Welcome!

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

CS 49 Week 1

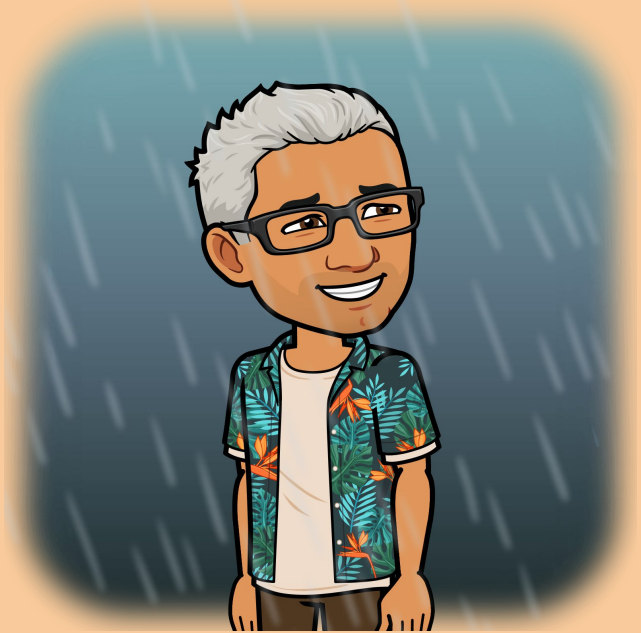
Surajit A. Bose

Agenda

- Goals for Week 1 Section
  - To get acquainted
  - To understand the purpose and setup of section
  - Using Karel the Robot, to understand the four-step process of writing a computer program
- Introductions
- Logistics
  - Section purpose and setup
  - Getting help
- Programming with Karel
  - Introducing Karel: World, Conditions, Commands
  - Overview of programming
  - 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 fifth 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
  - 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!

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 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

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 (~25 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 Infrastructure

- 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 forum for announcements and other questions
- 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

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

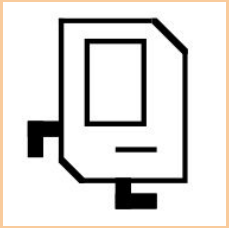
- Surajit's office hours
  - Fridays 12 noon–1p, directly after section
  - By appointment on Zoom
- Post on the section forum, usually get a response within 2 hours
- Email bosesurajit@fhda.edu or Canvas inbox, 24 hr turnaround
- The github repo has section materials, starter code, and solutions

---

- Canvas inbox for Lane
- The main forum for the course on the Code in Place site
- Lane's office hours
- Online or in-person tutoring (Room 3600)

Programming with Karel

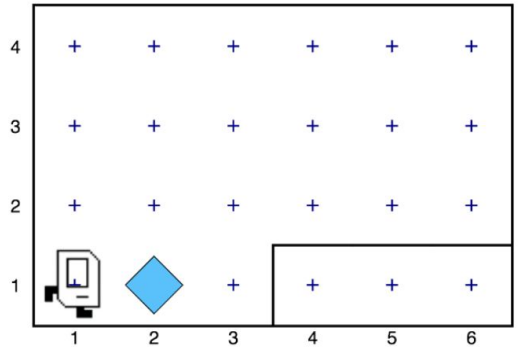
Karel the Robot



- 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()`
- Details of Karel's world, conditions, and commands are on 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.

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**

The Anatomy of a Program

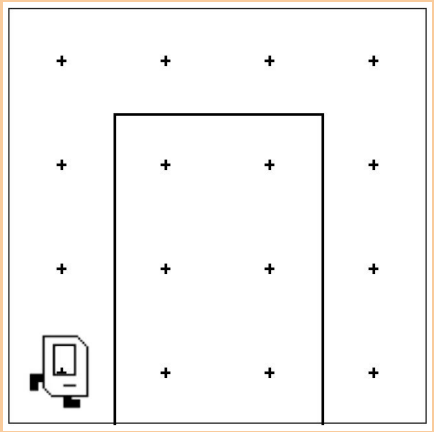
- File name, overview comment, your name, the date
- Import statement
- **`main()`** module with further comments. Indentation is important!
- Guard clause and invocation of **`main()`**

```
"""
archway.py

Get Karel over an archway

Programmer: Surajit A. Bose, Date: April 25, 2025
"""

from karel.stanfordkarel import *

def main():
    """
    Move Karel from one corner of the world to the other over an archway

    Preconditions:
    - Karel is on the bottom row leftmost corner facing east
    - There is an archway three squares high blocking Karel's way forward

    Postconditions:
    - Karel is on the bottom row rightmost corner facing east
    - Archway is behind Karel
    """
    pass    # Delete this line and write your code here!  :)


if __name__ == "__main__":
    main()
```

Section Problem: Archway

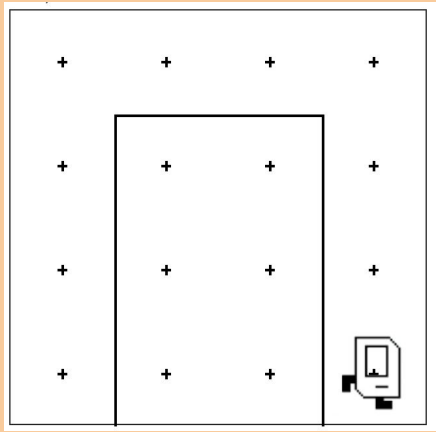
Understanding Archway

Preconditions: 1. Karel is on the bottom row leftmost corner, facing east:



2. There is an archway three squares high blocking Karel's way forward

Postconditions: 1. Karel is on the bottom row rightmost corner, facing east



2. Archway is behind Karel

Designing, Implementing, and Refining Archway

- Designing the solution
  - What steps are needed to get from preconditions to postconditions? Write out each step in English
- Implementing the solution
  - Translate the steps above into code
- Refining the solution
  - Are there any errors? If so, let's fix them
  - Abstract out repeated steps into functions

Decomposing a Problem: Functions

- In solving Archway, 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, Karel turns 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()
```

More about functions

- What other set of repeated steps can we abstract out into its own function? **`move_forward_three()`**
- Let's implement these two functions and incorporate them into our program
- Notice how when a function is called from **`main()`**, the flow of execution jumps to the called function
- Notice how when the called function's work is done, the flow of execution jumps back to the calling function, in this case, **`main()`**

That's all, folks!

Next up: Control Flow