Lesson 1: Meet Karel the Robot

Created: 2024-05-27T01:25:35.951175+10:00

1. Meet Karel

In the 1970s, a Stanford graduate student, Rich Pattis decided that it would be easier to teach the fundamentals of programming if students could somehow learn the basic ideas in a simple environment free from the complexities that characterize most programming languages. Rich designed an introductory programming environment in which students teach a robot to solve simple problems. That robot was named Karel, after the Czech playwright Karel ■apek, whose 1923 play R.U.R. (Rossum's Universal Robots) gave the word robot to the English language.Karel the Robot was quite a success. Karel has been used in introductory computer science courses all across the world and has been taught to millions of students. Many generations of students have learned how programming works with Karel, and it is still the gentle introduction to coding used at Stanford.What is Karel?Karel is a very simple robot living in a very simple world. By giving Karel a set of commands, you can direct it to perform certain tasks within its world. The process of specifying those commands is called programming. Initially, Karel understands only a very small number of predefined commands, but an important part of the programming process is teaching Karel new commands that extend its capabilities.Karel programs have much the same structure and involve the same fundamental elements as Python, a major programming language. The critical difference is that Karel's programming language is extremely small and as such the details are easy to master. Even so, you will discover that solving a problem can be challenging.By starting with Karel, you can concentrate on solving problems from the very beginning. Problem solving is the essence of programming. And because Karel encourages imagination and creativity, you can have quite a lot of fun along the way.def main(): move() for c in range(4): put_beeper() move() for c in range(3): while front_is_clear(): move() put_beeper() turn_left() move()
{"width":6,"height":4,"initialX":0,"initialY":0,"initialD":1,"initialB":1000,"cells":[]}Karel's worldKarel'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's commandsWhen Karel is shipped from the factory, it responds to a very small set of commands. CommandDescription 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. The empty pair of parentheses that appears in each of these commands is part of the common syntax shared by Karel and Python and is used to specify the…

2. Programming Karel

The simplest style of Karel program uses text to specify a sequence of built-in commands that should be executed when the program is run. Consider the simple Karel program below. The text on the left is the program. The state of Karel's world is shown on the right:# Program: FirstKarel # ---------- # The FirstKarel program defines a "main" function # with three commands. These commands cause Karel # to move forward one block, pick up a beeper and # then move ahead to the next corner. from karel.stanfordkarel import * def main(): move() pick_beeper() move(){"width":6,"height":4,"initialX":0,"initialY":0,"initialD":1,"initialB":1000,"cells":[{"x":1,"y":0,"wall":{"0":false,"1":false,"2":false,"3":false},"beepers":1,"colour":"BLANK"},{"x":2,"y":0,"wall":{"0":false,"1":true,"2":false,"3":false},"beepers":0,"colour":"BLANK"},{"x":3,"y":0,"wall":{"0":true,"1":false,"2":false,"3":true},"beepers":0,"colour":"BLANK"},{"x":4,"y":0,"wall":{"0":true,"1":false,"2":false,"3":false},"beepers":0,"colour":"BLANK"},{"x":5,"y":0,"wall":{"0":true,"1":false,"2":false,"3":false},"beepers":0,"colour":"BLANK"},{"x":3,"y":1,"wall":{"0":false,"1":false,"2":true,"3":false},"beepers":0,"c

olour":"BLANK"},{"x":4,"y":1,"wall":{"0":false,"1":false,"2":true,"3":false},"beepers":0,"colour":"BLANK"},{"x":5,"y":1,"wall":{"0":false,"1":false,"2":true,"3":false},"beepers":0,"colour":"BLANK"}]}Press the "Run" button to execute the program. Programs are typically written in a text editor or a special application called an IDE. This reader has the ability to execute programs in order to help you see how things work as you learn.The program is composed of several parts. The first part consists of the following lines:# Program: FirstKarel # ---------- # The FirstKarel program defines a "main" function # with three commands. These commands cause Karel # to move forward one block, pick up a beeper # and then move ahead to the next corner. These lines are an example of a comment, which is simply text designed to explain the operation of the program to human readers. Comments in both Karel and Python begin with the characters # and include the rest of the line. In a simple program, extensive comments may seem silly because the effect of the program is obvious, but they are extremely important as a means of documenting the design of larger, more complex programs. The second part of the program is the line:from karel.stanford import * This line requests the inclusion of all definitions from the karel.stanford library. This library contains the basic definitions necessary for writing Karel programs, such as the definitions of the standard operations move() and pick_beeper(). Because you always need access to these operations, every Karel program you write will include this import command before you write the actual program.The final part of the Karel program consists of the following function definition:def main(): move() pick_beeper() move() These lines represent the definition of a new function, which specifies the sequence of steps necessary to respond to a command. As in the case of the FirstKarel program itself, the function definition consists of two parts that can be considered separately: The first line constitutes the function header and the indented code following is the function body. If you ignore the body for now, the function definition looks like this:def main(): body of the function definition The first word in the function header, def, is part of Python's syntactic structure. It says that you are creating a new function. The next word on the header line specifies the name of the new function, which in this case is main. Defining a function means that Karel can now respond to a new command with that name. The main() command plays a special role in a Karel program. When you start a Karel program it creates a new Karel instance, adds that Karel to a world that you specify, and then issues the main() command. The eff…

3. Defining New Functions

In the last chapter we wrote a program to help Karel climb a simple ledge:# Program: FirstKarel # ---------- # Karel picks up a beeper and places it on a ledge. from karel.stanfordkarel import * def main(): move() pick_beeper() move() turn_left() move() turn_left() turn_left() turn_left() move() move() put_beeper() move(){"width":6,"height":4,"initialX":0,"initialY":0,"initialD":1,"initialB":1000,"cells":[{"x":1,"y":0,"wall":{"0":false,"1":false,"2":false,"3":false},"beepers":1,"colour":"BLANK"},{"x":2,"y":0,"wall":{"0":false,"1":true,"2":false,"3":false},"beepers":0,"colour":"BLANK"},{"x":3,"y":0,"wall":{"0":true,"1":false,"2":false,"3":true},"beepers":0,"colour":"BLANK"},{"x":4,"y":0,"wall":{"0":true,"1":false,"2":false,"3":false},"beepers":0,"colour":"BLANK"},{"x":5,"y":0,"wall":{"0":true,"1":false,"2":false,"3":false},"beepers":0,"colour":"BLANK"},{"x":3,"y":1,"wall":{"0":false,"1":false,"2":true,"3":false},"beepers":0,"colour":"BLANK"},{"x":4,"y":1,"wall":{"0":false,"1":false,"2":true,"3":false},"beepers":0,"colour":"BLANK"},{"x":5,"y":1,"wall":{"0":false,"1":false,"2":true,"3":false},"beepers":0,"colour":"BLANK"}]}Even though the FirstKarel program above demonstrates that it is possible to perform the turn_right() operation using only Karel’s built-in commands, the resulting program is not particularly clear conceptually. In your mental design of the program, Karel turns right when it reaches the top of the ledge. The fact that you have to use three turn_left() commands to do so is annoying. It would be much simpler if you could simply say turn_right() and have Karel understand this command. The resulting program would not only be shorter and easier to write, but also significantly easier to read.Defining New CommandsFortunately, the Karel programming language makes it possible to define new commands simply by including new function definitions. Whenever you have a sequence of Karel commands that performs some useful task--such as turning right--you can define a new function that executes that sequence of commands. The format for defining a new Karel function has much the same as the definition of main() in the preceding examples, which is a function definition in its own right. A typical function definition looks like this:def name(): commands that make up the body of the function In this pattern, name represents the name you have chosen for the new function. To complete the definition, all you have to do is provide the sequence of commands in the lines after the colon, which are all indented by one tab. For example, you can define turn_right() as follows:def turn_right(): turn_left() turn_left() turn_left() Similarly, you could define a new turn_around() function like this:def turn_around(): turn_left() turn_left() You can use the name of a new function just like any of Karel’s built-in commands. For example, once you have defined turn_right(), you could replace the three turn_left() commands in the program with a single call to the turn_right() function. Here is a revised implementation of the program that uses turn_right():# Program: BeeperPickingKarel # ---------- # The BeeperPickingKarel program defines a "main" # function with three commands. These commands cause # Karel to move forward one block, pick up a # beeper and then move ahead to the next corner. from karel.stanfordkarel import * def main(): move()

pick_beeper() move() turn_left() move() turn_right() move() move() put_beeper() move() def turn_right(): turn_left() turn_left() turn_left(){"width":6,"height":4,"initialX":0,"initialY":0,"initialD":1,"initialB":1000,"cells":[{"x":1,"y":0,"wall":{"0":false,"1":false,"2":false,"3":false},"beepers":1,"colour":"BLANK"},{"x":2,"y":0,"wall":{"0":false,"1":true,"2":false,"3":false},"beepers":0,"colour":"BLANK"},{"x":3,"y":0,"wall":{"0":true,"1":false,"2":false,"3":true},"beepers":0,"colour":"BLANK"},{"x":4,"y":0,"wall":{"0":true,"1":false,"…

4. Decomposition

As a way of illustrating more of the power that comes with being able to define new functions, it's useful to have Karel do something a little more practical than move a beeper from one place to another. The roadways often seem to be in need of repair, and it might be fun to see if Karel can fill potholes in its abstract world. For example, imagine that Karel is standing on the "road" shown in the following figure, one corner to the left of a pothole in the road. Karel's job is to fill the hole with a beeper and proceed to the next corner. The following diagram illustrates how the world should look after the program execution.If you are limited to the four predefined commands, the main() function to solve this problem would look like this:def main(): move() turn_left() turn_left() turn_left() move() put_beeper() turn_left() turn_left() move() turn_left() turn_left() turn_left() move() The initial motivation for defining the turn_right() function was that it was cumbersome to keep repeating three turn_left() commands to accomplish a right turn. Defining new functions has another important purpose beyond allowing you to avoid repeating the same command sequences every time you want to perform a particular task. The power to define functions unlocks the most important strategy in programming—the process of breaking a large problem down into smaller pieces that are easier to solve. The process of breaking a program down into smaller pieces is called decomposition, and the component parts of a large problem are called subproblems.As an example, the problem of filling the hole in the roadway can be decomposed into the following subproblems:Move up to the holeFill the hole by dropping a beeper into itMove on to the next cornerIf you think about the problem in this way, you can use function definitions to create a program that reflects your conception of the program structure. The main function would look like this:def main(): move() fill_pothole() move() The correspondence with the outline is immediately clear, and everything would be great if only you could get Karel to understand what you mean by fill_pothole(). Given the power to define functions, implementing fill_pothole() is extremely simple. All you have to do is define a fill_pothole() function whose body consists of the commands you have already written to do the job, like this:def fill_pothole(): turn_right() move() put_beeper() turn_around() move() turn_right() Here is the complete program. Notice how you can understand the programmer's intent simply from reading the main function. When you run the program, the line highlighting shows how a computer will execute it, step by step. However, because the program is nicely broken down we can understand it on a human thought level:from karel.stanfordkarel import * def main(): move() fill_pothole() move() # Fills the pothole beneath Karel's current position by # placing a beeper on that corner. For this function to work # correctly, Karel must be facing east immediately above the # pothole. When execution is complete, Karel will have # returned to the same square and will again be facing east. def fill_pothole(): turn_right() move() put_beeper() turn_around() move() turn_right() # Turns Karel 90 degrees to the right. def turn_right(): turn_left() turn_left() turn_left() # Turns Karel around 180 degrees. def turn_around(): turn_left() turn_left(){"width":5,"height":4,"initialX":0,"initialY":1,"initialD":1,"initialB":1000,"cells":[{"x":0,"y":0,"wall":{"0":true,"1":true,"2":false,"3":false},"beepers":0,"colour":"BLANK"},{"x":1,"y":0,"wall":{"0":false,"1":true,"2":false,"3":true},"beepers":0,"colour":"BLANK"},{"x":2,"y":0,"wall":{"0":true,"1":false,"2":false,"3":true},"beepers":0,"colour":"BLANK"},{"x":3,"y":0,"wall":{"0":true,"1":false,"2":false,"3":false},"beepers":0,"colour":"BLANK"},{"x":4,"y":0,"wall":{"0":true,"1":false,"2":false,"3":false},"beepers":0,"colour":"BLANK"},{"x":0,"y":…

5. For Loops

One of the things that computers are especially good at is repetition. How can we convince Karel to execute a block of code multiple times? To see how repetition can be used, consider the task of placing 42 beepers:Basic For LoopSince you know that there are exactly 42 beepers to place, the control statement that you need is a for loop, which specifies that you want to repeat some operation a fixed number of times. The structure of the for statement appears complicated primarily because it is actually much more powerful than anything Karel needs. The only version of the for syntax that Karel uses is:for i in range(count): statements to be repeated We will go over all the details of the for loop later in the class. For now you should read this line as a way to express, "repeat the statements in the function body count times." We can use this new for loop to place 42 beepers by replacing count with 42 and putting the command put_beeper() inside of the for loop code block. We call commands in the code block the body:# Program: PlaceManyBeepers # ---------- # Places 42 beepers using a for loop from karel.stanfordkarel import * def main(): move() # Repeat put_beeper many times for i in range(42): put_beeper() move(){"width":4,"height":4,"initialX":0,"initialY":0,"initialD":1,"initialB":1000,"cells":[]}The code

above is editable. Try to change it so that it places only 15 beepers.Matching Postconditions with PreconditionsThe previous example gives the impression that a for loop repeats a single line of code. However the body of the for loop (the statements that get repeated) can be multiple lines. Here is an example of a program that puts a beeper in each corner of a world:# Program: CornerBeepers # ---------- # Places one beeper in each corner from karel.stanfordkarel import * def main(): # Repeat the body 4 times for i in range(4): put_beeper() move() move() move() turn_left(){"width":4,"height":4,"initialX":0,"initialY":0,"initialD":1,"initialB":1000,"cells":[]}Pay very close attention to the way that the program flows through these control statements. The program runs through the set of commands in the for loop body one at a time. It repeats the body four times.Perhaps the single most complicated part of writing a loop is that you need the state of the world at the end of the loop (the postcondition) to be a valid state of the world for the start of the loop (the precondition). In the above example the assumptions match. Good times. At the start of the loop, Karel is always on a square with no beepers facing the next empty corner. What if you deleted the turn_left() at the end of the loop? The postcondition at the end of the first iteration would no longer satisfy the assumptions made about Karel facing the next empty corner. The code is editable. Try deleting the turn_left() command to see what happens!Nested LoopsTechnically the body of a for loop can contain any control flow code, even other loops. Here is an example of a for loop that repeats a call to a function which also has a for loop. We call this a "nested" loop. Try to read through the program, and understand what it does, before running it:# Program: CornerFiveBeepers # ---------- # Places five beepers in each corner from karel.stanfordkarel import * def main(): # Repeat once for each corner for i in range(4): put_five_beepers() move_to_next_corner() # Reposition karel to the next corner def move_to_next_corner() : move() move() move() turn_left() # Places 5 beepers using a for loop def put_five_beepers() : for i in range(5): put_beeper(){"width":4,"height":4,"initialX":0,"initialY":0,"initialD":1,"initialB":1000,"cells":[]}Based on work by Chris Piech and Eric Roberts at Stanford University.https://compedu.stanford.edu/karel-reader/docs/python/en/intro.html

6. While Loops

The technique of defining new functions, and defining for loops—as useful as they are—does not actually enable Karel to solve any new problems. Every time you run a program it always does exactly the same thing. Programs become a lot more useful when they can respond differently to different inputs.As an example. Let's say you wanted to write a program to have Karel move to a wall. But you don't simply want this program to work on one world with a fixed size. You would like to write a single program that could work on any world.# Program: MoveToWall # ---------- # Uses a "while" loop to move Karel until it hits # a wall. Works on any sized world. from karel.stanfordkarel import * # the program starts with main def main(): # call the move to wall function move_to_wall() # this is a very useful function def move_to_wall(): # repeat the body while the condition holds while front_is_clear(): move(){"width":7,"height":7,"initialX":0,"initialY":0,"initialD":1,"initialB":1000,"cells":[]}Try changing the world by clicking the numbers above the world. For any sized world, Karel will move until it hits a wall. Notice that this feat can not be accomplished using a for loop. That would require us to know the size of the world at the time of programming.Basic While LoopIn Karel, a while loop is used to repeat a body of code as-long-as a given condition holds. The while loop has the following general form:while test: statements to be repeated The control-flow of a while loop is as follows. When the program hits a while loop it starts repeating a process where it first checks if the test passes, and if so runs the code in the body.When the program checks if the test passes, it decides if the test is true for the current state of the world. If so, the loop will run the code in the body. If the test fails, the loop is over and the program moves on.When the program runs the body of the loop, the program executes the lines in the body one at a time. When the program arrives at the end of the while loop, it jumps back to the top of the loop. It then rechecks the test, continuing to loop if it passes. The program does not exit the loop until it gets to a check, and the test fails.Karel has many test statements, and we will go over all of them in the next chapter. For now we are going to use a single test statement: front_is_clear() which is true if there is no wall directly in front of Karel.Fencepost BugLet's modify our program above to make it more interesting. Instead of just moving to a wall, have Karel place a line of beepers, one in each square. Again we want this program to work for a world of any size:# Program: BeeperLineBug # ---------- # Uses a while loop to place a line of beepers. # This program works for a world of any size. # However, because each world requires one fewer # moves than put_beepers it always misses a beeper. from karel.stanfordkarel import * def main(): # Repeat until karel faces a wall while front_is_clear(): # Place a beeper on current square put_beeper() # Move to the next square move(){"width":7,"height":7,"initialX":0,"initialY":0,"initialD":1,"initialB":1000,"cells":[]}That looks great. Except for one problem. On every world Karel doesn't place a beeper on the last square of the line (look closely). When Karel is on the last square, the program does not execute the body of the loop because the test no longer passes -- Karel is facing a wall. You might be tempted to try switching the order of the body

so that Karel moves before placing a beeper. The code is editable so go try it!There is a deeper problem that no rearrangement of the body can solve. For the world with 7 columns, Karel needs to put 7 beepers, but should only move 6 times. Since the while loop executes both lines when a test passes, how can you get the program to execute one command one more time than the other?The bug in this program is an example of a programming problem called a fencepost e…

7. If Statements

The final core programming control-flow construct to learn are conditional statements (if and if/else).Basic ConditionalsAn if/else statement executes an "if" code-block if and only if the provided test is true for the state of the world at the time the program reaches the statement. Otherwise the program executes the "else" code-block.if test: if code-block else: else code-block To get a sense of where conditional statements might come in handy, let's write a program that has Karel invert a line of beepers. If a square previously had a beeper, Karel should pick it up. If a square has no beeper, Karel should put one down.# Program: UpAndDown # ---------- from karel.stanfordkarel import * def main(): while front_is_clear(): invert_beeper() move() # Prevent a fencepost bug invert_beeper() # Picks up a beeper if one is present # Puts down a beeper otherwise def invert_beeper(): # An if/else statement if beepers_present(): pick_beeper() else: put_beeper(){"width":8,"height":8,"initialX":0,"initialY":0,"initialD":1,"initialB":1000,"cells":[{"x":1,"y":0,"wall":{"0":false,"1":false,"2":false,"3":false},"beepers":1,"colour":"BLANK"},{"x":3,"y":0,"wall":{"0":false,"1":false,"2":false,"3":false},"beepers":1,"colour":"BLANK"},{"x":4,"y":0,"wall":{"0":false,"1":false,"2":false,"3":false},"beepers":1,"colour":"BLANK"},{"x":6,"y":0,"wall":{"0":false,"1":false,"2":false,"3":false},"beepers":1,"colour":"BLANK"}]}Note that an if statement does not need to have an else block -- in which case the statement operates like a while loop that only executes one time:if test: if code-block ConditionsThat last example used a new condition. Here is a list of all of the conditions that Karel knows of: TestOppositeWhat 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? Putting it all togetherCongrats! You now know all of the core programming control-flow blocks. While you learned them with Karel, methods, while loops, for loops, if/else statements work in the same way in almost all major languages, including Python.Now that you have the building blocks you can put them together to build solutions to ever more complex problems. To a large extent, programming is the science of solving problems by computer. Because problems are often difficult, solutions—and the programs that implement those solutions—can be difficult as well. In order to make it easier for you to develop those solutions, you need to adopt a methodology and discipline that reduces the level of that complexity to a manageable scale.Based on work by Chris Piech and Eric Roberts at Stanford University.https://compedu.stanford.edu/karel-reader/docs/python/en/intro.html

8. Code

Write and test any code here!from karel.stanfordkarel import * def main(): # your code here... move()