Python in Action

In *Python: A Whirlwind Tour*, you learnt to use Python in quite an abstract environment. In this worksheet, we'll build on that knowledge in an applied setting: that of programming a robot. For now, this robot will be in a simulator, but the code you will write for it will be very similar to the code for your real robot.

Using the simulator

Running a program

On your desktop on your lab machine, you will find a directory called `Portable Python`, which contains `PyScripter-Portable.exe`. Double click this to open the PyScripter development environment.

In PyScripter, open the file named `run.py` in the `robot-sim` folder on your desktop. This program allows you to launch the simulator. To try it out, run it (by clicking the green play icon on the toolbar, or pressing Ctrl+F9). A dialog box will ask you for a program name. Enter `test.py` and press ENTER. A test program will run in a simulated arena. (You may need to bring the simulator window to the front by clicking on it in your taskbar.)

To create your own program, click File > New, and write your code in that file. Save the file in the `robot-sim` folder on your desktop.

To run your program, run `run.py`, like you did before. Enter the name of your code file (including the `.py`) into the dialog box and press ENTER.

The robot object

In your program, you first need to import the module which lets you interact with your robot. To do so, start your program with the following code:

```
from sr.robot import *
```

Next, create a `Robot` object. We haven't covered objects, but for now you can think of them as collections of variables and functions.


```
R = Robot()
```

The variable `R` now contains a `Robot` object. This object will act as your connection to your robot, allowing you to give it commands and read from its sensors. On a real robot, you'll also use a `Robot` object, which is similar but with more features.

Below this, you can write your program code. You can still use `print` statements as normal.

Getting things moving: motors

Your robot is equipped with two wheels, each connected to a motor, and a caster. This allows it to drive forwards and backwards, and turn by setting the left and right wheels at different speeds (a method known as skid steering).

The `Robot` object you placed in variable `R` earlier contains a list of the motor controllers connected to it. The simulated robot has one controller, which can control two motors. Each motor is also an object, with a `power` variable which you can use to set its speed. For example, to drive straight forwards at half power, you might use the following piece of code:

```
R.motors[0].m0.power = 50
R.motors[0].m1.power = 50
```

If you run this in the simulator, the robot will just drive forward until it hits a wall.

A motor speed is simply a number between -100 (reverse at full power) and 100 (forwards at full power). What happens if you change the speed on the first line to `25`? What about a negative number?

Exercise: driving in a square

The `time` module contains the `sleep` function, which simply waits for a given number of seconds (see the Calling Functions section of the previous tutorial for a reminder about modules). For example, the following code will wait 1 second before printing the second message:

```
import time

print "Message 1"
time.sleep(1)
print "Message 2"
```


If you use `time.sleep` in a robot program, the motors will still move at the last speeds you set while the program sleeps.

Using the `time.sleep` function and one of the loops you learnt about in the last tutorial, make your robot drive in a square.

Finding things

Your robot's webcam is a very important sensing device. Using it, your robot can calculate the position of any libkoki markers in sight. There are markers on the tokens that your robot must pick up, other robots, and the arena walls (which are shown as thin blue rectangles in the simulator).

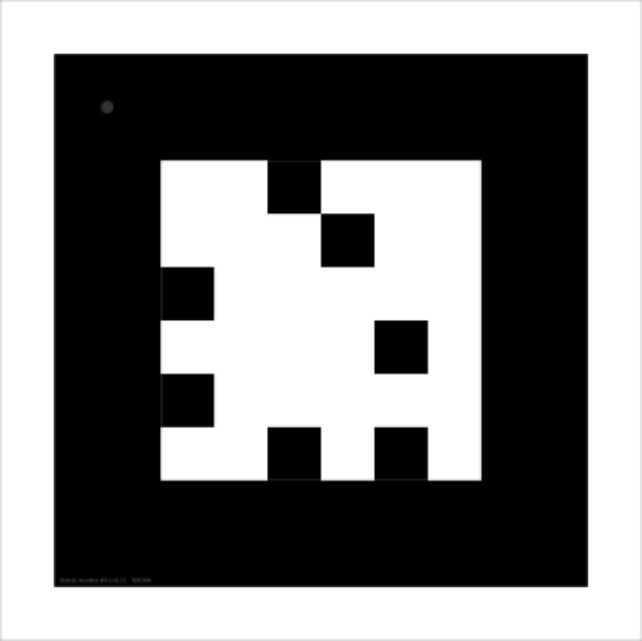

Figure 1: A libkoki marker

The `R.see()` function takes a picture with the webcam, looks for markers in it, and returns a list of all the markers it can detect. For example:

```
markers = R.see()
print "I can see",len(markers),"markers."
for m in markers:
    if m.info.marker_type == MARKER_TOKEN:
        print " - Token", m.info.offset, "is", m.dist, "metres away"
```


The marker objects are documented at https://www.studentrobotics.org/docs/programming/simulator#Interface.

The most useful properties of a marker object are:

- `m.info.marker_type`: this is a value indicating the type of the marker. Possible values are `MARKER_TOKEN`, `MARKER_ROBOT`, `MARKER_ARENA` (these are the ones on the arena walls), and `MARKER_PEDESTAL`.
- `m.info.offset`: the ID number of this marker. This can be used to distinguish between different tokens, wall markers, etc..
- `m.dist`: the distance between the webcam and the marker (in metres).
- `m.rot_y`: the number of degrees between straight ahead and the marker, where negative numbers are to the left, `0` is straight ahead, and positive numbers are to the right.

Exercise: find a certain marker

Make your robot turn on the spot until it can see wall marker number 0 (which is in the top-left corner in the simulator).

Exercise: lining up

Make your robot turn on the spot until it sees a token (any token). When it sees one, make it line up so that it is roughly straight ahead (the `m.rot_y` value will be useful). Finally, drive towards it. Try to refine your program so that the robot gets as close as possible to the token.

Picking things up

The red thing on the front of the simulator robot is a grabber. You can close it with `R.grab()` and open it with `R.release()`. If a token is nearby when you call `R.grab()`, it will turn green and be held by the grabber, moving with the robot until you call `R.release()`.

`R.grab()` returns `True` if it manages to pick something up. For example, you might use this code somewhere in your program:

```
if R.grab():
    print "Gotcha!"
else:
    print "Aww, missed!"
```


Exercise: picking up a token

Adapt your program from the last exercise to pick up the token when it finds it. The token must be within 0.4 metres of the robot and be in front of it to be picked up.

Putting it all together

You now know enough to make your simulator robot play the game which your real robot will take part in. Adapt your last program once more by making the robot take the token to a wall marker before releasing it.

Moving on to a real robot

The code that you have written for your robot in the simulator will mostly work on your real robot as well, with the exception of the `R.grab()` and `R.release()` functions. (You may wish to write your own replacements, specific to your grabbing mechanism.)

There are a number of features which your real robot will have which the simulator doesn't, most importantly the IO (Input/Output) and servo boards. The IO board allows you to interface with sensors, while the servo board will likely be quite important for your grabber. These are described in our programming documentation online at http://srobo.org/docs/programming/sr.

In Two Colours, there is a raised platform in the centre of the arena with six tokens on it. This is not modelled in the simulator, so you might need to adapt your code to drive around it.

Finally, here are a few things to consider when working in the real world:

- In the simulator, the vision system is completely reliable. Unfortunately, in the real world the system cannot see some markers when they are too far away. It is also quite sensitive to motion blur, so you will probably need to stop moving to use the camera.
- In the competition, you can only return your tokens to your own zone to get points, so you'll have to change which wall markers you head for depending on which zone you are in. This is available to your program as `R.zone`.
- Other robots! This is the hardest thing to prepare for. You should avoid collisions with another robot where possible. They will have a libkoki marker (`MARKER_ROBOT`) on each side.

