Learning Programming with Robots*

Edited by Otfried Cheong

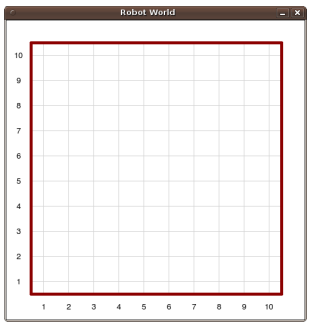
1 Robot world

We will learn programming by solving problems faced by a robot living in a small grid-like world. Our robot is capable of only four basic actions: moving one step forward, turning left, picking up and putting down beepers. Through the magic of programming, you will learn to have him combine those four basic actions in order to perform very complicated tasks.

Before we learn how to do all this, let's have a look at the robot's world. Open a Python interpreter window, and type in this code:

```
from cs1robots import *
create_world()
```

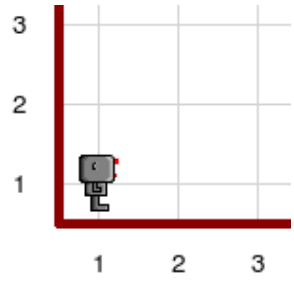
A new window will open showing a new robot world, like this:



This robot world consists of 10 vertical *avenues* and 10 horizontal *streets*. The robot can be placed at any intersection of an avenue with a street. So let's add a robot, by typing in the following line:

```
hubo = Robot()
```

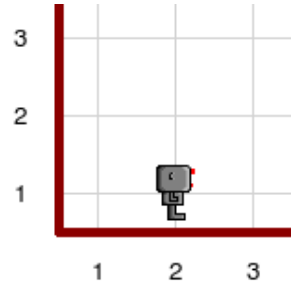
A robot will appear at the intersection of avenue 1 and street 1:



We can now start giving instructions to the robot. In Python, you give instructions to a robot by writing the robot's name, a dot, and the command:

```
hubo.move()
```

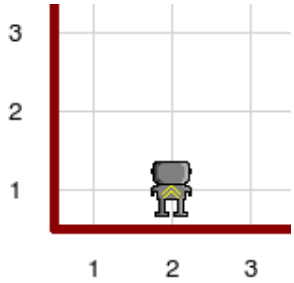
And immediately `hubo` will move one step forward:



Let's try a different instruction:

```
hubo.turn_left()
```

Our robot will turn to face northwards:



You can now control your robot by entering commands one by one—the robot executes each command as soon as you have entered it.

*These lecture notes are edited versions of the RUR-PLE lessons by André Roberge, which can be found at `http://rur-ple.sourceforge.net/`.

2 First program

But controlling a robot by typing in commands on a keyboard is not programming. "Programming" a robot means to design a whole sequence of instructions (an *algorithm*) for the robot, which is then executed automatically by the robot to solve some task.

Let's try this: Type the following text and save it as a file *square.py*:

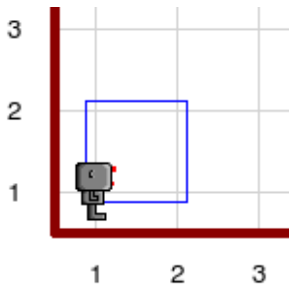
```
from cs1robots import *
create_world()
hubo = Robot()
hubo.move()
hubo.turn_left()
hubo.move()
hubo.turn_left()
hubo.move()
hubo.turn_left()
hubo.move()
hubo.turn_left()
```

Let your Python interpreter execute this code. A robot world window will open, and you will see the robot walking around a little square. Everything will happen quite fast, however, and it will be hard to see what is going on.

The first thing we can do is to turn on a *trace* for our robot, as follows:

```
from cs1robots import *
create_world()
hubo = Robot()
hubo.set_trace("blue")
...
```

This is much better already, as we can see precisely what the robot did:



If you also want to be able to observe the robot's movement in more detail, we can ask him to make a little break after each move:

```
from cs1robots import *
create_world()
hubo = Robot()
hubo.set_trace("blue")
hubo.set_pause(2)
...
```

The number in parentheses is the length of the break in seconds.

3 Bugs!

What happens when you mistype a command:

```
hubo.Move()
```

(Yes, Python distinguishes lower-case and upper-case letters.) Make sure you try it to see exactly what happens. In the future, if you see the same kind of complaint, it might help you discover what's wrong with your program. In other words, you will find how to recognize one kind of *bug*.

The following program demonstrates a different bug:

```
from cs1robots import *
create_world()
hubo = Robot()
hubo.move()
hubo.turn_left()
hubo.turn_left()
hubo.turn_left()
hubo.move()
```

What happens? Robots, like humans, cannot walk through walls. If you make him hit a wall, your program terminates.

4 Write programs for humans

Beginning programmers tend to think about programming as talking to the computer, and feel happy as long as the computer understands them. You should, however, strive to write programs that make it easy for other *people* to read your program and understand what it does. This takes a lot of practice and usually requires a fair bit of thinking. In the end, such programs are more likely to be correct, and can be maintained much more easily. (In practice, programs are never finished—there always are new features that need to be added or behavior that needs to be changed.)

We can use a trick of writing notes meant for other humans only (and not for the computer) inside the program. We call these notes *comments*.

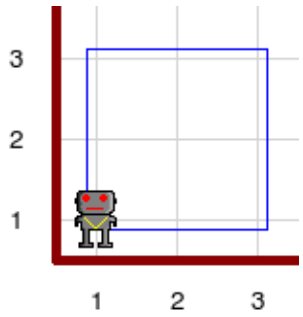
There are a few ways to write comments in a program. I'm going to teach you the simplest way used in Python: If a line starts with a hash symbol `#`, then this line is a comment and will be ignored by the Python interpreter:

```
# My first program
```

5 Turning left and turning right

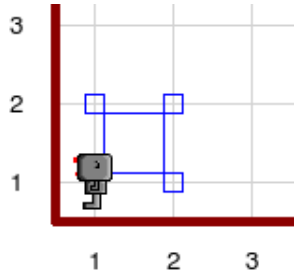
We already saw that our robot can turn left, as long as it is by exactly 90 degrees. There is no other steering mechanism—our robots are rather cheap.

Task: (Square) Write a program that makes `hubo` walk around a *bigger square* of size two, like this:



So how can we make the robot turn *right* instead of left? The only possible way to do this is by executing three left-turns. The following program let's our robot walk around a square *in clock-wise direction*:

```
from cs1robots import *
create_world()
hubo = Robot()
hubo.turn_left()
hubo.move()
# make a right turn
hubo.turn_left()
hubo.turn_left()
hubo.turn_left()
hubo.move()
# make a right turn
hubo.turn_left()
hubo.turn_left()
hubo.turn_left()
hubo.move()
# make a right turn
hubo.turn_left()
hubo.turn_left()
hubo.turn_left()
hubo.move()
```



It becomes quite tedious to write and read this code. This is because we repeat ourselves; in other words, the same sequence of instructions appears in many different places in the program. One basic rule of programming is: *Do not repeat yourself!*

In Python, one can give a simple name to a series of instructions. For example, we can define a `turn_right` command:

```
from cs1robots import *
create_world()
hubo = Robot()
hubo.turn_left()

def turn_right():
  hubo.turn_left()
  hubo.turn_left()
  hubo.turn_left()

hubo.move()
turn_right()
hubo.move()
turn_right()
hubo.move()
turn_right()
hubo.move()
```

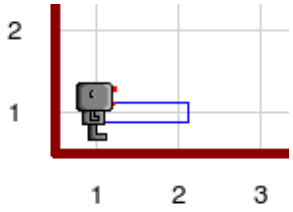
A function definition starts with the Python *keyword* `def`, the name of the function, a pair of parentheses, and a colon. The *body* of the function are the instructions that are executed when the function is *called*. Each line in the body has to be *indented* by the exact same amount (it doesn't matter how much—I always use two spaces).

A *keyword* like `def` is a special word that can be used in Python only to define a new function (it is for instance not allowed to define a function named "def").

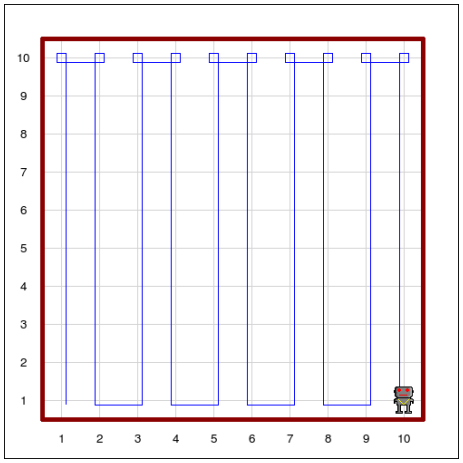
Task: (StepBack) Write a function `step_back` that makes `hubo` go *backward* one step, so that

```
hubo.move()
step_back()
```

will leave the robot at exactly the same position and orientation it started with:



Task: (ZigZag1) Write a program *zigzag.py* that makes your robot visit the entire world in a zigzag fashion:



Use functions to avoid repetitions and to make your code as clear as possible.

6 Beepers

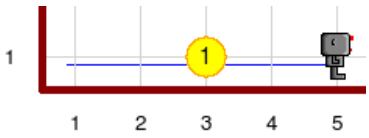
Beepers are small objects (drawn fairly big on the screen) that make a small beeping sound when they are turned on. A robot can only hear a beeper if he is standing right on top of them, on the same corner. He can pick them up and carry them in his pockets, or he can put them down.

You can instruct the robot to pick up a beeper through the `pick_beeper()` command, or to put some down, through the `drop_beeper()` command. If you ask a robot to pick up beepers where there are none, or to put down beepers if he doesn't carry any, you will get an error message and your program will terminate.

The following small program will drop a beeper at the intersection of avenue 3 and street 1:

```
from cs1robots import *
create_world()
hubo = Robot(beepers = 1)
hubo.set_trace("blue")
hubo.move()
hubo.move()
hubo.drop_beeper()
hubo.move()
hubo.move()
```

Note that we had to tell Python to create a robot that already carries one beeper. (Try what happens without this.)



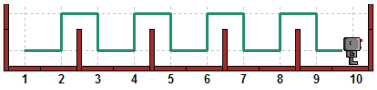
7 Worlds

So far, our robot has lived in a rather boring, empty world. We can let him play in more interesting worlds containing walls and beepers, by loading such a world instead of the `create_world()` call:

```
from cs1robots import *
load_world("worlds/hurdles1.wld")
hubo = Robot()
```

(You can also call `load_world()` without giving a filename. Then Python will let you choose a file when the program is run.)

Task: (Hurdles1) Hubo has entered a hurdles race, with obstaces as in the world file *hurdles1.wld*. Write a program that have him follow the path indicated below in his way to the finish line. He has finished when he has picked up the beeper.



Avoid repetition by defining useful functions.

Maybe you would rather like to build a more interesting world. You can do that using the `edit_world()` function:

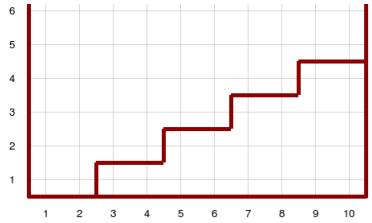
```
from cs1robots import *
create_world(avenues = 20, streets = 5)
edit_world()
save_world("myworld.wld")
```

The `edit_world` function places your world in an editable state. You can click on the world to add and remove walls. If you click with the left mouse button on an intersection, a beeper is placed there. You can remove beepers with the right mouse button. Press the *Return* key when you are done editing. Your program will then continue and save your world. (Again, you can call `save_world()` without giving a filename. Then Python will let you choose a file when the program is run.)

Note that you can only create or load a world *once* in a Python program or from the Python interpreter. To load another world, you have to rerun your program, or restart the interpreter.

8 Top-down design

Let's solve the following problem: Hubo delivers newspapers in his local neighborhood. Have him climb the stairs to the front door of a house, drop the newspaper (represented by a beeper) on the top step, and return to his starting point as illustrated below. The world file is *newspaper.wld*:



If we just type in commands to accomplish this task, we would end up with a sequence of more than 50 commands. This requires a fair bit of typing—and the more typing one does, the more errors can occur!

A better way to approach such a problem is to start with an *outline*:

- Climb (up) four stairs.
- Drop the newspaper.
- Turn around.
- Climb (down) four stairs.

Let's write this outline in a Pythonic form:

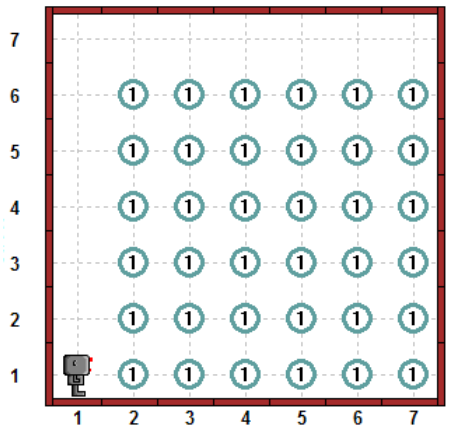
```
climb_up_four_stairs()
hubo.drop_beeper()
turn_around()
climb_down_four_stairs()
```

This is easy to read, and instead of one large problem we now need to solve a few small problems: How to climb up four stairs? We can decompose these smaller problems even further:

```
def climb_up_one_stair():
  hubo.turn_left()
  hubo.move()
  turn_right()
  hubo.move()
  hubo.move()

def climb_up_four_stairs():
  climb_up_one_stair()
  climb_up_one_stair()
  climb_up_one_stair()
  climb_up_one_stair()
```

This approach to solving a given problem is called *top-down design*, because we started at the top of the problem, with an outline for a solution, and then focused on each part of the solution one-by-one.

Task: (Harvest1) It's harvest time! Have the robot pick up all the carrots (represented by beepers) in this garden. The world file is *harvest1.wld*. Use top-down design.



9 Repetitions

We learnt the rule: *Do not repeat yourself!* We used functions for sequences of instructions that our robot needs to use repeatedly. But if we look at this function:

```
def turn_right():
  hubo.turn_left()
  hubo.turn_left()
  hubo.turn_left()
```

It still contains a lot of repetition: The same instruction is used three times. We can rewrite this using a *for-loop*:

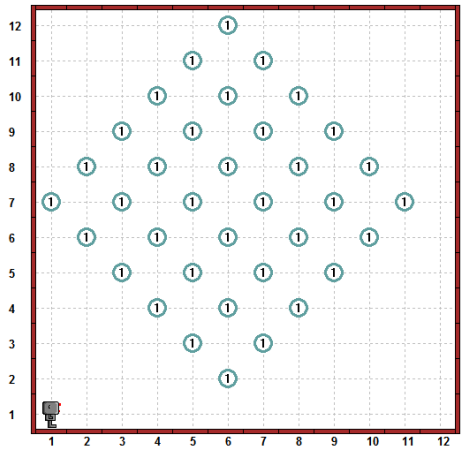
```
def turn_right():
  for i in range(3):
    hubo.turn_left()
```

We will see later what this means exactly, but for the moment it's enough to know that `hubo.turn_left()` will be repeated exactly three times.

Task: (Hurdles2) Redo the *Hurdles1* task above using for-loops.

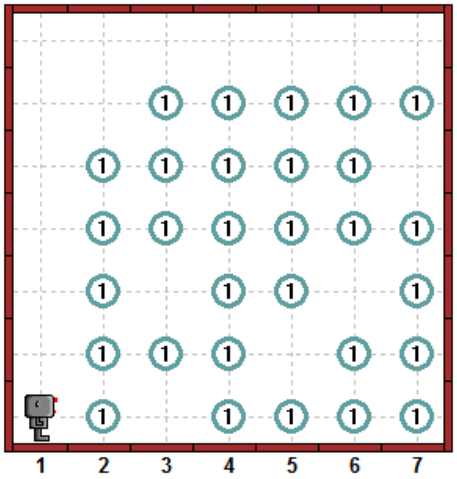
Task: (Harvest2) Redo the *Harvest1* task above using for-loops.

Task: (Harvest3) Some problem as *Harvest2*, but for the *harvest2.wld* world file:



10 If only the robot could decide...

How would we solve the harvesting problem on a world like this? (The world file is *harvest3.wld*.)



It would be nice if we could just run the progrem we wrote in task *Harvest2*. But it doesn't work, because calling `pick_beeper()` when there is no beeper causes an error.

If only `hubo` could check *if there is a beeper*, and pick it up only when a beeper is really there...

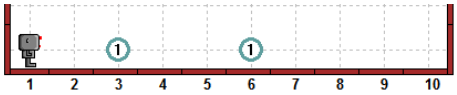
The Python *keyword* `if` does exactly this:

```
if hubo.on_beeper():
  hubo.pick_beeper()
```

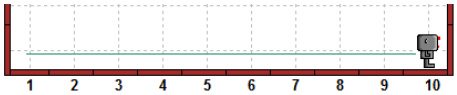
Let's look at the meaning of the above code:

- The instruction `if` implies that some condition, whose value is *true* or *false*, is going to follow;
- `hubo.on_beeper()` is a condition (or test) which is *true* if `hubo` is on top of a beeper and *false* otherwise;
- The colon (:) precedes the series of instructions that the robot must follow if the condition is *true*;
- The series of instructions to follow in that case is indented, just like we had in the case of function definitions.

Let's look at a simple example. Suppose we want the robot to take 9 steps, picking up any beepers that are there along the way. (We suppose that there can be at most one beeper at a given spot.) For example, the starting position might be like the following:



and we want the final position to be:



So, we want to:

- Take a step forward;
- Check to see if there is a beeper;
- Pick up the beeper if there is one; otherwise ignore and keep going;

repeating the above steps 9 times. Here is how we can do this:

```
def move_and_pick():
  hubo.move()
  if hubo.on_beeper():
    hubo.pick_beeper()

for i in range(9):
  move_and_pick()
```

Task: (Harvest4) Modify your program for the *Harvest2* task so that it works for the world *harvest3.wld*. Note that the new program should *also automatically* work for the original *harvest1.wld* world. Try it!

To be able to harvest in the fall, you have to put seeds in the ground in the spring. Can we write a program that will put seed potatoes in the ground, but skip any spot where there already is a potato? In other words, starting with the world *harvest3.wld*, we should end up with the world of *harvest1.wld*!

To do this, `hubo` needs to drop a beeper if there is *no* beeper at the current location. The following code will do this:

```
if not hubo.on_beeper():
  hubo.drop_beeper()
```

The Python word `not` inverts the sense of the condition: If `hubo.on_beeper()` is true, then `not hubo.on_beeper()` is false. If `hubo.on_beeper()` is false, then `not hubo.on_beeper()` is true. This is the simplest example of a Python *expression*.

Task: (Plant1) Write a program that will plant potatoes so that the field will look like *harvest1.wld* at the end. It should skip any spot where there already is a potato. (Note that you have to create your robot with sufficiently many beepers by using `hubo = Robot(beepers=36)`. Try your program with an empty world and with the worlds *harvest1.wld* and *harvest3.wld*.

11 What else?

In addition to being able to find out if he is standing next to one or more beepers, a robot can see if there is a wall in front of him, blocking his way. He can also turn his head to his left or his right and see if there is a wall there. You can ask him to have a look with the following tests:

```
hubo.front_is_clear()
hubo.left_is_clear()
hubo.right_is_clear()
```

Let's use the first one to have the robot explore his world. We will have him follow the boundary of his world by asking him to move forward, if there is no wall in front of him, and turn left otherwise. The following simple program is the basis of what is needed:

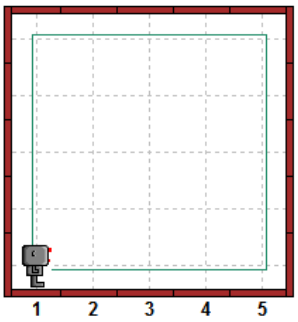
```
if hubo.front_is_clear():
  hubo.move()
else:
  hubo.turn_left()
```

We learnt a new Python keyword here: `else`. It introduces an alternative sequence of instructions. In this example, if there is no wall in front of `hubo`, then `hubo.front_is_clear()` is true, and the sequence after the `if` line is executed. If there is a wall, then `hubo.front_is_clear()` is false, and the sequence after the `else` line is executed. Do not forget the colon after `else`, and watch out for the correct indentation!

Let's repeat this simple conditional instruction many times to go around the world:

```
from cs1robots import *
create_world(avenues = 5, streets = 5)

hubo = Robot()
hubo.set_trace("blue")

def move_or_turn():
  if hubo.front_is_clear():
    hubo.move()
  else:
    hubo.turn_left()

for i in range(20):
  move_or_turn()
```

On a small world this gives the following end result:

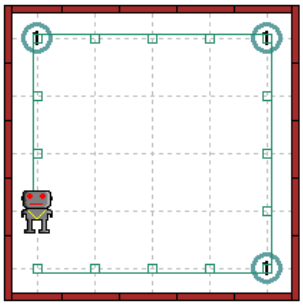


We can make this more interesting by doing a "dance" if we can move forward, and dropping a beeper if we have to turn. The following program does just that, going only partly around the world:

```
hubo = Robot(beepers = 10)

def dance():
  for i in range(4):
    hubo.turn_left()

def move_or_turn():
  if hubo.front_is_clear():
    dance()
    hubo.move()
  else:
    hubo.turn_left()
    hubo.drop_beeper()

for i in range(18):
  move_or_turn()
```

Notice how the instructions `dance()` and `hubo.move()` are aligned after the `if` statement and indented from it, indicating that they belong in the same block of instructions. The instructions `hubo.turn_left()` and `hubo.drop_beeper()`

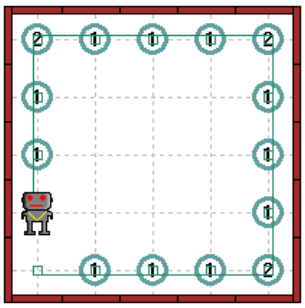
are similarly aligned, indented from the `else` statement to which they belong. The result of running the program is indicated below.



Now, what happens if we do not align the instruction `hubo.drop_beeper()` with `hubo.turn_left()`, but align it instead with the `else` statement, as indicated below?

```
def move_or_turn():
  if hubo.front_is_clear():
    dance()
    hubo.move()
  else:
    hubo.turn_left()
  hubo.drop_beeper()
```

Now, the definition of `move_or_turn()` includes a choice resulting in either a dance and a move forward, or a left turn, followed *every time* by the instruction `hubo.drop_beeper()`. The result of running this program is indicated below:

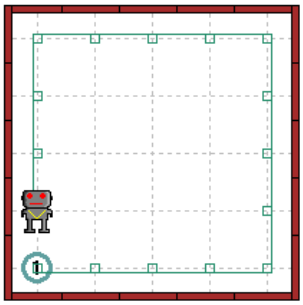


As you can see, after every step forward, a beeper has been put down. Each corner now has two beepers: one from the previous move to reach the corner, and one for the left turn after reaching the corner.

Now, suppose we align the `hubo.drop_beeper()` instruction with the `def` statement as indicated below?

```
def move_or_turn():
  if hubo.front_is_clear():
    dance()
    hubo.move()
  else:
    hubo.turn_left()
hubo.drop_beeper()
```

Now, `hubo.drop_beeper()` no longer belongs to the definition of the function `move_or_turn` at all, as it is not indented to be aligned with other instructions within the definition. It is a single instruction, the first in fact that the robot must follow, before executing the `move_or_turn()` function 18 times. The result is the following:



So, as you can see, much information is given through blank spaces (that is, the indentation of the instructions within blocks). Through practice, you will learn to use this to your advantage and realise that Python allows you to write very readable code by indenting instructions.

Our robot has become quite good at jumping hurdles. He now enters races of different lengths: short sprints and long races. He knows that he has reached the finish line when he is next to a beeper. Below, you will find three such race courses; the world files are *hurdles1.wld*, *hurdles2.wld*, and *hurdles3.wld*:

Task: (Hurdles3) Assume that there are no races longer than 20 units. Define a function that looks somewhat like the following:

```
def move_jump_or_finish():
  # test for end of race
  if not hubo.on_beeper():
    # is there a hurdle?
    if hubo.front_is_clear():
      hubo.move()
    else:
      jump_one_hurdle()
```

with an appropriate `jump_one_hurdle()` function, so that, other than definitions, the only instruction we need is:

```
for i in range(20):
  move_jump_or_finish()
```

Note that, in the above definition, the code is getting more and more indented as we introduce additional tests.

12 If, else, if, else...

The previous hurdles task required you to write an if/else within another one, all this because we wanted to give three choices to the robot: finish, move or jump. You may have noticed that this forced us to indent the code further and further. Imagine what would happen if we wanted to give 10 mutually exclusive choices; the resulting code would become hard to read. To help us in such situations, Python offers a keyword that represents the combination of an `else` statement followed by an `if` clause. That keyword is `elif`, which we can think of as an abbreviation for `else if`. With this new keyword, the above code can be written as:

```
def move_jump_or_finish():
  # test for end of race
  if hubo.on_beeper():
    pass  # race is over - do nothing
  # is there a hurdle?
  elif hubo.front_is_clear():
    hubo.move()
  else:
    jump_one_hurdle()
```

We can now better see, as they are indented the same way, that there are three possible choices. The `else` condition is executed only if all the previous conditions are false, so there is no condition associated with it.

The keyword `pass` which we introduced here is Python's way of saying "do nothing." It is necessary, because it is not possible to write an empty sequence of instructions (an empty *block*) in Python.

If we have more than three choices, all we need to do is add other `elif` statements

```
def move_jump_or_finish():
  # test for end of race
  if hubo.on_beeper():
    pass  # race is over - do nothing
  # is there a hurdle?
  elif hubo.front_is_clear():
    hubo.move()
  elif hubo.right_is_clear(): # never
    pass
  else:
    jump_one_hurdle()
```

Since we follow the bottom wall, `hubo.right_is_clear()` is always false, so the `pass` instruction is always ignored. Note that if we had used `hubo.left_is_clear()` instead, we would have gotten stuck forever as soon as we had reached the first hurdle. Try it for yourself!

13 For a while...

When we want to repeat some instructions until a certain condition is satisfied, Python gives us a simpler way to write this using a new keyword: `while`. Let me show you first how this would look using pseudocode:

```
While not on beeper,
... keep moving;
otherwise,
... stop.
```

You should agree that this expresses the same idea as before. Using Python code, here is how we would actually write it:

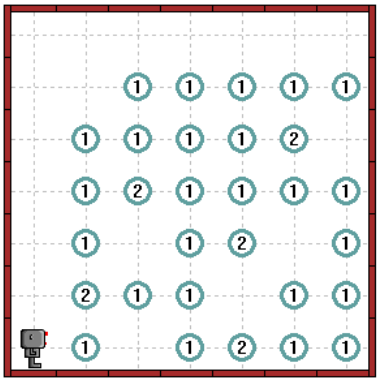
```
while not hubo.on_beeper():
  hubo.move()
```

No more need for a for-loop with a fixed number of repetitions.

Task: (Hurdles4) Use a `while` loop to rewrite the *hurdles3* program so that you don't have to use a for-loop of fixed length. In other words, the core of your program should look like the following:

```
while not hubo.on_beeper():
  move_or_jump()
```

Task: (Harvest5) Modify your program for the *Harvest4* task so that it also works when there is more than one carrot on one place, as in world file *harvest4.wld* below. All carrots must be harvested, and it should also work for the previous worlds *harvest1.wld* and *harvest3.wld*.



Task: (ZigZag2) Rewrite your program for *ZigZag1* so that the robot can visit an empty world of any size in zigzag fashion. Make sure that it works for even and odd numbers of streets and avenues. (You can assume that there are at least two streets and at least two avenues.)

14 Around the world in 80 days

Let's start by considering a simple problem: go around the world once, assuming there is no obstacle on the way. We have done this before, when we introduced the `front_is_clear()` test. Here's the outline of a solution which supposes that we carry at least one beeper at the beginning:

1. Put down a beeper to mark the starting point.
2. Move forward until facing a wall.
3. Turn left when facing a wall.
4. Repeat steps 2 and 3 until we find the beeper we had put down.
5. Finish when finding the beeper.

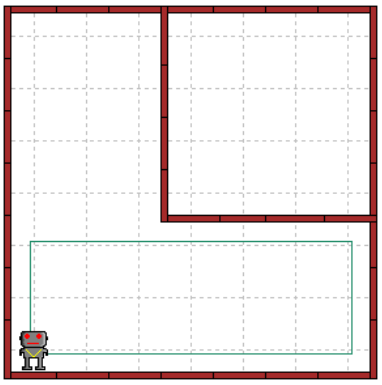
The key step is 4, where we have a repeating instruction with a test. Let's now translate the entire solution in proper code:

```
hubo.drop_beeper()
while not hubo.on_beeper():
  if hubo.front_is_clear():
    hubo.move()
  else:
    hubo.turn_left()
```

This does not work—the program terminates immediately after dropping the beeper. The problem is that we drop the beeper, and then immedately think that we are done. So we need to make one move after dropping the beeper:

```
hubo.drop_beeper()
hubo.move()
while not hubo.on_beeper():
  if hubo.front_is_clear():
    hubo.move()
  else:
    hubo.turn_left()
```

This works well on an empty world, but here is an example where it does not work:



The problem is that we assumed that we only had to move forward or turn left to go around the world; we never took into account situations where we would have wanted to make a right turn. What we need to do is first to check on the right side to see if there is still a wall; if not, we have to make a right turn. Here's a modified program that attempts to do just that:

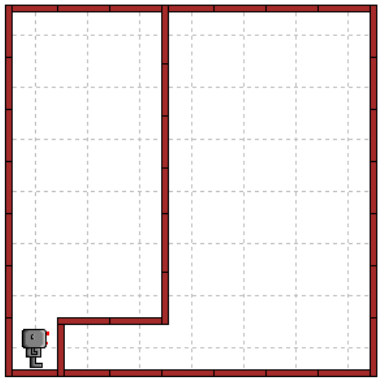
```
def turn_right():
  for i in range(3):
    hubo.turn_left()

hubo.drop_beeper()
hubo.move()
while not hubo.on_beeper():
  if hubo.right_is_clear():
    turn_right()
  elif hubo.front_is_clear():
    hubo.move()
  else:
    hubo.turn_left()
```

This still doesn't work. The robot gets in an *infinite loop* when there is no wall around him. (That's right: the `while` keyword makes it possible to write programs that never terminate.)

We need to have him move after turning right:

```
hubo.drop_beeper()
hubo.move()
while not hubo.on_beeper():
  if hubo.right_is_clear():
    turn_right()
    hubo.move()
  elif hubo.front_is_clear():
    hubo.move()
  else:
    hubo.turn_left()
```

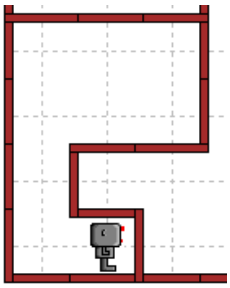
But this *still* does not always work:



Do you see why? We were too hasty in moving forward after putting down a beeper. We need to check if there is a wall preventing us from moving first. Here's a solution to the problem:

```
hubo.drop_beeper()
if not hubo.front_is_clear():
  hubo.turn_left()
hubo.move()
while not hubo.on_beeper():
  # continue as before
```

Can you imagine situations where this does not work? Well, here is one (try it!):



To make it work, we need to replace the `if` we just added by a `while`. Try it!

15 Clarifying our intent

We seem to have designed a program that works in all situations we are likely to encounter. This program, before we forget, is to allow the robot to explore his world, going around once. While the program is rather short, and its structure should be clear at this point, it might not be so obvious to someone who just happened to see it for the first time.

We discussed before that our goal should be to write programs so that they can be understood by a *human*. It's probably a good idea either to add comments and/or to introduce more meaningful words. Let's start by adding comments, somewhat more verbose than we think we might need.

```
# Define function for turning right
def turn_right():
  for i in range(3):
    hubo.turn_left()

# Mark the starting point with a beeper
hubo.drop_beeper()

# Find a clear direction and start moving
while not hubo.front_is_clear():
  hubo.turn_left()
hubo.move()

# We know we have gone around the world
# when we come back to the beeper.

while not hubo.on_beeper():
  if hubo.right_is_clear():
    # Keep to the right
    turn_right()
    hubo.move()
  elif hubo.front_is_clear():
    # move following the right wall
    hubo.move()
  else:
    # follow the wall
    hubo.turn_left()
```

While this sort of clarifies our intent for each instruction, it is not really that helpful in summarizing the *method* (also known as the *algorithm*) used in solving the problem. Therefore, these comments might not be as helpful to another reader as we might have wished. Reading over the comments, we note that the program has two parts:

1. mark the starting point;
2. follow the right wall until we come back to the start.

Let's rewrite this program so that these two parts

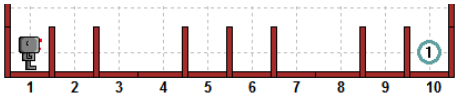
become clearer, and writing the comments differently.

```
# This program lets the robot go
# around his world counterclockwise,
# stopping when he comes back to his
starting point.

def turn_right():
  for i in range(3):
    hubo.turn_left()

def mark_starting_point_and_move():
  hubo.drop_beeper()
  while not hubo.front_is_clear():
    hubo.turn_left()
  hubo.move()

def follow_right_wall():
  if hubo.right_is_clear():
    # Keep to the right
    turn_right()
    hubo.move()
  elif hubo.front_is_clear():
    # move following the right wall
    hubo.move()
  else:
    # follow the wall
    hubo.turn_left()

# end of definitions, begin solution

mark_starting_point_and_move()

while not hubo.on_beeper():
  follow_right_wall()
```

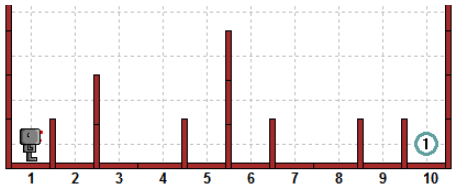
Isn't this clearer? This will also make it much easier for us to change ("maintain") the program if we want to change its behaviour in the future.

16 Hurdles and mazes

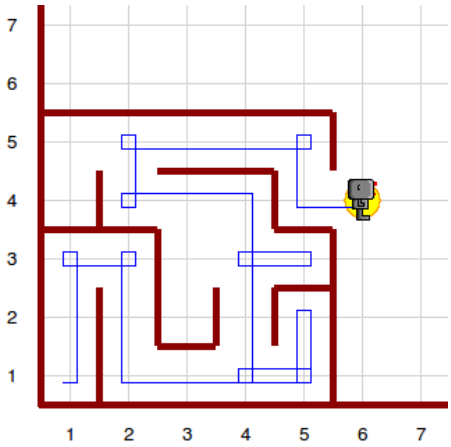
Change the program you just wrote to remove the `drop_beeper()` instruction. After saving it, try this slightly modified program with the following hurdle race (world file *hurdles3.wld*):



Surprise! Other than finishing facing in an unusual direction (perhaps bowing to the crowd after winning), the program we just wrote can solve the hurdle problem. It also works with the uneven hurdle (world file *hurdles4.wld*) illustrated below—which the program we wrote before for jumping over hurdles could not handle!



Even better, the same modified program can find the exit to a maze (world file *maze1.wld*):



17 Stepwise refinement

We started with a simple problem to solve (going around a rectangular world) and, by improving little by little (also called *stepwise refinement*), we manage to write a program that could be used to solve many other apparently unrelated problems. At each step, we kept the changes small, and *made sure we had a working solution*, before considering more complex problems. We also used more descriptive names for parts of the algorithm which made the program easier to read and, hopefully, to understand. This is a strategy you should use when writing your own programs:

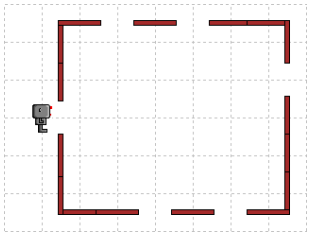
Steps to follow when writing a program:

- start simple;
- introduce small changes, one at a time;
- *make sure* that each of the changes you have introduced do not invalidate the work you have done before;
- add appropriate comments that don't simply repeat what each instruction does; and
- choose descriptive names.

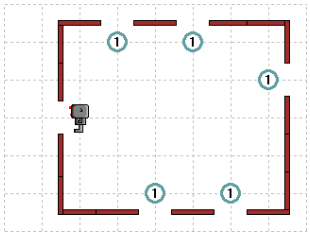
18 It's raining!

It was a beautifully sunny day. Hubo is playing outside with his friends. Suddenly, it started to

rain and he remembered that the windows in his house were all open. So he went back to his house and stopped in front of the door, unsure of how to proceed.

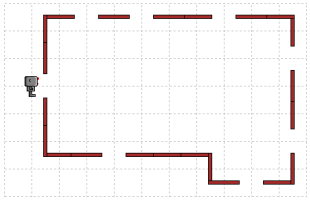


Task: (Rain1) Help Hubo close the windows of his house. A closed window has a beeper in front of it. When Hubo finishes his task, he will stand in the doorway, watching the rain fall, waiting for it to stop before he can go back and play outside, as illustrated below.



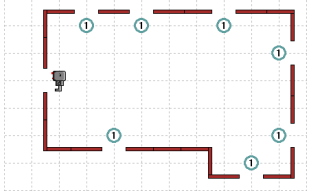
The world file is *rain1.wld*; make sure Hubo carries more than enough beepers to accomplish his task.

Task: (Rain2) Ami, Hubo's friend, lives in a bigger house. Ami was playing outside with Hubo when it started raining. Help Ami close the windows in her house. The starting position is:



The corresponding world file is *rain2.wld*.

When Ami is finished, she too will be standing in the doorway of her house as illustrated below.

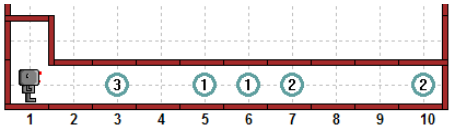


This problem is a bit more tricky to solve. Perhaps you should consider the possibility that Ami will need to step back sometimes...

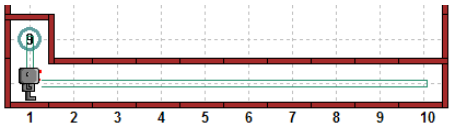
Verify to see if the program you wrote to help Ami close the windows in her house can be used by Hubo to do the same for his house. If not, change it so that it works for both Ami and Hubo.

19 After the storm

The wind blew really hard last night. There is litter everywhere outside Hubo's house. His parents asked him to go and clean up the driveway, as well as the path leading to the curb. They both are in a straight line, with garbage at random places as illustrated below.



Task: (Trash1) Hubo should collect all the litter, and put it in the garbage can, situated north of his starting point. The final situation should look like the following:



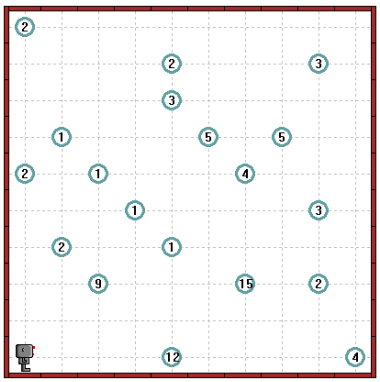
Important: To put all the trash in one pile, you will need to use the test `hubo.carries_beepers()` which I hadn't told you about ... yet! Try something like

```
while hubo.carries_beepers():
  # do something
```

Make sure your program works for both world files: *trash1.wld* and *trash2.wld*.

Task: (Trash2) Hubo's parents are so proud of his work, that they ask him to pick up all the garbage that got blown away in their backyard during the windstorm. Have him pick up all the garbage and bring it back with him to his starting position. Try to generalise from the program you just wrote to clean up the driveway.

Create your own world file, corresponding to a situation like the one illustrated below. Your solution should not depend on the exact location of the garbage, nor should it depend on the size of the yard!



20 The robot's compass

Our robot has a built-in compass, which allows him to tell if he is facing in the north direction or not. You can use this test in an `if` or `while` statement:

```
if hubo.facing_north():
  # do something
```

Task: (Return) Write a program that will allow Hubo to return to his usual starting position (avenue 1, street 1, facing east), starting from *any* position and orientation in an empty world. You can create robots with a given position and orientation like this:

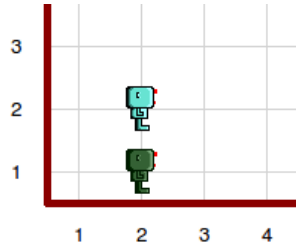
```
hubo = Robot(orientation = "W",
             avenue = 3, street = 5)
```

21 Hubo's family

Let's have a look at the following code:

```
from cs1robots import *

create_world()
hubo = Robot("light_blue")
hubo.turn_left()
hubo.move()
hubo.turn_left()
hubo.turn_left()
hubo.turn_left()

ami = Robot("green")
ami.move()
hubo.move()
```

The result looks like this:



We have two robots now—the light blue one is `hubo`, the green one is `ami` (available colors are `gray`, `green`, `yellow`, `purple`, `blue`, and `light_blue`).

If you run this interactively in the Python interpreter, you can now control both robots manually: When you say `hubo.move()`, Hubo makes a step. When you say `ami.move()`, then Ami makes a step.

Each robot is a Python *object*, and by calling *methods* of the object using the dot-notation you can inquire about the *state* of the object (as in `ami.on_beeper()`), or ask the object to perform some action and change its state (as in `hubo.turn_left()`).

Starting with the situation above, we can now write a small race:

```
for i in range(8):
  hubo.move()
  ami.move()
```

This is not very exciting as they are exactly equally fast - we'll see later how we can change that.

22 Functions with parameters

Earlier, we defined a function `turn_right()`. It still works—it will make `hubo` turn right. But now we have two robots—how do we make `ami` turn right? Should we define a new function? And then again another one for every robot we make?

What we need now is a function that can work for *any* robot:

```
def turn_right(robot):
  for i in range(3):
    robot.turn_left()
```

The word `robot` here is a *function parameter*. When we call the function, we have to provide a robot as a *function argument*:

```
turn_right(ami)
turn_right(hubo)
```

The function call `turn_right(ami)` executes the function `turn_right` with the parameter `robot` set to refer to the robot `ami`.