Welcome!

We'll get started shortly. Please take the Zoom poll in the meanwhile!

CS 49 Week 1

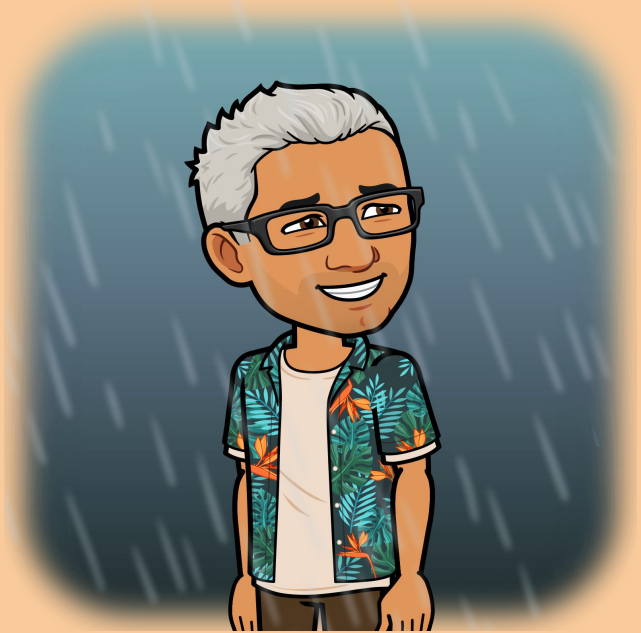
Surajit A. Bose

Agenda

- Introductions
- Logistics
  - Section purpose and setup
  - Getting help
- Introducing Karel
  - Karel's world
  - Karel conditions
  - Karel commands
- Programming with Karel
  - Overview of programming
  - Worked example: Move Beeper
  - Section problem: Archway

Introductions

A little about me



- My name is Surajit (he/him). If you are uncomfortable calling me by my first name, Mr Bose is okay too
- This is my fourth time volunteering as a section leader for CS 49
- I'm retired from a tech career
- In my free time I enjoy listening to Indian classical music
- My favorite cuisine is Burmese, and the best Burmese food I've ever eaten is in Palo Alto

What about you?

Please let us know:

- Your name, and if you would like to share them, your pronouns
- Where you're tuning in from: a city in the Bay Area? Somewhere else?
- Whether you have any prior programming/coding experience
- What you'd like to get out of this class
- Icebreaker question:
    - What is your favorite cuisine? (Italian, Chinese, Mexican, Indian, etc)
    - In what city have you eaten the best food from that cuisine?
- After your turn, you get to "popcorn" to someone, i.e., choose who goes next!

Logistics

What is Section?

- Section is your chief support system for CS 49
- The goal of section is to ensure your success in the class
- A small group of students meets weekly with a volunteer tutor
  - This is likely the smallest class meeting you'll have at Foothill
- Participating in section comprises 30% of your class grade
  - Please familiarize yourself with the section expectations page on Canvas
  - Active participation: camera on, ask questions, provide answers
    - You get participation points for attempting an answer, whether or not it's correct
    - You get participation points if you ask a question
  - To ensure that all of you get full participation points, I do call on students!

What happens during section?

- Quick Zoom poll at the start of the meeting (so please show up on time)
- Any needed logistics (~5 minutes)
- Overview of week's material (~15 minutes)
  - Usually, we will work through some coding exercises as review of concepts from the video lectures
  - This is also your opportunity to ask for clarification of any difficulties with the lecture or reading
- Coding practice (~20 minutes)
  - One or more section problems separate from required weekly assignments
  - Generally, three or so students work on these in breakout rooms
  - One student shares screen, all solve the problem together
- Discussion of solution with the entire group (~10 minutes)
- Optional lab/office hours after section time (1 hour)
  - For working on the weekly assignments or getting help

Section Norms

- Section is flipped—it is your time to get help with, review, and practice the week's material
- The assumption is that prior to section time, students will have:
  - Viewed the lecture videos
  - Done the reading
  - Begun work on the week's assignments
- If you're behind on the week's work, come anyway!
- Typically, camera and mic must be on
  - If you cannot have your camera on, please let Lane and me know
  - In noisy environments, please mute your mic when not speaking

Section Infrastructure (Slide 1 of 2)

- Section area of Code in Place website
  - Section Zoom link
  - Section problem descriptions
  - Coding area (IDE)
  - Section problem solutions, posted an hour after section
- Section github repo
  - Lecture slides from Mehran and Chris
  - Section slides such as these!
    - Clicking on links does not work in preview mode
    - Must download slides to click on links in them
  - Coding examples and starter code
  - Section problem solutions, different from the ones on the Code in Place site

Section Infrastructure (Slide 2 of 2)

- Pronto for announcements and messaging
  - A highly recommended short video on using Pronto within Canvas
  - Download and quickstart guide for using the mobile app
- Section announcements via Pronto
  - Usually two per week: one a couple days before section, one within a day after section
- Messaging via Pronto
  - Can DM me (best way to reach me for a quick though not necessarily immediate response)
  - Can post to section or course-wide Pronto discussions
  - Can DM Lane or others in this section or the course

How to get hold of me / get help from other resources

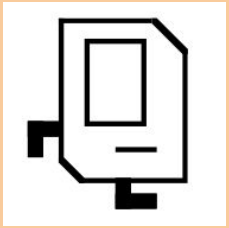
- Surajit's office hours
  - Fridays 12 noon–1p, directly after section
  - By appointment on Zoom
- Pronto DM for a quick response (usually within a couple hours)
- Email bosesurajit@fhda.edu or Canvas inbox, 24 hr turnaround
- The github repo has section materials, starter code, and solutions

---

- Pronto DM or Canvas inbox for Lane
- Lane's office hours
- Online or in-person tutoring at the STEM center (Room 4213)

Introducing Karel

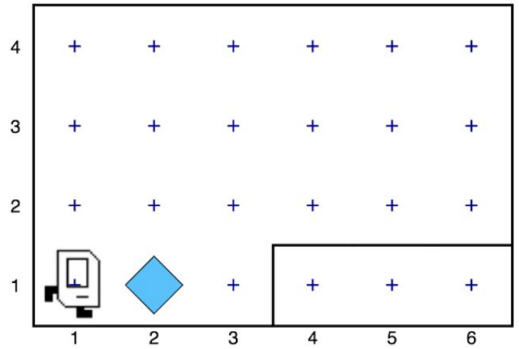
Karel the Robot



- Karel is a robot who occupies a certain **world**
  - The world has elements like corners, beepers, and walls that Karel interacts with or navigates
- Karel and its world have certain **conditions**
  - Conditions are statements that can be either `True` or `False`
  - In our world, for example, a condition can be "It is raining"
  - Determining whether a condition is `True` or `False` is called **evaluating** the condition
- Karel can perform certain **commands** to navigate its world
  - Commands are simple instructions like `move()` or `turn_left()`
- Let's look at Karel's world, conditions, and commands in the next few slides

Karel's World

Karel's world is defined by rows running horizontally (east-west) and columns running vertically (north-south). The intersection of a row and a column is called a corner. Karel can only be positioned on corners and must be facing one of the four standard compass directions (north, south, east, west). A sample Karel world is shown below. Here Karel is located at the corner of 1st row and 1st column, facing east.



Several other components of Karel's world can be seen in this example. The object in front of Karel is a beeper. As described in Rich Pattis's book, beepers are "plastic cones which emit a quiet beeping noise." Karel can only detect a beeper if it is on the same corner. The solid lines in the diagram are walls. Walls serve as barriers within Karel's world. Karel cannot walk through walls and must instead go around them. Karel's world is always bounded by walls along the edges, but the world may have different dimensions depending on the specific problem Karel needs to solve.

Karel Conditions

Test	Opposite	What it checks
`front_is_clear()`	`front_is_blocked()`	Is there a wall in front of Karel?
`beepers_present()`	`no_beepers_present()`	Are there beepers on this corner?
`left_is_clear()`	`left_is_blocked()`	Is there a wall to Karel's left?
`right_is_clear()`	`right_is_blocked()`	Is there a wall to Karel's right?
`beepers_in_bag()`	`no_beepers_in_bag()`	Does Karel have any beepers in its bag?
`facing_north()`	`not_facing_north()`	Is Karel facing north?
`facing_south()`	`not_facing_south()`	Is Karel facing south?
`facing_east()`	`not_facing_east()`	Is Karel facing east?
`facing_west()`	`not_facing_west()`	Is Karel facing west?

Karel Commands

Command	Description
`move()`	Asks Karel to move forward one block. Karel cannot respond to a `move()` command if there is a wall blocking its way.
`turn_left()`	Asks Karel to rotate 90 degrees to the left (counterclockwise).
`pick_beeper()`	Asks Karel to pick up one beeper from a corner and stores the beeper in its beeper bag, which can hold an infinite number of beepers. Karel cannot respond to a `pick_beeper()` command unless there is a beeper on the current corner.
`put_beeper()`	Asks Karel to take a beeper from its beeper bag and put it down on the current corner. Karel cannot respond to a `put_beeper()` command unless there are beepers in its beeper bag.

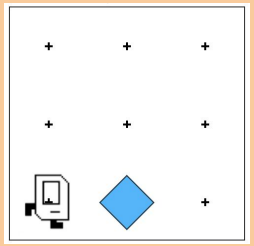
Programming with Karel

An Overview of Programming

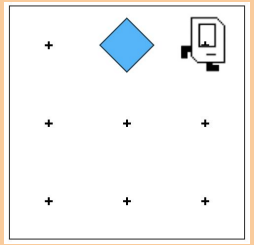
- A program is a sequence of steps. Each step is an instruction or **command**
- The steps are intended to get from a certain condition to a desired result
  - The starting condition is the **precondition**
  - The desired result is the **postcondition**
- Solving a programming problem involves
  - Identifying the precondition and postcondition. This is **understanding the problem**
  - Figuring out what steps are needed to get from precondition to postcondition. This is **designing the solution** . Usually, the solution is designed using ordinary language, not code
  - Coding the steps. This is **implementing the solution** . This is very hard, almost impossible, if you have not designed the solution first; it is very easy if you have designed the solution well
  - Finally, making the solution better: going over the program to fix errors, deal with special cases, or identifying ways to make the steps more efficient. This is an **iterative process of refinement**

Worked Example: Move Beeper

- Preconditions:
  - Karel is in the bottom corner, facing east
  - There is a beeper immediately in front of Karel



- Postconditions:
  - Karel is in the top right corner, facing east
  - Beeper has been moved and is immediately behind Karel



- What steps are needed to get from preconditions to postconditions?

The Anatomy of a Program

- File name, overview comment, your name, date
- Import statement
- **`main()`** module with further comments. Indentation is important!
- Guard clause and invocation of **`main()`**

```
"""
main.py

Karel moves a beeper from the bottom row to the top row

Programmer: Surajit A Bose, Date: 2025.02.03
"""

from karel.stanfordkarel import *

def main():
    """Move Karel and beeper to top row

    Preconditions:
    - Karel is in the bottom left corner, facing east
    - There is a beeper immediately in front of Karel

    Postconditions:
    - Karel is in the top right corner, facing east
    - Beeper has been moved and is immediately behind Karel
    """
    pass   # Delete this line and write your code here!  :)

# There is no need to edit code beyond this point
if __name__ == '__main__':
    main()
```

Decomposing a Problem: Functions

- In solving Move Beeper, we saw that to have Karel turn right, we had to use the `turn_left()` command three times
- This makes the code hard to follow; wouldn't it be nice to just say `turn_right()`?
- Also, in another program (such as Archway, coming up!) Karel might need to turn right more than once. Wouldn't it be nice to be able to reuse `turn_right()` instead of repeating `turn_left()` three times over and over?
- Programmers abstract out self-contained and/or repeated lines of code into smaller chunks called **functions**
- We can write `turn_right()` as a function and **call** that function inside `main()`
- `main()` is also just another function, called in the last line of our program

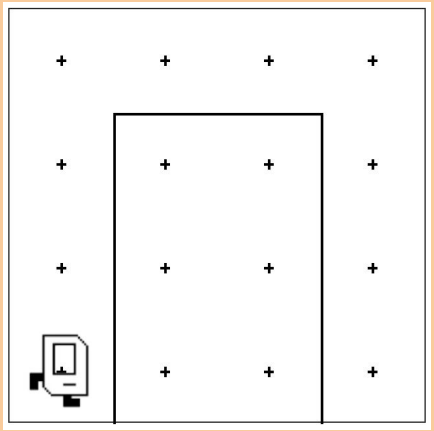
Sample function: `turn_right()`

```
# function header has keyword def, the function name, parentheses, and a colon
def turn_right():
    # function body has commands indented in a block below the function header
    turn_left()
    turn_left()
    turn_left()
```

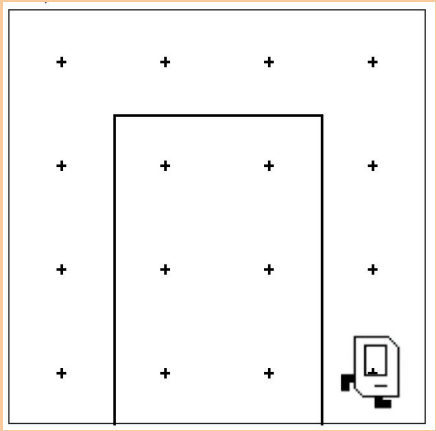
Section Problem: Archway

Section Problem: Archway

Karel is in a world with an archway, like so:



Karel needs to get over the archway, like so:



Coding Archway

- Understanding the problem
  - What are the preconditions?
  - What are the postconditions?
- Designing the solution
  - What steps are needed to get from preconditions to postconditions?
  - What repeated steps can be abstracted out into self-contained functions?
    - `turn_right()`
    - `move_three_steps()`
- Implementing the solution
  - Write the code for `main()`, calling the functions as though they're already written
  - Write each of the two helper functions
- Refining the solution: fixing any errors

That's all, folks!

Next up: Control Flow