6.01: Introduction to EECS I

Welcome to 6.01

February 1, 2011

6.01: Introduction to EECS I

The **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.

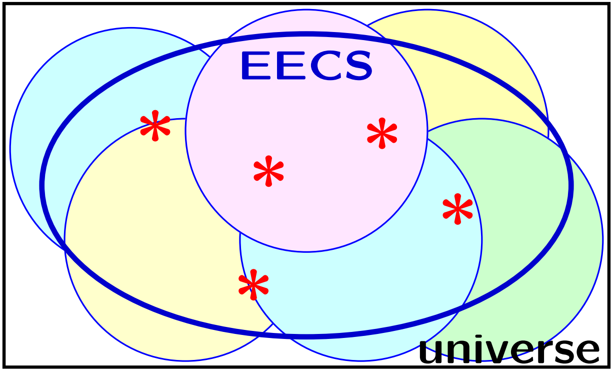
Goal is to convey a distinct perspective about engineering.

6.01 Content and Pedagogy

6.01 is organized in four modules (each rep. broad area of interest):

- Software Engineering
- Signals and Systems
- Circuits
- Probability and Planning

Approach: focus on **key concepts** to pursue **in depth**



6.01 Content and Pedagogy

6.01 is organized in four modules:

- Software Engineering
- Signals and Systems
- Circuits
- Probability and Planning

Pedagogy: practice — theory — practice

Intellectual themes are developed in context of a mobile robot.

Not a course about robots — robots provide versatile platform.

Module 1: Software Engineering

Focus on abstraction and modularity.

Topics: procedures, data structures, objects, state machines

Lab Exercises: implementing robot controllers as state machines

$$\begin{tikzcd}\text{SensorInput} \arrow[r] & \boxed{\text{Brain}} \arrow[r] & \text{Action}\end{tikzcd}$$

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

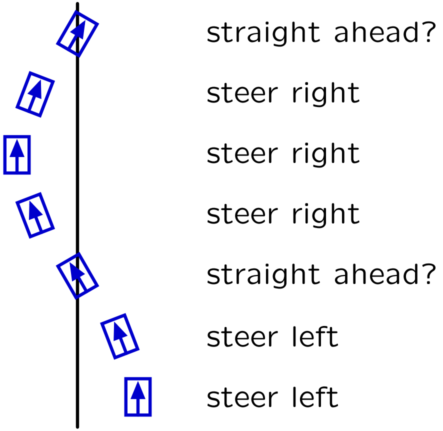
Primitives – **C**ombination – **A**bstraction – **P**atterns

Module 2: Signals and Systems

Focus on discrete-time feedback and control.

Topics: difference equations, system functions, controllers.

Lab exercises: robotic steering



Themes: modeling complex systems, analyzing behaviors

Module 2: Signals and Systems

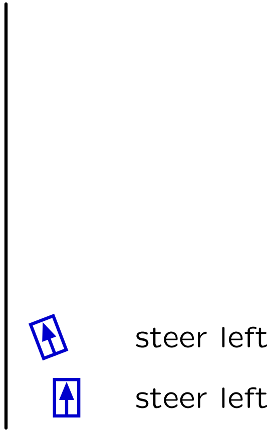
Example: steering a car

Algorithm: steer left when car is right of center and vice versa.

Module 2: Signals and Systems

Example: steering a car

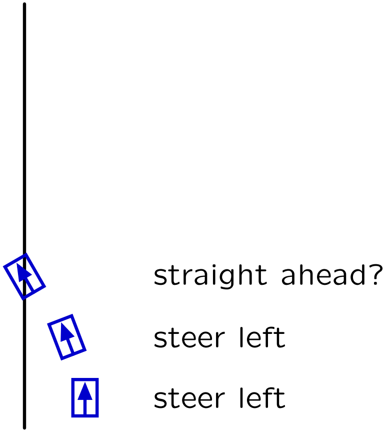
Algorithm: steer left when car is right of center and vice versa.



Module 2: Signals and Systems

Example: steering a car

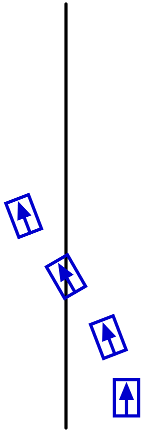
Algorithm: steer left when car is right of center and vice versa.



Module 2: Signals and Systems

Example: steering a car

Algorithm: steer left when car is right of center and vice versa.



steer right

straight ahead?

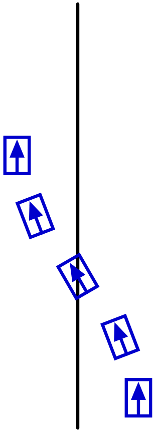
steer left

steer left

Module 2: Signals and Systems

Example: steering a car

Algorithm: steer left when car is right of center and vice versa.



steer right

steer right

straight ahead?

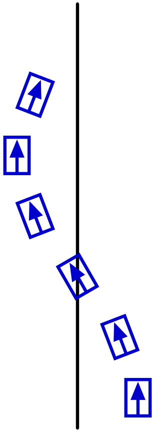
steer left

steer left

Module 2: Signals and Systems

Example: steering a car

Algorithm: steer left when car is right of center and vice versa.



steer right

steer right

steer right

straight ahead?

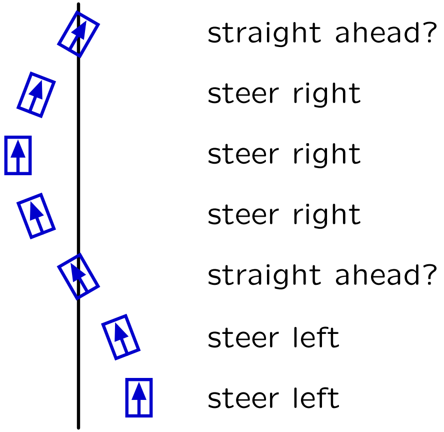
steer left

steer left

Module 2: Signals and Systems

Example: steering a car

Algorithm: steer left when car is right of center and vice versa.



Bad algorithm → poor performance.

Here we get persistent oscillations!

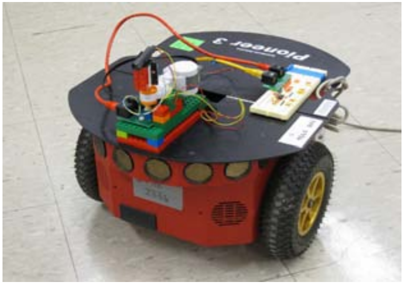
Module 3: Circuits

Focus on resistive networks and op amps.

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

Lab Exercises: build robot “head”:

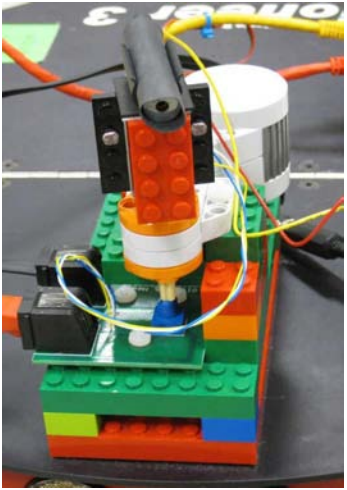
- motor servo controller (rotating “neck”)
- phototransistor (robot “eyes”)
- integrate to make a light tracking system

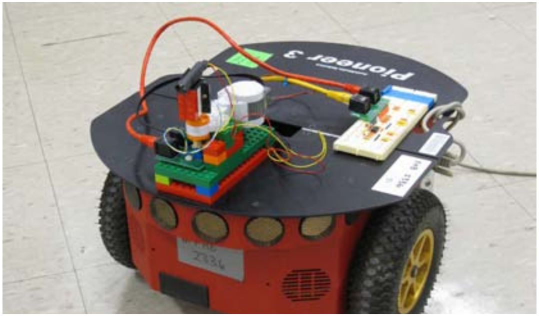


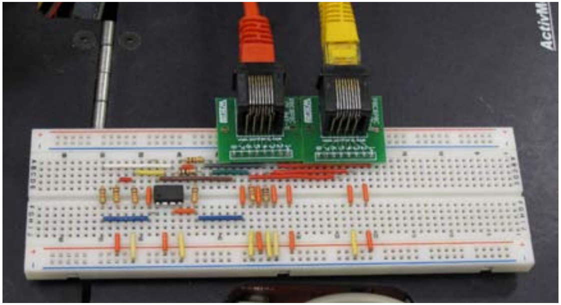
Themes: design and analysis of physical systems

Module 3: Circuits

Lab Exercises: build robot “head”:







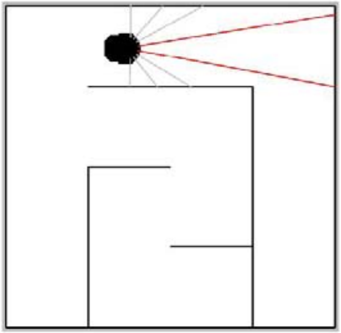
Module 4: Probability and Planning

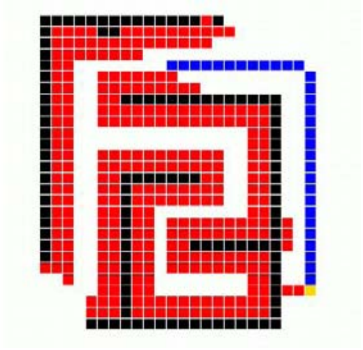
Modeling uncertainty and making robust plans.

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





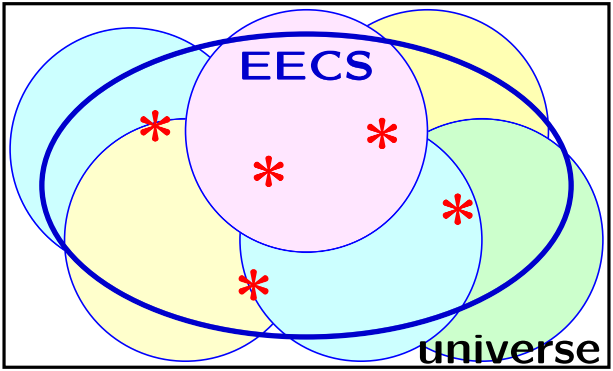
Themes: Robust design in the face of uncertainty

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 week) 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.

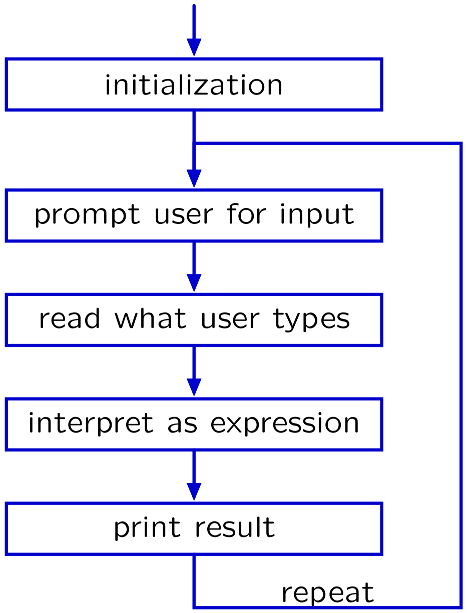
If you have little or no Python programming experience

- **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.



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**

$$\underbrace{(\ \underbrace{3}_{\text{integer}} * \underbrace{8}_{\text{integer}}\ )}_{\text{integer}} - 2$$

Example 2: **natural language**

$$\underbrace{\text{Apples}}_{\text{noun}}\ \text{are good as snacks.}$$

$$\underbrace{\underbrace{\text{Apples}}_{\text{noun}}\ \text{and}\ \underbrace{\text{oranges}}_{\text{noun}}}_{\text{noun}}\ \text{are good as snacks.}$$

Design Principle

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

$\rightarrow$ 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
```

Capturing Common Patterns

Procedures provide a mechanism for defining new operators.

```
>>> square(2)+square(4)
20
>>> square(3)+square(4)
25
```

Define a new operation that captures this pattern.

```
>>> def sumOfSquares(x,y):
...     return square(x)+square(y)
...
>>> sumOfSquares(3,4)
25
```

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
```

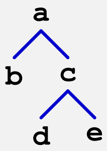
Check Yourself

The list

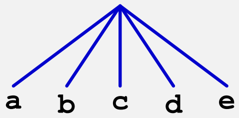
```
[a,[b,[c,[d,e]]]]
```

is best represented by which of the following figures?

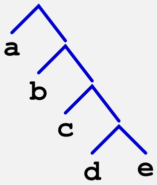
1.



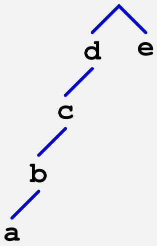
2.



3.



4.



5. none of the above

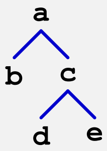
Check Yourself

The list

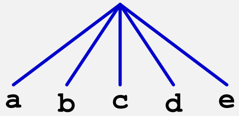
`[a,[b,[c,[d,e]]]]`

is best represented by which of the following figures? 3

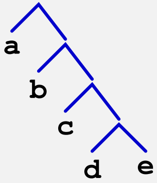
1.



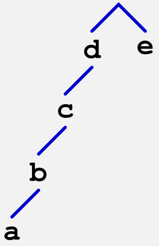
2.



3.



4.



5. none of the above

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**.

```
>>> class Student:
...     school = 'MIT'
...     def calculateFinalGrade(self):
...         ...
...         return theFinalGrade
>>> mary = Student()
>>> mary.section = 3
>>> john = Student()
>>> john.section = 4
```

Instances

– **inherit** 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**.

```
>>> 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**.

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

binding environment

b	3
x	2.2
foo	-1012

```
>>> b
3
>>> a
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
NameError:  name ’a’ is not defined
```

Environments

Assignments change the environment.

```
>>> a = 3
>>> a
3
>>> b = a + 2
>>> b
5
>>> b = b + 1
6
```

In general,

– **evaluate** the right-hand side
– **bind** the left-hand side to that value

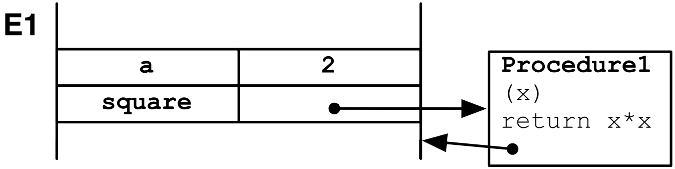
Environments

A similar binding occurs when a procedure is **defined**.

```
>>> a = 2
>>> def square(x):
...     return x*x
...
```

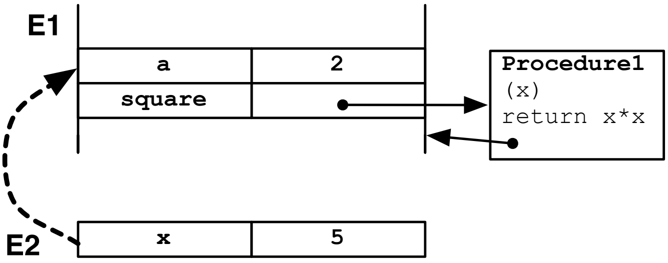
The procedure name (**square**) is bound to a procedure that has an argument (**x**) and a body (**return x*x**).

The procedure (**procedure1**) contains a pointer to the environment (**E1**) in which it was defined.



Environments

Calling a procedure creates a new environment.



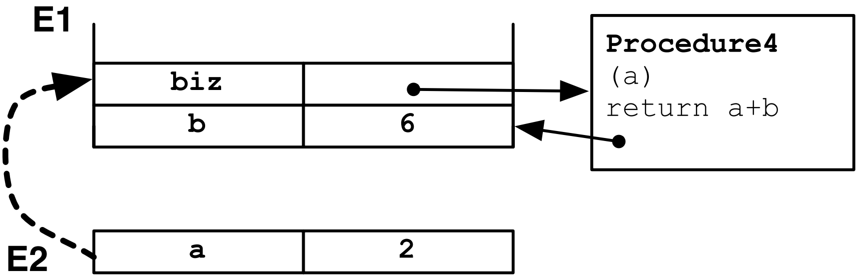
Evaluating an expression of the form **square(a+3)**:

– 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

Using environments to resolve non-local references.

```
>>> def biz(a):
...     return a+b
...
>>> b = 6
>>> biz(2)
8
```



Environments in OOP

Python implements classes and instances as environments.

Example: represent the following subject data:

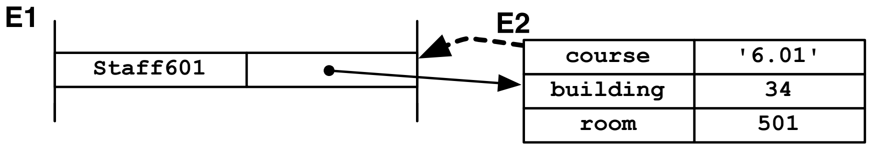
name	role	age	building	room	course
Pat	Prof	60	34	501	6.01
Kelly	TA	31	34	501	6.01
Lynn	TA	29	34	501	6.01
Dana	LA	19	34	501	6.01
Chris	LA	20	34	501	6.01

We can define a **class** to contain the common information.

Environments in OOP

When Python evaluates the definition of a **class**, it creates an **environment**.

```
>>> class Staff601:
...     course = '6.01'
...     building = 34
...     room = 501
...
```



Environments in OOP

Attributes are set/accessed using dot notation.

```
>>> Staff601.room
501
>>> Staff601.coolness = 11
```

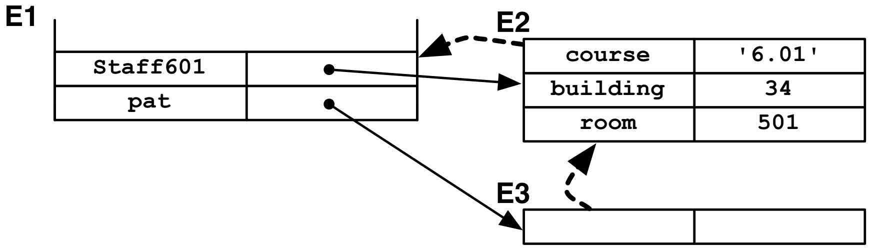
Rules of evaluation:

- 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

Environments in OOP

Creating an **instance** of a class creates another new environment.

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



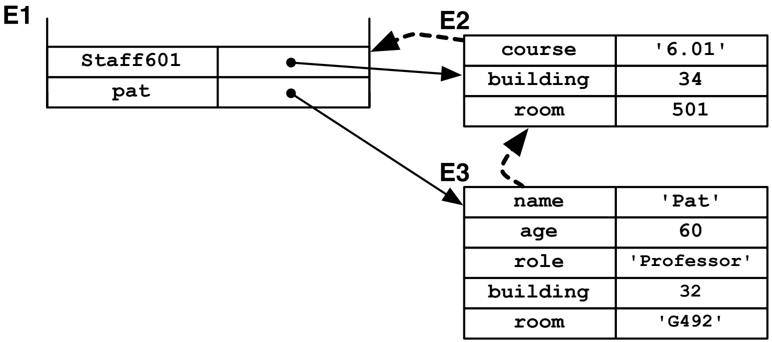
The parent of the new environment is the environment associated with the **class**.

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

Environments in OOP

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

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



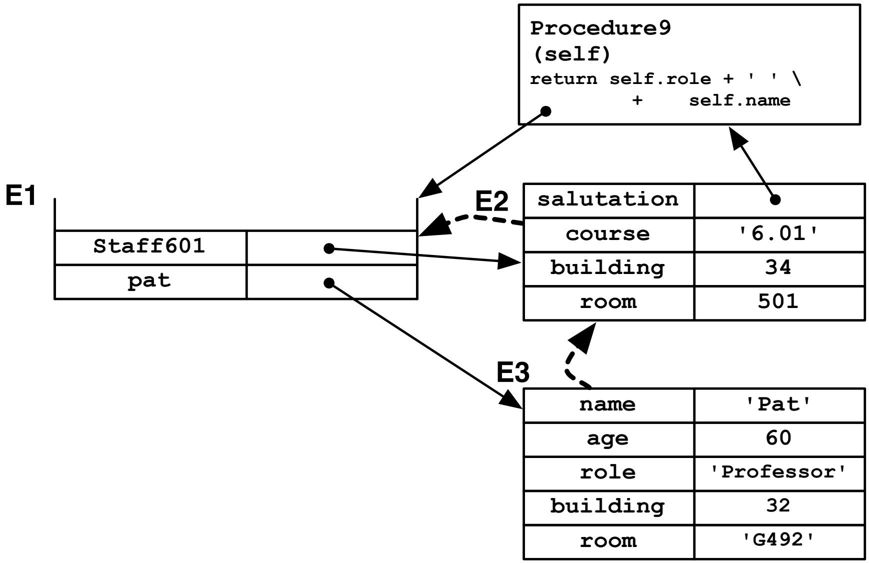
Environments in OOP

Methods that are added to a **class** are accessible to all **instances**.

```
>>> class Staff601:
...     def salutation(self):
...         return self.role + ' ' + self.name
...     course = '6.01'
...     building = 34
...     room = 501
>>> pat.name = 'Pat'
>>> pat.age = 60
>>> pat.role = 'Professor'
>>> pat.building = 32
>>> pat.office = 'G492'
```

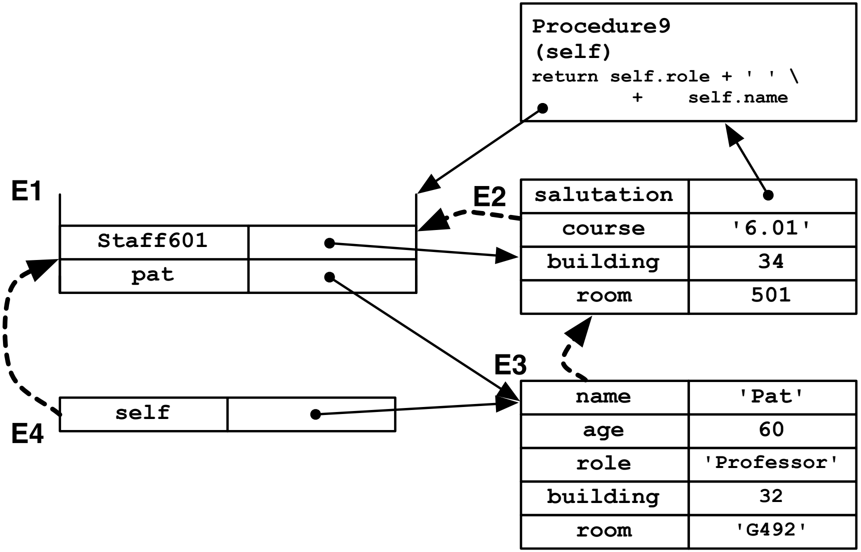
Environments in OOP

Methods that are added to a **class** are accessible to all **instances**.



Environments in OOP

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



```
>>> pat.salutation()
```

Environments in OOP

We can streamline creation of instances by specifying **__init__**.

```
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.

	procedures	data
Primitives	`+, *, ==, !=`	numbers, booleans, strings
Combination	`if, while, f(g(x))`	lists, dictionaries, objects
Abstraction	`def`	classes
Patterns	higher-order procedures	super-classes, sub-classes

We will develop compositional representations throughout 6.01.

- software systems
- signals and systems
- circuits
- probability and planning

This Week in Lab

- Software Lab: **Object-Oriented Programming**
- Design Lab: add **Polynomial** support to Python
- Nano-Quiz 1: first 15 minutes of Design Lab 1
  (you can retake this nano-quiz at the end of the term)
- If you are new to Python and/or programming:
  - **work through Python Tutor first**
    these take priority over other lab assignments
  - attend Python help session on Sunday Feb 6
    (where you can sign up for “new programmer” extension)

Go to your assigned section.
Email to request change.