can consider other courses to get into the subject more gradually:
  - CS 61AS: "Self-paced" CS61A. Uses Scheme rather than Python.
  - CS 10: The Beauty and Joy of Computing (course for non-majors).
- If you decide to do so, please be sure to officially drop the course, so that we can clear the waiting list.

Getting Help

- We don't expect you to *go* it alone!
- The staff is here to help. Feel free to make free use of lab assistants, TAs, and me by email or in person during office hours.
- And don't forget our message/discussion board: Piazza. This is where students help each other (there are lots of you, and someone probably knows the answer to your question).
- We *very strongly* suggest that you form or join a study group.

Mandatory Warning

- We allow unlimited collaboration on labs.
- On homework, again feel free to collaborate, but you'll get the most out of it if you try to work out answers for yourself first.
- On projects, feel free to talk to others (e.g., via Piazza), but we expect that you and your partner (if any) submit original work.
- Don't share your code with others (other partnerships).
- You can take small snippets 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.
- We can search the web for solutions, too. We have computers and we know how to use them.
- Most out-and-out copying is due to desparation 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: glue operations together;
- 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:

Type	Values	Literals (Denotations)
Integers	$0\quad -1\quad 16\quad 13$	`0 -1 0o20 0b1101`
	36893488147419103232	`0x20000000000000000`
Boolean (truth) values	true, false	`True False`
"Null"		`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)

- ... but not conveniently. So now we add more complex types:

Type	Values	Literals (Denotations)
Strings	pear,	`"pear"`
	I ♡ NY	`"I \u2661 NY"`
	Say "Hello"	`"Say \"Hello\""`
Tuples		`(), (1, "Hello", (3, 5))`
Ranges	0–10, 1–5	`range(10), range(1, 5)`
Lists		`[], [1, "Hello", (3, 5)]`
		`[ x**3 for x in range(5) ]`
Dictionaries		`{ "Paul" : 60, "Ann" : 59,`
		`"John" : 56 }`
Sets	$\{\}, \{1, 2\}$,	`set([]), { 1, 2 },`
	$\{x \mid 0 \leq x < 20$	`{ x for x in range(20) if prime(x) }`
	$\wedge\ x$ is prime	
and many others		

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 callee (left of the parentheses), a function.
  - Evaluate the arguments (within the parentheses).
  - The callee then tells what to do and what value to produce from the operands' values,
- For the convenience of the reader, though, Python employs a great deal of "syntactic sugar" to produce familiar notation:

```
(3 + 7 + 10) * (1000 - 8) / 992 - 17
```

Combining and Defining

- Certain primitives are needed to allow conditional execution:

```
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:

```
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:

```
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):

```
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.
- BCDBC (Be Curious; Don't Be Clueless)
- RTFM
- Have fun!