

Welcome!

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

CS 49 Week 2

Surajit A. Bose

Agenda

- Goal for Week 2 section: To understand how to use control flow in designing and implementing coding solutions
- Review of lecture concepts
 - Conditionals: **if**, **if-else**
 - Loops: **for**, **while**
 - Functions
- Worked example: [Five Corridors](#)
- Section problem: [Hospital Karel](#)

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 boesesurajit@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)

Control Flow: Conditionals, Loops

Control Flow

- The order in which instructions are executed is not always linear
- We discussed how in [Move Beeper](#), calling **turn_right()** causes program execution to jump out of **main()** to the called function, then back to main once **turn_right()** is completed
- A function call therefore affects the program's flow of control
- Conditionals and loops are other mechanisms used for control flow
- Conditionals include **if** and **if-else**
- Loops include **for** and **while**
- Let's look at each of these in turn

Python Conditionals



if Statement

- An **if** statement performs the block of code indented inside it only when the associated condition is **True**, and only once
- An example **if** statement that you might see and use with Karel:

```
if front_is_clear():  
    move()
```

- Note that **move()** is indented below the **if** clause. As with functions, indentation is meaningful and necessary in all Python conditionals and loops. It demarcates the **block** of commands that will be performed as part of that conditional or loop
- There can be several commands under a single **if**, all part of the same block

if-else Statement

- An **if-else** statement performs a block of code when the associated condition is **True**, or a different block of code when the condition is **False**
- The appropriate indented block is performed only once
- An example **if-else** statement that you might see and use with Karel:

```
if no_beeper_present():  
    move()  
else:  
    pick_beeper()
```

- There can be several commands within each block

Python Loops



while Loop

- A **while** loop repeatedly performs a block of code until its associated condition evaluates to **False**
- The condition is evaluated before each pass through the loop
- If the condition is **True**, the indented block is executed
- This process repeats until the condition becomes **False**
- An example **while** loop that you might see and use with Karel:

```
while front_is_clear():  
    move()
```

- This loop is also called an *indefinite loop* because we don't know in advance how many times it will run

for Loop

- A **for** loop performs its block of code a specified number of times
- An example **for** loop that you might see and use with Karel:

```
for i in range(3):  
    turn_left()
```

- **i** is a conventional name for the counter, from integer. The underscore **_** is also a very common counter. But you can call the counter anything, even **jane** or **bob**
- The counter is incremented after each pass through the loop
- This loop is also called a *definite loop*, because we know when it ends: when the counter reaches the specified number

Functions

Functions

- The process of breaking down a problem into smaller, self-contained building blocks is **decomposition**
- These smaller building blocks are **functions**
- A function is a sequence of steps that achieves a specific outcome
- Any set of steps that needs to be repeated could be made a function
- So could any logically self-contained portion of the program
- A sample function you might see and use with Karel:

```
def turn_right():  
    for _ in range(3):  
        turn_left()
```

Advantages of using functions

- **Readability** : a single statement `turn_right()` is easier to understand than three consecutive `turn_left()` statements
- **Repeatability** : If Karel needs to turn right several times in a program (e.g., as in [Archway](#)), using this function can help us avoid writing the same code over and over again
- **Reliability** : Using functions makes the code modular and easier to debug, as we can test each function out after writing it to make sure it works rather than testing the whole program at once

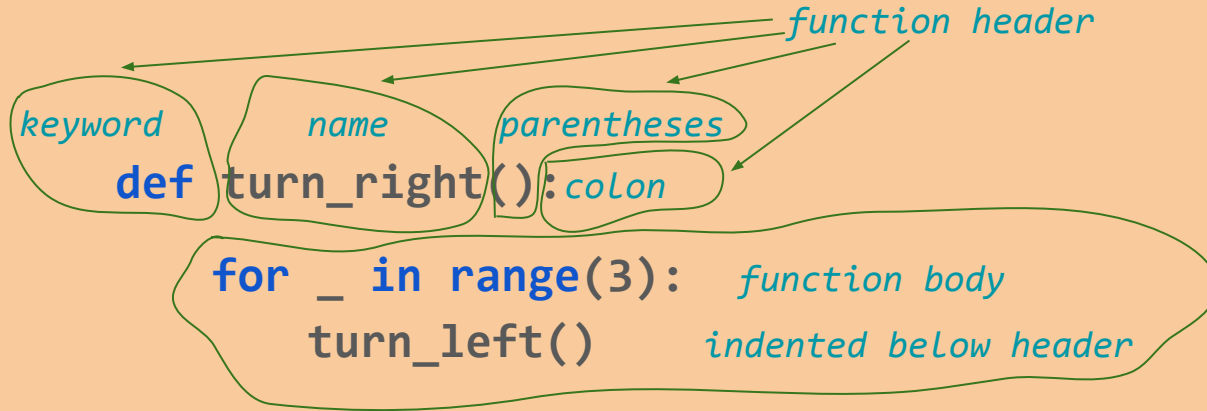
Function Structure (Slide 1 of 2)

- The function header is at the top of the function definition:

```
def function_name():
```

- The header starts with the keyword **def** followed by the function name
- The function name should briefly but clearly state what the function does
- The function name is followed by a pair of parentheses ()
- The function header ends with a colon :
- The function body is an indented block below the header. It includes all the commands necessary for the function to do its work

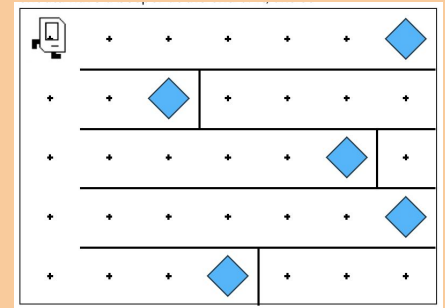
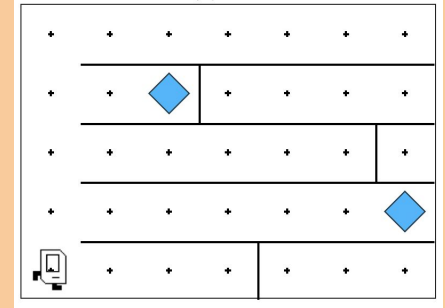
Function Structure (Slide 2 of 2)



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



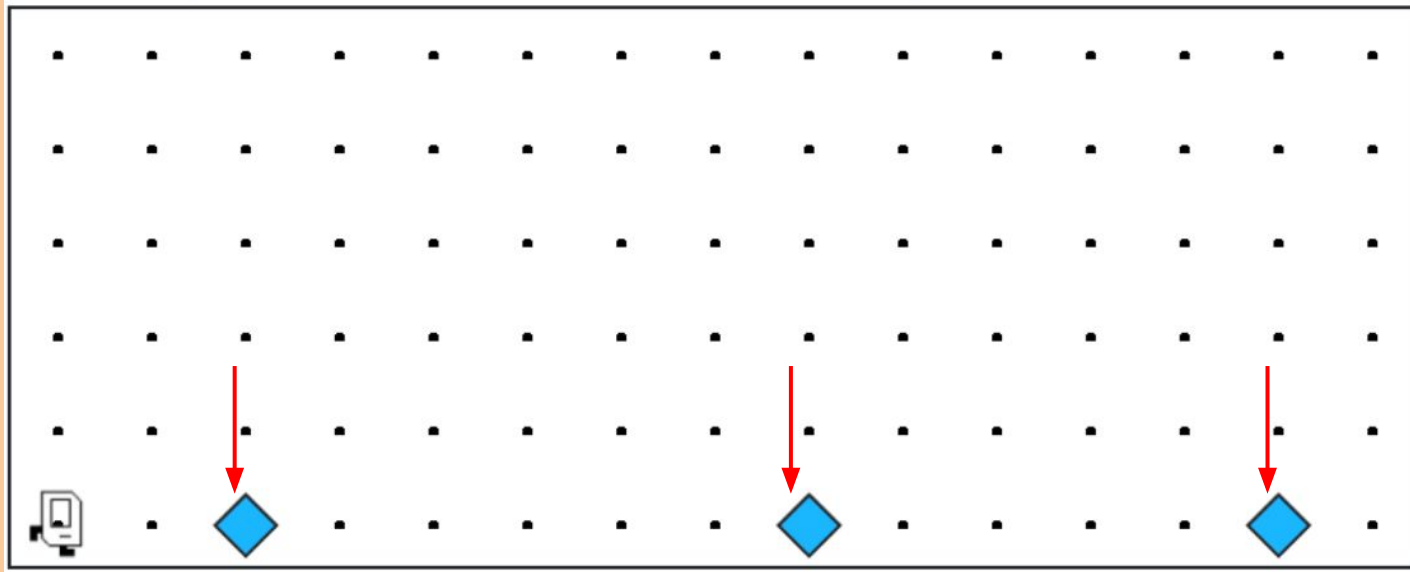
Designing, Implementing, and Testing the Solution

- 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: Hospital Karel

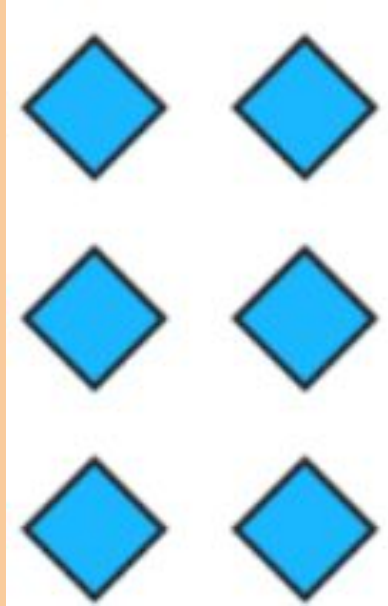
Hospital Karel Preconditions (Slide 1 of 3)

Karel starts facing east at [1, 1] with an infinite supply of beepers in its bag. Each beeper in the bottom row represents a location where Karel should build a hospital:



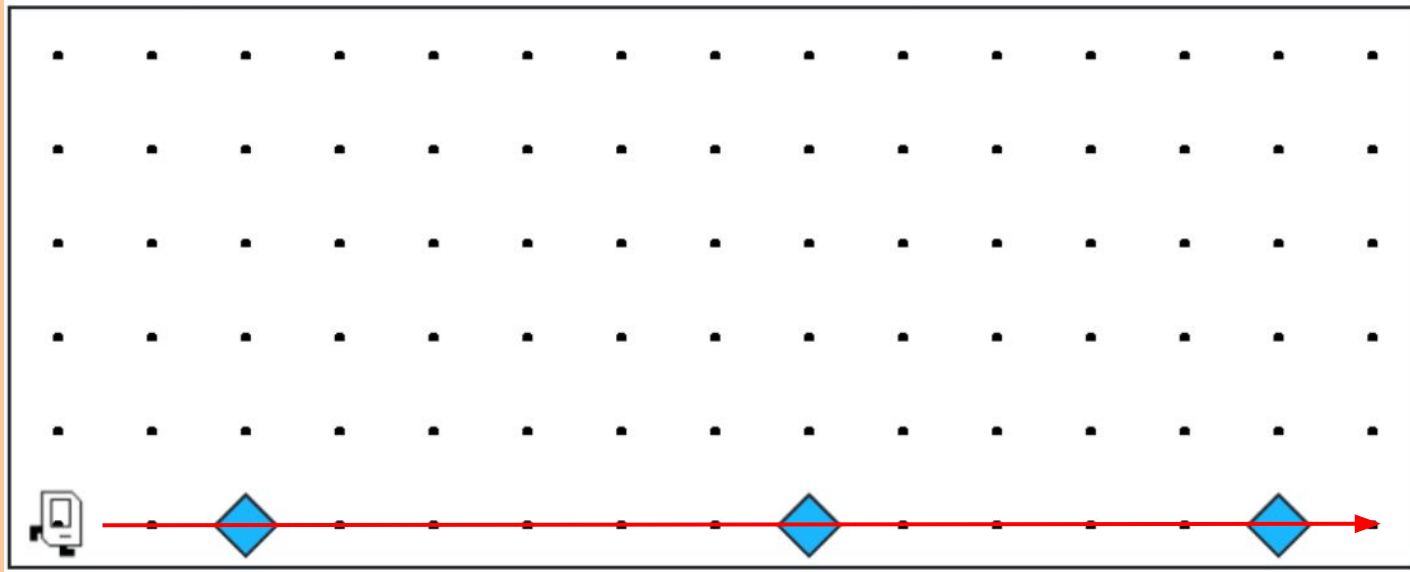
Hospital Karel Preconditions (Slide 2 of 3)

The hospitals Karel should build are a 3 x 2 rectangle of beepers:



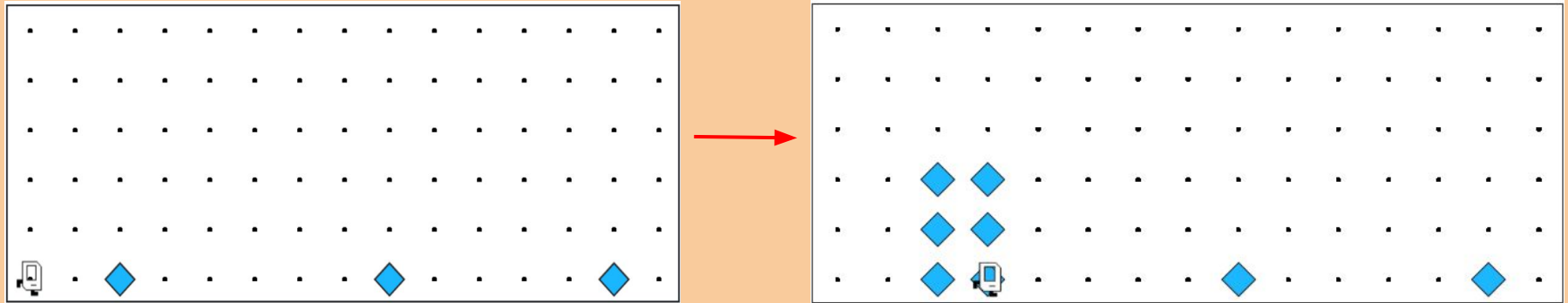
Hospital Karel Preconditions (Slide 3 of 3)

Karel must walk the row and build a hospital at each location. The beepers indicating the locations will be spaced such that there will be no overlapping or hitting walls:



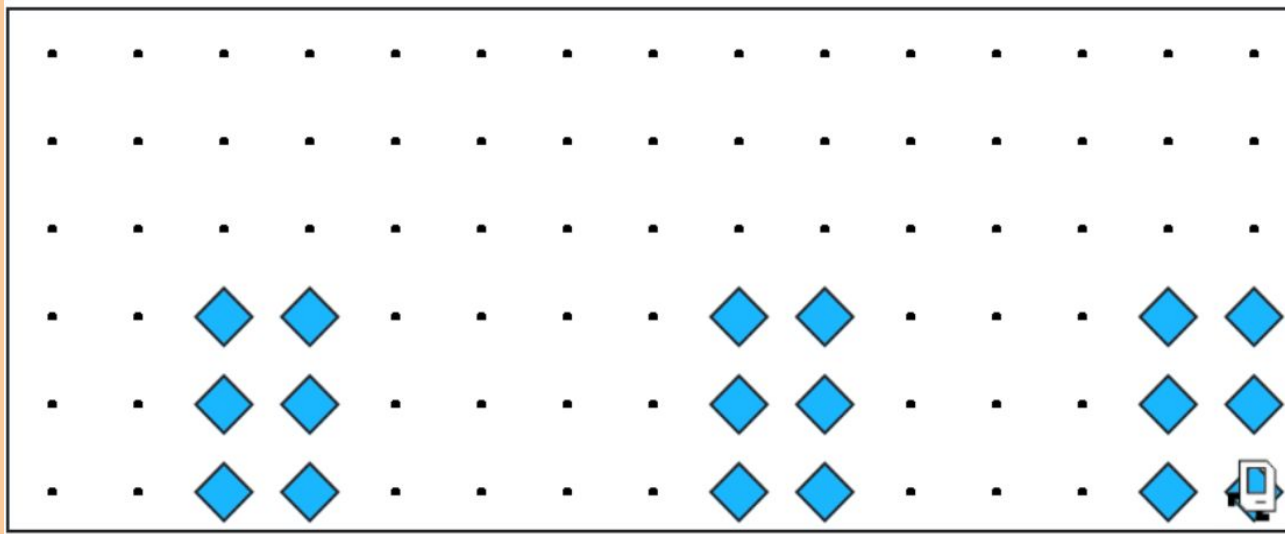
Hospital Karel Postconditions (Slide 1 of 2)

Here is the state after Karel has built the first hospital:



Hospital Karel Postconditions (Slide 2 of 2)

For the initial set of conditions shown, the result should look like this. Karel should not run into the wall after building the hospital that extends to the final corner.



Hospital Karel Hints

- Decompose the solution thoroughly
- You might want to consider whether building a hospital can be further broken down into twice building a column of three beepers. If you do this, make sure you don't end up with two beepers at the starting point for each hospital
- Be prepared to iterate over the solution. Tackle fencepost problems (such as the possibility of a hospital's being located on the leftmost corner) *after* you have tackled the general case
- Your program should work for any world that meets the preconditions; do not engineer it specifically for one world, e.g., by counting the number of hospitals and/or corners

That's all, folks!

Next up: Problem Solving