Welcome!

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

CS 49 Week 3

Surajit A. Bose

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)
- One-on-one code reviews with Sarah or Lia

Agenda

- Logistics
  - No section meeting next week, but course materials and assignments are still due as usual
  - Section slides and other materials for week 4 will be posted on the GitHub repo on Monday
- Goals for Week 3 Section:
  - Understand style guidelines
  - Understand how to design and refine coding solutions
- Review of lecture concepts
  - Coding style
  - Program design
  - Testing
- Worked example: Five Corridors
- Section problem: Spread_Beepers
- Optional bonus slides (won't be covered in section)

Lecture Review

Code Style Guidelines (Slide 1 of 2)

- Function names:
    - Short and descriptive, e.g. **`move_to_wall()`**
    - Typically the function name specifies an action and so uses an imperative verb
    - For this course, **`use_snake_case`** for function names
- Function structure:
    - What does Chris say about how long a function should be?
    - What does he say about the number of indentation levels?
- Code comments
    - **`"""Triple quoted strings"""`** for documenting entire program and each function
    - Obvious functions like **`turn_right()`** can have minimal to no comments
    - Complex functions should have pre- and postconditions specified, as should the entire program
    - ***`# Inline comments`*** as needed for specific lines of code, e.g., to mention fencepost condition

Code Style Guidelines (Slide 2 of 2)

- Formatting
  - Do not mix tabs and spaces for indentation; check style guide for your organization
  - Leave two blank lines between one function and the next
  - Be consistent about spaces, e.g., between function name, parentheses, and colon in header
- Example: Spring_Flowers solution on GitHub

The Process of Problem Solving: Stepwise Refinement

- First, **understand** the pre- and postconditions
- While **designing the solution**, identify the repeatable, self-contained building blocks that will be the functions: **decompose** thoroughly
- When coding, assume the functions are done
  - Write out the function headers
  - Use the **`pass`** keyword for the bodies
- Assemble the functions in **`main()`** to solve the overall problem
- **Implement** each building block by writing out its function body
- **Test** iteratively and **refine** the solution:
  - Check that each function works as expected
  - Check that all functions put together in **`main()`** solve the entire problem as expected

Program Design Guidelines

- Solve for the general case unless otherwise specified. E.g., for Hospital Karel, the solution should work:
  - For any number of hospitals: zero, one, two, or more
  - In a world with any number of columns, odd or even (can assume at least two columns)
  - Whether the hospital site is on the first column, in the middle, or the last column but one
- Do not over-optimize. E.g., for Hospital Karel:
  - The preconditions specify: "there is room to build the hospitals without overlapping or hitting walls."
  - Therefore, do not write checks to make sure that Karel can build a hospital without hitting a wall

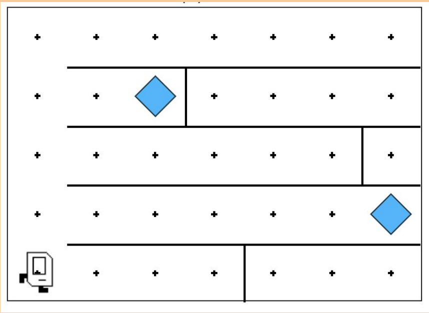
Program Testing Guidelines

- With loops, first write and test **one** pass through before putting the steps inside the loop
- Test each function as written
- When testing:
    - Check for syntax errors such as incorrect indentation or missing colons
    - Check program logic to ensure the problem is solved fully and correctly, including fenceposts
    - Check for efficiency, i.e., whether the solution can be made shorter (fewer steps == faster code)
    - Note the **iterative nature** of programming!
- Finally, check that the program is well documented and meets style guidelines
- Demonstration of these

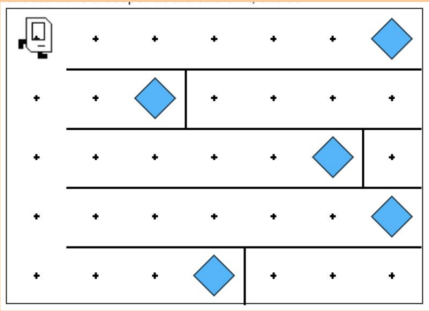
Worked Example: Five Corridors

Understanding Five Corridors

- Preconditions:
    - Karel is facing east at the bottom left corner of a world with five rows
    - Each row is a corridor ending at a wall
    - Some rows may have a beeper directly before the wall



- Postconditions:
    - Karel is facing east at the top left corner of the world
    - All rows have a beeper before the wall



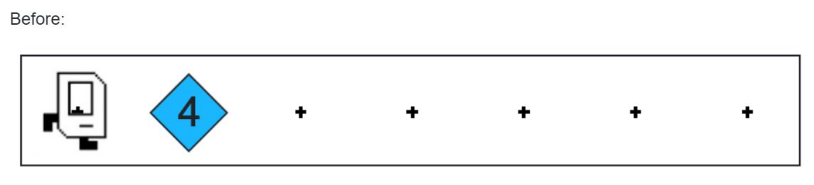
Designing, Implementing, and Testing the Solution

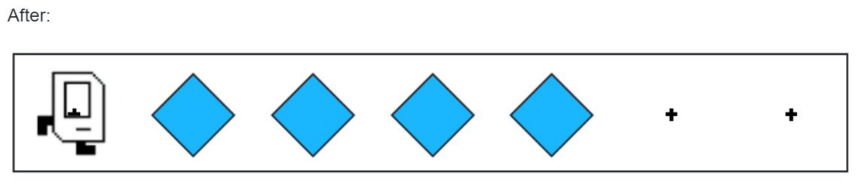
- Design:
    - What are the steps involved in the overall solution?
    - What does Karel need to do for **one** row?
    - How many times does Karel need to do this?
    - What would be a good control flow mechanism to use?
    - What are likely candidates for functions? I.e., what repeatable, modular building blocks can we use in the steps?
- Implement and test:
    - Write out all function headers and use **`pass`** for the bodies
    - Write out **`main()`** assuming all the functions have been implemented
    - Implement and test each function iteratively
    - Test **`main()`** and refine the overall solution

Section Problem: Spread Beepers

Understanding Spread Beepers

- Preconditions:
  - Karel is at the bottom right corner of a one-row world, facing east
  - Karel already has some unknown number of beepers in its bag—possibly infinite
  - Directly in front of Karel is a pile of an arbitrary number of beepers
  - The world is wide enough for the beepers in the pile to be spread out along the row





- Postconditions:
  - The beepers have been spread out across the row
  - The number of beepers spread out in the row is the same as the number in the pile
  - Karel is back at the bottom right of the row, facing east

Design Considerations and Suggestions for Spread Beepers

- What steps are needed for Karel to pick up **one** beeper from the pile and put it down in the appropriate spot?
- After placing one beeper where it should go, what does Karel need to do? What steps are involved in this?
- Generalize the above two bullet points with a loop
  - Given that the pile has an arbitrary number of beepers, should we use a **`for`** loop or a **`while`** loop?
  - If the latter, what is the loop condition?
- What should happen for the program to end?

Goodbye, Karel!

Next up: Python console programming

Bonus: Extending Spread Beepers

About Bonus Slides

- Slides with a green background are bonus slides
- They are for exercises or material beyond what's covered in the course
- They're strictly for fun
- We won't go over them during section
- Feel free to ignore them!

Extending Spread Beepers

- The preconditions for Spread Beepers specify:
  - The pile of beepers is directly in front of Karel
  - There is only one row in the world
- What if these these preconditions were different?
  - What if the pile of beepers could be anywhere, not just directly in front of Karel?
  - What if there were an arbitrary number of rows in the world?
- Can you design and implement a solution for these different preconditions?
- You can see one solution at work here
  - This is not the most efficient solution: Karel returns to the first column of the row after placing a beeper, then moves all the way back to the pile
  - Can you make the solution more efficient such that Karel does not go all the way to the first column after placing each beeper?