Liz Steen

6.01: Introduction to EECS I

software engineering – feedback and control – circuits – probability and planning

Announcements **Information & Policies** **Calendar**
Software **Online Tutor** **Reference Material**

Subject Information and Policies

Questions: Please send questions to `6.01-help@mit.edu`. This will ensure a quicker answer than questions to individual staff members.

Description

6.01 explores fundamental ideas in electrical engineering and computer science, in the context of working with mobile robots. Key engineering principles, such as abstraction and modularity, are applied in the design of computer programs, electronic circuits, discrete-time controllers, and noisy and/or uncertain systems.

Prerequisites: None
Corequisite: 8.02
Units: 2-4-6
Lectures: T 9:30-11:00, room 32-123
Lab Section 1: T 11:00-12:30 and R 9:30-12:30, room 34-501
Lab Section 2: T 2-3:30 and R 2-5, room 34-501

Attendance at the lab sessions is mandatory. Contact your lab instructor if you must miss a lab session because of illness or other extraordinary situation (excused by a Dean).

Staff

Name	Role	Office	email at mit.edu
W. Eric L. Grimson	Instructor	38-401	welg
Denny Freeman	Instructor	36-889	freeman
Tomas Lozano-Perez	Instructor	32-G492	tlp at csail
Tim Lu	Instructor	E17-438	timlu
Ali Mohammad	Instructor	34-501	alawi

Russ Tedrake	Instructor	32-380B	russt at csail
Kendra Beckler	T. A.	34-501	kkb
Nicole Bieber	T. A.	34-501	nbieber
Sam Davies	T. A.	34-501	sdavies
Daw-Sen Hwang	T. A.	34-501	dawsen
Evan Iwerks	T. A.	34-501	iwerks

Office Hours

All office hours will be in 34-501.

- Monday: 6:00PM to 9:00PM
- Tuesday: 7:00PM to 9:00PM
- Wednesday: 6:00PM to 11PM

Textbook

The textbook for the course will be a set of notes (around 300 pages) written by the staff. The notes are available as a PDF file on the reference tab of this page. You will also be able to order a bound copy from CopyTech for approximately $23.

6.01 Grade

Your grade in 6.01 will be the weighted average of the following component grades:

Online Tutor Exercises	5%
Online Tutor Software Labs	10%
Online Tutor Design Labs + Interviews	20%
Homework Problems	5%
Nanoquizzes	10%
Midterm 1	10%
Midterm 2	15%
Final Exam	25%

Collaboration Policy

We encourage students to discuss assignments in this subject with other students and with the teaching staff to better understand the concepts. *However, when you submit an assignment under your name, we assume that you are certifying that the details are entirely*

your own work and that you played at least a substantial role in the conception stage.

You will work with a partner in the design labs. You and your partner can equally share all results, code, and graphs that you develop as a team. However, tutor questions about the software labs and design labs are individual. You alone are responsible for any written text that you hand in.

You should not use results from other students (from this year or from previous years) in preparing your solutions to online tutor problems, nanoquizzes, exams, or written answers. You should not take credit for computer code or graphics that were generated by other students unless you developed those materials while working with your assigned lab partner. Students should never *share* their solutions with other students.

Incidents of plagiarism will result in a grade of zero on the assignment and, at the discretion of the staff, be reported to the Committee on Discipline (COD). More information about what constitutes plagiarism can be found at http://web.mit.edu/academicintegrity/

Due Dates, Lateness Penalties, and Extension Policy

Due Dates for all assignments are given on the Online Homework Tutor. Assignments must be completed by the scheduled due dates unless officially excused by a Dean. Participation in sports, music, interviews, projects, etc. are not official excuses for late work (you can use your extensions for these activities). Unless officially excused, your grade for late assignments will be multiplied by 0.5. **Written homework assignments will not be accepted more than one week late, except by special arrangement with an instructor.**

Each student will be allowed exactly **two extensions** that can be applied to all the assignments for any single week. Extensions do not cover nano-quizzes, interviews or exams. Instructions for requesting extensions will be posted here after the beginning of the term.

Once you request to use your extension, it cannot be rescinded. Extended assignments are due one week after the original due date. Extensions cannot be applied to interviews or to nanoquizzes or to exams.

Nanoquizzes

A short (15-minute) on-line nanoquiz will be given prior to each design lab session. The purpose of these nanoquizzes is to provide motivation and feedback for learning the materials presented in the weekly lectures, readings, and on-line tutor problems. Nanoquizzes will generally consist of a simple question from this week's assignments and a more difficult question from previous weeks.

The nanoquizzes can only be accessed on-line during the first 15 minutes of your design lab session in 34-501 (i.e., starting at 5 minutes past the hour). Contact one of your lab instructors if you must miss a nanoquiz because of illness or other extraordinary situation (excused by a Dean).

Nanoquiz Makeups. You can pick any two nanoquizzes to take again (plus any additional nanoquizzes that were officially excused by a Dean). Participation in sports, music, interviews, projects, etc. are not official excuses for missing Nanoquizzes (you can use your makeups for these activities). Nanoquizzes can only be retaken at a makeup session near the end of the term, from 4pm to 9pm (except as supported by a Dean). You will have 15 minutes to complete each makeup nanoquiz. Your grade on the makeup nanoquiz will **replace** your previous nanoquiz grade.

Midterm Exams

Midterm exams will be given in the evening of October 12 and November 10 (see Calendar page). The exams will cover all materials contained in lectures, on-line tutor problems, nanoquizzes, software labs, and design labs up to the date of the exam.

Final Exam

A three-hour final exam will be given during the Final Examination Period at the end of the semester. The final exam will be comprehensive across all materials in this subject, however, materials since the midterms will be weighted more heavily. The final exam will be scheduled by MIT's Registrar's Office. Conflicts with the scheduled time must be resolved by scheduling a conflict examination with MIT's Registrar's Office.

Regrade Policy

If you find a grading error in an examination or homework assignment, please submit your exam/homework along with a cover sheet that describes the error that you found to your TA. We will review your concern and then regrade the entire exam/homework to try to eliminate the error that you identified as well as any other grading errors. Requests for regrades must be made **within one week** of the date when the graded exam/homework was returned.

Advanced Lab Assistant Option

Students with **substantial background** in EE and CS can satisfy the requirements of 6.01 by serving as a lab assistant, as follows:

- complete the tutor exercises and software labs (same as regular 6.01)
- prepare for design lab by attending the Tuesday staff meeting from 4-7pm
- help students as a lab assistant during ONE of the regularly scheduled design labs
- take midterm and final exams (same as regular 6.01)

Grading for the advanced LA option has the following weights:

- Exams: 50%
- Advanced preparation (reading, software labs): 10%
- Staff lab session attendance: 20%
- Participation and engagement as an LA: 30%

Thus, students in this option still register for 6.01; they simply satisfy the course requirements in a different manner. As part of the teaching staff, participants in this option are expected to attend all staff sessions and serve as an LA in prearranged design labs (or to make prior arrangements with the faculty in charge). Failure to do so will result in a substantial grade penalty. If you are interested, speak to an instructor.

Advanced Programming Option

For students with **substantial programming experience**, we will be offering a separately graded 3-unit subject that you can do in addition to 6.01. This subject will involve an additional weekly meeting and 4 programming projects spread across the term, each taking 2-3 weeks. If you are interested, speak to an instructor.

Python + Computational Examples

Today's plan

- Will defer the standard admininstrivia on course mechanics to first real lecture on Tuesday
- First hour will be spent talking about Python and computational thinking
- Remainder of first lab will be spent working through programming examples
  - If you don't have a lot of experience with Python or programming, you can spend that time working through the Python tutor
  - If you are an experienced programmer, you can jump into the assigned problems
- Regular schedule will begin next week

Sun - Python tutorial

Outline

- Compositional systems
- Python interpretation
- Object-oriented programming

Reading: Course notes 1, 2, 3.1—3.5, A.1, A.3—A.4

Theme: Computational systems

OOP

design, analize, maintain complex systems

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

abstraction

Allows one to isolate behavior of module from details of module

- easier to design by suppressing details

don't want to worry about all of the layers

-control complexity

seperate operation + use

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
  - $x + y$
  - $x - y$
  - $x * y$
  - x / y
  - $-x$
  - (x)

are primitives

Note abstraction – in $x + y$, x or y could themselves be complex expressions

6.01 is about Compositional Systems

- In computer programs
- In control systems
- In circuits
- In estimation and decision making

In each case, will learn

- primitives
- ways of combining primitives
- ways of abstracting to create new "primitives"
- patterns by which combinations are typically used

Compositional Systems in Software

	Procedures	Data
Primitives	`+, *, ==`	numbers, strings
Combination	`if, while, f(g(x))`	lists, dicts, objects
Abstraction	`def`	ADTS, classes
Patterns	higher-order procedures	generic functions inheritance

No spagetti code

-abstraction

Why Compositional Systems: Declarative vs. Imperative Knowledge

Declarative knowledge captures statements of fact: "what is true" knowledge.

- The square root of x is that non-negative y such that $y * y = x$.
- This doesn't tell us how to find a square root, though it does tell us how to recognize one if we see it.

Imperative knowledge captures methods for inferring new information: "how to" knowledge

- Start with a guess g
- If $g * g$ is close to x, stop, return g
- Otherwise take a new guess by averaging g and x/g
- Repeat

Compositional systems let us capture these computational patterns: they help us put together primitives to infer new knowledge, in a manner that suppresses details and supports abstraction.

ways to infer square root

primitives – avg
└ close enough

Python Interpreter

Need a language for capturing computational patterns

Syntax: What sequences of symbols, numbers, words make a legal program

Semantics: What a program means

The definition of the **interpreter** is the definition of the semantics of the language

Python Shell:

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

Need to define process of interpretation!

Interpretation

The **interpreter** is the ultimate imperative knowledge:

- It defines the rules for composing simpler expressions (or computations) to create more complex expressions (or computations)
- It defines the set of legal expressions in a language (or compositional system)
- It defines the steps by which a value or meaning is associated with an expression

how does Python take an expression + return a value

Python Expressions: Primitives

Simple data primitives are things like numbers, strings.

```
>>> 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
>>> ''this is a string''
'this is a string'
>>>str(3)
'3'
```

– integer division rounds

Python Expressions: Primitives

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

```
>>> 2 + 3
5
>>> (3 * 8) - 2
22
```

↑ follows order of operations

Understanding the Interpreter

- The interpreter captures the rules of evaluation of expressions
- The interpreter uses chains of environnments to keep track of values associated with names
- A variable has meaning only with respect to an environment
- An expression is always evaluated with respect to an environment
- Understanding how the interpreter creates and uses environments help understand how programs capture patterns of computation

binding → names + values } environment

Interpreting Expressions

Define a procedure **I** that takes a Python program as input and an environment in which to interpret it and:

- Returns a value, and possibly
- Changes something about the internal state of the computer

Note – we can actually write such a procedure! For purposes of 6.01, however, we will simply use the interpreter provided for us; the goal here is to describe how the interpreter evaluates expressions, as that defines our language's behavior.

Input	I (Input, E)
num	*num*
expr1 + expr2	I (expr1, E) + I (expr2, E)

Rules apply recursively, e.g., *expr1* might be a complex expression that requires further evaluation

Variables

- Want to give names to values of expressions, so that we can compactly refer to them
- Assignments are one way to do this (will see others shortly)

Examples:

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

creates a binding

Variables

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

b	3
x	2.2
foo	-1012

name value

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

Assignments change the environment

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

Rule: get value of RHS using interpreter's evaluation rules; bind LHS to that value in environment

Interpreter

- The value of an expression depends on the environment
- Assignments change the environment

Input	Side Effect	I (Input, E)
var		E[*var*]
var = expr	E[*var*] = I (expr, E)	

Procedures

Need way of capturing common patterns: + give it a name

```
2*2
3*3
(8+4)*(8+4)

def square(x):
    return x * x
```

procedure / function

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

Evaluating Procedure Definitions

Input	Side Effect on E
`def` *var* (*args*) *body*	E[*var*] = Procedure(*args*, *body*, E)

Note that evaluating a `def` creates a binding for the procedure name together with the body of the procedure and the environment in which it was created; it does not actually evaluate (or call) the procedure.

glues together into structure

Calling a Procedure

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

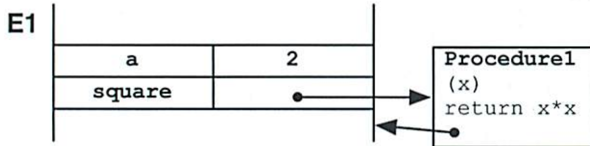
1. Evaluate `<expr0>` in calling environment
2. Evaluate (`<expr1>, ..., <exprn>`)
3. A *new environment* is created:
   - binds the parameters of the procedure to argument values
   - has as parent env. that in which the procedure was defined
4. The procedure body is evaluated in the new environment

Input	I (Input, E)
e0 (*e1*, ..., *eN*)	proc = I (*e0*, E) v1 = I (*e1*, E) ... newE = Env(proc.args, (v1, ..., vN), E) I (proc.body, newE)

Calling a Procedure

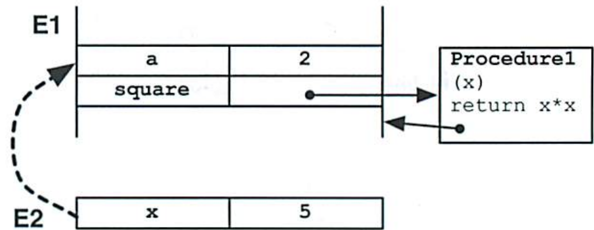
```
>>> square(a + 3)
```

evaluated in this environment ($E1$):



Calling a Procedure

1. evaluate `square` in $E1$ and get `Procedure1`.
2. evaluate `a + 3` in $E1$ and get 5.
3. Create $E2$:
   - binds `x` to 5
   - has $E1$ is its *parent*.



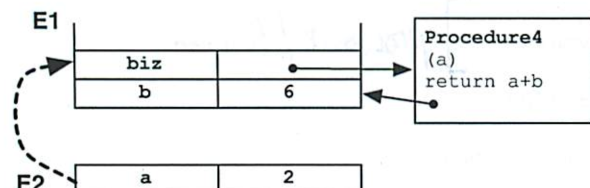
The dotted line indicates *parent environment*

4. evaluate `x * x` in $E2$, return 25

can build long, complex chain

Non-local reference

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

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



goes up parent chain

= 8

Data Structures

Procedures capture common patterns of computation; let's us abstract away details and use as if were a primitive

We also need ways of grouping data elements together into more complex structures that can be treated as primitives.

Python has several important ones, which you should explore:

- Lists
- Tuples
- Dictionaries

We will come back to these next time, as well.

Classes and Instances

One important way to group information together is to aggregate common data elements and common procedures for manipulating those data elements into a single structure

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

Object-oriented programming
- aggregates of data

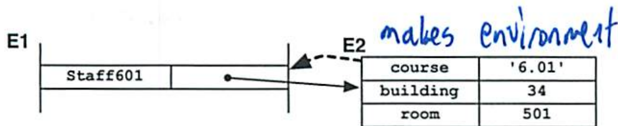
6.01 Domain

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

Class Definition

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

Create commonality



Evaluation of class definition creates an environment!

Accessing and setting *attributes* of the class

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

Class Definition

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

Accessing and setting *attributes* of the class

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

Note 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

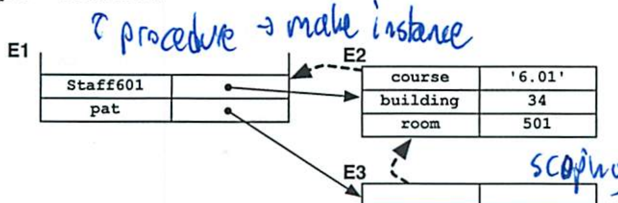
Make

Instances

- specific versions

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

(procedure → make instance



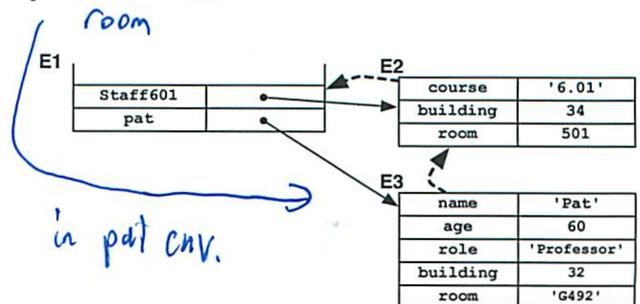
Evaluating class instantiation creates a new environment, scoped by parent environment of class; any init expressions (see later) are evaluated wrt to this environment; environment is returned as value

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

Instances

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

(office crossed out, "room" written above)

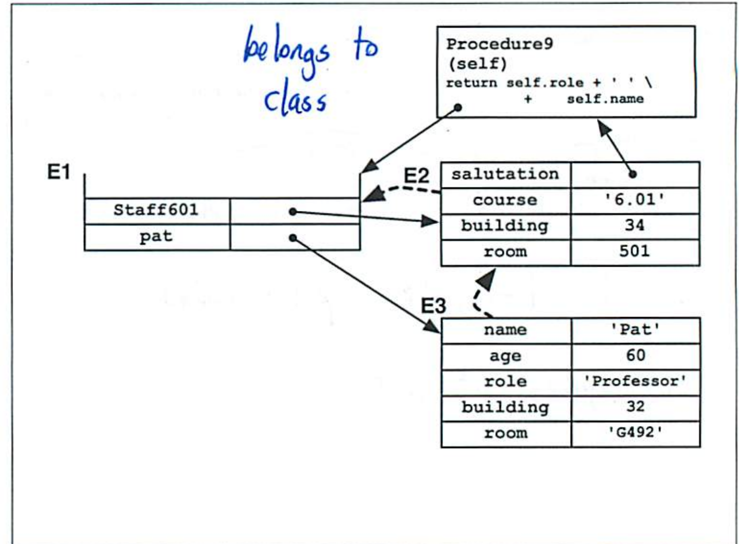


Methods

```
class Staff601:
    course = '6.01'
    building = 34
    room = 501
    def salutation(self):
        return self.role + ' ' + self.name
```

« common method / procedures

↑ points to instance



Calling Methods

```
>>> Staff601.saluation(pat)
```

- Evaluate `pat` to get the instance E_3.
- Make a new environment, E_4, binding `self` to E_3. The parent of E_4 is E_1, because we are evaluating this procedure call in E_1.
- Evaluate `self.role + ' ' + self.name` in E_4.
- In E_4, we look up `self` and get E_3, look up `role` in E_3 and get `'Professor'`, etc.
- Ultimately, we return `'Professor Pat'`.

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

is **exactly** equivalent to

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

(*)

Initialization

We can specify particular computations to perform whenever we create an instance.

```
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
```

built in

To create an instance

```
>>> pat = Staff601('Pat', 'Professor', 100000)
```

String Methods

```
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 __str__(self):
        return self.salutation()

>>> pat = Staff601('Pat', 'Professor', 100000)
>>> print pat
Professor Pat
```

Without `__str__` method, we would get:

```
<__main__.Staff601 instance at 0x9e19a80>
```

Rest of today

- If you are new to Python and/or programming:
  - **Work through the Python Tutor first**
  - Come to office hours on Sunday, to get a 'free' extension on this week's work
- Work through this week's Software Lab assignment
- First nanoquiz at end of session – can retake at end of term if you need to; this gives you sense of what we expect

> If you have a problem with your assigned section, please email welg@mit.edu.

Software Lab 1: Intro to Python and OOP

1 Setup

For this lab, it will be easiest to use one of our laptops or desktop machines. If you have already installed Python on your own laptop, you can use it, instead. If you haven't installed Python yet, and would like help, please bring your laptop to evening or weekend office hours.

- **Using a lab laptop or desktop machine**
  - Log in using your Athena user name and password.
  - Click once on the Terminal icon (usually on the bottom left of the screen.) In the terminal window, type `athrun 6.01 setup`. This step is only done for the first lab; for subsequent labs, do `athrun 6.01 update`. It will create a folder in your Athena account called `Desktop/6.01`.
- **Using your own laptop**
  - Go to the course web page: `http://mit.edu/6.01`
  - Go to the calendar tab, and download the zip file for software lab 1. Unzip it.
  - When we mention finding a file in `Desktop/6.01/...`, look for it in the folder you got by unzipping the archive.
- **Using course notes in lab**
  - Click once on the Firefox icon at the top left of the screen.
  - Go to `http://mit.edu/6.01`.
  - Click on `Reference Material` in the navigation bar.
  - Click on `Course Notes`.
  - In the popup window, click on `Open with`, choose `Document Viewer` from the pull-down list and click `OK`.

- **Using the online Tutor**
  - If you have not already registered for the 6.01 tutor, do so now. (Note that the online homework tutor is different from the online Python tutor, which you may have been using to prepare for 6.01. You need to register separately for the homework tutor.)
    - ⋆ Click once on the Firefox icon at the top left of the screen.
    - ⋆ Go to `http://mit.edu/6.01`.
    - ⋆ Click on `Online Tutor` in the navigation bar.
    - ⋆ Under the Homework Tutor section, click on the `register here` link and follow instructions
- **Writing and running Python programs**
  - In the Terminal window, type `idle &`.
  - You can type Python expressions in Idle's `Python Shell` window.
  - You can write your programs in a file and test them using `Run Module`. For example:
    - ⋆ Click Idle's `File` menu, select `New Window`, and write `print 'Hello World'` in the window.
    - ⋆ Click Idle's `File` menu, select `Save as`, navigate to `Desktop/6.01/lab1/swLab/`, and enter the file name `test.py`.

      When using the lab laptops, if you find yourself in a file dialog box that seems to be far away from your home directory, you can always type ~ (the tilde character) in the box, followed by the Enter key; that should take you to your home directory, which contains your `Desktop` folder.
    - ⋆ Click Idle's `Run` menu, then select `Run Module`.
    - ⋆ Look at the `Python Shell` window: you should see `Hello World`.

2 Exercises

> **If you have already worked through** our Python programming tutor and/or have had other Python experience, then go ahead and do the problems below.
>
> **If not**, then please work through the Python tutor. To register for the Python tutor, go to the course web page, click on the `Online Tutor` link, and register for the **Python Tutor**. You will get instructions on how to log onto that tutor. If you need extra help in Python, come to our help session on Sunday. At that session, you can sign up for a free 'new programmer' extension on the work of this week.

2.1 Simple Looping Procedures

Open the file `Desktop/6.01/lab1/swLab/sl1Work.py` and complete the definition of the `myAdd` procedure. This procedure should take a list as an argument, and should return the sum of the elements of the list. If the argument list is empty, the output should be 0. **Do not use the built-in Python `sum` procedure – we want you to get practice in writing looping procedures.**

Debug it in Idle until it seems correct. confused how to start list data structure → seperate sheet

Wk.1.3.1	Check your results by copying the text of your procedure from Idle and pasting it into the tutor problem Wk.1.3.1.

Similarly, complete the definition of the `myMul` procedure, which will compute the product of the elements of the list supplied as argument. If the argument list is empty, the output should be 1.

Wk.1.3.2	Check your results by copying the text of your procedure from Idle and pasting it into the tutor problem Wk.1.3.2.

2.2 Factorial

Open the file `Desktop/6.01/lab1/swLab/sl1Work.py` and complete the definition of the `fact` procedure, so that `fact(n)` returns the value of $n!$ (i.e., $n * (n-1) * (n-2) * ... * 1$)

Debug it in Idle until it seems correct.

Wk.1.3.3	Check your results by copying the text of your procedure from Idle and pasting it into the tutor problem Wk.1.3.3.

2.3 Reverse

Open the file `Desktop/6.01/lab1/swLab/sl1Work.py` and complete the definition of the `myReverse` procedure. This procedure should take a list as input, and return a new list as output, whose values are in the opposite order to the input.

Debug it in Idle until it seems correct.

Wk.1.3.4	Check your results by copying the text of your procedure from Idle and pasting it into the tutor problem Wk.1.3.4.

2.4 Object-Oriented Practice

run through on web

Wk.1.3.5	Get some practice with object-oriented concepts in this tutor problem.

Wk.1.3.6	Get some more practice with object-oriented concepts in this tutor problem.

2.5 Two-dimensional vectors

Open file `Desktop/6.01/lab1/swLab/sl1Work.py` and complete the definition of the `V2` class; it represents two-dimensional vectors and supports the following operations:

- Create a new vector out of two real numbers: `v = V2(1.1, 2.2)`.
- Convert a vector to a string.
- Add two `V2`s to get a new `V2`.
- Multiply a `V2` by a scalar (real or int) and return a new `V2`.

Step 1. Define the basic parts of your class, with an `__init__` method and a `__str__` method, so that if you do

```
print V2(1.1, 2.2)
```

it prints

```
V2[1.1, 2.2]
```

Exactly what gets printed as a result of this statement depends on how you've defined your `__str__` procedure; this is just an example. Remember that `str(x)` turns `x`, whatever it is, into a string.

Step 2. Write two accessor methods, `getX` and `getY` that return the `x` and `y` components of your vector, respectively. For example,

```
>>> v = V2(1.0, 2.0)
>>> v.getX()
1.0
>>> v.getY()
2.0
```

Step 3. Define the `add` and `mul` methods, so that you get the following behavior:

```
>>> a = V2(1.0, 2.0)
>>> b = V2(2.2, 3.3)
>>> print a.add(b)
V2[3.2, 5.3]
>>> print a.mul(2)
V2[2.0, 4.0]
```

```
>>> print a.add(b).mul(-1)
V2[-3.2, -5.3]
```

Step 4. A cool thing about Python is that you can overload the arithmetic operators. So, for example, if you add the following method to your V2 class

```
def __add__(self, v):
    return self.add(v)
```

— cool!

then you can do

```
>>> print V2(1.1, 2.2) + V2(3.3, 4.4)
V2[4.4,6.6]
```

Add to the class the `__add__` method, which should call your `add` method to add vectors, and the `__mul__` method, which should call your `mul` method to multiply the vector by a scalar. The scalar will always be the second argument.

← do as well or get weird error (don't forget to read)

Test your implementation in Idle until it seems correct to you.

Wk.1.3.7	Check your results by copying the text of your procedure from Idle and pasting it into the tutor problem Wk.1.3.6.

concatinates w/ [+] like Java script

V2 - make instance
cast as floats

$$\frac{1.1 \cdot 1 \cdot 48{,}000}{12}$$

int ≠ float

don't make sloppy mistakes when copy + pasting

6.01 Intro to Python and OOP — Fall 2010 6

6.01 Design Lab 1

9/9

lists

- flexible
- contain anything
- sequence
- mutable like strings

L=[ ] empty

L=[1, 2, 3, 4]

L = ['abc', ['def', 'ghi']]

L[i] index

len(L) = empty

L.append

~~[illegible]~~

if

7 > 8 → false

[1, 2] = [1, 2] → true

[1, 2] is [1, 2] → false

(2)

```
if <boolExpr>:
    <statement>
else:
    <statement>
```

(indent) (indent)

For

```
for x in a:
    print x, len(x)
```

(how I actually did it)
- weird thinking about companant pieces
- and how to do basic things

but failed big time on tutor

opps - must include 1st line

can see what they did
- they did not check that 0

Factorial → how will I do?

total = total · (n - i) total = 1

5! = 5 · 4 · 3 · 2 · 1

5 · (5-0)

1 · 5-0 → 5 · (5-1) → 20 · (5-2)

③

fact (2)

- so I was close
- put in some logging code to see what going on

~~[illegible]~~ 5 -

- so second part was right
- calc error when I was doing it manually

```
ans = 1
for i in range (2, n+1):
    ans = ans * i
    return ans
```

← somewhat better

recursive

```
if n <= 1
    return 1
else
    return n * fact (n-1)
```

↑

~~that should~~ don't forget about this

(4)

My Reverse

reverse the list

Start at end

L[::]

len = lenght

for x ~in_

m= [[len -!]

- - -

Their ans of carse much shorter

for x in L

ans = [x] + ans

add that piece

i did not know that worked on lists

6.01 Lists

9/12

- only key info for me notes

are <u>pointers</u>

b=a

<u>same</u> list
<u>not</u> a copy

if want a copy
c=list(a)
c=a[:]

but only 1 level!

- otherwise copy, deep copy

---

tuple = like a list, not mutable (changable)

a=(1,2,3)
a=1,2,3
b=1,
need comma w/ tuple 1 element

String = tuples of characters

Procedures

- in a different environment

②

but can't write

```
a=3
def b():
    a=a+1    ← a has already been evaluated
    print a
```

does support "global a"

- so it changes the global a
- don't make a new binding

procedures like #s

Can pas procedures as augments

Square(square(x))

Think C-Spy Reading

9/12

- what I found interesting

all about problem solving

~~*~~* = exponential

~~does~~ - integer division (rounds down)

can multiply stringes to repeat them

"Fun" * 3" = "FunFunFun"

Comments

~~///~~ ⤵

#

int + float different

convert w/ float(15) = 15.0

if 1 # in * ÷ is float → ans will be

% = modulus operator, returns remainder

like php can do "if isDivisible(6,3)" don't need == True
(isDivisible(6,3) → true)

5.8 → if use int() it would round up

- but others would not know what it was doing
- better to error

- why you write tests too, I believe

I see why people like python → beutiful lang

Comma at the end of ~~[struck]~~ print supresses new line

for ~~is~~ is like php foreach

Strings are like arrays w/ each letter an entry
can do slices
print s[0:5] → prints first 6 characters
-nicer than php substr
* Starts counting at 0 *
Strings are immutible
- must create new string
never thought about how to do find
lists can be nested ← like php arrays
- was in course notes
range(10) → [0,1,2,3,4,5,6,7,8,9]
"in" checks if value is in the list
del deletes item in a list
nest lists for matricles
tuples are immutable
new type: dictionaries = {}
- associative arrays
10.2 - I can see where OOP feels natural
- functions came along w/ that object
Sparse matrix example cool
Same open + close functions for file handling as php

③

but contents must be a string, so str( )
Or use pickle plug in (like seralize?)

Class = user defined compound type
define it and then create instances
(I like this explination more than (Car, Dog, etc)
Sameness interseting point
Shallow = same ~~the the~~ reference] opposet what I would think
deep = Same value

~~One~~ note aliasing happens
- So import copy module + use it
  - shallow copies 1st level
  - if it has other embedded objects use deep copy

remember my v2 work about making new objects
- like when you have a modifier + when you don't
- Some programs only use modifiers!

algorith = <u>general solution</u>

⊗ methods are different than functions
- defined inside a class
- different syntax

time. print( )

(4)

This was my V2 problem I think
- Can define outside class as a function
- go back + check
- wanted to add self

V2.add(V2b)

Could have done ~~X~~ add vectors(V2, V2b)

returned a vector
return V2 (__, __)

Polymorthism - methods that work on more than 1 type
- (not what I thought it meant)

Can have mutiple objects
(I like the card example
- seems natural)

and "magic methods" (? right name) __add__ is sneaky but cool

Class Deck inits all the cards

inheritance - new class that is modified version of old class
- sometime useful, but not always

class Hand (Deck):

6.01 HW

(scratchpad, not well formatted)

9/12

> a, b = (2,3) ← so this is faster way for assign

> a + b
5

> a, b = [2,3]

> a+b
5 ← oh so it was not a list

> [a, b] = (2, 3)

> a+b
5 ← wtf?? where read about this? (ask)

> (a, b) = [a, b] — must not be a list

same again!

(a, b) = (2, 3, 4)

errors - too many values to unpack

(a, b) = 2

error int object **not** iterable

(a, (b, c)) = (2, (3, 4))

works

(a, (b, c)) = (2, 3, 4) does not! - it does 1 at a time or something

② OOP pratice now

- may need some help

Modifier

~~Se~~ a. x = Thing()
↑b

b.x = 7

a.x.x
Thing b
b.x
7

they are easy
- just need to work though + think
- really good exercises
- focus

~~But~~ c = a.get = Thing()
↑b

c.x = b.x = 7

~~a.set~~ parenthesse -> eval 1st!

a.set (1-7)
a.x = -6

c.x = 3
a.get = -6 so error

(now I see what ref means
does make sense - at least w/ my prior experience)

(3)

a = Thing

b = Thing

a.x = Thing()
↑b

b.x = Thing()
↑a

: infinite loop?

True

---

a.x =
└ thing b
  b.x =
  └ thing a
    a.x =
    └ thing b

~~Part 2: Thing 2~~

⟶

(4)

Part 2: Thing 2

oh builds appon

Thing Mangle

- increments x by 1
- sets hasBeenMangled true

a = Thing ← starting here fresh? - we should

a.x = 5

6

b = Thing

b.x = Thing

b.x.x = 3

mangle b.x

b.x.x = 4

b.x.hasBeen = true

---

c = Thing

mangle that

- ?? error since c.x not defined? ✓ bingo

(5) <u>Part 3</u> <u>Thing mangle</u>

Ans: Thing.mangle
↑ new method
(old was a function, right?)

replace arg w/ self

<u>Part 4 More Mangling</u>

mangled (z)
a = new Thing
a.set (z) ← don't access a directly
a.mangle ( )
return a ↑ must have

it gives cryptic error messages

<u>1.4.3 Assign</u>

↓ no self - not in a class

assign things (Thing1, Thing2) "pure function" I think
Thing1.set (Thing2.~~set~~ get)

<u>Part 3</u>

but can't do new Thing (5)
↑ not defined

(6)

Sum of all things

takes list of things

for n in thinglist

sum = ~~thing.~~ n.get()+sum

~~return sum~~

must return a Thing, not an int

Still no → says did not fulfill problem

- perhaps use built in sum?

sum = Sum(~~x~~.get())
^ thingList

since removing for loop

Still must use "a list comprehension"

- documentation is hard to understand

Something with [___ for ___]

[sum = sum + n.get ~~()~~ for n in thingList]

Or change sum variable name

now it returns [1,5] not the sum

- but at least it passes the format check

but how to do it?

- or do it my way + leave that code in - trick system

~~and also~~ and return empty list as Thing - not int

(7)

Answer

```
def sumofAllThings(listOfThings):
    a = Thing()
    a.set(sum([t.get() for t in listofThings]))
    return a
```

here is were we should have used sum!

1.4.4 Part 1 Thing clone

- easy make new Thing
- copy value

Part 2 str

* remember must

2 convert

```
"bla" + str(self.get())
```

Done

Staff

Lecturer: Eric Grimson

Instructors	TAs
Dennis Freeman	Kendra Beckler
Tomas Lozano-Perez	Nicole Bieber
Tim Lu	Sam Davies
Ali Mohammad	Daw-Sen Hwang
Russ Tedrake	Evan Iwerks

Plus many excellent undergraduate Lab Assistants (LAs)

Course Goals and Course Coverage

- 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

Capture patterns, supress details

- Software (2.5 weeks, throughout) ← learn in 2 weeks!
- Linear systems/Control (3 weeks)
- Circuits/Sensing (3 weeks)
- Probability/Localization (2 weeks)
- Search/Planning (3 weeks)

Come up w/ good models

Course Mechanics

- **Lecture**: Tue 9:30AM 32-123
- **Software Lab**: Tue, 11:00 or 2:00 in 34-501
  - done individually
  - some problems due in lab, some two days later
- **On-line tutor** (register via 6.01 web page; different login from the Python tutor) problems
- **Written homework** problems 3, first Thur posted
- **Reading** (assigned on calendar web page)
- **Nano-quiz** (at the beginning of design lab) Thur
  - 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 or 2:00 in 34-501
  - lab work done with partner (randomly assigned)
  - some check-offs due in lab, some a week later
- Two **interviews** (individual)
- Two **midterms** and a **final exam**
- Advanced programming **option** (separate 3-unit subject)
  - If you are interested, see Prof. Grimson

Outline

- Data structures and procedures
- Functional programming style
- Inheritance
- Procedures as first-class objects

Reading: 3.4.6, 3.5.4, 3.6, 4.1, A.1, A.2

I did get up to speed w/ Python quickly

Collecting data: data structures

- Would like a way to gather data together into structures that can be manipulated as a single entity
- Have seen classes as one mechanism
- Simple linear mechanisms: list and tuples
  - Lists are mutable; tuples are not
  - Linear collection of elements
  - Kind of PCAP – elements can themselves be complex data structures (abstraction)

Lists: Creation

```
>>> foo = [1,2,3,4]
>>> foo
[1, 2, 3, 4]
>>> bar = list('abcd')
>>> bar
['a', 'b', 'c', 'd']
```

Lists: Accessing and copying

```
>>> bar = list('abcd')
>>> bar[0]
'a'
>>> bar[3]
'd'
>>> bar[-1]
'd'
```

last (should really be -0, but oh well)

```
>>> bar[1:]
['b', 'c', 'd']
>>> bar[:1]
['a']
>>> bar[:]
['a', 'b', 'c', 'd']
```

Lists: Mutation

```
>>> bar = list('abcd')
>>> foo = bar
```

points to same list

```
>>> foo
['a', 'b', 'c', 'd']
>>> bar[0] = 'z'
>>> foo
['z', 'b', 'c', 'd']
>>> bar
['z', 'b', 'c', 'd']

>>> oof = tuple('abcd')
>>> oof[0] = 'z'
Traceback (most recent call last):
  File ''<pyshell#35>'', line 1, in <module>
    oof[0] = 'z'
TypeError: 'tuple' object does not support item assignment
```

this is what confused me yesterday

Manipulating Lists: List Comprehension

- Could write procedures that loop over indices into a list, computing functions of each element
- Ideally would like to think about doing things to elements of a list without worrying about looping structure
- List comprehension provides such an abstraction
- Note how this abstraction can change your mode of thinking – focus on manipulating a list as if it were a single entity; rather than getting bogged down in the details of the list itself!

```
> foo = [1,2,3,4,5]
> def doubleIt(lst):
      return [x*2 for x in lst]
> doubleIt(foo)
[2, 4, 6, 8, 10]
> doubleIt(doubleIt(foo))
[4, 8, 12, 16, 20]
```

abstraction, suppresses details

Dictionaries: Read about them

"associative array"

```
>>> d = {'a':7, 'b':8}
>>> d['a']
7
>>> d['c']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'c'
>>> d[100] = 'a hundred'
>>> d
{'a': 7, 'b': 8, 100: 'a hundred'}

>>> d.has_key('a')
True
>>> d.has_key('d')
False
>>> for k in d:
        print k, d[k]
a 7
b 8
100 a hundred
```

What are standard ways of structuring software systems?

- Object oriented programming
  - Focal point is an instance – a collection of related data elements and procedures for manipulating them
  - Organize in hierarchies that inherit or share methods and data – simulations
- Procedural (or imperative) programming
  - Focal point is sequences of instructions and changes in state – data structures are mutated to reflect state of computation (older)
  - Organize around looping constructs and state changes
- Functional systems
  - Focal point is idea of mathematical function – procedures that convert input into output without mutation or side effect
  - Organize in collections of functions that support modular connections

OO

Functional Programming Style: An Example

Return the name of the team with the most efficient offense

```
bigLeague = [['patriots', [1000, 1100, 900]],
             ['colts', [350, 315, 400]],
             ['raiders', [100, 150, 225, 117]]]
```

most avg. yards

↑ # yards of offence

Standard procedural style

```
def bestTeam(yardsData):
    winner = None
    bestYards = -1
    for i in range(len(yardsData)):
        total = 0
        numYards = len(yardsData[i][1])
        for j in range(numYards):
            total = total + yardsData[i][1][j]
        avg = total / float(numYards)
        if avg > bestYards:
            bestYards = avg
            winner = yardsData[i][0]
    return winner
```

procedural (what I have done) esp w/ Tecker

good on small size problems

Functional Style

Not OOP

- Procedures are 'first class': treated the same as any other kind of data
- Procedures generally designed around concept of transforming input to output (like a mathematical function) without side-effects or mutation

  Only pure functions
- Very small procedures with specific tasks, and useful names – modular design allows for reuse of procedures

Split into functions

Did this later (6.005)

Most of the program

```
# Return the name of the team with highest average yards
def bestTeam(yardsData):
    (name, yards) = argMax(avgYards, yardsData)
    return name

# Given record for a single team, return its average yards
def avgYards(teamData):
    (name, yards) = teamData
    return listMean(yards)

# Return the mean (average) of a list of numbers
def listMean(data):
    return sum(data) / float(len(data))

>bestTeam(bigLeague)
'patriots'
```

unpack

no indicies anywhere!

Argmax!

Given a procedure that takes an item and returns a number, and a list of items, return the item for which the procedure returns the highest number

```
def argMax(f, items):
    bestItem = None
    bestScore = None
    for item in items:
        newScore = f(item)
        if bestScore == None or newScore > bestScore:
            bestScore = newScore
            bestItem = item
    return bestItem
```

is 1 for loop

and mutation

Recursion

- An alternative way to do simple iteration
- A natural generalization of functional programming
- The only convenient way to do some operations on nested lists
- Powerful way to think about PCAP

Fundamental idea:
Define a procedure f:

- *recursively* in terms of f applied to *simpler* arguments
- with a non-recursive *base case* for the *simplest* arguments

OR

- Given a problem, assume you can solve a simpler version of it
- Decide how to use solution to simpler problem, plus simple operations, to construct larger solution
- Decide for what size problem you can solve directly

always need a base case – an escape clause

Recursion on natural numbers

base case: 0 or 1

recursive case: $f(n)$ defined in terms of $f(n-1)$

Exponentiation:

$$b^n = \begin{cases} b \cdot b \cdot \ldots \cdot b \\ b \cdot b^{n-1} \end{cases}$$

$$f(n) = \begin{cases} 1 & \text{if } n = 0 \\ bf(n-1) & \text{otherwise} \end{cases}$$

simpler version of same problem

$$b^n = \begin{cases} (b^{n/2})^2 & \text{if } n \text{ even} \\ b \cdot b^{n-1} & \text{if } n \text{ odd} \end{cases}$$

$$f(n) = \begin{cases} 1 & \text{if } n = 0 \\ bf(n-1) & \text{if } n \text{ odd} \\ (f(n/2))^2 & \text{if } n \text{ even} \end{cases}$$

Slow exponentiation

```
def expo(b, n):
    if n == 0:
        return 1
    else:
        return b * expo(b, n-1)
```

```
expo(2, 10)
 expo args: (2, 10)
  expo args: (2, 9)
   expo args: (2, 8)
    expo args: (2, 7)
     expo args: (2, 6)
      expo args: (2, 5)
       expo args: (2, 4)
        expo args: (2, 3)
         expo args: (2, 2)
          expo args: (2, 1)
           expo args: (2, 0)
           expo result: 1
          expo result: 2
         expo result: 4
        expo result: 8
       expo result: 16
      expo result: 32
     expo result: 64
    expo result: 128
   expo result: 256
  expo result: 512
 expo result: 1024
1024
```

How does the time it takes to compute b^n grow as n grows?

linear n

Fast exponentiation

```
def fexpo(b, n):
    if n == 0:
        return 1
    elif n%2 == 1:
        return b * fexpo(b, n-1)
    else:
        return fexpo(b, n/2)**2
```

```
fexpo(2, 10)
 fexpo args: (2, 10)
    fexpo args: (2, 5)
       fexpo args: (2, 4)
          fexpo args: (2, 2)
             fexpo args: (2, 1)
                fexpo args: (2, 0)
                fexpo result: 1
             fexpo result: 2
          fexpo result: 4
       fexpo result: 16
    fexpo result: 32
 fexpo result: 1024
1024
```

How does the time it takes to compute b^n grow as n grows?

log(n)

Towers of Hanoi

Dr. Quandry

Move a stack of 64 discs of different sizes, such that at no time does a larger disc cover a smaller one.

hard to do procedurally

```
def Hanoi(n, From, To, Spare):
    if n == 0:
        print 'move from ' + From + ' to ' + To
    else:
        Hanoi(n-1, From, Spare, To)
        Hanoi(1, From, To, Spare)
        Hanoi(n-1, Spare, To, From)
```

How does the time it takes to compute b^n grow as n grows?

exponental n^2

n = 1000
log n = 10
linear = 1000
quadratic 10000
exponental way big!

Inheritance

Just as

- **Classes** capture shared attributes among their **instances**,
- **Superclasses** capture shared attributes among their **classes**.

Superclasses are environments

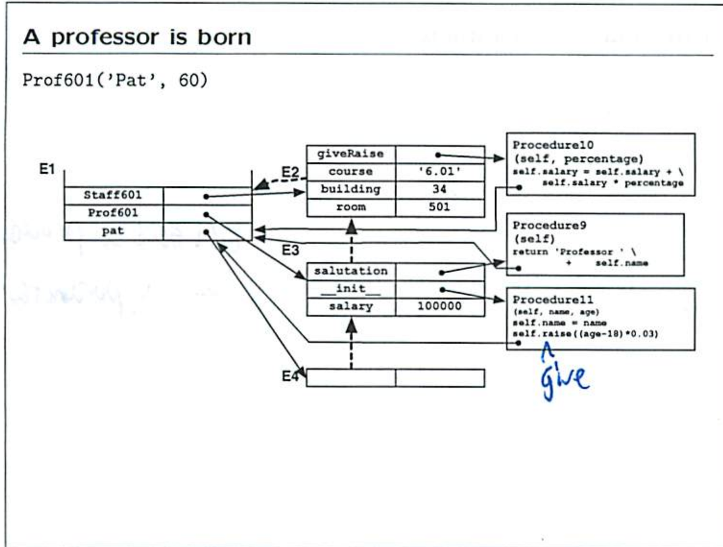
Classes are environments whose parent can be a superclass environment

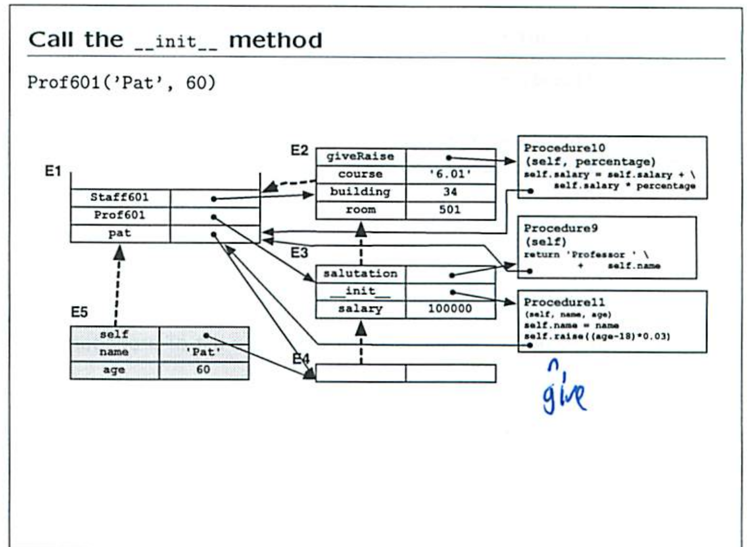
Instances are environments whose parent is a class environment

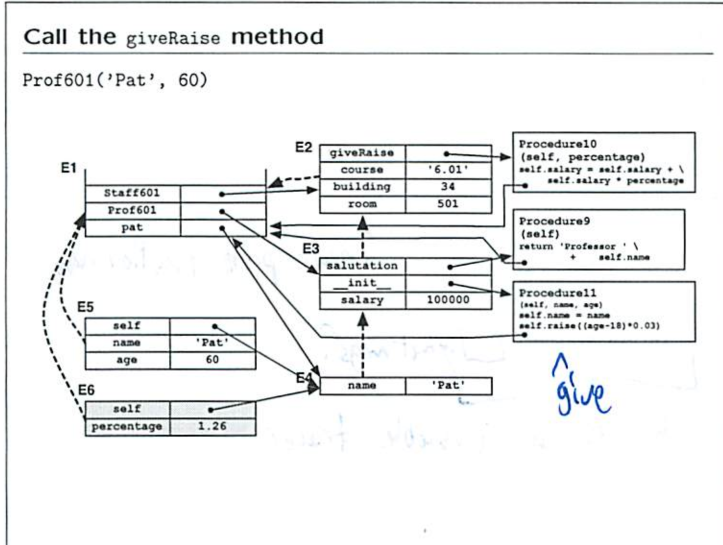
Staff6.01

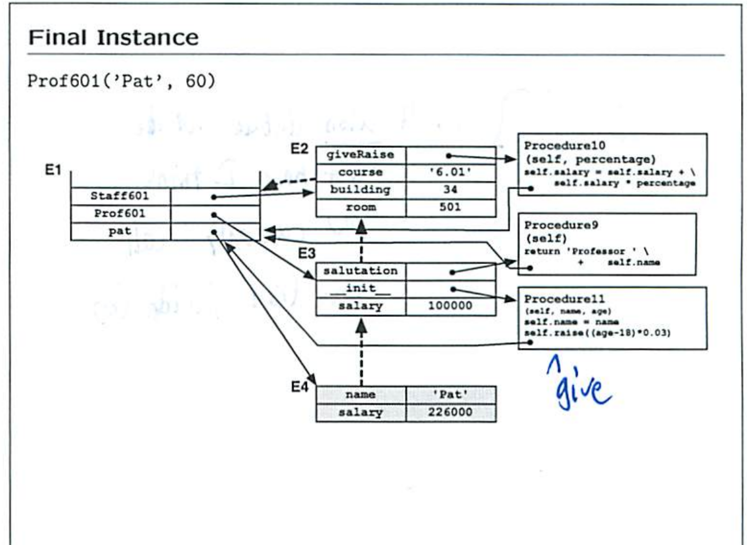
```
class Staff601:
    course = '6.01'
    building = 34
    room = 501

    def giveRaise(self, percentage):
        self.salary = self.salary + self.salary * percentage

class Prof601(Staff601):
    salary = 100000

    def __init__(self, name, age):
        self.name = name
        self.giveRaise((age - 18) * 0.03)

    def salutation(self):
        return 'Professor' + self.name
```

this environment inherits from superclass









Procedures and instances are first-class

They can be

- stored in variables or other data structures
- passed as a parameter to a procedure
- returned as a result from a procedure

Constructing procedures

```
>>> lambda x: x + 1
<function <lambda> at 0x2e4ddb0>
```

greek: make a procedure

```
>>> (lambda x: x + 1)(4)
5
```

Can separate concept of procedure as description of computational process from concept of abstraction by naming

```
def foo(a):
    def bar(x):
        return x + a
    return bar(6)
```

← here give it a name

```
>>> foo(3)
9
```

Storing procedures

```
>>> procs = [lambda x: x + 1,
             lambda x: x + 2,
             lambda x: x + 3]
>>> [p(100) for p in procs]
[101, 102, 103]
```

How could we apply the second procedure in a list of procedures to the argument 200?

Procedures as parameters

```
def applyTo5(p):
    return [p(x) for x in range(5)]

def proc(x):
    return 'this is a ' + str(x)

> applyTo5(proc)
['this is a 0', 'this is a 1', 'this is a 2',
 'this is a 3', 'this is a 4']

> applyTo5(lambda x: [x, x+x, x*x, x**x])
[[0, 0, 0, 1], [1, 2, 1, 1], [2, 4, 4, 4],
 [3, 6, 9, 27], [4, 8, 16, 256]]
```

can send a procedure as a parameter

Procedures as return values

```
def foo(a):
    def bar(x):
        return x + a
    return bar

>>> foo(6)
<function bar at 0x2e45dd0>

>>> foo(6)(3)
9

>>> thing = foo(5)
>>> thing(3)
8
```

could also define outside — but now I think you can only call bar from inside foo

Trace

```
def thing(x):
    return x + x * x

def trace(f):
    def tracedFun(arg):
        print 'Arg:', arg
        result = f(arg)
        print 'Result:', result
        return result
    return tracedFun
```

never prints function names

what args?

Write a reusable tracer

Check Yourself

```
>>> thing(4)

>>> trace(thing)

>>> tracedThing = trace(thing)
>>> tracedThing(4)

>>> tracedThing(tracedThing(4))

>>> trace(thing)(3)
```

20

<function ___ at ___ >

I knew it was weird

Arg: 4 Result: 20 20

This Week

Software lab: Practice with programming and OOP classes

Design lab: Building a complex class for polynomials

Homework 1: Symbolic calculator with ideas from Python's eval, and state machines. Due in parts, see Homework Tutor for details.

To get help:

- Email 6.01-help@mit.edu
- Go to lab hours (see course web page for times)
- Remember to check your due dates/times on the tutor

Software Lab 2: More OOPs

1 Setup

For this lab, it will be easiest to use one of our laptops or desktop machines. If you have already installed Python on your own laptop, you can use it, instead. If you haven't installed Python yet, and would like help, please bring your laptop to evening or weekend office hours.

- **Using a lab laptop or desktop machine**
  - Log in using your Athena user name and password.
  - Click once on the Terminal icon (usually on the bottom left of the screen.) In the terminal window, type `athrun 6.01 setup`. This step is only done for the first lab; for subsequent labs, do `athrun 6.01 update`. It will create a folder in your Athena account called `Desktop/6.01`.
- **Using your own laptop**
  - Go to the course web page: `http://mit.edu/6.01`
  - Go to the calendar tab, and download the zip file for software lab 1. Unzip it.
  - When we mention finding a file in `Desktop/6.01/...`, look for it in the folder you got by unzipping the archive.
- **Using course notes in lab**
  - Click once on the Firefox icon at the top left of the screen.
  - Go to `http://mit.edu/6.01`.
  - Click on `Reference Material` in the navigation bar.
  - Click on `Course Notes`.
  - In the popup window, click on `Open with`, choose `Document Viewer` from the pull-down list and click `OK`.
- **Using the online Tutor**
  - If you have not already registered for the 6.01 tutor, do so now:

  - ⋆ Click once on the Firefox icon at the top left of the screen.
  - ⋆ Go to `http://mit.edu/6.01`.
  - ⋆ Click on `Online Tutor` in the navigation bar.
  - ⋆ Under the Homework Tutor section, click on the `register here` link and follow instructions

- You can type Python expressions in idle's `Python Shell` window.
- You can write your programs in a file and test them using `Run Module`. For example:
  - Click idle's `File` menu, select `New Window`, and write `print 'Hello World'` in the window.
  - Click idle's `File` menu, select `Save as`, navigate to `Desktop/6.01/lab1/swLab/`, and enter the file name `test.py`.

    When using the lab laptops, if you find yourself in a file dialog box that seems to be far away from your home directory, you can always type ~ (the tilde character) in the box, followed by the Enter key; that should take you to your home directory, which contains your `Desktop` folder.
  - Click idle's `Run` menu, then select `Run Module`.
  - Look at the `Python Shell` window: you should see `Hello World`.

2 Exercises

If you have already worked through our Python programming tutor and/or have had other Python experience, then you should be all set to work on the problems below.

If you are still trying to get up to speed on Python, then please continue to work through the Python tutor. If you attended our extra help session last weekend, you should have signed up for a free 'new programmer' extension on the work of last week, and we will extend the same extension to those who signed up for this week.

2.1 Fibonnaci

Open the file `Desktop/6.01/lab1/swLab/sl1Work.py` (same as last week) and complete the definition of the `fib` procedure, so that `fib(n)` returns n^{th} Fibonnaci number. Recall that `fib(n)` is equal to the sum of `fib(n-1)` and `fib(n-2)`, that `fib(0)` is 0 and that `fib(1)` is 1.

Debug it in idle until it seems correct.

Wk.2.1.1	Check your results by copying the text of your procedure from idle and pasting it into this tutor problem.

2.2 Inheritance

Wk.2.1.2	Get some practice with inheritance in this tutor problem.

2.3 Rotating V2

Last week, you created two-dimensional vectors as a class, and provided a set of methods for supporting manipulation of those instances. This week, we want to extend the class by adding one more method, `rotate`, that creates a new vector which represents the original vector rotated about the origin.

You could go back and edit your class definition from last week to add a new method, but what if you had not written the original class yourself and did not have the original code? Instead, we will use inheritance, to define a subclass of `V2`. The `V2R` class should behave just like `V2` except that:

- it always converts its input coordinates to floating point numbers, using Python's `float`, e.g. `float(1)` return `1.0`.
- it has a new `rotate` method.

You should not make any assuptions about the implementation of the `V2` class, in particular, you should not assume that you know how the `__init__` method of the `V2` class works or what the instance variables are; you should only use the methods of the class.

Open `Desktop/6.01/lab1/swLab/sl1Work.py` (same as last week, which should have your definition of the `V2` class) and do the following:

Step 1. Define the basic parts of your class, with an `__init__` method and a `__str__` method, so that if you do

```
print V2R(1, 2)
```

it prints something like

```
V2R[1.0, 2.0]
```

Exactly what gets printed as a result of this statement depends on how you've defined your `__str__` procedure; this is just an example. Remember that `str(x)` turns `x`, whatever it is, into a string.

Step 2. Define the `rotate` method, which rotates a `V2` (around the origin) by some angle θ, yielding a new `V2`. You may want to import the math module in order to use trigonometric functions. You should get the following behavior:

```
>>> a = V2R(1.0, 2.0)
>>> print a.rotate(math.pi/2)
V2R[-2.0, 1.0]
```

Wk.2.1.3	Check your results by copying the text of your procedure from idle and pasting it into this tutor problem.

2.4 Two-dimensional line segments

Now we want you to create a two-dimensional line segment, which is composed of two (rotatable) vectors, and start vector and an end vector. Think of a line segment an arbitrary line between a starting point and a termination point.

Open `Desktop/6.01/lab1/swLab/sl1Work.py` (which should have your definition of the `V2R` class) and add the definition of the `Seg2` class; it represents two-dimensional segment and supports the following operations:

- Create a new segment out of two vectors: `u = V2R(1.1, 2.2), v = V2R(-1.5, 3.4), seg = Seg2(u, v)`.
- Convert a segment to a string.
- Translate a Seg2 by adding a vector to both the start point and end point, yielding a new Seg2.
- Multiply a Seg2 by a scalar (real or int) and return a new Seg2.
- Rotate a Seg2 (around the origin) by some angle θ, yielding a new Seg2. This involves rotating the end points of the segment.

Step 3. Define the basic parts of your class, with an `__init__` method and a `__str__` method, so that if you do

```
print Seg2(u, v)
```

where u and v are V2R instnaces, it prints something like

```
Seg2[V2R[1.1, 2.2], V2R[-1.5, 3.4]]
```

Exactly what gets printed as a result of this statement depends on how you've defined your `__str__` procedure; this is just an example.

Step 4. Write two accessor methods, `getStart` and `getEnd` that return the `Start` and `End` components of your segment, respectively. For example,

```
>>> s = Seg2(V2R(1, 2), V2R(-1, 3))
>>> s.getStart()
V2R[1.0, 2.0]
>>> s.getEnd()
V2R[-1.0, 3.0]
```

Step 5. Define the `translate` and `scale` methods, so that you get the following behavior:

```
>>> a = V2R(1.0, 2.0)
>>> b = V2R(2.2, 3.3)
>>> c = V2R(3.3, 4.4)
>>> s = Seg2(a,b)
>>> print s.translate(c)
Seg2[V2R[4.3, 6.4], V2R[5.5, 7.7]]
>>> print s.scale(2)
Seg2[V2R[2.0, 4.0], V2R[4.4, 6.6]]
```

Step 6. Define the `rotate` method, so that you get the following behavior:

```
>>> a = V2R(1.0, 2.0)
>>> b = V2R(2.2, 3.3)
>>> s = Seg2(a,b)
>>> print s.rotate(math.pi/2)
Seg2[V2R[-2.0, 1.0], V2R[-3.3, 2.2]]
```

Test your implementation in idle until it seems correct to you.

Wk.2.1.4	Check your results by copying the text of your procedure from idle and pasting it into this tutor problem.

2.5 More Inheritance

Wk.2.1.5	Get some more practice with inheritance in this tutor problem.

2.6 List practice

Wk.2.1.6	Get some practice with list structures in this tutor problem.

Wk.2.1.7	Get some practice with list comprehensions in this tutor problem.

6.01 More Python and OOP — Fall 2010 6

SW Lab 2 2

(Scratchpad)

9/14

fib

- ~~have~~ i try to make it recursive

error int object not iterable

i have an accumulator, or some sort of iterator

only n-1 + n-2

fib 5 = fib (5-1) + fib (5-2) + ~~fib (5-3) + fib (5-4) + fib (5-5)~~

fib 4 + fib 3 + fib 2 + 1 0

fib 4-1 + fib 4-2 + fib 1 + 0 + fib 2 + 1 + 0 + 1

so try to do in 1 function

fib (0) = 0 ✓

fib (1) = 1 ✓

fib (2) = fib 2-1 + 0 ✓

1 0 = 1

fib (3) = fib 3-1 + fib 3-2 + fib 3-3

fib 2-1 + 1 + 0

1 + 0 + 1 + 0 = 2

②

fib 5 = fib 5-1 + fib 5-2 not ...

now that returns right

[eaiser than I thought

did not need to think through full recursion

2.2 like thing

foo. get a

return a + NN. get 'a1

self. n += 1

error - int + string

(a1) ooh it concats

2.3 V2R

- not assuming beyond what is defined =
only use interface

rotate
arand origion
converts input to float

③

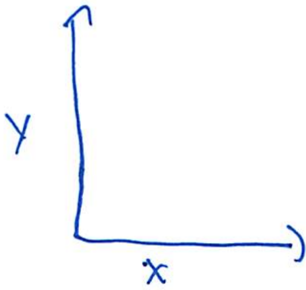
__init__ ✓

rotate (θ)

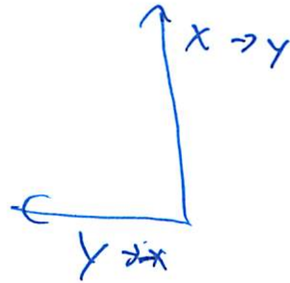
- need to think through trig

import math

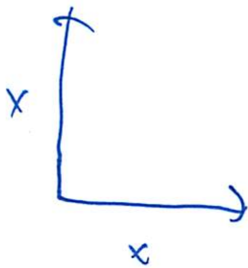
pi/2 = 90°



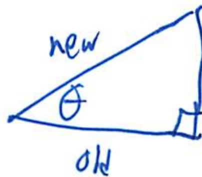
↻ 90°



pi/4 = 45°



↻ 45°

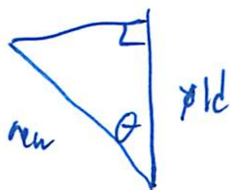


~~tan θ~~ ~~sin~~ θ =

$\cos\theta = \frac{old}{new}$

new cos θ = old

new = old / cos θ



new = old / cos θ

make sure self.getX() ← don't forget ()

(4)

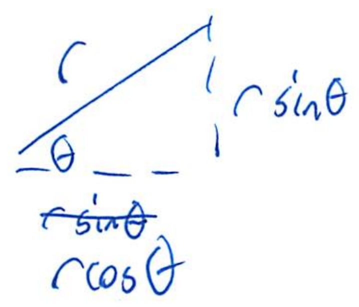
__init__

V2.__init__(float(x), float(y))

don't save to somewhere

got it running but rotate error

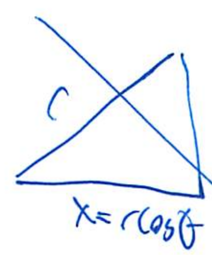
think in polar coords (r, θ)



$\sin\theta = \frac{opp}{r}$

but need to convert x, y to r

was right



$y = r\sin\theta$ $r = \frac{y}{\sin\theta}$

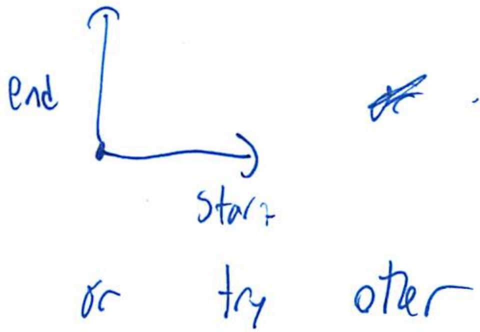
$r = \frac{x}{\cos\theta}$

(too many variables?)

skip for now time pressure

(5)

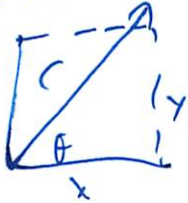
Try 2.4



or try other

---

go back 2.3



~~$r = \tan\left(\frac{y}{x}\right)$~~

$r = \tan\theta = \frac{y}{x}$

$\theta = \tan^{-1}\left(\frac{y}{x}\right)$

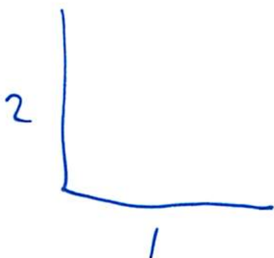
$r = \sqrt{x^2 + y^2}$

so $x =$ ~~$\tan\left(\frac{y}{x}\right)$~~ $\cos\theta$

$y =$ ~~"~~ $\sin\theta$

$\sqrt{x^2+y^2}\ \cos\theta$

(r, θ)



$\sqrt{1^2 + 2^2}$ $\sqrt{5}$ $\sqrt{5}\cos($~~1.5707963~~$)$ $\frac{\pi}{2}$ 0

① Ok so 2.4 got extended

TA told me to look up rotate formula

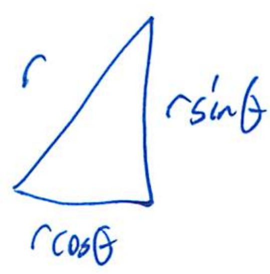
q = inital angle

θ = rotation

$x = r\cos q$

$y = r\sin q$

} so I had this



$x' = r\cos(q+\theta)$

$y' = r\sin(q+\theta)$

↑ duh add to angle not some sort of new angle w/ that

$x' = r\cos\theta\cos q - r\sin q\sin\theta$

$y' = r\sin q\cos\theta + r\cos q\sin\theta$

$$x' = x\cos\theta - y\sin\theta$$
$$y' = y\cos\theta + x\sin\theta$$

I really hate the time pressure of the class

(7) I felt really time pressured today

- hard to think
- don't learn
- feel bad, need min to unwind

most pressured in a few months

SW Lab 2 More (HW) 9/15

2.4 2D Line Segments

? don't fully understand

how can you have start + end vector?

init + print works first try

use getX() not [0]

---

2 accessor methods

get start + end ✓

- easy

---

now translate

? move over?

by a certain vector

or what is it doing

- just adding

Should I build getStartX()

or will getStart().getX() work???

bingo → getting the hang of this!

② What is scale

- multiply by a scalar (real or int)
- return ~~self~~
  ↳ seg2

↑ what is a real?
int, long, float

just do multiplication ✓ → when relaxed easy and kinda fast

---

rotate method

- oh no
- well reuse old method
  - or just <u>call it</u>!

smart! - done

~~all~~ Passed 1st try!

2.5 Inheritance

Account Dollars (inital).

deposit dollars()

define ~~the~~ Account Pounds

- but where there are 2 $ to a GBP

1 ~~GBP~~ GBP = 2 USD

.5 " = 1 "

~~when~~ when calling something (self, etc...) and I had a print instead of a return

③ 2.6 Nesting

Give a python ~~sta~~ statement which evals to that pic
- weird
- yeah multidimension lists

Part 2 Sharing 1
- opps it was pointing not to ~~box~~ cell, but whole box

2.7 List Comprehension
- this was kinda confusing

[expression for variable in list]

ie [math.sqrt(x) for x in primes if x > 5]

~~like listed nets~~

ie [x·y for x in primes for y in [1,2,3]]
- like ~~X~~ nested for loop
- got it 1st try
- weird but kinda useful

part 2 is weird
- I did a sum in one of the other things
    - but I think I cheated in it

how w/ 2 lists
got it 1st try

(Done lab)

Design Lab 2: Polynomial Class

Design labs are generally done with partners, but this one should be done individually.

- **Using a lab laptop or desktop machine**
  - Log in using your Athena user name and password.
  - Click once on the Terminal icon (usually on the bottom left of the screen.) If you have not already done this, in the terminal window, type `athrun 6.01 setup`.

    Type `athrun 6.01 update` to get the latest batch of files.
- **Using your own laptop**
  - Go to the course web page: `http://mit.edu/6.01`
  - Go to the calendar tab, and download the zip file for design lab 2. Unzip it.

The design lab for this week is to implement a Python class that provides methods for performing algebraic operations on polynomials.

Representation

We can represent a polynomial as a list of coefficients starting with the highest-order term. For example, here are some polynomials and their representations as lists:

$x^4 - 7x^3 + 10x^2 - 4x + 6$	`[1, -7, 10, -4, 6]`
$3x^3$	`[3, 0, 0, 0]`
8	[8]

> **Wk.2.2.1 Part 1** It is a little bit tricky to implement addition and multiplication of polynomials. Do Part 1 of tutor problem Wk.2.2.1 before you start programming, and **be sure you understand the results in the example transcript near the end of this handout.**

Operations

Edit the definition of the `Polynomial` class in `Desktop/6.01/lab2/designLab/dl2Work.py`. Your class should have one attribute and several methods:

- An attribute called `coeffs`, which is the list of coefficients used to create the instance. It must have this name or the tests in the tutor will fail.
- `__init__(self, coefficients)`: initializes the `coeffs` attribute to be a list of *floating-point* coefficient values.
- `coeff(self, i)`: returns the coefficient of the x^i term of the polynomial. For example, the coefficient of term 3 of $x^4 - 7x^3 + 10x^2 - 4x + 6$ is -7.
- `add(self, other)`: returns a new `Polynomial` representing the sum of `Polynomials` `self` and `other`. **Be sure that performing any operation on polynomials, e.g. `p1 + p2`, *does not change* the original value of `p1` or `p2`.**
- `mul(self, other)`: returns a new `Polynomial` representing the product of `Polynomials` `self` and `other`
- `__str__(self)`: converts a `Polynomial` into a string. Do the simplest thing that shows the coefficients; remember that `str(x)` turns `x`, whatever it is, into a string.

  After you're done with everything else, go back and change your `__str__` method to print polynomials out as they are shown in the transcript at the end. This is not required; do it only if you have time and interest.
- `val(self, v)`: returns the numerical result of evaluating the polynomial at $x = v$.
- `roots(self)`: returns a list containing the root or roots of first or second order polynomials (for orders other than 1 and 2, just print an error message saying that you don't handle them).

  For second-order (quadratic) polynomials, return real roots (a single number) if possible, and otherwise return complex numbers. Python has built-in facilities for handling complex numbers: `3 + 2j` stands for a number with a real part of 3 and an imaginary part of 2. You can take square roots of complex numbers by using a fractional exponent:

  ```
  >>> (3 + 2j)**0.5
  (1.8173540210239707+0.55025052270033747j)
  ```

  To take the square root of a negative number, you first have to convert it to a complex number; you can do this by adding `0j` to it or by using `complex(x, 0)`, where `x` is the number.

  ```
  >>> (-3)**0.5
  Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
  ValueError: negative number cannot be raised to a fractional power
  >>> (-3 + 0j)**0.5
  (1.0605402120460133e-16+1.7320508075688772j)
  ```

  You can get the real part of a complex number `z` by using `z.real`.

Try to do things as simply as possible. Don't do anything twice. If you need some extra procedures to help you do your work, you can put them in the same file as your class definition, but outside the class (so, put them at the end of the file, with no indentation).

Operator overloading

In order to use expressions like `p1 + p2`, `p1 * p2`, and `p1(3)`, for addition, multiplication, and evaluation, respectively, define the specially-named methods `__add__`, `__mul__`, and `__call__`. So for example, include

```
def __add__(self, other):
    return self.add(other)
def __mul__(self, other):
    return self.mul(other)
def __call__(self, x):
    return self.val(x)
```

Multiplication will be similar to addition. Also, in order to have your polynomials printed out nicely by the Python shell, you can add this line to your class:

```
def __repr__(self):
    return str(self)
```

which says that the the shell should print the string returned by the `__str__` method.

Sample transcript

```
>>> p1 = Polynomial([1, 2, 3])
>>> p1
1.000 z**2 + 2.000z + 3.000
>>> p2 = Polynomial([100, 200])
>>> p1.add(p2)
1.000 z**2 + 102.000z + 203.000
>>> p1 + p2
1.000 z**2 + 102.000z + 203.000
>>> p1(1)
6.0
>>> p1(-1)
2.0
>>> (p1 + p2)(10)
1323.0
>>> p1.mul(p1)
1.000 z**4 + 4.000 z**3 + 10.000 z**2 + 12.000z + 9.000
>>> p1 * p1
1.000 z**4 + 4.000 z**3 + 10.000 z**2 + 12.000z + 9.000
>>> p1 * p2 + p1
100.000 z**3 + 401.000 z**2 + 702.000z + 603.000
>>> p1.roots()
[(-1+1.4142135623730947j), (-1-1.4142135623730947j)]
>>> p2.roots()
[-2.0]
>>> p3 = Polynomial([3, 2, -1])
>>> p3.roots()
[-1.0, 0.33333333333333331]
```

```
>>> (p1 * p1).roots()
Order too high to solve for roots.
```

Wk.2.2.1 After you have debugged in idle, check and submit your results by copying the text of your class and associated definitions from idle and pasting it into the tutor problem Wk.2.2.1.

Optional

There's a particularly elegant way to implement the `val` method, using *Horner's Rule*. For computing the value of a polynomial, it structures the computation of

$$a_n x^n + a_{n-1} x^{n-1} + \cdots + a_2 x^2 + a_1 x + a_0$$

as

$$(\cdots(a_n x + a_{n-1})x + \cdots + a_1)\,x + a_0$$

In other words, we start with a_n, multiply the entire result by x, add a_{n-1}, multiply by x, and so on, until we reach a_0. For example, we'd evaluate $8x^3 - 3x^2 + 4x + 1$ as

$$((8 \cdot x - 3) \cdot x + 4) \cdot x + 1$$

For fun, try implementing `val` with Horner's rule. Think about how many multiplication operations it takes to evaluate a polynomial using Horner's rule, compared to the usual way.

Design Lab 2

9/16

- polynomials

$x^4 - 7x^3 + 10x^2 - 4x + 6 \rightarrow [1, -7, 10, -4, 6]$

↑ by virtue of position

- tutor problem - simple math

product

- forget??
- mutliply each coefficent? no - remember

$(4x)(4x) = 16x^2 = (4x)^2$

$(5x^2)(-2x^3) = -10x^5$

$(x+3)(x+2)$

$x^2 + 3x + 2x + 5$

$x^2 + 5x + 5$

$(3x^3 + 2x - 4) \times (2x + 7)$

$6x^4 + 4x^2 - 8x + 21x^3 + 14x - 28$

6 21 4 6 -28

(2)

$(3x^3 + 2x - 4)(2x)$

$6x^4 + 4x^2 - 8x$

6 0 4 -8 0

21 0 14 -28

Now on to code

coeffs

ith term

- what is 0
- its not just [in]

[1 -7 10 -4 6]

0 1 2 3 4

4 3 2 1 0 ✓done

(I like how they build it up)

add(self, other)

- return new, don't change either

(3)

I really think I should reverse order of list
- Then simple to add list
- reversed - was a bit tricky w/ 0 index etc
  - I just guessed + checked

Write quick print method!

other must be a polynomial

Should also do the __ add __

~~method~~

Why is it running so much! - it reverses for printing!
- Or is it auto reversed?

or use that list function thing

[x + y for x in self.coeffs for y in other.coeffs]

now acting weird though

add works now w/ lists at same lenght
- what if not same length?
  - either write checker + fixer (ugly)
  - or write a more robust script
  - or a returner - if list does not exist → return 0 (already have function!

63

Cool I think my

max Index function and self.coeff(i) returning 0 on error (using try)

print a+b works!

now multiply
- this will be much harder

[0, 1, 2]

perhaps go though each in other one at a time

[
0 - do nothing
1, shift everything to the right
but depending on its placement
- well first string multiplies

pos [
last → ~~every~~ my 0 pos → just multiply
1 → move everything ← 1 and multiply
2 → move everything over 2 + multiply
(is this right?)

[>1 → multiply by that

(5) Go back at example

[3 0 2 -4] [2 0]

[6 0 4 -8 0] ← oh wow that shifting does work!
- if only I did that in hs!

So take other

for x in other

if x = 0 just multiply ← sub function (private)

move everything x positions, then muliply
(sub function (private))

getting kinda confused on what is object
- since got into mulilating!
- which is larger work w/
- but solve it assuming one is larger

I want it to append at end
- reverse, append, reverse
↑ This is ugly since it should take self

or use insert (i, x)

(6)

Think it might be working now

- let try

- don't multiply by 0!

→ skip

- so now I have

[6, 0, 4, -8, 0, 0, 0, 0, 0, 0, 0]

- don't do max leng → do len other

-

- Works on mine!
- works on reverse
- try some others
  - tried all from above

Val

v=5 [1, 2, 3] → $(5)^2 + 2(5) + 3$

i: 2 1 0

3^0

let me test

[1, -7, 10, -4, 6] at v = 1

... $+ 10(1)^2 + -4(1) + 6$ just add → 6 ✓

7

now $r=2$

$1(2)^4 + -7(2)^3 + 10(2)^2 - 4(2) + 6$

$16 + -7 \cdot 8 + 10 \cdot 4 - 8 + 6$

$16 - 56 + 40 - 8 + 6 = -2$ ✓

Should be good

---

roots

- what is this again mathmatically
- oh rootin' w/ the quadratic formula
- look up my app from 8/9th grade!

2nd order ~~[a, b, c]~~

$5x^2 + 3x + 7$

[5, 3, 7] so len = 3

a	b	c
2	1	0

need good example

$(x+1)(x+2)$

$X2 + 3x \cdot 2$

- but wrong - finish for hw

(8) At home now

- not being graded on style
- fixed roots
  - needed extra parenthese
- now w/ complex #
  - should be automatic
  - but use ** .5 not sqrt()
    and call complex on the number
    - negitive numbers need 0J added to it
    - done
- ok time for testing

Operation overloading
- need call and repr

---

now run transcript
- oh I never did nice print
- this is optional I think
- also they always return floats
- multiply not even close
  - for roots should return I numbers?
  - p2.root errors since a = 0 - oh need to do manually
    100x + 200

(9)

$100x + 200$

$100x = -200$

$x = -2$

- different method
- implemented
- remember for list order does not matter

Ok try pasting it in + see how I fail

- got every one wrong!
- oh they send in tuples
  - but where feeding it into polynomial system
- or perhaps they are not printing python code
- now this time it produced something different!
  - 3rd result

but how is add so wrong → tested right?

- called twice
- try simpler cases

$[1, 0, 1] + [2, 1]$

- worked!
- calc what they have manually

(10)

- yeah it looks to be right
- 3rd step now
  - ~~but this is a 4th result!~~
  - same as the result the python got
- unless my understanding of this is wrong
  - why is ~~an~~ idle + tutor getting diff values?
  - something about floats?
- and why did code work well w/ simple examples from the log?
- and still works fine?

- even roots fails!
  - oh it wants x, y pairs ??
  - or real and imaginary seperate
- now # all messed up
  - on web ans were kinda the same
  - different in idle
- email help sent
  - completely confuse
- let me try having in't make sure it is floating pt
  - nope

(1) ———————————————— (email) 9/18

Oh what I was producing is reverse
- but ~~chang temi~~ are they reaching in to my program?
- w/o interface?

And they told me how they wanted roots

Now need to wait
- or do thw calc
- but thw exercises before that

———————————————— (email)

they want coeffs to be in proper format
- so I have to ~~run~~ rewrite all code?
- grrr
- complex, make sure to check
- add works again
- mul seems to be working what I thought, but formula wrong?

$[3, 0, 2, -4]$ $[2, 7]$

$3x^3 + 2x - 4$ $2x + 7$

$6x^4 + 4x^2 - 8x + 21x^3 + 14x - 28$

yeah I am not adding ~~write~~ right

(12) start from front
Then add each pice
not multiply w/ what you have?

Or can I start from back
Yeah
- but then for other one ~~add~~ start fresh + add
- call it step

~~new~~ real ans:

$6x^4 + 21x^3 + 4x^2 - 8x + 14x - 28$

[6, 21, 4, 6, -28]

✓ correct

now try transcript
- bingo
- so I was doing it wrong the whole time

---

time for a check
but the paper says return real root when possible
- yeah that is what my code will do

Bingo code works!

(13)

Oh on final submit it screwed up
Since the complex thing returned +0 J

-grr

I had complex in there
- is the only way to do it w/ if statement

Well w/e -> too late now

(1)

Wk 2 HW Exercises

9/18

intro to recursion
class notes 2.3.2
- but those notes don't exist :(
- email sent

---

base cases → 1 or more
recursive cases → 1 or more

---

recursive add
- this looks stupid
- try it out
- works
- kinda weird

Part 2 : b must be int ✓
a can be any number ✓
- what about negitive
- b must be positive ✓
- a can be any ✓

(2)

Sub Part 3

got it
but I forgot to change add -> sub in function call

#2 Slow mod

- modulus operation
- %
- WP: finds the remainder of division of 1 # by another

---

recursive function
a, b - positive integers
can only + - and simple "if" tests

- decompose problem

Slow mod (5, 2)

5 - 2 - ..... = 1

while -> do recursive instead?

if a - b > b
 Slow mod (a-b, b)
else
 return a - b

3

Is it not returning properly?

return SlowMod(a-b, b)

There we go

SlowMod(6, 2)

- wrong
- do if (a-b) >= b:

slowMod(4, 6)

- wrong
- if b > a ← at very top
  return a

#3 Inheritance + State Machines

Prof says read Course Notes 3.6 via email
- for recursion

All of chap 4 is state machines
- the topic for next few weeks
- should read it fully

3.6 Recursion

9/18

an interpreter is also recursive

function/procedure is a black box

Can get a lot of work done - solve a little bit each time

Calls itself from inside itself

have base case(s) and recursive case(s)

```
add(m, n):
    if n == 0:
        return m
    else:
        return 1 + add(m, n-1)
```

So add(2, 2)

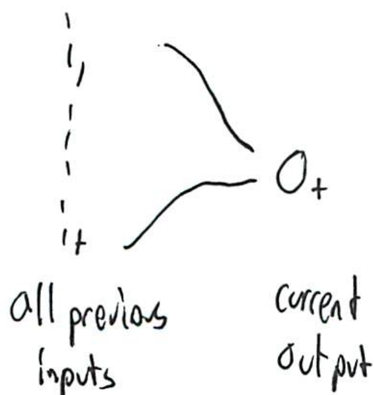
gives

add(2,2)
  add(2, 1)
    add(2,0) - base case
    2
  3
4

structure → base case → answer

Course Notes 4 State Machines Reading 9/28

- method of modeling system whose output depends on the entire history of their inputs → not just the most recent input
- use cases
  - UI → keyboard, mouse
  - Conversations → it depends on what has been said
  - State of spacecraft → valves open?
  - Sequential patterns in DNA
- can be continuous time or discrete time
  - continuous time ↓ continuous space, differential eq to explain system dynamics, but sensor's output is discrete
  - discrete time ↓ we will use
- job of embedded system → perform transduction from a stream (∞ sequence) of input values to a stream of outputs



- but this complex as previous inputs grow

(2) So we will define <u>states</u> we want to look at
- difficult sometimes to know what states to look at

3 ways will use in this class

1. <u>Synthetically</u> → can specify a "program" for robot
   - inputs → sensor readings
   - outputs → control commands
2. <u>Analytically</u> → can describe the global properties of a system
   - inputs → a simple command to entire system
   - output → simple measure of state of system
   - <u>ie</u> will a global system oscillate or diverge, etc
3. <u>Predictively</u> → can describe how the environment works
   - <u>ie</u> where I will end up if I drive down a certain road
   - <u>ie</u> which path should I take through space to reach a certain state

(state, system, etc are all used by Course 6, ESD very abstractly - kinda mathmatical)

build complex state machines by combining primitive state machines

3) Primitive State Machines

Specify a transducer as a SM by specifing:

- Set of states S
- Set of Inputs I → input vocabulary
- Set of outputs O → output vocabulary
- next-state function $n(i_t, s_t)$
  - maps input at time t and the state at time t to the state at time $t+1$, s_{t+1}
- output function $o(i_t, s_t)$
  - maps input at time t and that state at time t to the output at t, o_t
- initial state s_0
  - state at time 0

Examples

- tick tock machine
  - 1, 0, 1, 0, 1 - finite-state
  - ignores input
- Controller of digital watch
  - transduces inputs (button presses) into seq. of outputs (segments of display)

(4)

Simplest state machine is a pure function

- no state
- output is $o_t = i_t + 1$
- immediate input → output relationship

linear, time invariant systems in chap 6

Language Acceptor

- returns true if input in a certain pattern a, b, c
- uses states 0, 1, 2 to stand for situations
- state 3 = example not expected
- once machine goes into 3 → it can never exit

$S = \{0, 1, 2, 3\}$

$I = \{a, b, c\}$

$O = \{True, False\}$

$$n(s,i) = \begin{cases} 1 & \text{if } s=0 \; i=a \\ 2 & \text{"} \; s=1 \; i=b \\ 3 & \text{"} \; s=2 \; i=c \\ 4 & \text{otherwise} \end{cases}$$

$$o(s,i) = \begin{cases} \text{False} & \text{if } n(s,i) = 3 \\ \text{true} & \text{otherwise} \end{cases}$$

$s_0 = 0$

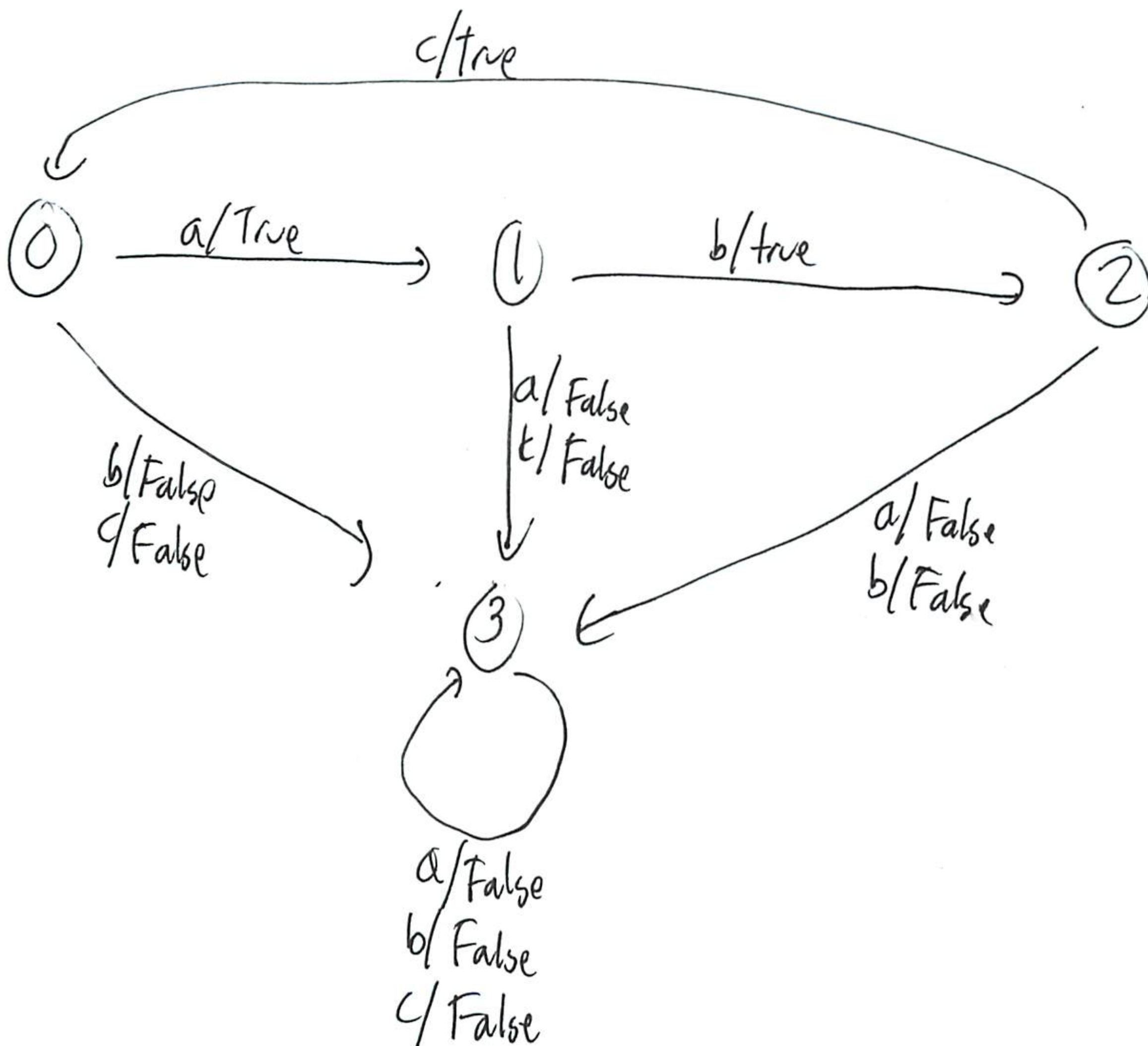
(5)

State transition diagram

- each circle is a state
- arcs are possible transactions the machine can make
  - labeled input/output

    $n(s,i)$ ↓ ↑ $o(s,i)$

  - directional w/ arrows
- must be an arc coming out of every state for every possible input



(6) ~~SM~~ State Machine Tables

time	0	1	2	...
input	i_0	i_1	i_2	...
state	s_0	s_1	s_2	...
output	O_1	O_2	O_3	...

now feed our machine with a, b, c, a, c, a, b

time	0	1	2	3	4	5	6	7	
input	a	b	c	a	c	a	b		
state	0	1	2	0	1	3	3	3	...
output	True	T	T	T	F	F	F	F	

- for class of regular language patterns see stack machine

Up + Down Counter

- starts at 0
- if gets "u" then increments
  - "d" decrements
- output = new state

S = integers

I = {u, d}

O = integers

$$n(s,i) = \begin{cases} s+1 & \text{if } i=u \\ s-1 & \text{if } i=d \end{cases}$$

$$O(s,i) = n(s,i)$$

$$s_0 = 0$$

(7)

Delay

input → output after some delay

S = anything $\quad n(s,i) = i$

I = " $\quad o(s,i) = s$

O = " $\quad S_0 = 0$

So it takes an input and stores it for 1 cycle

time	0	1	2	3	4	5
input	3	1	2	5	4	
State	0	3	1	2	5	4
output	0	3	1	2	5	

Accumulator

(remember s = old state, n = new state)

S = numbers $\quad n(s,i) = s + i$

I = " $\quad o(s,i) = n(s,i)$

O = " $\quad S_0 = 0$

Average

- current val + previous val

S = numbers $\quad n(s,i) = i$

I = " $\quad o(s,i) = (s+i)/2$

O = "

(boring though
- but if wanted all avg need 2 variables)

(8) State Machine in Python

- build up simple infrastuture
- w/ OOP
- SM abstract class as superclass

```
class Accumulator (SM):
    startState = 0
    def getNextValue(self, state, input):
        return (state + inp, state + inp)
```

↑ note does not change state of machine
since we just want this function to test what new value can be

(I see now how you can use this powerful abstract concept in plenty of situations)

make in instance by calling start
then have it actually change w/ step(inp)

↑ using word inp not input to avoid bugs w/ built in input()

(9)

if we had an `__init__` method w/ start as argument
→ Then we are defining a set of machines
  ↑ technical case

have start be a seperate method
 - (why??)

```
step (self, inp) :
    (s, o) = self.getNextValue(self.state, inp)
    self.state = s
    return o
```

Can make a transduce() for ~~range~~ list of inputs

```
* induce(self, inputs)
      Self.start()
      return [self.step(inp) for inp in inputs]
```

Can define some default methods

Thoughts 6.01

9/18

6.01 is like Christmas
- a new puzzle
- mental challenge
  - want to play with it, engrossed
- don't want it to get old + stale
- a mental challenge

Like traveling as well

Back to HW

9/18

Counting State Machine (SM)

~~Subclass~~ def getOutput

- returns just output

- example of Count Mod 5 class
- build Counting State Machine class
  - need to define start state
    - get next values
  - get output ~~pre defined~~ will be defined in subclass
- Then another subclass: Alternate Zeros
  - ~~other~~ even steps 0 = i
  - odd steps = 0
  - So $i_0, 0, i_2, 0, i_4, 0 \ldots$
- study Staff601 + SM classes
  - super class method can refer to subclass methods

(2) But their is no input in this

- Just ignore

w/ get Next Value

1st return = state
2nd " = output

((Cool how you can have 2 returns!
(- can php do that?)

- remember don't change state here
- don't define % 5 here
  - this is generic state machine

~~state - set~~

Step
- set state

try running
- have compiled SM code

- now giving me init error on init sm
  - but I never called it ??
  - 3 arguments given, takes 2
  - someone online says that it is a module, not a class
    - so what is the class?

③ dir(sm) should tell you what module defines

- but does nothing - if run it direct in terminal it does

So how to move forward

- why can we not see SM
- another person mentions subclassing a module

Still don't get this module thing

- we did not do an example in class ← typical MIT
- tomorrow's class

- go back + read old notes on OCW
- or just implent SM myself
  - good
- now why does get Output require the state and an input ?!
  - self.state → well c.state
    - input does not matter
  - works
- but get next value also needs ~~input~~ def in Count Mod 5
  - or does it?
  - do we not care about its output, its wrong

or def get Next value

return State +1, get ~~O~~utput (state + 1)

- much better
  - now need Alt 0s
    - then check

(4)

- how will I track even odd
  - well just one state
  - can determine even out on the fly
- Cool works
  - try it out
- wrong
  - it demands an transduce method
  - Oh it custom defines a getOutput method - cool
  - but it shold return a list???
    - why/how?
  - get output needs to take inputs
    - but how??
    - how does input change it in the definition?
  - oh for aternate() → output = input on even
    not state
    ~~we don't care about~~ 2r
    never output state
    ↑ don't confuse new state & output!

⑤

Big mistake!

get Next Val

return state + 1, self.getOutput(state, inp)

new state — current output

↑ not state + 1

* we are trying to break things up in parts

Done!

3.7 Implementing an Interpreter

3/19

lang Spy = simple syntax of Scheme
+ OOP of Python

- fully parenthesized
  - around every sub expression
  - easy for computer
  - hard for humans
- assume someone wrote tokenizer that breaks input into a list of lists
- features
  - integers
  - + - * / = ← test
  - assignment
  - functions
  - function definition
  - if a b c
    - a ← test, b ← true, c ← false
  - compound expressions

②

Evaluating

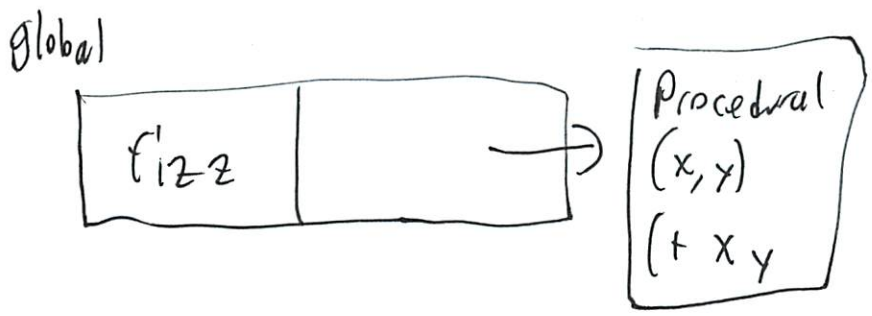
- using an Interperter (executes code directly)
- first just return if we have an integer
- if "begin" - return compound expression
- how symbols : anything beside q. # or ()
- need an environment dictionary
  - + dictionary
  - + parent env
- say a symbol is <u>bound</u> if it is defined
- relationship key → value = <u>binding</u>
- need to be able to perform a <u>lookup</u>
  - first in env's dictionary
  - then in parent's env
  - else error
- need an <u>add binding</u> operation
- functions
  - if an expression is <u>not</u> a special symbol → its a function call
  - 2 types of functions
    - primitive - built in, do actual work
    - user defined, set of primitive functions

③

function definition have 4 parts

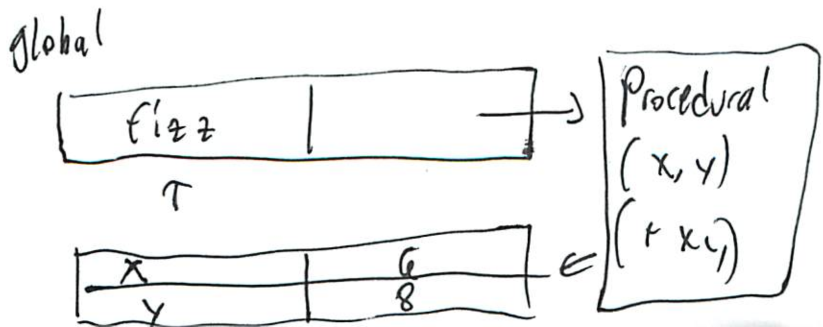
1. def. → special symbol
2. Function name
3. formal parameters
4. body

when define store the parameters, body, enviroment where defined



function calls

- trickiest part
- first eval each expression
  - ↳ get function instance, 6, 8
- make a new Env
  - ↳ child of env where it was defined
  - not env where it was called (does not matter)



4

Now eval expression in the new env

x, y } are in new env

+ } in global env

Can do a trace to see what is going on

(how do you do that → seems helpful)

<u>if</u>

- can't use primitive built-in if function
  - we can't stop recursion
  - (I don't really get why)

Complete!

<u>OOP Spy</u>

- modeled on Python's OOP but simpler

* Classes and instances are both environments
- Only need to add 2 syntatic features:

<u>attribute lookup</u>

obj.a → (attr obj a)

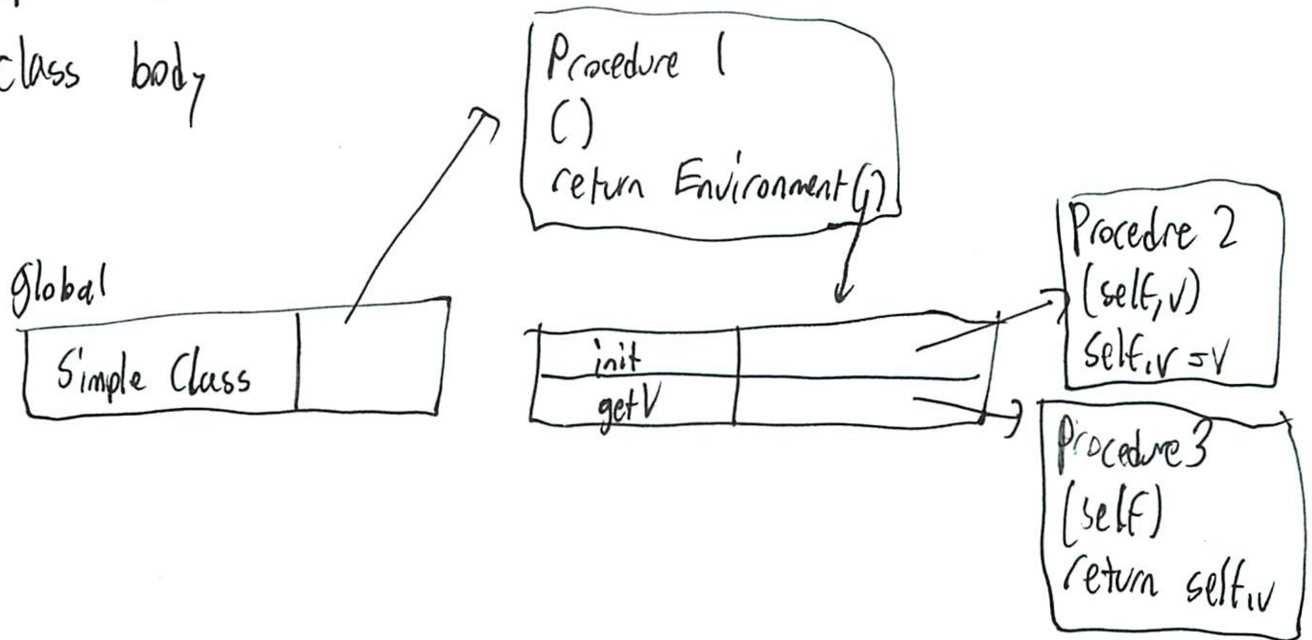
look up a in env obj

<u>Class definition</u>

lot of sub parts (4)

(5)

1. Symbol "class"
2. name
3. super class
4. class body



bind class to procedure to make it
- not directly to parent env

3 steps to process

1. Make env for class
2. Eval class body - use regular Spy
3. Make a constructor function - to make instances

Homework 1: Calculator

1 Mechanics

Assignment (as opposed to tutor hw problems)

This assignment consists of three major parts, each of which can be entered into a problem on the tutor. The first two parts are due **one week** from the start of the assignment; the third part is due **a week after that**. See the tutor for the due dates. Your code will be tested for correctness by the tutor, but also graded for organization and style by a human. We will deduct points for repeated and/or excessively complex code.

Hand in, printed and stapled, in 34-501, before the beginning of *your* nano-quiz on September 30:

- A printed version of your commented code, including the code you submitted to the three tutor problems and the code for the extension you chose.
- A transcript demonstrating it running on the test cases discussed below and any additional test cases.
- An answer to the 'check yourself' problem for the extension you chose.

Do your work in the file `6.01/lab2/designLab/hw1Work.py`, which you can get via `athrun 6.01 update` on Athena, or by downloading a zip file from the calendar page.

You can discuss this problem, at a high level, with other students, but your program must be your own work.

2 Symbolic Calculator

We will construct a simple symbolic calculator that reads, evaluates, and prints arithmetic expressions that contain variables as well as numeric values. It is similar, in structure and operation, to the Python interpreter.

To make the parsing simple, we assume that the expressions are *fully parenthesized*: so, instead of `a = 3 + 4`, we will need to write `(a = (3 + 4))`. In other words, in any expression which involves subexpressions that are not simple elements, those subexpressions will recursively be enclosed within parentheses. Thus any complex expression contains an expression, an operator and another expression, and each of these expressions, if not a number or a variable, is itself contained within parentheses.

The following is a transcript of an interaction with our calculator, where the % character is the prompt. After the prompt, the user types in an expression, the calculator evaluates the expression, possibly changing the environment, and prints out both the value of the expression and the new environment.

```
>>> calc()
% (a = 3)
None
   env = {'a': 3.0}
```

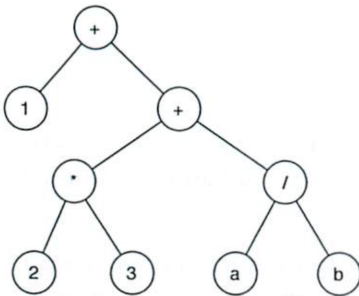
```
% (b = (a + 2))
None
   env = {'a': 3.0, 'b': 5.0}
% b
5.0
   env = {'a': 3.0, 'b': 5.0}
% (c = (a + (b * b)))
None
   env = {'a': 3.0, 'c': 28.0, 'b': 5.0}
```

3 Syntax Trees

The calculator operates in two phases. It

- *Parses* the input string of characters to generate a *syntax tree*; and then
- *Evaluates* the syntax tree to generate a value, if possible, and does any required assignments.

A syntax tree is a data structure that represents the structure of the expression. The nodes at the bottom are called *leaf* nodes and represent actual primitive components (numbers and variables) in the expression. Other nodes are called *internal* nodes. They represent an operation (such as addition or subtraction), and contain instances of *child* nodes that represent the arguments of the operation. The following tree represents the expression `(1 + ((2 * 3) + (a / b)))`. Note the use of parentheses to separate each subexpression:



We can represent syntax trees in Python using instances of the following collection of classes. These definitions are incomplete: it will be your job to fill them in.

```
class BinaryOp:
    def __init__(self, left, right):
        self.left = left
        self.right = right

    def __str__(self):
        return self.opStr + '(' + \
               str(self.left) + ', ' +\
               str(self.right) + ')'
    __repr__ = __str__
class Sum(BinaryOp):
```

```
    opStr = 'Sum'
class Prod(BinaryOp):
    opStr = 'Prod'
class Quot(BinaryOp):
    opStr = 'Quot'
class Diff(BinaryOp):
    opStr = 'Diff'
class Assign(BinaryOp):
    opStr = 'Assign'
class Number:
    def __init__(self, val):
        self.value = val
    def __str__(self):
        return 'Num('+str(self.value)+')'
class Variable:
    def __init__(self, name):
        self.name = name
    def __str__(self):
        return 'Var('+self.name+')'
```

Leaf nodes are represented by instances of `Number` and `Variable`. Internal nodes are represented by instances of `Sum`, `Prod`, `Quot`, and `Diff`. The superclass `BinaryOp` is meant to be a place to put aspects of the binary operators that are the same for each operator, in order to minimize repetition in coding.

We could create a Python representation of `(1 + ((2 * 3) + (a / b)))` with

```
Sum(Number(1.0), Sum(Prod(Number(2.0), Number(3.0)), Quot(Variable('a'), Variable('b'))))
```

Note that we will be converting all numbers to floating point to avoid problems with division later on.

In addition to numerical expressions, the language of our calculator includes assignment 'statements', which we can represent as instances of an assignment class. They differ from the other expressions in that they are not compositional: an assignment statement has a variable on the left of the equality and an expression on the right, and it cannot itself be part of any further expressions. Because assignments share the same initialization and string methods, we have made `Assign` a subclass of `BinaryOp`, but they will require very different handling for evaluation.

4 Parsing

Parsing is the process of taking a string of characters and returning a syntax tree. We'll assume that we parse a single line, which corresponds to a single expression or assignment statement. The processing happens in two phases: tokenization and then parsing a token sequence.

4.1 Tokenization

A *tokenizer* takes a sequence of characters as input and returns a sequence of tokens, which might be words or numbers or special, meaningful characters. For instance, we might break up the string:

```
'((fred + george) / (voldemort + 666))'
```

into the list of tokens (each of which is, itself, a string):

```
['(', '(', 'fred', '+', 'george', ')', '/', '(', 'voldemort', '+', '666', ')', ')']
```

We would like our tokenizer to work the same way, even if the spaces are deleted from the input:

```
'((fred+george)/(voldemort+666))'
```

Our special, single-character tokens will be:

```
seps = ['(', ')', '+', '-', '*', '/', '=']
```

Step 1. Write a procedure `tokenize(inputString)` that takes a string of characters as input and returns a list of tokens as output. The output of `tokenize('(fred + george)')` should be `['(', 'fred', '+', 'george', ')']`. There are other test cases in the `hw1Work.py` file.

Wk.2.4.1	After you have debugged your code in Idle, submit it via this tutor problem.

4.2 Parsing a token sequence

The job of the parser is to take as input a list of tokens, produced by the tokenizer, and to return a syntax tree as output. Parsing Python and other programming languages can be fairly difficult, and parsing natural language is an open research problem. But parsing our simple language is not too hard, because every expression is either:

- a number, or
- a variable name, or
- an expression of the form

  ```
  ( expression op expression )
  ```

  where op is one of `+, -, *, /, =`.

This language can be parsed using a simple *recursive descent* parser. A good way to structure your parser is as follows:

```
def parse(tokens):
    def parseExp(index):
        <your code here>
    (parsedExp, nextIndex) = parseExp(0)
    return parsedExp
```

The function `parseExp` is a recursive function that takes an integer `index` into the `tokens` list. This function returns a pair of values:

- the expression found starting at location `index`. This is an instance of one of the syntax tree classes: `Number`, `Variable`, `Sum`, etc.
- the index beyond where this expression ends. If the expression ends at the token with index 6, then the returned value would be 7.

In the definition of this function we make sure that we call it with the value `index` corresponding to the start of an expression. So, we need to handle only three cases. Let `token` be the token at location `index`. The cases are:

- If `token` represents a number, then make it into a `Number` instance and return that, paired with `index+1` . Note that the value attribute of a `Number` instance should be a Python floating point number.
- If `token` represents a variable name, then make it into a `Variable` instance and return that, paired with `index+1`. Note that the value attribute of a `Variable` instance should be a Python string.
- Otherwise, the sequence of tokens starting at `index` must be of the form:

  ```
  ( expression op expression )
  ```

  Therefore, `token` must be `'('` . We need to:

  – Parse an expression (using `parseExp`), getting a syntax tree that we'll call `leftTree` and the index for the token beyond the end of the expression.

  – The token beyond `leftTree` should be a single-character operator token; call it op.

  – Parse an expression (using `parseExp`) starting beyond op, getting a syntax tree that we'll call `rightTree`.

  – Use op to determine what kind of internal syntax tree instance to make: construct it using `leftTree` and `rightTree` and return it as the result of this procedure, paired with the index of the token beyond the final right paren.

We will give you two useful procedures:

- `numberTok` takes a token as an argument and returns `True` if the token represents a number and `False` otherwise.
- `variableTok` takes a token as an argument and returns `True` if the token represents a variable name and `False` otherwise.

It is also useful to know that if `token` is a string representing a legal Python number, then `float(token)` will convert it into a floating-point number.

We have implemented `__str__` methods for the syntax-tree classes. The expressions print out similarly to the Python expression that you would use to create the syntax tree:

```
>>> parse(tokenize('(1 + ((2 * 3) + (a / b)))'))
Sum(Num(1.0), Sum(Prod(Num(2.0), Num(3.0)), Quot(Var(a), Var(b))))
```

It is **very important** to remember that this is simply the string representation of what is actually an instance of the syntax tree class Sum.

Here are some examples:

```
>>> parse(['888'])
Num(888.0)
>>> print parse(['(', 'fred', '+', 'george', ')'])
Sum(Var(fred), Var(george))
>>> print parse(['(', '(', 'a', '*', 'b', ')', '/', '(', 'cee', '-', 'doh', ')' ,')'])
Quot(Prod(Var(a), Var(b)), Diff(Var(cee), Var(doh)))
>>> print parse(tokenize('((a * b) / (cee - doh))'))
Quot(Prod(Var(a), Var(b)), Diff(Var(cee), Var(doh)))
```

Step 2. Implement `parse` and test it on the examples in the work file, or other strings of tokens you make up, or on the output of the tokenizer. Start by making sure it handles single numbers and variable names correctly, then work up to more complex nested expressions.

Wk.2.4.2	After you have debugged your code in Idle, submit it via this tutor problem. You should include only the code you wrote for `parse`.

5 Evaluation

Once we have an expression represented as a syntax tree, we can evaluate it. We will start by considering the case in which every expression can be evaluated fully to get a number; then we'll extend it to the case where expressions may remain symbolic, if the variables have not yet been defined.

For our calculator, just as for Python, expressions are evaluated with respect to an *environment*. We will represent environments using Python dictionaries (which you should read about in the Python documentation at
`http://docs.python.org/tutorial/datastructures.html#dictionaries`), where the keys are variable names and the values are the values of those variables.

5.1 Eager evaluation

Here are the operation rules of the basic calculator, which tries to completely evaluate every expression it sees. The value of every expression is a number. The evaluation of *expr* in *env* works as follows:

- If *expr* is a `Number`, then return its value.
- If *expr* is a `Variable`, then return the value associated with the *name* of the variable in *env*.
- If *expr* is an arithmetic operation, then return the value resulting from applying the operation to the result of evaluating the left-hand tree and the result of evaluating the right-hand tree.
- If *expr* is an assignment, then evaluate the expression in the right-hand tree and find the name of the variable on the left-hand side of the expression; change the dictionary *env* so that the variable *name* is associated with the value of the expression from the right-hand side. Note that all the values in the environment should be floating point numbers.

Optional: You can make your program more beautiful and compact, using functional programming style, by storing the procedures associated with each operator in the subclass. The Python module `operator` provides definitions of the procedures for the arithmetic operators. Here is an example of using operators.

```
import operator
>>> myOp = operator.add
>>> myOp(3, 4)
7
```

Step 3. Write an `eval` method for each of the expression classes that might be returned by the parser. It should take the environment as an argument and return a number. In real life, we would worry a lot about error checking; for now, just assume that you are only ever given perfect expressions to evaluate.

Test your program incrementally, using expressions like:

```
>>> env = {}
>>> Number(6.0).eval(env)
6.0
>>> env['a'] = 5.0
>>> Variable('a').eval(env)
5.0
>>> Assign(Variable('c'), Number(10.0)).eval(env)
>>> env
{'a': 5.0, 'c': 10.0}
>>> Variable('c').eval(env)
10.0
```

You may find it useful to use the `testEval` procedure to test your code.

5.2 Putting it all together

Now, it's time to put all your pieces together and test your calculator. The work file defines `calc`, a procedure that will prompt the user with a `'%'` character, then read in the next line of input that the user types into a string called `inp`. On the following line, you should make whatever calls are

necessary to tokenize, parse, and evaluate that input. The procedure will print the result of the evaluation, as well as the state of the environment after that evaluation.

For debugging, it can be easier to type in all the expressions at once. The `calcTest` procedure in the work file takes a list of strings as input, and processes them one by one (much the way Idle works when you ask it to 'run' a Python file). You can use `testExprs` in the work file, as input to this procedure for testing. And feel free to make up test cases of your own.

Step 4. Fill in the `calcTest` procedure, so that it calls your code, and make sure it works on the examples. Here is the desired behavior of the lazy evaluator on `testExprs`:

```
>>> calcTest(testExprs)
% (2 + 5)
7.0
   env = {}
% (z = 6)
None
   env = {'z': 6.0}
% z
6.0
   env = {'z': 6.0}
% (w = (z + 1))
None
   env = {'z': 6.0, 'w': 7.0}
% w
7.0
   env = {'z': 6.0, 'w': 7.0}
```

Note that this is due at a later date than the earlier problems.

Wk.2.4.3	After you have debugged your code in Idle, submit it via this tutor problem. You should include the class definitions for `Sum`, `Prod`, `Quot`, `Diff`, `Assign`, `Number`, `Variable` and any other class or procedure definitions that they depend on.

6 Extensions

You should do **one** of these extensions to the calculator. Include your solution and your answer to the corresponing Check Yourself question in your written paper.

6.1 Tokenizing by State Machine

Step 5. Write a state machine class, called `Tokenizer`, whose input on each time step is a single character and whose output on each time step is a either a token (a string of 1 or more characters) or the

empty string, '', if no token is ready. `Tokenizer` should be a subclass of `sm.SM`. Remember that the state of a state machine can be a string.

Here are some examples. Note that there **must be** a space at the end of the string.

```
Tokenizer().transduce('fred ')
['', '', '', '', 'fred']
Tokenizer().transduce('777 ')
['', '', '', '777']
Tokenizer().transduce('777 hi 33 ')
['', '', '', '777', '', '', 'hi', '', '', '33']
Tokenizer().transduce('**-)( ')
['', '*', '*', '-', ')', '(']
Tokenizer().transduce('(hi*ho) ')
['', '(', '', 'hi', '*', '', 'ho', ')']
Tokenizer().transduce('(fred + george) ')
['', '(', '', '', '', 'fred', '', '+', '', '', '', '', '', '', 'george', ')']
```

Step 6. Now, write a procedure `tokenize(inputString)` that takes a string of characters as input and returns a list of tokens as output. The output of `tokenize('(fred + george) ')` should be `['(', 'fred', '+', 'george', ')']`. To do this, your procedure should:

- Make an instance of your `Tokenizer` state machine.
- Call its transduce method on the input string, **with a space character appended to the end of it.** An important thing to understand about Python is that almost any construct that iterates over lists or tuples will also iterate over strings. So, even though `transduce` was designed to operate on lists, it also operates on strings: if we feed a string into the `transduce` method of a state machine, it will call the `step` method with each individual character in the string.
- Remove the empty strings to return a list of good tokens.

Check Yourself 1.

- Explain precisely why you need a space character appended to the end of the input to the `Tokenizer` input.
- Compare and contrast your two tokenizer implementations.

Include your answer in your write-up.

6.2 Lazy partial evaluation

To make the calculator flexible, we will allow you to define an expression, like `(d = (b + c))`, even before `b` and `c` are defined. Later, if `b` and `c` are defined to have numeric values, then evaluating `d` will result in a number.

Step 7. Change your `eval` methods, so that they are lazy, and can handle symbolic expressions for which we do not have values of all the symbols.

- If the expression is a `Variable`, test to see if it is in the dictionary. If it is in the dictionary, return the result or evaluate the value for the variable in the environment, otherwise, simply return the variable. (The Python expression `'a' in d` returns `True` if the string `'a'` is a key in dictionary `d`).
- When you evaluate an assignment *do not* evaluate the right hand side; simply assign the value of the variable in the environment to be the unevaluated syntax tree. Notice this means that the values in the environment will always be syntax trees and not numbers as in eager evaluation. This is called *lazy evaluation*, because we don't evaluate expressions until we need their values.
- If your expression is an arithmetic operation, evaluate both the left and right subtrees. If they are both actual numbers, then return a new number computed using the appropriate operator, as before. If not, then make a new instance of the operator class, whose left and right children are the results of having evaluated the left and right children of the original expression (because the evaluation process may have simplified one or the other of the arguments) and return the operator node. This is called *partial evaluation* because we only evaluate the expression to the degree allowed by the variable bindings.
- When you look a variable up in the environment, evaluate the result before returning it, because it might be a symbolic expression.

If you want to check whether something is an actual number (float or int), you can use the `isNum` procedure defined in the work file.

Note that, if you are writing your `eval` method in the `BinaryOp` class, you will need to be able to make a new instance of the subclass that `self` belongs to (e.g. `Sum`). Python provides a `__class__` method for all objects, so that `self.__class__` can be called to create a new instance of that same class.

Here are some ideas for testing eval by itself:

```
>>> env = {}
>>> Assign(Variable('a'), Sum(Variable('b'), Variable('c'))).eval(env)
>>> Variable('a').eval(env)
Sum(Var(b), Var(c)]
>>> env['b'] = Number(2.0)
>>> Variable('a').eval(env)
Sum(Num(2.0), Var(c))
>>> env['c'] = Number(4.0)
>>> Variable('a').eval(env)
6.0
```

```
>>> calcTest(lazyTestExprs)
% (a = (b + c))
None
   env = {'a': Sum(Var(b), Var(c))}
```

```
% (b = ((d * e) / 2))
None
   env = {'a': Sum(Var(b), Var(c)), 'b': Quot(Prod(Var(d), Var(e)), Num(2.0))}
% a
Sum(Quot(Prod(Var(d), Var(e)), 2.0), Var(c))
   env = {'a': Sum(Var(b), Var(c)), 'b': Quot(Prod(Var(d), Var(e)), Num(2.0))}
% (d = 6)
None
   env = {'a': Sum(Var(b), Var(c)), 'b': Quot(Prod(Var(d), Var(e)), Num(2.0)), 'd': Num(6.0)}
% (e = 5)
None
   env = {'a': Sum(Var(b), Var(c)), 'b': Quot(Prod(Var(d), Var(e)), Num(2.0)), 'e': Num(5.0), 'd': Num(6.0)}
% a
Sum(15.0, Var(c))
   env = {'a': Sum(Var(b), Var(c)), 'b': Quot(Prod(Var(d), Var(e)), Num(2.0)), 'e': Num(5.0), 'd': Num(6.0)}
% (c = 9)
None
   env = {'a': Sum(Var(b), Var(c)), 'c': Num(9.0), 'b': Quot(Prod(Var(d), Var(e)), Num(2.0)), 'e': Num(5.0), 'd':
Num(6.0)}
% a
24.0
   env = {'a': Sum(Var(b), Var(c)), 'c': Num(9.0), 'b': Quot(Prod(Var(d), Var(e)), Num(2.0)), 'e': Num(5.0), 'd':
Num(6.0)}
% (d = 2)
None
   env = {'a': Sum(Var(b), Var(c)), 'c': Num(9.0), 'b': Quot(Prod(Var(d), Var(e)), Num(2.0)), 'e': Num(5.0), 'd':
Num(2.0)}
% a
14.0
   env = {'a': Sum(Var(b), Var(c)), 'c': Num(9.0), 'b': Quot(Prod(Var(d), Var(e)), Num(2.0)), 'e': Num(5.0), 'd':
Num(2.0)}

>>> calcTest(partialTestExprs)
% (z = (y + w))
None
   env = {'z': Sum(Var(y), Var(w))}
% z
Sum(Var(y), Var(w))
   env = {'z': Sum(Var(y), Var(w))}
% (y = 2)
None
   env = {'y': Num(2.0), 'z': Sum(Var(y), Var(w))}
% z
Sum(2.0, Var(w))
   env = {'y': Num(2.0), 'z': Sum(Var(y), Var(w))}
% (w = 4)
None
   env = {'y': Num(2.0), 'z': Sum(Var(y), Var(w)), 'w': Num(4.0)}
% z
6.0
   env = {'y': Num(2.0), 'z': Sum(Var(y), Var(w)), 'w': Num(4.0)}
% (w = 100)
None
   env = {'y': Num(2.0), 'z': Sum(Var(y), Var(w)), 'w': Num(100.0)}
% z
102.0
   env = {'y': Num(2.0), 'z': Sum(Var(y), Var(w)), 'w': Num(100.0)}
```

Check Yourself 2. What happens if you evaluate

```
(a = 5)
(a = (a + 1))
a
```

- Using eager evaluation?
- Using lazy evaluation?

Include your answer in your write-up.

HW Assign 1

9/18

? formal HW assignment

(Even more 6.01 HW!)
- (so many different types of HW)
- need to read parser chap I think

9/19

- read chap
- So symbolic calc is just like is described
  - except seems a little simpler
  - uses our state machine ?!
  - state is a list
    - ah! - good idea rather than new states!

- ? just copy code from course notes?

2 phases
- parse syntax tree
- evaluates

represent syntax tree w/ collection of classes
- definitions incomplete

Wow long representation of class creation
- over kill?

Well we will need to parse it
- 2 parts:

<u>tokenization</u> - <u>input</u> string
- <u>output</u> series of tokenizers

②

(neatness counts in this assignment!)

- So tokenizer

((fred + george) / (voldemort + 666))

[(, (, fred, +, george,), /, (, voldemort, +, 666,),)]

- so ignore spaces
- single char tokens : [(,), +, -, *, /, =]
- test cases in file

Ok lets try!

- first go through adding chars to temp
- till reach single char token
- append temp to ans
- append single char token
- repeat

Oh from yesterday "import lib601.sm as sm"
- grrrrr

---

after 1st try: fairly good!
- just need to ignore spaces
- and sometimes temp is empty
  - don't append that

③

Done!

- took only ~~8~~ 10 min!
- but failed one test case where space is a seperator
  - this is complex since in every other case we do not care about spaces
    - only if it is the only seperator → before and after are letters
    - can look forward + backward
    - or store space if nothing else comes up
      - which is cleaner
    - both are some sort of state machine
- picked 1st
  - if first or last char - no as well
- and don't append if seperator is a space
- done → checks off on web
  25 min total

Now Parse Part 2

- ~~takes 2 the~~ input list of tokens ~~and~~
- output syntax tree

④

- hard to parse python and have not yet parsed natural lang
- but our syntax easy to parse
- <u>every expression</u>
  - #
  - variable name
  - expression of the form (expression op expression)

    op: +, -, *, /, =

- can parse using simple recursive decent parser
- they provide a basic ~~scratch pad~~ structure
- returns expression at starting "index" and ending "index"
- put in a call for it to start at begining

<u>3 cases</u> (token is token at starting indx)

- token ~~is~~ #, make a # instance (floating pt)
  return it
  return indx +1
- token = variable name → make Variable ~~[illegible]~~ instance
  return it
  return index +1

- Otherwise in the (expression op expression) form
  - first token = (
  - parse the expression inside
    - recursively using Parse Exp()
    - get syntax tree → left Tree
  - —— next token = op (single char)
  - parse right expression
    - "op" ——→ ")"
    - well use parseExp()
  - use op to decide what kind of internal syntax tree to make
    - construct w/ left tree, right Tree
    - return it

2 procedures provided
- numberTok → true if a number
- variableTok → true if a variable name

__str__ method implemented
- with that complex thing I complained about earlier
- all the instances

(feeling pressured from other hw).

(6) 9/20

Work on Parser

- implement # first
  - oh already have a is number!
- Variable
  - easy feo
- but what about the (— op —)
  - recursive
  - so eaiser
- also how does parsed Ex accumulate?
  - seems to be over ritten
    - deal w/ later

So left Tree =

remember we are dealing w/ tokens - not strings!

?how return a syntax tree
- is that a type?

else if = elif ≡ stupid

how is value being set as next index

now how to return a syntax tree?
- class = sum
- oh class determines what it is
- and they are subclasses of binary op
- return Sum(leftTree, rightTree), index + 4

(7) Is it adding the return index to the result?

anyway stuff is not getting overwritten
- So I guess it is all recursive

- and some of parenthees stuff screwed up
- index return # prob bad

- and assignment?
- return type assign

- yeah for some reason next int being added to output
fred is also being split by tokenizer
- hmmm!
fr, e, d
- fixed, was at wrong indent level
- too many comments :)

- ok since left tree is not taking index return
and right tree + opp should base off that!
- fixed # in ans
- and one more test did not error!
- can do one (ccc - bbb) but not multiple
- just playing w/ return #
- that did it → 1 not 2
- all tests in IDLE passed → try in tutor

(8) tutor

- should always return floats
  - change 'at number
- ~~Ad~~ m' and I have a Quot where should be Prod ~~product~~ and Diff where should be quot

good now!

Ok 3rd part due in another week

- although now that own it, want to tackle
- but wait till after lecture + other hw
- when I get started w/ 6.01 don't want to stop!

6.01: Introduction to EECS I

Capturing Common Patterns: State Machines

Week 3 *September 21, 2010*

Outline

- State machines : a new PCAP system
  - Properties of state machines
  - Examples of state machines
  - Implementing in Python
  - State machines as a PCAP

Reading: 3.4.6, 3.5.4, 3.6, 4.1, 4.4, A.1, A.2

Primitives
Abstraction
Patterns
Combonators

Remembering

- Programs thus far:
  - Purely functional – output depends only on input
  - Object oriented – methods only really depend on state of attributes and input

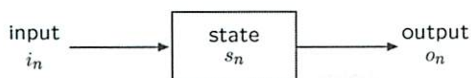
What if you want a procedure that can remember some results of its computations from one call to the next – whose output depends on entire history of inputs?

State machines

State machines are a way of organizing and managing the memory of a computation over time; way of modeling systems whose output depends on entire history of inputs

- If o_t is output at t, and i_t is input, we want a mapping such that $(i_1, \ldots, i_t) \mapsto o_t$
- Compare this with functional programming, where $i_t \mapsto o_t$
- Too complicated, so look for set of states, where each state captures essential properties of history of inputs, and determines the next state and output
- Need to find set (finite?) of states

State machines



On the nth **step**, the system time sequence

- gets **input** i_n
- generates **output** o_n and
- moves to a new **state** s_{n+1}

Output and next state depend on input and current state (which captures essence of history of inputs)

State machines: example usage

- User interfaces (if mouse click, and previous input = X, do Y)
- Modeling conversations, natural dialogue systems (to what does "it" or "this" refer in a conversation?)
- State of a space craft (what is level of fuel, oxygen, given that certain valves are currently open?)
- Video games, e.g Quake, WarCraft (actions of agents determined by sequence of inputs of users)

↳ early video games

State machines: generic usage

- Synthetically – specify program (for robots) for a system embedded in world – inputs are sensor data, outputs are control commands
- Analytically – analyze properties of coupled system (control system and environment it is controlling)
- Predictively – plan trajectories through space of external world to reach desired goal state, choose between alternatives

Will use state machines for all three purposes

Turnstile

Given an embedded system, can supply it with a stream (infinite sequence) of inputs, it will "transduce" stream to give stream of outputs

Want to model such a system, here is an example

Inputs = {coin, turn, none}

Outputs = {enter, pay}

States = {locked, unlocked}

$$\text{nextState}(s, i) = \begin{cases} \text{unlocked} & \text{if } i = \text{coin} \\ \text{locked} & \text{if } i = \text{turn} \\ s & \text{otherwise} \end{cases}$$

$$\text{output}(s, i) = \begin{cases} \text{enter} & \text{if nextState}(s, i) = \text{unlocked} \\ \text{pay} & \text{otherwise} \end{cases}$$

s_0 = locked

↑ every state machine

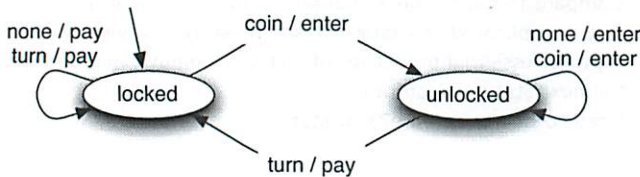
↑ not i = coin

PCAP!

does a new state every millisecond

State-transition Diagram

- Nodes represent states ~ circles
- Unlabeled arrow goes to start state
- Arcs represent transitions: label is input / output



Turn Table

time	0	1	2	3	4	5	6
state	L	L	U	U	L	L	U
in	None	coin	None	turn	turn	coin	coin
out	pay	enter	enter	pay	pay	enter	enter

Transition tables

We will use state transition tables to examine the evolution of a state machine.

For each column in the table, given the current input value and state we can use the output function to determine the output in that column; and we use the next-state function applied to that input and state value to determine the state in the next column.

In general we have:

time	0	1	2	...
input	i_0	i_1	i_2	...
state	s_0	s_1	s_2	...
output	o_1	o_2	o_3	...

State Machines in Python

SM Class:

Methods that are shared among all state machines

- `start(self)` – initialize state
- `step(self, input)` – get next state, compute and return output
- `transduce(self, inputs)` – initialize, then process sequence of inputs, return sequence of outputs

Turnstile Class:

Attributes that are shared among all turnstiles

- `startState`
- `getNextValues(self, state, inp) - transition diagram`

Turnstile Instance:

Attributes of this particular turnstile, including current state

- `state`

* don't confuse output, new state! ~out is display screen

State can be any data structure

Turnstile Class

```
class Turnstile(SM):
    startState = 'locked'

    def getNextValues(self, state, inp):
        if inp == 'coin':
            return ('unlocked', 'enter')
        elif inp == 'turn':
            return ('locked', 'pay')
        elif state == 'locked':
            return ('locked', 'pay')
        else:
            return ('unlocked', 'enter')
```

represents the transition diagram

Turn, Turn, Turn

```
testInput = [None, 'coin', None, 'turn', 'turn', 'coin', 'coin']

ts = Turnstile()

ts.transduce(testInput)
Start state: locked
In: None   Out: pay    Next State: locked
In: coin   Out: enter  Next State: unlocked
In: None   Out: enter  Next State: unlocked
In: turn   Out: pay    Next State: locked
In: turn   Out: pay    Next State: locked
In: coin   Out: enter  Next State: unlocked
In: coin   Out: enter  Next State: unlocked
['pay', 'enter', 'enter', 'pay', 'pay', 'enter', 'enter']
```

SM Abstraction

- Have built our example on abstraction of state machine
- Can think in terms of states, transitions, without worrying about details
- But need underlying substrate – will provide one, which makes some assumptions!

SM Class

```
class SM:
    def start(self):
        self.state = self.startState

    def step(self, inp):
        (s, o) = self.getNextValues(self.state, inp)
        self.state = s
        return o

    def transduce(self, inputs):
        self.start()
        return [self.step(inp) for inp in inputs]
```

Note that `getNextValues` should not change the state.
State is managed by `start` and `step`.

Accumulator

Here is a very simple state machine – just adds up inputs

```
class Accumulator(SM:
    startState = 0

    def getNextValues(self, state, inp):
        return (state + inp, state + inp)
```

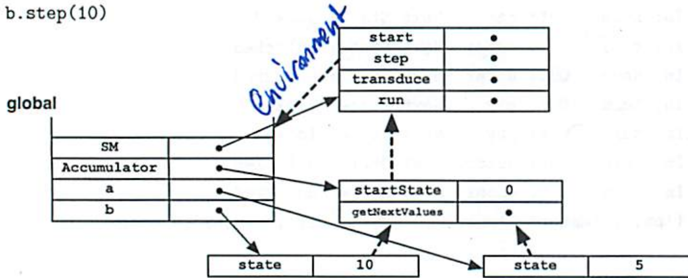
lecture same as course notes

Check Yourself

```
>>> a = Accumulator()
>>> a.start()
>>> a.step(7)
>>> a.step(-2)
>>> a.state
???    5
>>> b = Accumulator()
>>> b.start()
>>> b.step(10)
>>> b.state
???    10
```

Classes and Instances for Accumulator

```
a = Accumulator()
a.start()
a.step(7)
a.step(-2)
b = Accumulator()
b.start()
b.step(10)
```



State machines as acceptors

Another standard use of a FSM is as an acceptor

- decides whether input is legal
- only outputs true or false

Examples include natural language parsers, user interfaces

FSM "defines" the language – determines all legal "words" that are accepted

– handy in Natural Lang Processing + other things

A language acceptor

same as book

A language acceptor

$$S = \{0, 1, 2, 3\}$$

$$I = \{a, b, c\}$$

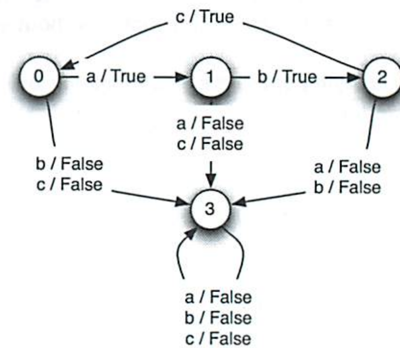
$$O = \{\mathit{true}, \mathit{false}\}$$

$$n(s,i) = \begin{cases} 1 & \text{if } s = 0, i = a \\ 2 & \text{if } s = 1, i = b \\ 0 & \text{if } s = 2, i = c \\ 3 & \text{otherwise} \end{cases}$$

$$o(s,i) = \begin{cases} \mathit{false} & \text{if } n(s,i) = 3 \\ \mathit{true} & \text{otherwise} \end{cases}$$

$$s_0 = 0$$

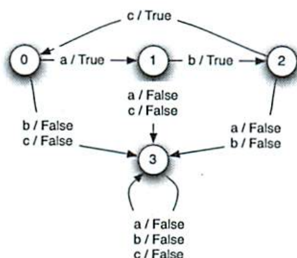
State Transition Diagram



State transition diagram for language acceptor

Simulation

time	0	1	2	3	4	5	6	7
input	'a'	'b'	'c'	'a'	'c'	'a'	'b'	
state	0	1	2	0	1	3	3	3
output	True	True	True	True	False	False	False	



Python implementation

```
class ABC(SM):
    startState = 0
    def getNextValues(self, state, inp):
        if state == 0 and inp == 'a':
            return (1, True)
        elif state == 1 and inp == 'b':
            return (2, True)
        elif state == 2 and inp == 'c':
            return (0, True)
        else:
            return (3, False)
```

Simulation

```
>>> abc.transduce(['a', 'b', 'c', 'a', 'c', 'a', 'b'], verbose =
True)
Start state: 0
In: a Out: True Next State: 1
In: b Out: True Next State: 2
In: c Out: True Next State: 0
In: a Out: True Next State: 1
In: c Out: False Next State: 3
In: a Out: False Next State: 3
In: b Out: False Next State: 3
[True, True, True, True, False, False, False]
```

I'm thirsty

Let's model a Pepsi machine. For simplicity, assume a Pepsi only costs 10 cents.

$$\text{Inputs} = \{5, 10, \text{GimmePepsi}, \text{GimmeMyMoney}\}$$

$$\text{Outputs} = \{\text{Kerplunk}, \text{Return5}, \text{Return10}, \text{None}\}$$

$$\text{States} = \{\text{NoCoins}, \text{Have5}, \text{Have10}\}$$

$$s_0 = \text{NoCoins}$$

issue Pepsi

See sheet

PCAP System for State Machines

- **Primitives**: Basic state machines
- **Combinators**: Ways of connecting them to make new state machines
- **Abstraction**: Naming combined machines

This week's design lab and exercises will focus on state-machine combinators and abstraction.

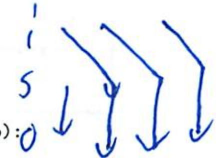
We will also use these ideas in future weeks to model robot control and sensing systems

State Machine Primitives

Accumulator

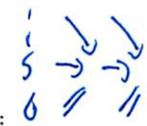
Delay

```
class Delay(SM):
    def __init__(self, v0):
        self.startState = v0
    def getNextValues(self, state, inp):
        return (inp, state)
```



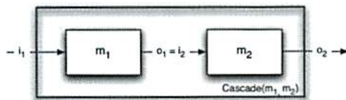
Increment

```
class Increment(SM):
    def __init__(self, incr):
        self.startState = incr
    def getNextValues(self, state, inp):
        return (safeAdd(inp, state), safeAdd(inp, state))
```

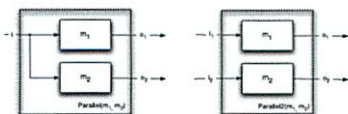


Dataflow Combinators

Cascade:

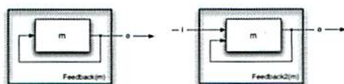


Parallel:



alternates for choices

Feedback:



Example cascade

```
m1 = Delay(99)
m2 = Delay(22)
C = Cascade(m1, m2)
```

time	0	1	2	3	4	5	6
m1 input	3	8	2	4	6	5	
m1 state	99	3	8	2	4	6	5
m1 output	99	3	8	2	4	6	
m2 input	99	3	8	2	4	6	
m2 state	22	99	3	8	2	4	6
m2 output	22	99	3	8	2	4	

Example Feedback

We would like a machine that counts:

$$\text{Inputs} = \text{Numbers}$$
$$\text{Outputs} = \text{Numbers}$$
$$\text{States} = \text{Numbers}$$
$$\text{nextState}(s, i) = i + 1$$
$$\text{output}(s, i) = \text{nextState}(s, i)$$
$$s_0 = 0$$

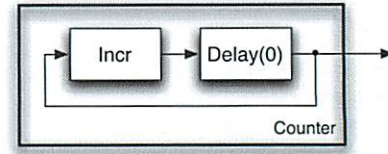
We have

$$o[t] = i[t] + 1$$

Suppose we connect input to output: $i[t] = o[t]$

We **cannot** satisfy these equations! A FSM cannot have a direct dependence of its output on its input!!

Example Feedback



Let's delay our output, to line it up with the next input:

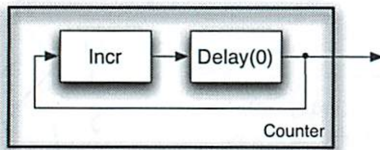
$$o_i[t] = i_i[t] + 1$$
$$o_d[t] = i_d[t-1]$$
$$i_i[t] = o_d[t]$$
$$i_d[t] = o_i[t]$$

delay is important

must sync. inputs!

First two equations describe machines, second two describe wires

Example Feedback



Now we have

$$o_i[t] = i_i[t] + 1$$
$$= o_d[t] + 1$$
$$= i_d[t-1] + 1$$
$$= o_i[t-1] + 1$$

Now output will be one greater at each time step!

Sequence Combinators

- *Terminating* state machines have a `done` method, that returns `True` when the state machine has terminated.
- Combine terminating SMs with:
  - `sm.Sequence`: takes a list of state machines, runs each until termination and starts the next one
  - `sm.Repeat`: takes a state machine and a count, runs the state machine until termination the specified number of times, then terminates

glue together simple machines

This Week

Software lab: Practice with simple state machines

Design lab: Controlling robots with state machines

To get help:

- Email `6.01-help@mit.edu`
- Go to lab hours (see course web page for times)
- Remember to check your due dates/times on the tutor

6.01 Lecture Extra

9/21

Function

$i_t \to O_t$

$O_t = f(i_t)$

If n values for input, n possible outputs
- 1 to 1 match

State

$(i_1, i_2, i_3 \ldots i_t) \to O_t$

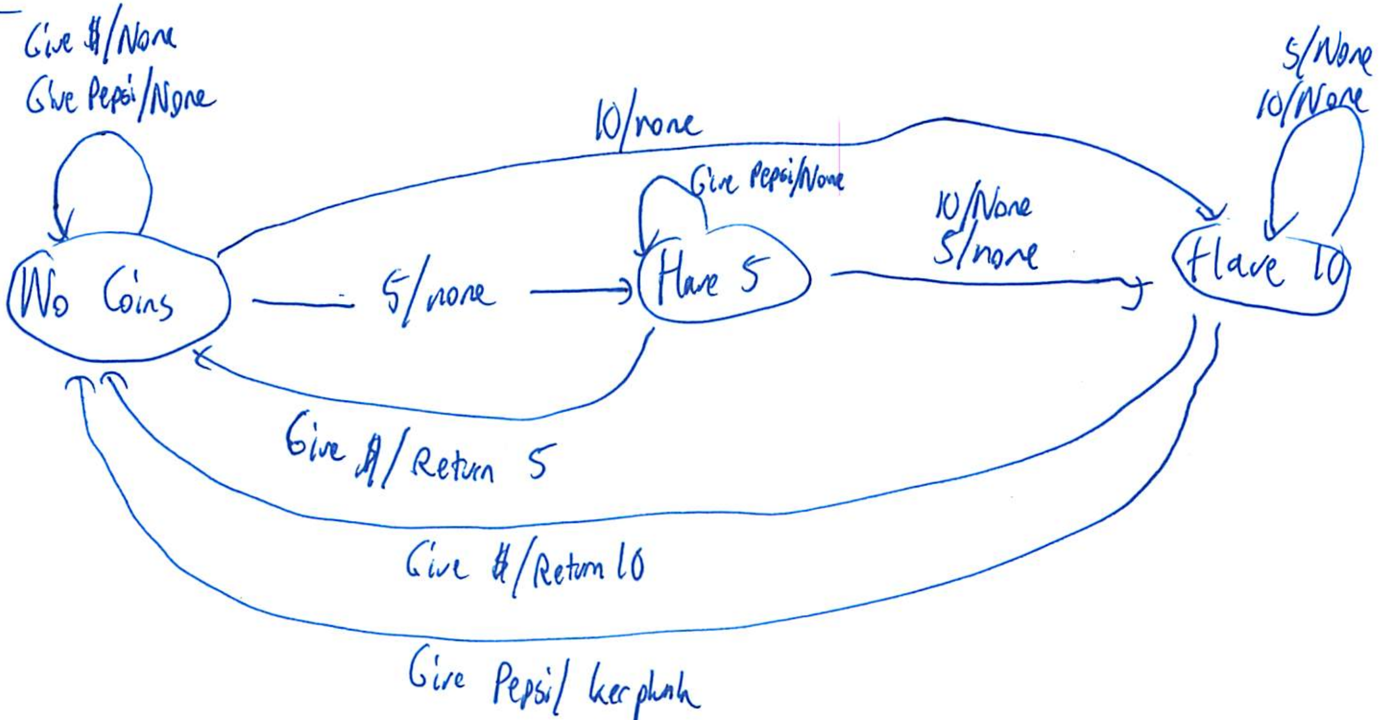
procedure depends on increasing # of inputs

If n values (input), n^t possible outputs

We want some states such that

$O_t(s_t, i_t)$ $\quad S_t(i_t, S_{t-1})$

Pepsi



1. Start w/ expected behavior
2. Then have every input at every output

②

Made a design decision to eat the charge

If wanted it to be fair

- don't have state for each combo of 5+10
- add extra state variable → balance
    - check when give money back
- or put extra $ in + return it

When system gets complex + are their pieces you can break off

Software Lab 3: State Machines

1 Setup

If you have already installed Python on your own laptop, you can use it. If you haven't installed Python yet, and would like help, please bring your laptop to evening office hours. Otherwise, please use a lab laptop or desktop.

- **Using a lab laptop or desktop machine**
  - Log in using your Athena user name and password.
  - Click once on the Terminal icon (usually on the bottom left of the screen.) In the terminal window, type `athrun 6.01 update`. It will create a `Desktop/6.01/lab2/swLab` directory and put a file in it.
- **Using your own laptop**
  - Go to the course web page: `http://mit.edu/6.01`
  - Go to the calendar tab, and download the zip file for software lab 2. Unzip it.
  - When we mention finding a file in `Desktop/6.01/...`, look for it in the folder you got by unzipping the archive.

2 Exercises

Wk.3.1.all	Do these problems on the tutor. You can test your code in idle by implementing your programs in the file `lab3/swLab/sl3Work.py`, which imports the `sm` module and has some useful test cases.

6.01 Software Lab

9/21

Do tutor problems

State Machine

Old State	input 0	1	2
0	1	3	0
1	2	0	0
2	3	1	0
3	0	2	0

? what are rules ?

output = same as new state

inital state = 0

- Count foward + backward mod 4 + reset input
- Counts how many 0 + 1 inputted + has reset input

* neither seem right ? what is pattern?

2 = reset

input 0 → output 1 new state = 1

state, input

1, 0 → 2

2, 2 → 0

0, 0 → 1

1, 0 → 2

②

~~2,0~~ 2,0 → 3

3, 1 → 2

2, 1 → 1

1, 1 → 0

0, ? → 3   1

3, ? → 2   1

2, ? → 0   2

0, ? → 2   0

1, ? → 2   0

2, ? → 3   0

oh wow all right! → forwards + backwards mod 4 (still don't see it)

---

2. Turnstile

- same as lecture
  - locked / coin
    - or just look up lecture notes
    - slightly different
  - new state, output
  - just look at diagram!   Done ✓

③ Double Delay State Machine

- delays input 2 steps
- need to give it first 2 inputs
- data structure should be a list
- need to reinstall to 2.6.6 since 2.7 does not work
  - done

- or have it be a dictionary, not list
- or can be list, just must be careful
- ok got it working
  return [state [1], inp], state [0]
- just need some more practice w/ this

Part C) comment machine → do later

9/28 night

in = strings

out = { input character or none }

= comments ''' """ to make multi line string

'''
blah → multi line ~~comment~~ string
'''

(4) So go through string, outputting None
till get to # ~~tha~~
then output each letter till newline '\n'

- seems kinda easy
- state [index, if in comment]
  - index ↑ int, if in comment ↑ boolean
- oh need py 2.6 on desktop
- ok - now just # and \n outputted
  - output when True
  - and don't output \n
- and output text 'None' not value None
  - oh no wait we want value None
  - did not know that existed
- That was easy
- first word option - easy, variation on theme
- think I got implementing SM now
  - it was just that their code was not working
- done for the ~~week~~ day

Design Lab 3: Controlling Robots

1 Materials

This lab should be done with a partner. Each partnership should have:

- A lab laptop.
  - Log in using one partner's Athena user name and password.
  - Click once on the Terminal icon (usually on the bottom left of the screen.)
  - If you have not already done this, in the terminal window, type `athrun 6.01 setup`. Otherwise, do `athrun 6.01 update` to get the latest batch of files.
- A robot and a serial cable.
  - The serial cable is a long beige or gray cable. Most of the robots already have one attached.
  - *Warning: if your robot starts to go too fast or get away from you,* ***pick it up!!***
- A white foam-core board with bubble-wrap on one side.

Be sure to mail all of your code and data to your partner. You will both need to bring it with you to your first interview.

2 Simple Brains

1. **Run a brain in the simulator.**
   - **a.** In the Terminal window, type `soar &`.
   - **b.** Click soar's **Simulator** button, navigate to `Desktop/6.01/lab3/designLab/worlds` and choose `tutorial.py`, click Open. This loads a specific virtual world into our robot simulator.
   - **c.** Click soar's **Brain** button, navigate to `Desktop/6.01/lab3/designLab/smBrain.py`, and click Open. This loads a specific state machine definition into the robot simulator. That state machine describes the actions that the robot will take in response to sensed information about the virtual world surrounding it.
   - **d.** Click soar's **Start** button, and let the robot run for a little while.

e. Click soar's **Stop** button.

f. Notice the graph that was produced; it shows a 'slime trail' of the path that the robot followed while the brain was running. You can just close the window. (If you don't want the brain to produce a slime trail, you can set the `drawSlimeTrail` argument to the `RobotGraphics` constructor in the `smBrain.py` file to be `False`).

2. Modify the brain and run it.

a. In the Terminal window, type `idle &` to open up an Idle environment.

b. Click Idle's **File** menu, select **Open...**, navigate to `Desktop/6.01/lab3/designLab/smBrain.py`, and click **Open**.

c. The state machine that controls the robot's actions is defined by the `MySMClass` definition. Think of this state machine as taking sensory data as input, and returning as output instructions to the robot on how to behave (we'll see more about this kind of state machine modeling of a robot and world next lecture). The `io.Action` object returned as the output by the `getNextValues` method of the `MySmClass` tells the robot how to change its behavior, and has two attributes that are important to us:

⋆ `fvel` specifies the forward velocity of the robot (in meters per second)

⋆ `rvel` specifies the rotational velocity of the robot (in radians per second), where positive rotation is counterclockwise

d. Find the place where the velocities are set in the brain, and then modify it so that it makes the simulated robot rotate in place.

e. Save the file.

f. Go back to the soar window and click the **Reload Brain** button

g. Run the brain by clicking the **Start** and then the **Stop** buttons.

3. Run it on the robot

a. Connect the robot to your laptop, making sure the cable is tied around the handle in the back of the robot.

b. Power on the robot, with a switch on the side panel.

c. Click soar's Pioneer button, to select the robot.

d. One partner should be in charge of keeping the robot safe. Keep the cable from getting tangled in the robot's wheels. **If the robot starts to get away from you, pick it up, then, turn it off using the switch on the robot.**

e. Click soar's Start button. You should be able to hear the sonar sensors making a ticking noise.

3 Sonars

The `inp` argument to the `getNextValues` method of `MySMClass` is an instance of the `soar.io.SensorInput` class, which we have imported as `io.SensorInput`. It has two attributes, `odometry` and `sonars`. For this lab, we will just use the `sonars` attribute, which contains a list of 8 numbers representing readings from the robot's 8 sonar sensors, which give a distance reading in meters. The first reading in the list (index 0) is from the leftmost (from the robot's perspective) sensor; the reading from the rightmost sensor is the last one (index 7).

Now we will investigate the behavior of the sonar sensors. **Don't spend more than 10 or 15 minutes experimenting with the sonars. When you're done, ask a staff member for a checkoff.**

- Modify the brain so that it sets both velocities to 0, and uncomment the line `print inp.sonars[3]`. Reload the brain and run it. It will print the value of `inp.sonars[3]`, which is the reading from one of the forward-facing sonar sensors.
- From how far away can you get reliable distance readings? What happens when the closest thing is farther away than that?
- What happens with things very close to the sensor?
- Does changing the angle between the sonar transducer and the surface that it is pointed toward affect the readings? Does this behavior depend on the material of the surface? Try bubble wrap versus smooth foam core.
- Now, set the `sonarMonitor` argument to the `RobotGraphics` constructor to be `True`.

  Reload the brain and run it. This will bring up a window that shows all the sonar readings graphically. The length of the beam corresponds to the reading; red beams correspond to "no valid measurement". Test that all your sonars are working by blocking each one in turn. If you notice a problem with any of the sensors, talk to the staff.

> *Checkoff 1.* Explain to a staff member the results of your experiments with the sonars. Demonstrate that you know your partner's name and email address.

Make the robot move forward to approximately 0.5 meters of an obstacle in front of it and keep it at that distance, even if the obstacle moves back and forth. Do this by editing the `getNextValues` method of `MySMClass`; there is no need to change any other part of the brain. Don't set the forward velocity higher than 0.3. Debug it in simulation, by clicking soar's **Simulator** button and choosing `tutorial.py`. Once it seems good, run it on a real robot, by choosing soar's **Pioneer** button.

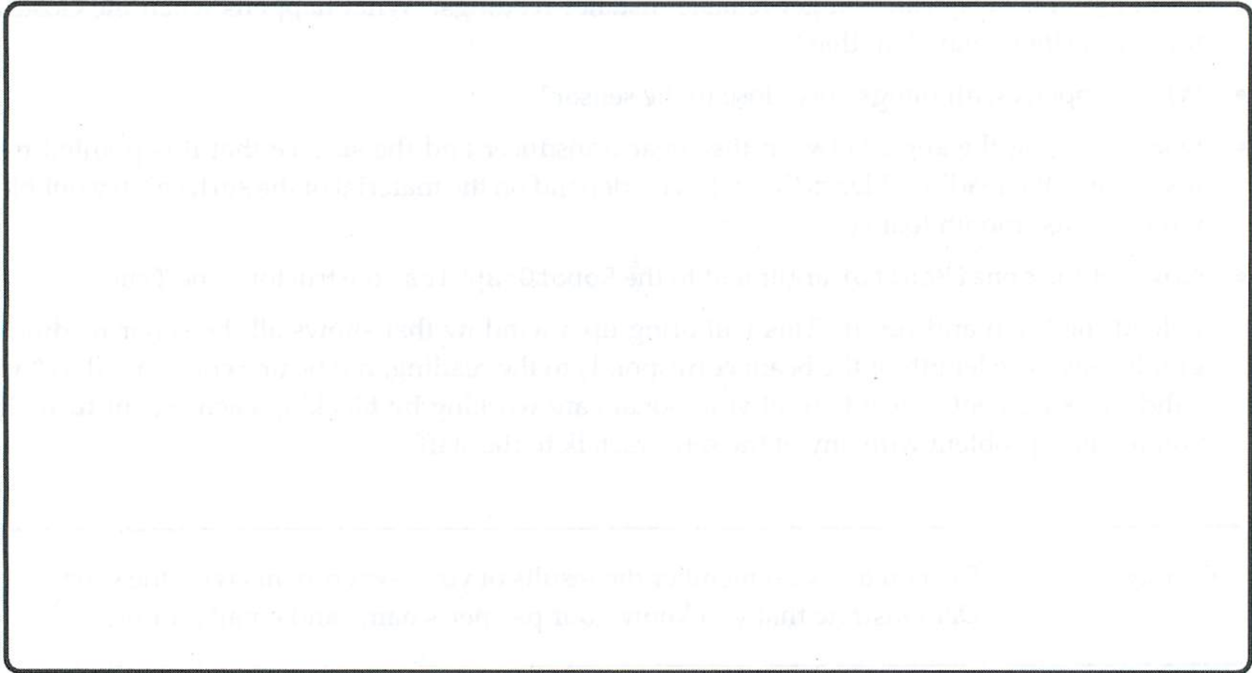
> *Checkoff 2.* Demonstrate your distance-keeping brain on a real robot to a staff member.

4 Following Boundaries

Our goal now is to build a state machine that controls the robot to do a more complicated task:

1. When there is nothing nearby, it should move straight forward.
2. As soon as it reaches an obstacle in front, it should follow the boundary of the obstacle, keeping the right side of the robot between 0.3 and 0.5 meters from the obstacle.

Draw a state-transition diagram that describes each distinct situation (state) during wall-following and what the desired output (action) and next state should be in response to the possible inputs (sonar readings) in that state. Start by considering the case of the robot moving straight ahead through empty space and then think about the input conditions that you encounter and the new states that result. Think carefully about what to do at both inside and outside corners. Remember that the robots rotate about their center points. Try to keep the number of states to a minimum.



Checkoff 3.	Show your state-transition diagram to a staff mamber. Make clear what the conditions on state transitions are, and what actions are associated with each state.

Copy your current `smBrain.py` file to `boundaryBrain.py` (you can do this with **Save As** in idle), and modify it to implement the state machine defined by your diagram. Make sure that you define a `startState` attribute and a `getNextValues` method.

Try hard to keep your solution simple and general. Use good software practice: do not repeat code, use helper procedures with mnemonic names, try to use few arbitrary constants and give the ones you do use descriptive names.

To debug, add print statements that show the relevant inputs, the current state, the next state, and the output action.

Record a slime trail of the simulated robot following a sequence of walls; make sure that it can handle outside and inside corners. Going around very sharp corners or hairpin turns, such as the L in `tutorial.py`, is not required, but is extra cool.

Checkoff 4.	Demonstrate your boundary follower to a staff member. Explain why it behaves the way it does. **Mail your code to both partners.**

9/23

works w/ 2

how count 2 adjecent

- did not get it in time :(

- always true

- big fail - spent another 15 min trying - got a much better still wrong

---

Des Lab 3

partner tgarv@mit

Max listed distance = 5

↳ error

1.5 is about max we could get (meters)

- constrained by tables

- bubble / non bubble works about the same

.17 is smallest

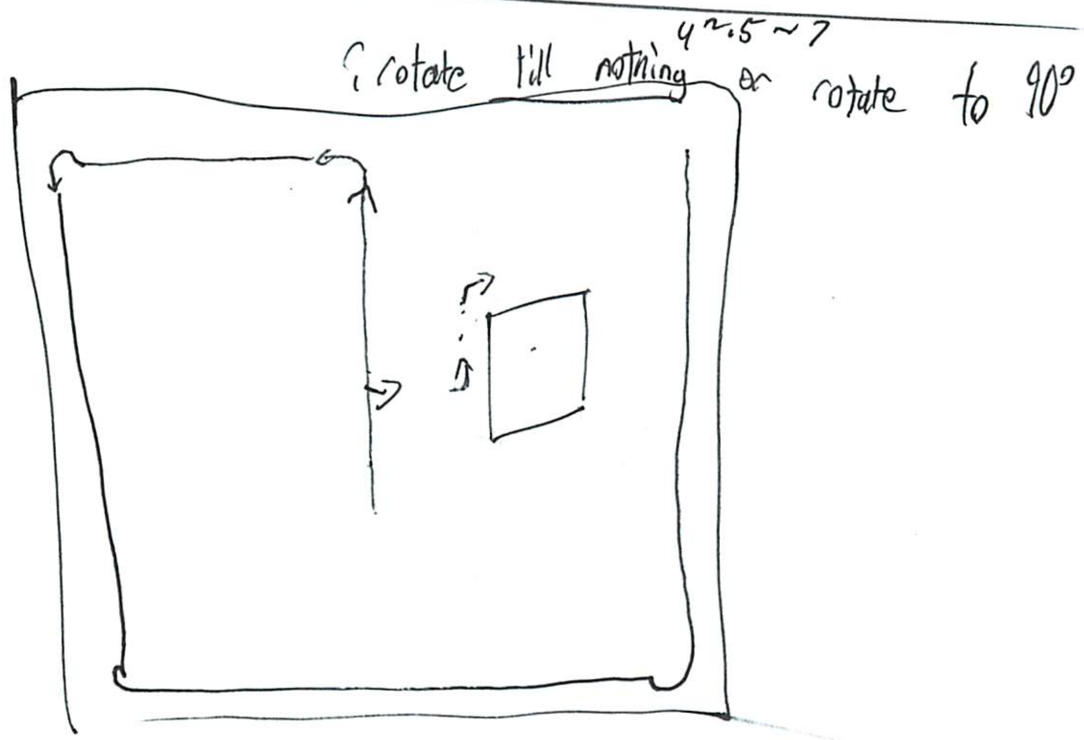
bubble wrap theoretically disperse waves

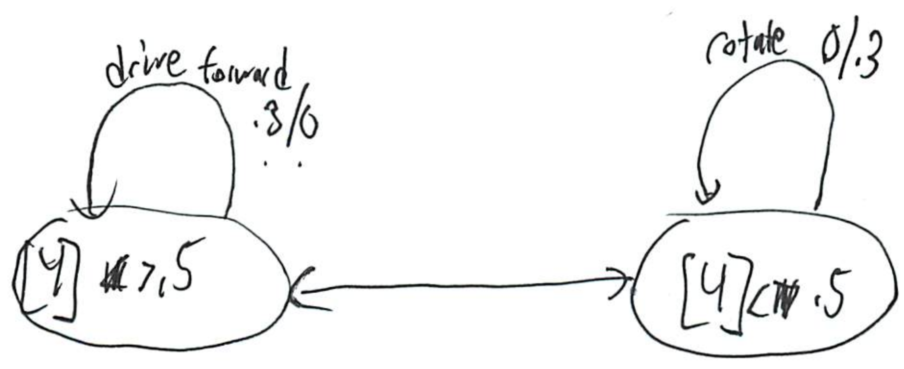
- seems to work better

(2)

Means move it back + forth till distance = .5
then stop

-easish





should include something on sensor 7

-could rotate till [4] < .5 and [7] > .3
or

(4) - no pg 3??

start rotate [4] < .5

end rotate [4] > .5 and [7] > .3

- ? rotate state to store

- or reeval on each loop

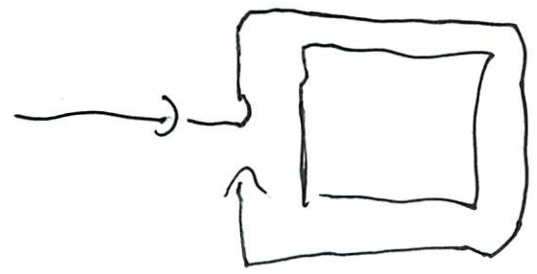
[4] < .5 or [7] < .3 → rotate

↑ reeval for gap (perhaps .3 - .5)

↑ start rotating ↑ end rotating

- could also take into account

- need to go around a sq as well



(5)

Need to look on left side

? how to know which way to turn

- always turn ~~right~~ left

if [7] > 1 → turn right

- don't care 4

- turn right [4] < .5 or [1] < .3

~~if [1] > 1 || < 1.5 → turn left~~

~~[4] < .5 or [1] > .3~~

always want right side following

if 7 > 1 turn R till 7 > .3 & 7 < .5 + 4 > .5

if 4 < .5 turn L till 4 > .5 + 7 > .3 + 7 < .5

bubbles run, rotate, stop

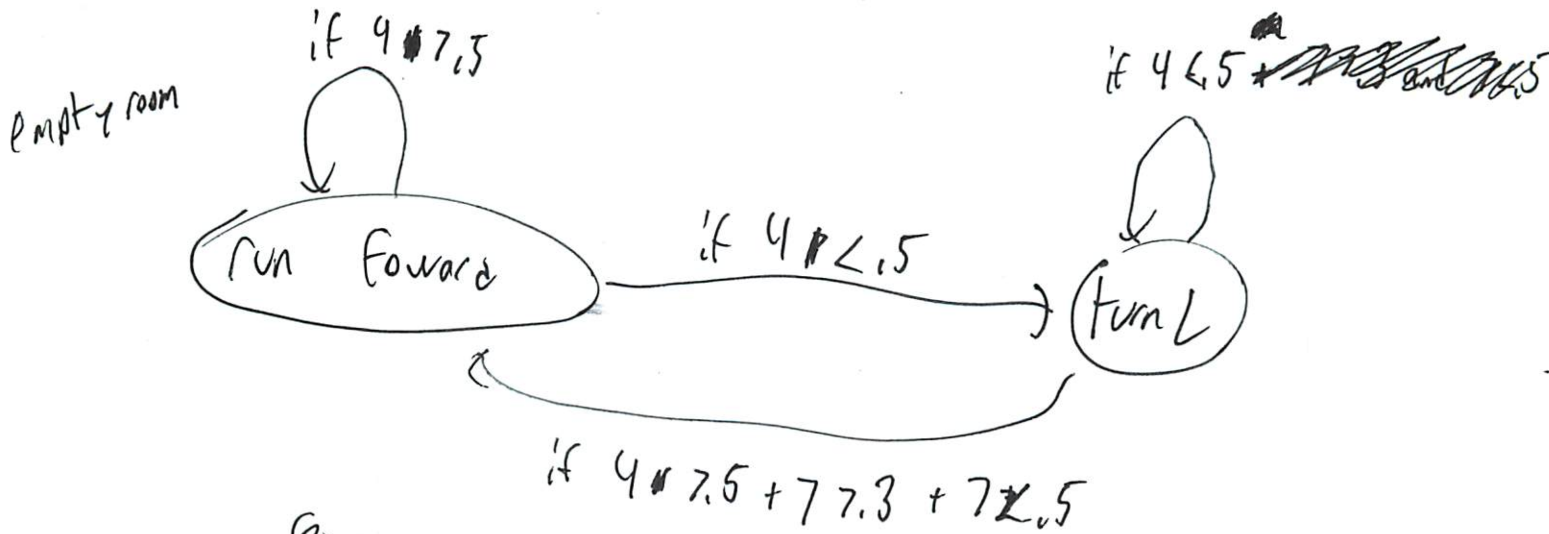
stateless
- almost

(Coco)

(6)

Need some states

- turn R
- turn L
- run forward



Object

start

③ else

run forward

② if 4 < .5

turn L

~~if 4 < .5~~ or ~~7 > .6~~

else

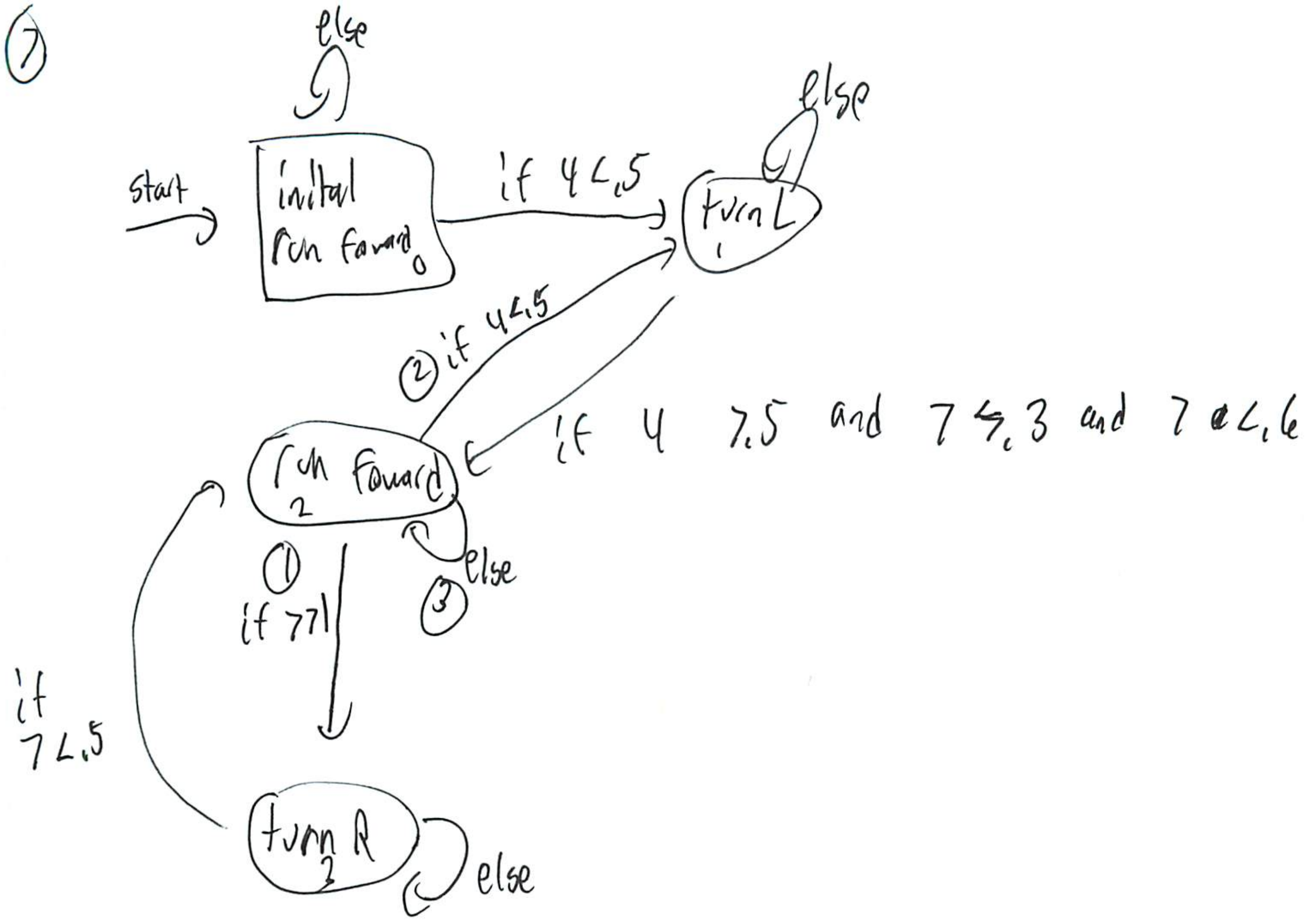
if 4 > .5 and 7 > .3 and 7 < .6

if 7 > 1 ①

if 7 < .5

turn R

7 < .5



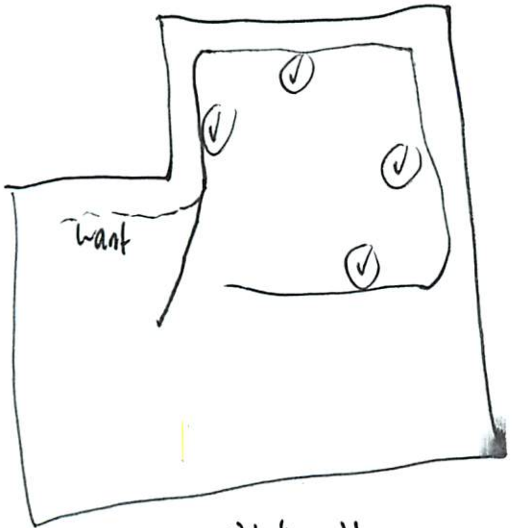
roughly correct, but
-2 diff radius
L → tight
R → wide

check off 3 ✓

---

define states as # 0, 1, 2, 3 -see above
int

works ok



R

∞← h∞

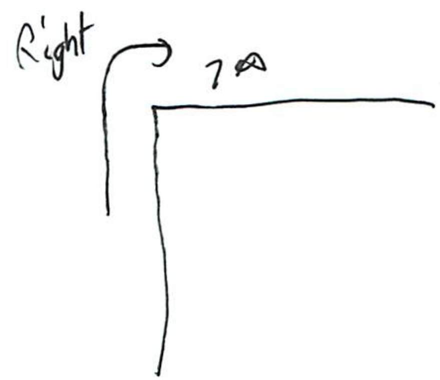
∞← L

when $7 + 4 = \infty$ (ie > 1)

turn right

- we forgot to set R to — when in state forward 2
  (didn't notice)
- next — he overshot in corner
  - Thomas: just change turn radius
  - genius!
  - I would have changed more — when so simple
- when we turn ~~right~~ left — lower the .6 to

Now this corner



- ? build a state rotation accumulator

(8) (9)

or

(3) → (1) .

now $7 < .5$

$7 < .5$ and $4 > 1$ → run foward

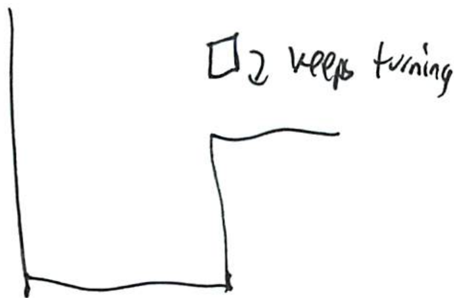
- i could play w/ speeds more

Right

if $7 > 1$ ← so fix by then turning left

$4 < .5$ if too close

turn L



now not doing whole trial right

- seems to be different each time

- cut down 1

- remove the $< .3$

---

can install sonar at home

(10) 9/23

Working at home by myself now

- also can retry nano quiz
- edison :)
- nanoquiz cont.
  - I need to really think out + plan
  - I was down by ~~[illegible]~~ 10 too
    - half the time

my check ans did seem to work:

~~Oh so all I'd had now was~~

correct!

- I just typed search patterns wron

So mistakes

1. Arrived on time; need to arrive 10 min early to set everything up
2. Tried to do it too simple, bypassed SM
3. Got flustered

Lets see their ans

- They did standard SM

state = [prev, na, nb]

(11) Looked at prev and inp

- prev = a + input = c
- never thought to look at prev
- I would always look at advance
- they incremented na + nb
  - not searched each time
- just diff ways to do I guess

Back to robot

- concentrate
- don't just random guess
- it actually runs faster on my pc! - faster processor

don't be lazy!

---

in state 2 dont get that close on 7

- watch for wall + avoid
  - something I deleted
- comment on all rules
  - implement mirrors
- stuck when removed 4 < .5 or 6 > 1
- no ~~t~~ it iterates b/w right + left
- add comments for each rule

Doing a really close turn regardless

having it print reasons is good!

Want it to turn a nice 90°

- clener
- but how do you know when at 90
- now it turns wherever C. 3
  - I don't want to build wait timers
  - ? This seems to be a common problem of where I don't want to add new feature - try to find way around takes longer
- too far from object - turn right
  - need now
  - have when 7 7 1
    - make it .5
    - numbers slowly ↓
  - still too close?
- needs to watch on 1-3 when turning right
  - guess should do on left
    - or I just did that
- worked very well, just need to get back to start!

- much more disiplined knowing more w/ status printout
- each diretion is a mishmosh
- we should unify to clean up
- Should I make decisions stateless
  ie - if 7 < .3 ↶
  4 < .3 ↓
  1 < .1 ↷
- can I unify?

~~what~~

What I have

0: inital forward
- straightfoward ;)

1: left

7 < .3 → really close to wall
turn left
4 > .5 → front is clear
go straight
else ~~turn left~~
turn left

} seems pretty clear but only 1 direction

(14)

State = 2 → straight

7 <.2 → about to crash on right!
turn left

1 <.2 → about to crash on left!
turn right

7 7.5 → turn right to follow objects
turn right

4 <.5 → front not clear
turn right

) ? should be global

State = 3 → right

1 <.3 → about to crash on ~~right~~ left
(↙ diff values)
turn left

) ??

7 <.3 → about to crash right
turn right

4 7 1 and 7 <.5 → straight is clear + within wall
go straight

? do I need

4 <.5 → about to crash

So that worked twice really nice!

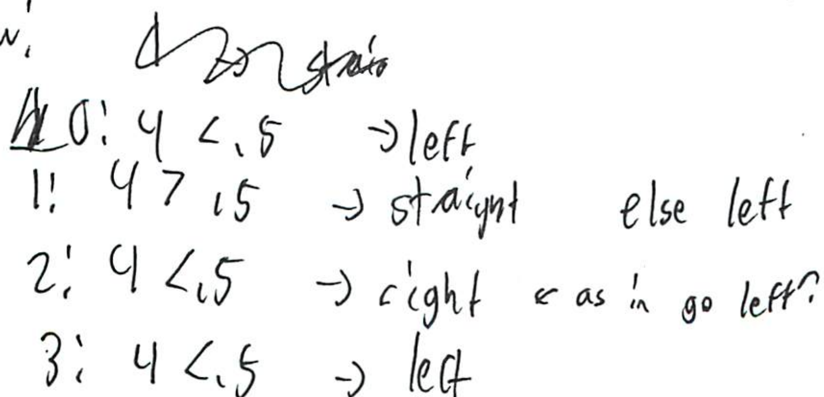
- now I want to clean up
- about to crash global
- and clean up right

So putting crash avoidance for R+L at top worked just as well
- dropped some lines of code

I wish we had another world to try

No right clean up
- don't check that kinda close to wall
  ↑ useless to do
- worked!
- had a lot of usless code

- can I make ~~stra~~ 4 ~~#~~ < .5 always go same direction?



So always go left
- Cool works
- don't really need a left state anymore!
  - ~~tha~~ - handled in global
- 0 state has no rules

"can I do it stateless
- ~~the~~ prof says can't be done

I think inital mistake I made w/ so many rules
was trying to make it follow walls nice

Oh still need to break out of 0
- that is why don't need state
- well says 1

Took too much out
- needs to break out of state 1
- oh well
  - ~~abandon clean 2~~
- needed front not clear, turn right in 2
  - ~~I think I am going to abandon clean 2~~

nope got it to work — about half of previous code 62 vs 47
and works on other map! er third

HW1 Assign Part 2: Put it Together

- put pieces together + test
- calc → prompts the user → goes to inp
- tokenize, → parse → evaluate
  - print evaluation + new env
- Test procedure
  - need to fill in ??
  - oh no - you pass it expressions and it "types it"
  - or do you put something in?
  - oh input should be test Exprs
    - ↳ but then it needs to calculate!!
  - now ec the rest of the problem
    - ~~this this~~ no guidance → extensions next
  - so put it togeter tokenize → parse → evaluate ✓ ✓
  - now need to evaluate - how?
    from test
    ~~~~.eval(env)
    ↑ not defined anywhere
    - oh skipped section 5 - eval
    ~~~~

(2)

eval

- have syntax tree ✓
- First only #
- Then w/ variables /env
  - using dictionary

eager

- computes every expression it sees
- 1. if # return value.

  * write for each expression class *

  - so just for number → print it!
  - easy → done ✓

2. if Variable → return value from dict
   - assume already in dict + print
   - done

3. arithmetic do the opp left op right
   - can use functional programming style
   - what form self.left + self.right?
     - or need .eval? yeah - otherwise instance

③

Done for sum
- copy for − * ÷

4. Assignment
- name on right
- value on left
- Change (mutilate?) env
- return nothing
- oh — but Variable ( ) returns its value
  - null - so must override
  - or get name manually

done!

I like this assignment → PCAP

Back to put together
- done really quick now

Ok time to submit
- 1st test failed no 6.01 lib

then it all passed!

Done ✓

- now an extension

(4)

Extension - pick 1

- tokenizing SM
  - seems silly, work around-ish
  - could use SM practice
- lazy eval
  - seems more helpful
  - store unassigned variables

6.2 Lazy Partial eval

(d = (b + c))

^ before b + c are defined

only eval d when called

1. If variable test in dict
   - Yes, return ~~variable~~ value
     - ↳ test 'a' in d → True
   - No - return variable
     - ↳ as Var('a')
2. When evaluate assignment - assign the syntax trees don't evaluate
   - = self.right

5

3. Operations ; evaluate
   - if numbes, return like befor
   - if not, make new instance of operator class
     - partial eval
     - still eval left + right
       - See if you can make some impact
4. When look up variable in env
   - that is where?
   - - Variable
   - eval it

- Oh they have isNum
  - I used NumberTok
- And hint on making class
  - let me try 1st

- ; Change print of variable to show it?
- NumToken erroring
  - change isNum
- still erroring
  - do isNum(left) and isNum(right)

(6)

That must be the hint

Self.__class__ makes new instance

- Stuck on why it says unsupported operand types for +
  instance + instance
- Variable is returning something wrong
  - or Sum can't eval right
  - even if I just return None
  - it prints fine
    - just won't evaluate
    - will do left + right
      - and eval left + right
      - but Sum won't do it!
  - or perhaps that is right
    - because they are not # - can't eval!
      - yet!
  - or Check if ~~num defined~~ variable defined as number
    - if so return that
  - but that is not really what we are supposed to do
    - always eval
    - and Sum should check what is up

7

Go back to normal eval + see if still works
- its storing some variables as Number(5.0)
  - did it do that before?
  - not evaling
    - should always eval
- but what if sum?
- just go through old fashioned + make sure still works
-
  - oh my isNum I had wrong
  - doing normal eval gives ∞ loop now
    - because debug code in Sum
- When assigning a to sum a+b it does not eval
  - Though numbers
  - well that was lazy eval
  - Oh as designed
  - When print should be good
    - thats when ∞ loop is back
    - and true
      - since it ref variable A in there
      - so bad example
  - Oh actually working now
    - cool

(8)

So what went wrong?

- the ~~a~~ isNum(left) and isNum(right)

  not [isNum(left) + isNum(Right)]

  ↑ duh that is true + false

  or did I have isNum(left + right)

  ↑ which still tries the addition

and in Variable

- if it is a # return, don't eval

but both I think were not the problem ???

- and don't use __class__

---

test just like sheet

- cool done

- Now they want ~~ans~~ some written ans

- and print lots of stuff