6.01: Introduction to EECS I

Welcome to 6.01

Staff

<table>
<thead>
<tr>
<th>Instructors</th>
<th>TAs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sarah Finney</td>
<td>Tom DiCicco</td>
</tr>
<tr>
<td>Leslie Kaelbling</td>
<td>Ali Mohammad</td>
</tr>
<tr>
<td>Tomas Lozano-Perez</td>
<td>Christina Wright</td>
</tr>
<tr>
<td>Antonio Torralba</td>
<td>Chuan Zhang</td>
</tr>
</tbody>
</table>

Plus many undergraduate Lab Assistants (LAs)

Course Goals

- Design and analysis of complex systems via abstraction and modularity
- Importance of models for analysis and synthesis
- Dealing with partially specified problems
- Basic skills in EE and CS

Lab: Mobile Robots

Lab: Circuits

Course Coverage

- Software (2 weeks, throughout)
- Linear systems/Control (3 weeks)
- Circuits/Sensing (3 weeks)
- Probability/Localization (2 weeks)
- Search/Planning (2 weeks)
Course Mechanics

- Lecture: Tue 9:30AM (some at different times)
- Software Lab: Tue 11:00 (sec 1) or 2:00 (sec 2)
  - done individually
  - some problems due in lab; some on Thursday
- On-line tutor (register via 6.01 web page; different login from the Python tutor) problems
- Nano-quiz (at the beginning of design lab)
  - easy question from Tuesday lecture or software lab or tutor probs
  - harder question on previous material
  - open book
  - don’t be late!!

More Course Mechanics

- Design lab: Thu 9:30 (sec 1) or 2:00 (sec 2) in 34-501
  - lab work done with partner (randomly assigned)
  - in-lab check-offs
- Weekly written questions (individual)
- Two interviews (individual)
- Two midterms and a final exam
- Advanced programming option (separate 3-unit subject)

Compositional Systems

The most powerful way of building complex systems.

What does it mean for a system to be compositional?

- Set of primitive objects
- Ways of combining primitive objects to get a new object
- New objects can be used and combined in all the ways that primitive objects can

Some compositional systems

Natural Numbers

- Zero is a natnum
- If $x$ is a natnum, then $x + 1$ is a natnum

Arithmetic expressions

- A numeral is an arithmetic expression
- If $x$ and $y$ are arithmetic expressions, then so are
  - $x + y$
  - $x - y$
  - $x \cdot y$
  - $x / y$
  - $-x$
  - $(x)$

6.01 is about Compositional Systems

- In computer programs
- In control systems
- In circuits
- In estimation and decision making
Compositional Systems in Software

<table>
<thead>
<tr>
<th>Procedures</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>+, *, ==</td>
<td>numbers, strings</td>
</tr>
<tr>
<td>if, while, f(g(x))</td>
<td>lists, dicts, objects</td>
</tr>
<tr>
<td>def</td>
<td>ADTS, classes</td>
</tr>
<tr>
<td>higher-order procedures</td>
<td>generic functions</td>
</tr>
<tr>
<td>inheritance</td>
<td></td>
</tr>
</tbody>
</table>

Python Interpreter (Shell)

- Prompts the user for an expression (>),
- Reads what the user types in,
- Evaluates the expression, and
- Prints out the resulting value

Like Scheme, Perl, Ruby... interpreted
Unlike C, C++, Java... compiled

Python Expressions

Like a calculator, apply operators in order of precedence until a single value remains.

```python
>>> 2 + 3
5
>>> (3 * 8) - 2
22
>>> 2.0
2.0
>>> 0.1
0.10000000000000001 # Note, not exact
>>> 1.0 / 3.0
0.33333333333333331 # Note, not exact
>>> 1 / 3
0 # will change in Python 3.0
```

Variables

A binding environment specifies a mapping between variable names and values.

```
>>> b
3
>>> a
NameError: name 'a' is not defined
```

Assignments

```python
>>> a = 3
3
>>> a = 2*a + a/3
>>> b = a
>>> b = b + 1
>>> b
??
```

Lists

```python
>>> a = [2, 4, 9]
```

```
>>> a[0]
2
>>> a[2]
9
>>> a[-1]
9
```
List Slices

```python
>>> a = [2, 4, 9]
>>> a[1:2]
[4]
>>> a[1:]
[4, 9]
>>> a[:2]
[2, 4]
>>> a[:]
[2, 4, 9]
```

List Operations

The `+` operator produces a merged list

```python
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> c = a + b
>>> c
[1, 2, 3, 4, 5, 6]
>>> a
[1, 2, 3]
```

List Mutation

```python
a[1] = -3
```

List Operations

```python
>>> a = [1, 2, 3]
>>> a + [4]
[1, 2, 3, 4]
>>> a
[1, 2, 3] # did not modify a
>>> a = [1, 2, 3]
>>> a.append(4)
>>> a
[1, 2, 3, 4] # modified a
```

Structure Sharing

```python
>>> b = a
```

Now, `a` and `b` are both names for the same list structure!

```python
>>> b[2] = 1
```

Copying a List

```python
>>> a = [1, 2, 3]
>>> b = list(a)
>>> b[2] = 1
>>> b
[1, 2, 1]
>>> a = [1, 2, 3]
```
Check Yourself

>>> a = [1, 2, 3]
Which of the following Python statements, when evaluated, would give rise to this memory structure:

1. b = [[1,2,3], [1,2,3]]
2. b = [a, a]
3. b = a + a
4. b = [a]+[a]

Iterating Over Lists

numbers = [1, 3, 4]
result = 0
for x in numbers:
    result = result + x * x

Compare to:

numbers = [1, 3, 4]
result = 0
for i in range(len(numbers)):
    result = result + numbers[i] * numbers[i]

Procedures

def square(x):
    return x * x

>>> square(6)
36
>>> square(2 - square(2))
4

Calling a Procedure

<expr0>(<expr1>, ..., <exprn>)

1. Evaluate <expr0> in calling environment
2. Evaluate (<expr1>, ... has as parent env. that in which the procedure was defined
4. The procedure body is evaluated in the new environment

Calling a Procedure

>>> square(a + 3)
evaluated in this environment (E1):

E1

Procedure1
(x)
return x\times x

The dotted line indicates parent environment

4. evaluate x * x in E2, return 25
Non-local reference

```python
def biz(a):
    return a + b
```

```console
>>> b = 6
>>> biz(2)
```

Classes and Instances

- **Instance**: collection of data and procedures
- **Class**: what is common among a collection of instances

### 6.01 Domain

<table>
<thead>
<tr>
<th>name</th>
<th>role</th>
<th>age</th>
<th>building</th>
<th>room</th>
<th>course</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pat</td>
<td>Prof</td>
<td>60</td>
<td>34</td>
<td>501</td>
<td>6.01</td>
</tr>
<tr>
<td>Kelly</td>
<td>TA</td>
<td>31</td>
<td>34</td>
<td>501</td>
<td>6.01</td>
</tr>
<tr>
<td>Lynn</td>
<td>TA</td>
<td>29</td>
<td>34</td>
<td>501</td>
<td>6.01</td>
</tr>
<tr>
<td>Dana</td>
<td>LA</td>
<td>19</td>
<td>34</td>
<td>501</td>
<td>6.01</td>
</tr>
<tr>
<td>Chris</td>
<td>LA</td>
<td>20</td>
<td>34</td>
<td>501</td>
<td>6.01</td>
</tr>
</tbody>
</table>

### Instances

```console
>>> pat = Staff601()
```

```console
>>> pat.name = 'Pat'
>>> pat.age = 60
>>> pat.role = 'Professor'
>>> pat.building = 32
>>> pat.office = 'G492'
```

Accessing and setting attributes of the class

```console
>>> Staff601.room
501
>>> Staff601.coolness = 11  # out of 10, of course...
```
Methods
class Staff601:
    course = '6.01'
    building = 34
    room = 501
    def salutation(self):
        return self.role + ' ' + self.name

Calling Methods
>>> Staff601.saluation(pat)
• Evaluate pat to get the instance E3.
• Make a new environment, E4, binding self to E3. The parent of E4 is E1, because we are evaluating this procedure call in E1.
• Evaluate self.role + ' ' + self.name in E4.
• In E4, we look up self and get E3, look up role in E3 and get 'Professor', etc.
• Ultimately, we return 'Professor Pat'.

Initialization
class Staff601:
    def __init__(self, name, role, salary):
        self.name = name
        self.role = role
        self.salary = salary
    def salutation(self):
        return self.role + ' ' + self.name
    def giveRaise(self, percentage):
        self.salary = self.salary + self.salary * percentage

To create an instance
>>> pat = Staff601('Pat', 'Professor', 100000)

String Methods
class Staff601:
    def __str__(self):
        return self.salutation()

>>> print pat
Professor Pat
Without __str__ method, we would get:
<__main__.Staff601 instance at 0x9e19a80>