Chapter 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.

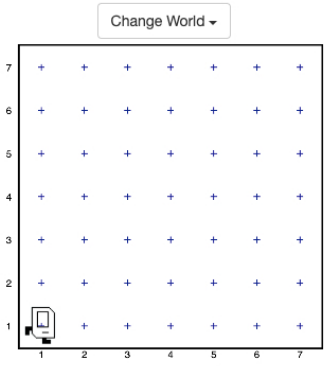
As an example, let's say you wanted to write a program to have Karel move to a wall. But you don't simply want this program to work on one world with a fixed size. You would like to write a single program that could work on any world.

```
# File: MoveToWall.py
# ---------------------------
# 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()
```

▶ Run Program | Show Text Descriptions



Try changing the world by clicking the "Change World" dropdown above the world. For any sized world, Karel will move until it hits a wall. Notice that this feat can not be acomplished using a `for` loop. That would require us to know the size of the world at the time of programming.

Basic While Loop

In 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.

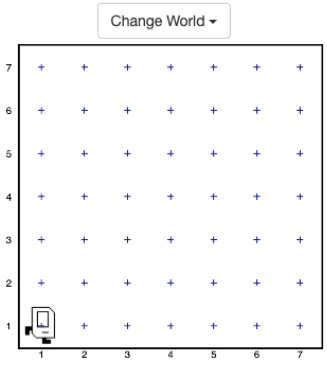
Fencepost Bug

Let'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:

```
# File: BeeperLineBug.py
# ---------------------------
# 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 *

# program starts at main
def main():
   # repeats until karel faces a wall
   while front_is_clear():
      # place a beeper on current square
      put_beeper()
      # move to the next square
      move()
```

▶ Run Program | Show Text Descriptions

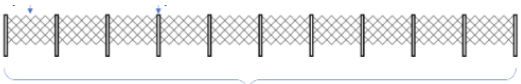


That looks great. Except for one problem. On every world Karel doesn't place a beeper on the last square of the line (look closely). When Karel is on the last square, the program does not execute the body of the loop because the test no longer passes -- Karel is facing a wall. You might be tempted to 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 error**. The name comes from the fact that if you want to build a fence made of panels which have one fence post on either size, the number of fence posts is always one greater than the number of panels. How many fence posts, for example, do you need to build a fence with 10 panels? The answer is 11, as illustrated by the following diagram:

"panel" "fencepost"

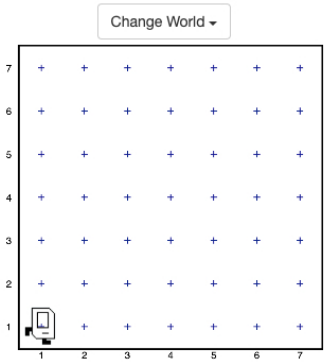


10 panels, 11 fenceposts

Once you discover it, fixing this bug is actually quite easy. Before Karel stops at the end of the world, all that the program has to do is place a final beeper:

```
# File: BeeperLine.py
# -------------------------------
# Uses a while loop to place a line of beepers.
# This program works for a world of any size.
from karel.stanfordkarel import *

# program starts at main
def main():
    # repeats until karel faces a wall
    while front_is_clear():
        # place a beeper on current square
        put_beeper()
        # move to the next square
        move()
    # solves the fencepost bug
    put_beeper()
```



▶ Run Program | Show Text Descriptions

Next Chapter