Nicholas Bowman, Sonja Johnson-Yu, Kylie Jue
CS 106AP

Assignment #1
June 24, 2019

Assignment #1: Karel the Robot

On-time deadline: 11:59 PM on Tuesday, July 2
Extended deadline: 11:59 PM on Wednesday, July 3
This assignment should be done individually.

This assignment consists of five Karel programs. There is a starter project including all of these problems on the CS106AP web site under the "Assignments" tab. Before you start on this assignment, make sure to read Handout #6 (Using Karel in PyCharm) in its entirety. When you are ready to start working on these programs, you need to:

1. Download the starter project as described in Handout #6 (Using Karel in PyCharm).
2. Edit the program files so that the assignment actually does what it's supposed to do. This will involve a cycle of coding, testing, and debugging until everything works.
3. Once you have gotten each part of the program to run correctly in the default world associated with the problem, you should make sure that your code runs properly in all of the worlds that we have provided for a given problem. Instructions on how to load new worlds for Karel to run in can be found in Handout #6 (Using Karel in PyCharm).
4. Submit your assignment on Paperless as described in Handout #8 (Submitting Assignments). Remember that you can submit more than one version of your project, but only the most recent submission will be graded. If you discover an error after you've made a submission, just fix your program and submit a new copy.

The five Karel problems to solve are described on the following pages. There is also a bonus problem described at the end that is **optional** to complete.

Please remember that your Karel programs must limit themselves to the language features described in the *Karel reference guide* (Handout #7) and covered in lectures 1-4. You may not use other features of Python (including variables, parameters, break, and return), even though the PyCharm-based version of Karel accepts them.

Problem 1 (`CollectNewspaperKarel.py`)

Your first task is to solve a simple story-problem in Karel's world. Suppose that Karel has settled into its house, which is the square area in the center of Figure 1.

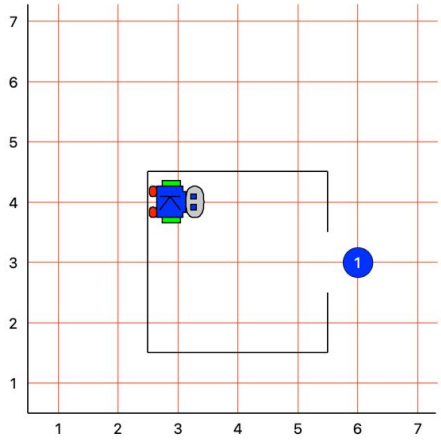

Figure 1: Karel's starting state for **`CollectNewspaperKarel.py`**

Karel starts off in the northwest corner of its house as shown in the diagram. The problem you need to solve is to get Karel to collect the newspaper. The newspaper, like all objects in Karel's world, is represented by a beeper. You must get Karel to pick up the newspaper located outside the doorway and then to return to its initial position.

This simple exercise is meant to help you get you started programming with Karel. You can assume that every part of the world looks just as it does in the diagram: the house is exactly this size, the door is always in the position shown, and the beeper is just outside the door. Thus, all you have to do is write the sequence of commands necessary to have Karel:

1. Move to the newspaper,
2. Pick it up, and
3. Return to its starting point.

Although the program is not many lines of code, it is still worth getting some practice with decomposition. In your solution, include a function for each of the steps shown in the outline.

Your program should run successfully in the following world (all worlds are located in the **`worlds/`** folder): **`CollectNewspaper.kwld`** (default)

Problem 2 (`ColumnBuilderKarel.py`)

Your second task is to help Karel rebuild some of the columns in the Main Quad that were destroyed in the 1989 earthquake! Unfortunately, Karel is unsure of where the columns should be rebuilt. Luckily, there are still some rubble piles that indicate where columns used to exist.

In this problem, Karel starts facing east in the southwest corner of the world. Avenues 2 and 3 (the two street corners directly in front of Karel's starting position) are the only two potential locations where columns may need to be rebuilt. If either or both of those avenues have rubble (represented by a single beeper at the street corners), Karel should rebuild a three-beeper-tall column. Keep in mind that the base of the column will already

exist due to the rubble. If an avenue does not have rubble (represented by an empty corner), Karel should continue on without building a column.

Two sample runs are exhibited in Figures 2 and 3 on the following page. In the first world, there are two piles of rubble; therefore, Karel builds two columns. In the second world, there is only a single pile of rubble on the third avenue so Karel only builds a single column.

You can assume that:

- There are only two possible locations with rubble (the intersections of street 1 with avenues 2 and 3).
- The world will always be of size 5-by-5.
- Karel should always finish on the fourth avenue facing east.

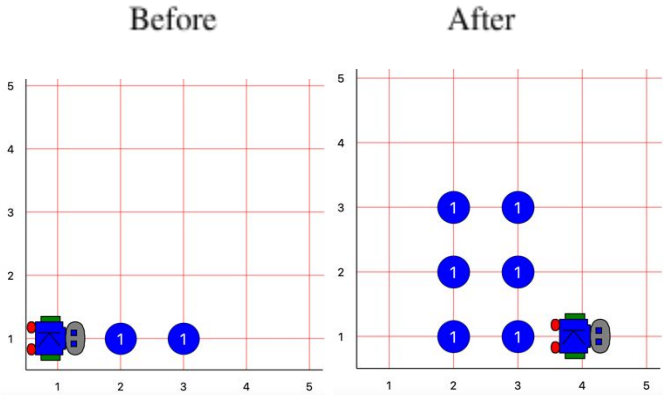

Figure 2: Example run 1 for **`ColumnBuilderKarel.py`**

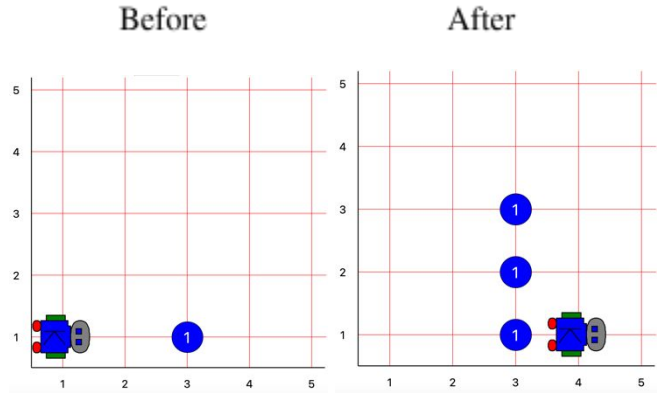

Figure 3: Example run 2 for **`ColumnBuilderKarel.py`**

Your program should run successfully in all of the following worlds (all worlds are located in the **`worlds/`** folder): **`ColumnBuilder1.kwld`** (default), **`ColumnBuilder2.kwld`**, **`ColumnBuilder3.kwld`**, **`ColumnBuilder4.kwld`**

Problem 3 (`TripleKarel.py`)

Your third task is to help Karel paint the exterior of some oddly-shaped buildings, using beepers of course! For this problem, Karel starts facing west next to a "building"

(represented by a rectangle constructed from walls) whose sides span one or more street corners. Karel's goal is to paint all of the buildings present in the world by placing beepers on three of the sides of each of the buildings.

We recommend breaking down the problem into the following steps:

1. First, Karel should paint one side of the rectangle, placing beepers on all corners that are adjacent to the wall of the building. Note that there's a boundary detail here: **the last square where Karel ends should not have a beeper on it**.
2. Next, Karel should accomplish the task of painting a single rectangle. Think about how you can use the functionality of the previous subtask to help you accomplish this goal. You may need to write a small amount of code to reposition Karel in between painting individual walls of a building.
3. Finally, the overall **`TripleKarel`** problem is just painting all three buildings in the world. Again, you may need to write a small amount of code to reposition Karel in between painting individual buildings.

Figures demonstrating the before and after stages of each of the three steps are shown below.

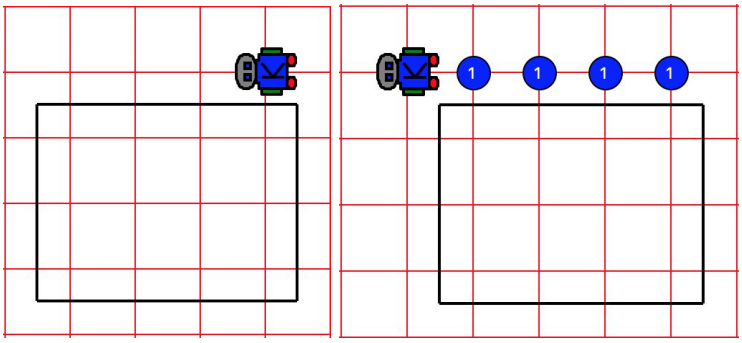

Figure 4: After you've completed the first step of **`TripleKarel.py`**, Karel should be able to paint one side of one building.

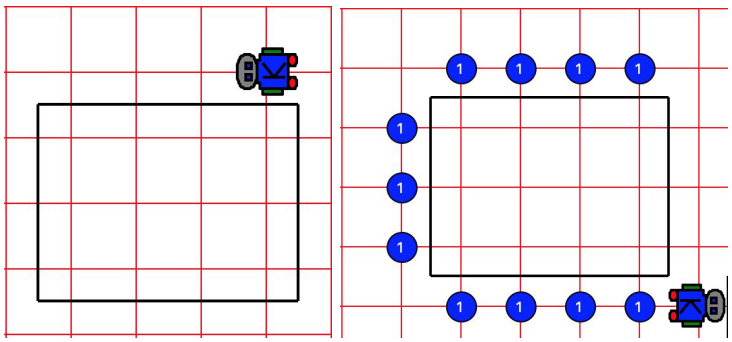

Figure 5: After you've completed the second step, Karel should be able to paint one building.

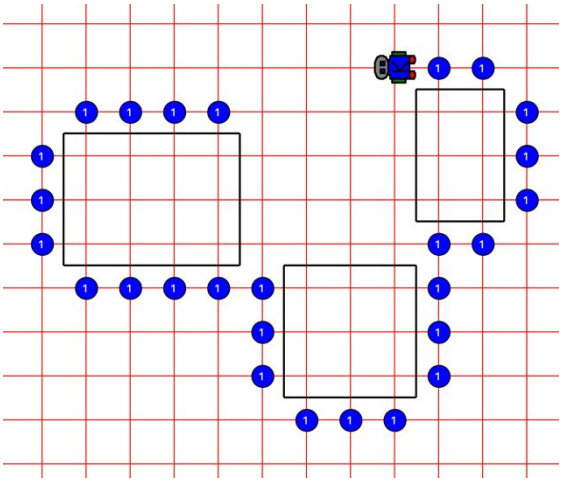

Figure 6: After you have painted the whole world, the end result should look like this.

You can assume that:

- Karel will always start facing west at the upper right corner of the leftmost building (at the position where the first beeper should be placed).
- Although buildings may be of varying sizes, there will always be exactly three of them, and their relative position to one another will always be the same (as displayed in Figure 6). If you are still confused about what assumptions you can make about the world, see the additional **`Triple`** world files we have included.

Your program should run successfully in all of the following worlds (all worlds are located in the **`worlds/`** folder): **`Triple1.kwld`** (default), **`Triple2.kwld`**, **`Triple3.kwld`**

Problem 4 (`FillPotholeKarel.py`)

Your fourth task is to help Karel fill potholes on Campus Drive. We'll start with a simpler version of the task – a segment of Campus Drive with only two potholes (pictured in **Figure 7 and 8**) – and then generalize our code to work on a road with any number of potholes. We recommend breaking the problem down into the following steps (which we'll walk you through below):

1. Write a function called **`fillPothole()`** that fills a single pothole (beginning and end states pictured in **Figure 7**).
2. Use your **`fillPothole()`** function to solve the problem for the two-pothole world with seven avenues (beginning and end states pictured in **Figure 8**).
3. Generalize your code so that it will work in any size world where there is any number of avenues and potholes can be at any avenue, not just the second and fifth (beginning and end states pictured in **Figure 9**).
4. Generalize your code so that Karel doesn't fill potholes that are already filled (beginning and end states pictured in **Figure 10**).

As always, we want to make sure that Karel works in different worlds. Maybe there are segments of Campus Drive that have more potholes than others or segments that have potholes that have already been filled in by someone else.

Specifically, we recommend starting with the default **`TwoPotholes.kwld`** file and then changing the world file to **`RegularPotholes.kwld`** for step 3 and to **`AllPotholes1.kwld`** for step 4. When finished, you should make sure that your final program runs in **`AllPotholes2.kwld`** and that it still works for **`TwoPotholes.kwld`** and **`RegularPotholes.kwld`**.

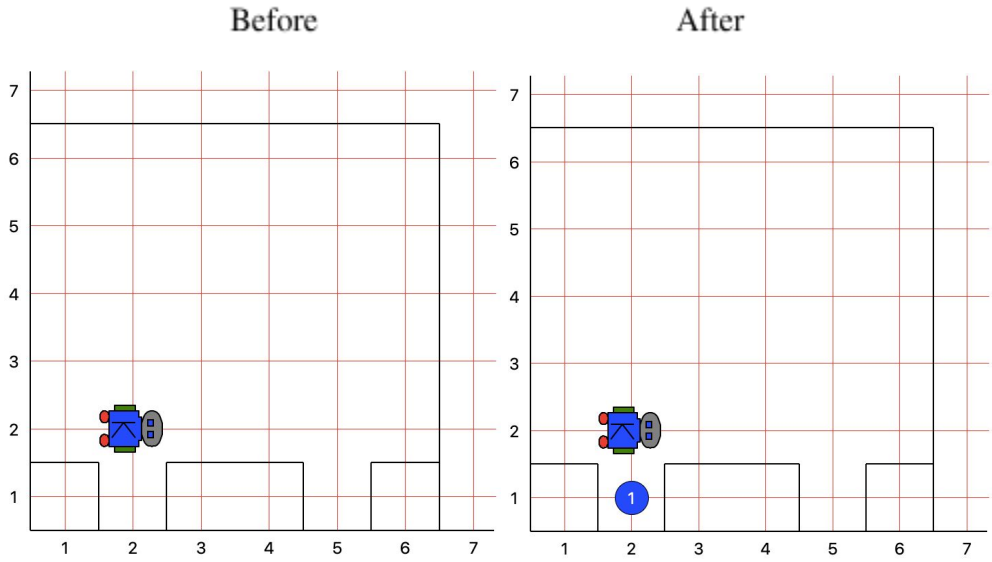

Figure 7: The beginning and end states for the **`fillPothole()`** function.

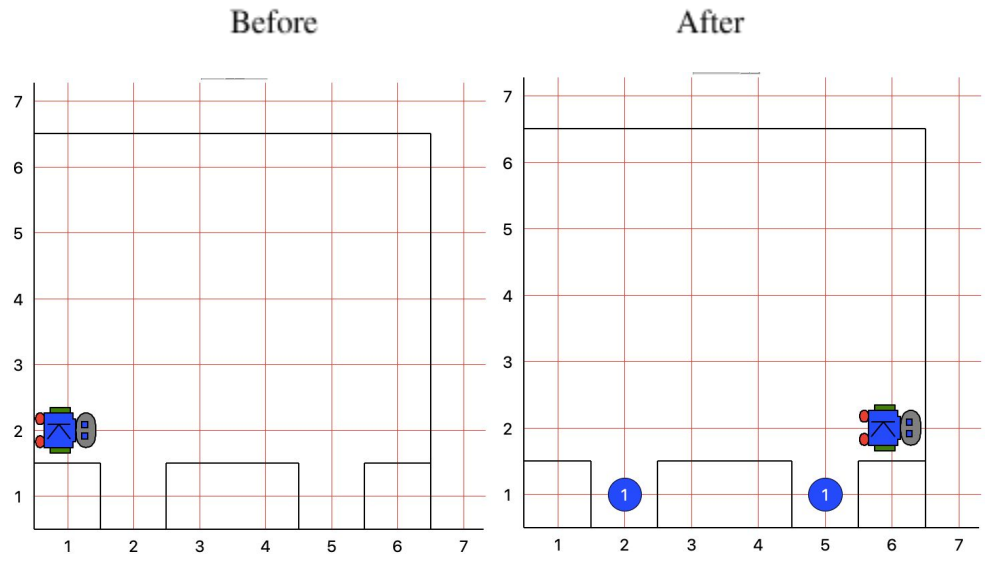

Figure 8: The beginning and end states for Karel in a simple two-pothole world.

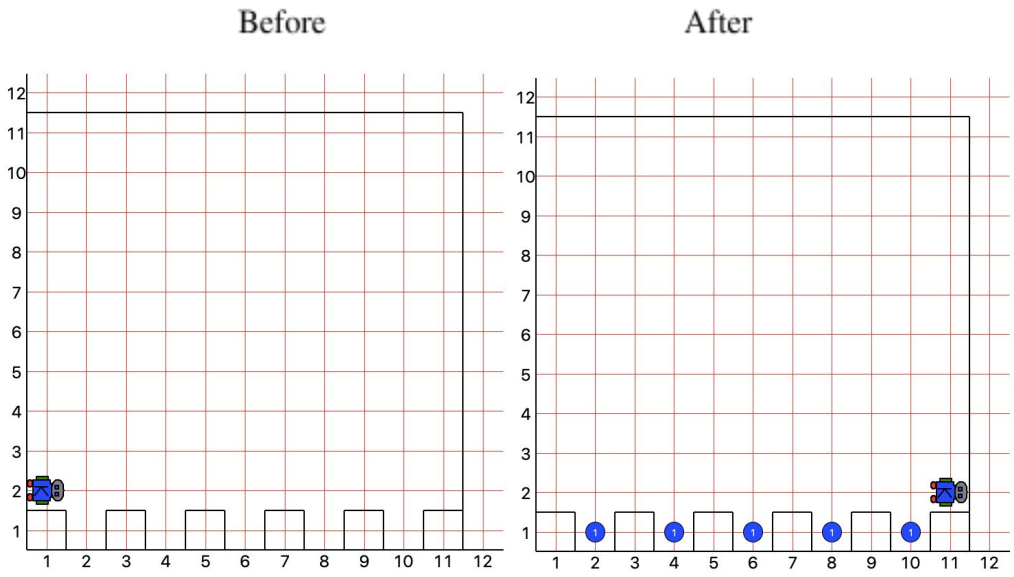

Figure 9: The beginning and end states for Karel in a world with many potholes.

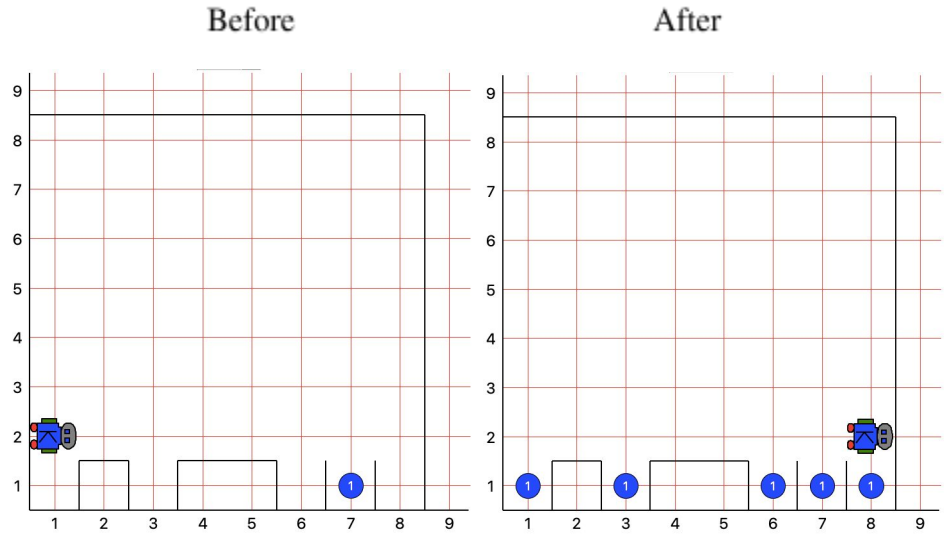

Figure 10: The beginning and end states for Karel in a world with some potholes already filled.

You should note that:

- Your final program shouldn't make any assumptions about the number of avenues in the world.
- Potholes may need to be filled in both the first and last avenues of the world.

Your program should run successfully in all of the following worlds (all worlds are located in the **`worlds/`** folder): **`TwoPotholes.kwld`** (default), **`RegularPotholes.kwld`**, **`AllPotholes1.kwld`**, **`AllPotholes2.kwld`**

Problem 5 (`CheckerboardKarel.py`)

Your fifth and final task is to get Karel to create a checkerboard pattern of beepers inside an empty rectangular world, as illustrated in **Figure 11**. (Karel's final location and the final direction it is facing at the end of the run do not matter.)

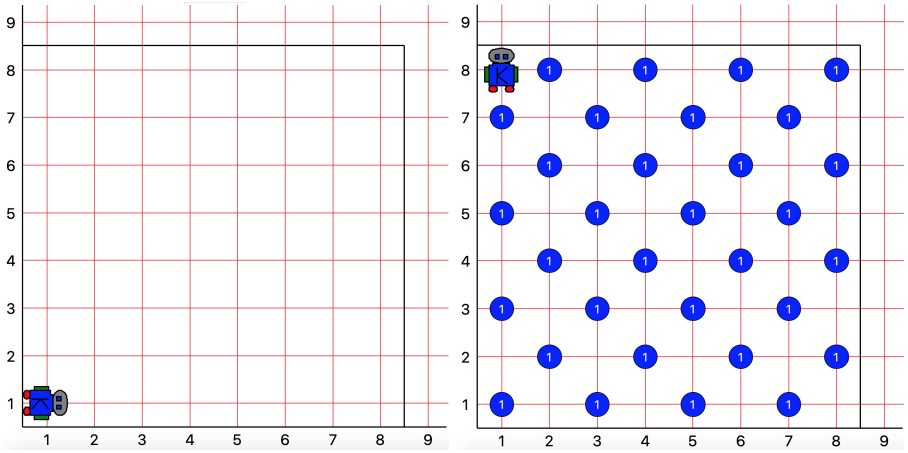

Figure 11: The beginning and end states for `CheckerboardKarel`.

This problem has a nice decomposition structure along with some interesting algorithmic issues. As you think about how you will solve the problem, you should make sure that your solution works with checkerboards that are different in size from the standard 8x8 checkerboard shown in the example above. Some examples of such cases are discussed below.

Odd-sized checkerboards are tricky, and you should make sure that your program generates the following pattern in a 5x3 world:

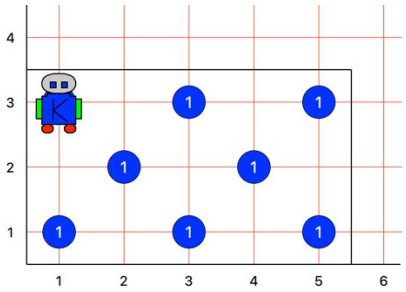

Figure 12: Karel should generate this checkerboard pattern for a 5x3 world.

Other special cases you should consider are worlds with only a single column or a single row. The starter code folder contains several sample worlds with these special cases, and you should make sure that your program works for each of them.

This problem is hard: Try simplifying your solution with decomposition. Can you checker a single row/column? Make the row/column work for different widths/heights? Once you've finished a single row/column, can you make Karel fill two? Three? All of them? Incrementally developing your program in stages helps break it down into simpler parts and is a wise strategy for attacking hard programming problems.

Your program should run successfully in all of the following worlds (all worlds are located in the `worlds/` folder): **`Checkerboard8x8.kwld`** (default), **`Checkerboard8x1.kwld`**, **`Checkerboard1x8.kwld`**, **`Checkerboard7x7.kwld`**, **`Checkerboard6x5.kwld`**, **`Checkerboard3x5.kwld`**, **`Checkerboard40x40.kwld`**, **`Checkerboard1x1.kwld`**

Problem 6 (Bonus) (`MidpointKarel.py`)

This problem is a bonus problem that is not required. Students who successfully complete this problem will be awarded a small amount of extra credit on the assignment.

As an exercise in solving algorithmic problems, program Karel to place a single beeper at the center of 1st Street. For example, if Karel starts in a 5x5 world, it should end standing on a beeper as pictured in **Figure 13**.

Note that the final configuration of the world should have only a single beeper at the midpoint of 1st Street. Karel is allowed to place as many additional beepers wherever it wants to along the way, but it must pick them all up again before it finishes.

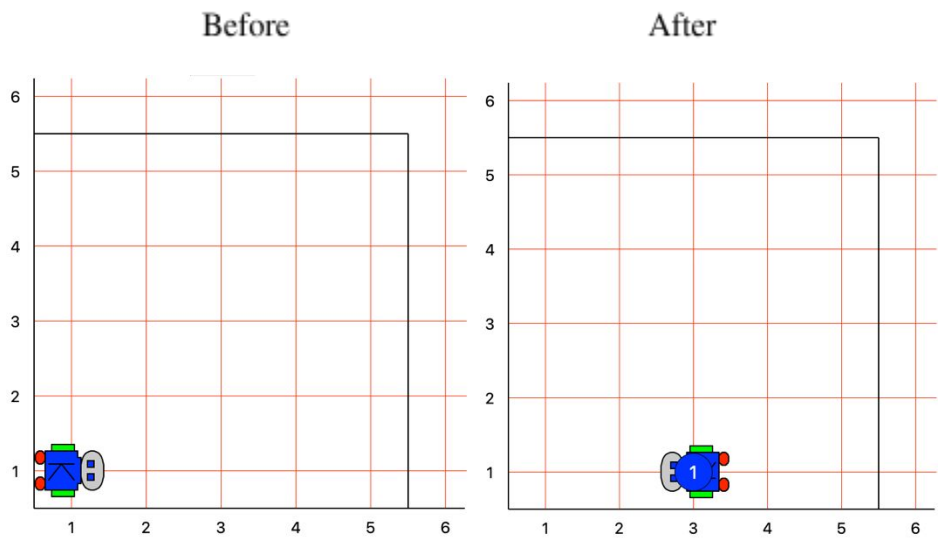

Figure 13: The beginning and end states for **`MidpointKarel`**

In solving this problem, you may count on the following facts about the world:

- Karel starts facing east at 1st Avenue and 1st Street.
- The initial state of the world includes no interior walls or beepers.
- The world need not be square, but you may assume that it is at least as tall as it is wide.

Your program, moreover, can assume the following simplifications:

- If the width of the world is odd, Karel must put the beeper in the center square. If the width is even, Karel may drop the beeper on either of the two center squares.
- It does not matter which direction Karel is facing at the end of the run.

There are many different algorithms you can use to solve this problem so feel free to be creative!

Your program should run successfully in all of the following worlds (all worlds are located in the **`worlds/`** folder): **`Midpoint5.kwld`** (default), **`Midpoint1.kwld`**, **`Midpoint2.kwld`**, **`Midpoint8.kwld`**

Submission

Following the instructions in Handout #8 (Submitting Assignments), you should submit the following files (do not include any files not included in this list!):

- **`CollectNewspaperKarel.py`**
- **`ColumnBuilderKarel.py`**
- **`TripleKarel.py`**
- **`FillPotholeKarel.py`**
- **`CheckerboardKarel.py`**

If you did the bonus portion of the assignment, you should also submit:

- **`MidpointKarel.py`**

Advice, Tips, and Tricks

With the exception of **`CollectNewspaperKarel`**, all of the Karel problems you will solve should work in a variety of different worlds that match the problem specifications. We have provided a comprehensive list of worlds at the end of each problem in which your Karel program should run successfully. All of these provided worlds exist in the **`worlds/`** folder in the starter code. These worlds are all of the files with which we will be evaluating your code while grading, **so make sure to run your code on all of the worlds we have provided.**

As mentioned in class, it is just as important to write clean and readable code as it is to write correct and functional code. A portion of your grade on this assignment (and the assignments that follow) will be based on how well-styled your code is. Make sure to follow PEP8 standards we discussed in class, and before you submit your assignment, take a minute to review your code to check for stylistic issues like those mentioned on the next page.

Have you added comments to your methods? To make your program easier to read, you can add comments before and inside your methods to make your intention clearer. Good comments give the reader a clue about what a method does and, in some cases, how it works. We recommend writing pre- and post-conditions for each function, as shown in the table below.

Not-So-Good Code	Good Code

`def fill_row_with_beepers():`
`  while front_is_clear():`
`  put_beeper()`
	`def fill_row_with_beepers():`
	`  """`
	`  Makes Karel move to the end of`

```
  move()
put_beeper()
```

```
the row, dropping a beeper
before each step it takes.

Pre-condition: None

Post-condition: Karel is facing
the same direction as before,
and every step between Karel's
old position and new position
has had a beeper added to it.
"""
while front_is_clear():
  put_beeper()
  move()
put_beeper()
```

Did you decompose the problem? There are many ways to break these Karel problems down into smaller, more manageable pieces. Decomposing the problem elegantly into smaller sub-problems will result in a small number of easy-to-read methods, each of which performs just one small task. Decomposing the problem in other ways may result in methods that are trickier to understand and test. Look over your code and check to see whether you've decomposed the problem into smaller pieces. Does your code consist of a few enormous methods (not so good), or many smaller methods (good)?

This is not an exhaustive list of stylistic conventions, but it should help you get started. As always, if you have any questions on what constitutes good style, feel free to stop by to see the course helpers in Tresidder with questions, come visit us during office hours, or email your section leader with questions!