Welcome to 6.01

Intellectual themes in 6.01 are recurring themes in EECS:
- design of complex systems
- modeling and controlling physical systems
- augmenting physical systems with computation
- building systems that are robust to uncertainty

Intellectual themes are developed in context of a mobile robot.

Goal is to convey a distinct perspective about engineering.

Module 1: Software Engineering
Focus on abstraction and modularity.

Topics: procedures, data structures, objects, state machines

Lab Exercises: implementing robot controllers as state machines

Abstraction and Modularity: Combinators
- Cascade: make new SM by cascading two SM's
- Parallel: make new SM by running two SM's in parallel
- Select: combine two inputs to get one output

Themes: PCAP
- Primitives - Combination - Abstraction - Patterns

Module 2: Signals and Systems
Focus on discrete-time feedback and control.

Topics: difference equations, system functions, controllers.

Lab exercises: robotic steering
Module 3: Circuits

Focus on resistive networks and op amps.

Topics: KVL, KCL, Op-Amps, Thevenin equivalents.

Lab Exercises: build robot “head”:
• motor servo controller (rotating “neck”)
• phototransistor (robot “eyes”)
• integrate to make a light tracking system

Themes: design and analysis of physical systems

Module 4: Probability and Planning

Modeling uncertainty and making robust plans.

Topics: Bayes’ theorem, search strategies

Lab exercises:
• Mapping: drive robot around unknown space and make map.
• Localization: give robot map and ask it to find where it is.
• Planning: plot a route to a goal in a maze

Themes: Robust design in the face of uncertainty

6.01 Content and Pedagogy

6.01 is organized in four modules:
• Software Engineering
• Signals and Systems
• Circuits
• Probability and Planning

Approach: focus on key concepts to pursue in depth

Pedagogy: practice — theory — practice

Course Mechanics

• Lecture: Tue 9:30AM 10-250
• Reading (assigned on calendar web page)
• On-line tutor problems (register via 6.01 web page)
  – practice concepts from lectures and readings
  – prepare for software and design labs
• Software Lab: 1.5 hours in 34-501
  – individual exercises, on-line checking and submission
  – some problems due in lab, some due (a few days) later
• Design lab: 3 hours in 34-501
  – lab work done with partner (new partner each week)
  – some check-offs due in lab, some due (a few days) later
• Written homework problems (4 total)
• Nano-quiz (15 minutes at the beginning of design lab)
  – help keep on pace; open book; don’t be late
• Two interviews (individual)
• Two midterms and a final exam
• Advanced Lab Assistant Option

Module 1: Software Engineering

6.01 makes use of programming both as a tool and as a way to express and explore important ideas.

Today’s agenda
• Python interpreter
• hierarchical programming constructs
• hierarchical data constructs
• object-oriented programming (OOP)

Reading: Course notes, chapters 1–3

Special Note to First-Time Programmers

Exercises in weeks one and two are intended to ensure that everyone reaches a minimum level of familiarity with Python.

If you have little or no Python programming experience
• work through the Python tutor problems; these take priority over other assignments in software and design labs
• attend Python help session Sunday Feb 6 (where you can sign up for a free “new programmer” extension on work due this week).

If at end of week 2, you do not feel prepared to continue 6.01, you can switch registration from 6.01 to 6.00 (offer expires Feb 14).
Python Interpreter
After initializing, Python executes its interpreter loop.

initialization

prompt user for input

read what user types

interpret as expression

print result

repeat

Python Interpreter
Numbers and strings are interpreted as data primitives.

Example (user input in red)

>>> python
>>> 2
2
>>> 5.7
5.7000000000000002
>>> 'Hello'
'Hello'

Python Expressions
Expressions are interpreted as combinations of primitives.

>>> 2+3
5
>>> 5.7+3
8.6999999999999993
>>> 'Hello' + 'World'
'Hello World'

Not all combinations make sense.

>>> 'Hello' + 3
Traceback (most recent call last):
  File "<stdin>" , line 1 , in <module>
TypeError: cannot concatenate 'str' and 'int' objects

Python Expressions
Python expressions can be used in the same ways that primitives are used.

>>> (3 * 8) - 2
22
>>> 24 - 2
22

We refer to such systems as compositional.

Compositional Systems
Compositional systems are familiar in many contexts.

Example 1: arithmetic expressions

\[
\begin{align*}
(3 & \times 8) - 2 \\
\text{integer} & \text{ integer} \\
\text{integer}
\end{align*}
\]

Example 2: natural language

Apples are good as snacks.

nour

Apples and oranges are good as snacks.

nour

nour

Design Principle
We would like to take advantage of composition to reduce the conceptual complexity of systems that we build.

→ make the composition explicit
Capturing Common Patterns

Procedures can be defined to make important patterns explicit.

>>> 2*2
4
>>> 3*3
9
>>> (8+4)*(8+4)
144

Define a new operation that captures this pattern.

>>> def square(x):
...     return x*x
... >>> square(6)
36

Composition allows hierarchical construction of complex operations.
Hierarchical construction reduces conceptual complexity and facilitates design of complicated systems.

Composition of Data Structures

Lists provide a mechanism to compose complicated data structures.

Lists of primitives (integers, floats, booleans, strings):

>>> [1, 2, 3, 4, 5]

Heterogeneous lists:

>>> [1, 'start', 2, 'stop']

List of lists:

>>> [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

Lists are compositional.

Variables

A variable associates a name with an expression (much as def associates a name with a procedure).

Examples:

>>> b = 3
>>> x = 5 * 2.2
>>> y = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> y[0]
[1, 2, 3]
>>> y[-1]
[7, 8, 9]
>>> y[-1][1]
8

Check Yourself

The list 
[a, [b, [c, [d, e]]]]
is best represented by which of the following figures?

1. 2. 3. 4.

Object-Oriented Programming (OOP)

Classes provide a convenient way to aggregate procedures and data in a single structure.

>>> class Student:
...     school = 'MIT'
...     def calculateFinalGrade(self):
...         ...     return theFinalGrade

Classes can include attributes (data) and methods (procedures).
**Instances**

Classes can be used to define instances.

```python
>>> class Student:
...     school = 'MIT'
...     def calculateFinalGrade(self):
...         return theFinalGrade

mary = Student()
mary.section = 3
```

Instances
- `__init__` the methods and attributes of their class
- can also contain new attributes and/or methods

John and Mary share the same school but have a different section.

**Classes, Sub-Classes, and Instances**

Classes can be used to define sub-classes.

```python
>>> class Student601(Student):
...     lectureDay = 'Tuesday'
...     lectureTime = '9:30-11'
...     def calculateTutorScores(self):
...         return theScores

Sub-classes
- inherit the methods and attributes of their class
- can also contain new attributes and/or methods
```

**Environments**

Python associates names with values in binding environments.

```python
>>> b = 3
>>> x = 2.2
>>> foo = -506 * 2
```

<table>
<thead>
<tr>
<th>binding environment</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
</tr>
<tr>
<td>x</td>
</tr>
<tr>
<td>foo</td>
</tr>
</tbody>
</table>

Traceback (most recent call last):
File "<stdin>", line 1, in <module>
NameError: name 'a' is not defined

**Environments**

A similar binding occurs when a procedure is defined.

```python
>>> def square(x):
...     return x*x

>>> a = 2
>>> b = 6
>>> biz(2)
8
```

The procedure name `square` is bound to a procedure that has an argument (x) and a body (`return x*x`).

**Environments**

Using environments to resolve non-local references.

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

>>> b = 6
>>> biz(2)
8
```

Evaluating an expression of the form `square(a+3)`:
- evaluate the name `square` in the calling environment (E1) to determine the procedure to execute (Procedure1)
- evaluate argument (a+3) in calling environment to get value (5)
- create a new environment (E2, whose parent is E1)
- bind parameter (x) to previously evaluated argument value (5)
- evaluate procedure body (`return x*x`) in E2
Environments in OOP
When Python evaluates the definition of a class, it creates an environment.

```python
>>> class Staff601:
...     course = '6.01'
...     building = 34
...     room = 501
...
>>> Staff601
Environments in OOP
Creating an instance of a class creates another new environment.

```python
>>> pat = Staff601()
Environments in OOP
The parent of the new environment is the environment associated with the class.

```python
>>> pat.course
'6.01'
>>> Staff601.course
'6.01'

Environments in OOP
Methods that are added to a class are accessible to all instances.

```python
>>> class Staff601:
...     def salutation(self):
...         return self.role + ' ' + self.name
...     course = '6.01'
...     building = 34
...     room = 501
...     def role(self):
...         return 'Professor'
...     def office(self):
...         return 'G492'
...
>>> pat.name = 'Pat'
>>> pat.age = 60
>>> pat.role = 'Professor'
>>> pat.office = 'G492'
```

Environments in OOP
Attributes are set/accessed using dot notation.

```python
>>> Staff601.name
'Pat'
>>> Staff601.role
'Professor'
>>> Staff601.office
'G492'

Rules of evaluation:
- First variable name is evaluated, points to an environment
- Second variable name is evaluated with respect to that environment, leading to binding of name and value; value is returned, or value is bound

New attributes can be added to `pat` without changing `Staff601`.

```python
>>> pat.name = 'Pat'
>>> pat.age = 60
>>> pat.role = 'Professor'
>>> pat.office = 'G492'
```
Environments in OOP

```python
>>> Staff601.salutation(pat)

Procedure9
def salutation(self):
    return self.role + ' ' + self.name

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)
```

Compositional Systems — Summary

Composition is a powerful way to build complex systems.

**PCAP** framework to manage complexity.

<table>
<thead>
<tr>
<th></th>
<th>procedures</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primitives</td>
<td>+, *, ==, !=</td>
<td>numbers, booleans, strings</td>
</tr>
<tr>
<td>Combination</td>
<td>if, while, f(g(x))</td>
<td>lists, dictionaries, objects</td>
</tr>
<tr>
<td>Abstraction</td>
<td>def</td>
<td>classes</td>
</tr>
<tr>
<td>Patterns</td>
<td>higher-order procedures</td>
<td>super-classes, sub-classes</td>
</tr>
</tbody>
</table>

We will develop compositional representations throughout 6.01.

- software systems
- signals and systems
- circuits
- probability and planning
6.01SC Introduction to Electrical Engineering and Computer Science
Spring 2011

For information about citing these materials or our Terms of Use, visit: http://ocw.mit.edu/terms.