



2

Personal Robots

Every Pleo is autonomous. Yes, each one begins life as a newly-hatched baby Camarasaurus, but that's where predictability ends and individuality begins. Like any creature, Pleo feels hunger and fatigue - offset by powerful urges to explore and be nurtured. He'll graze, nap and toddle about on his own -when he feels like it! Pleo dinosaur can change his mind and his mood, just as you do.

From: www.pleoworld.com

Opposite page: Pleo robots
Photo courtesy of UGOBE Inc.

Most people associate the personal computer (aka the PC) revolution with the 1980's but the idea of a personal computer has been around almost as long as computers themselves. Today, on most college campuses, there are more personal computers than people. The goal of One Laptop Per Child (OLPC) Project is to “provide children around the world with new opportunities to explore, experiment, and express themselves” (see www.laptop.org). Personal robots, similarly, were conceived several decades ago. However, the personal robot ‘revolution’ is still in its infancy. The picture on the previous page shows the Pleo robots that are designed to emulate behaviors of an infant Camarasaurus. The Pleos are marketed mainly as toys or as mechatronic “pets”. Robots these days are being used in a variety of situations to perform a diverse range of tasks: like mowing a lawn; vacuuming or scrubbing a floor; entertainment; as companions for elders; etc. The range of applications for robots today is limited only by our imagination! As an example, scientists in Japan have developed a baby seal robot (shown on the opposite page) that is being used for therapeutic purposes for nursing home patients.

Your Scribbler robot is your personal robot. In this case it is being used as an educational robot to learn about robots and computing. As you have already seen, your Scribbler is a rover, a robot that moves around. Such robots have become more prevalent in the last few years and represent a new dimension of robot applications. Roaming robots have been used for mail delivery in large offices and as vacuum cleaners in homes. Robots vary in the ways in which they move about: they can roll about like small vehicles (like the lawn mower, Roomba, Scribbler, etc.), or even ambulate on two, three, or more legs (e.g. Pleo). The Scribbler robot moves on three wheels, two of which are powered. In this chapter, we will get to know the Scribbler in some more detail and also learn about how to use its commands to control its behavior.

The Scribbler Robot: Movements

In the last chapter you were able to use the Scribbler robot through Myro to carry out simple movements. You were able to start the Myro software, connect to the robot, and then were able to make it beep, give it a name, and move it around using a joystick. By inserting a pen in the pen port, the

scribbler is able to trace its path of movements on a piece of paper placed on the ground. It would be a good idea to review all of these tasks to refresh your memory before proceeding to look at some more details about controlling the Scribbler.



The Paro Baby Seal Robot.
Photo courtesy of National Institute of Advanced Industrial Science and Technology, Japan (paro.jp).

If you hold the Scribbler in your hand and take a look at it, you will notice that it has three wheels. Two of its wheels (the big ones on either side) are powered by motors. Go ahead turn the wheels and you will feel the resistance of the motors. The third wheel (in the back) is a free wheel that is there for support only. All the movements the Scribbler performs are controlled through the two motor-driven wheels. In Myro, there are several commands to control the movements of the robot. The command that directly controls the two motors is the `motors` command:

```
motors(LEFT, RIGHT)
```

`LEFT` and `RIGHT` can be any value in the range `[-1.0...1.0]` and these values control the left and right motors, respectively. Specifying a negative value moves the motors/wheels backwards and positive values move it forward. Thus, the command:

```
motors(1.0, 1.0)
```

will cause the robot to move forward at full speed, and the command:

```
motors(0.0, 1.0)
```

will cause the left motor to stop and the right motor to move forward at full speed resulting in the robot turning left. Thus by giving a combination of left and right motor values, you can control the robot's movements. Myro has also

provided a set of often used movement commands that are easier to remember and use. Some of them are listed below:

```
forward(SPEED)
backward(SPEED)
turnLeft(SPEED)
turnRight(SPEED)
stop()
```

Another version of these commands takes a second argument, an amount of time in seconds:

```
forward(SPEED, SECONDS)
backward(SPEED, SECONDS)
turnLeft(SPEED, SECONDS)
turnRight(SPEED, SECONDS)
```

Providing a number for SECONDS in the commands above specifies how long that command will be carried out. For example, if you wanted to make your robot traverse a square path, you could issue the following sequence of commands:

```
forward(1, 1)
turnLeft(1, .3)
forward(1, 1)
turnLeft(1, .3)
forward(1, 1)
turnLeft(1, .3)
forward(1, 1)
turnLeft(1, .3)
```

of course, whether you get a square or not will depend on how much the robot turns in 0.3 seconds. There is no direct way to ask the robot to turn exactly 90 degrees, or to move a certain specified distance (say, 2 ½ feet). We will return to this later.

You can also use the following movement commands to translate (i.e. move forward or backward), or rotate (turn right or left):

```
translate(SPEED)
rotate(SPEED)
```

Additionally, you can specify, in a single command, the amount of translation and rotation you wish use:

```
move(TRANSLATE_SPEED, ROTATE_SPEED)
```

In all of these commands, `SPEED` can be a value between `[-1.0...1.0]`.

You can probably tell from the above list that there are a number of redundant commands (i.e. several commands can be specified to result in the same movement). This is by design. You can pick and choose the set of movement commands that appear most convenient to you. It would be a good idea at this point to try out these commands on your robot.

Do This: Start Myro, connect to the robot, and try out the following movement commands on your Scribbler:

First make sure you have sufficient room in front of the robot (place it on the floor with a few feet of open space in front of it).

```
>>> motors(1, 1)
>>> motors(0, 0)
```

Observe the behavior of robot. Specifically, notice if it does (or doesn't) move in a straight line after issuing the first command. You can make the robot carry out the same behavior by issuing the following commands:

```
>>> move(1.0, 0.0)
>>> stop()
```

Go ahead and try these. The behavior should be exactly the same. Next, try making the robot go backwards using any of the following commands:

```
motors(-1, -1)
move(-1, 0)
backward(1)
```

Again, notice the behavior closely. In rovers precise movement, like moving in a straight line, is difficult to achieve. This is because two independent motors control the robot's movements. In order to move the robot forward or backward in a straight line, the two motors would have to issue the exact same amount of power to both wheels. While this technically feasible, there are several other factors than can contribute to a mismatch of wheel rotation. For example, slight differences in the mounting of the wheels, different resistance from the floor on either side, etc. This is not necessarily a bad or undesirable thing in these kinds of robots.

Under similar circumstances even people are unable to move in a precise straight line. To illustrate this point, you can try the experiment shown on right.

For most people, the above experiment will result in a variable movement. Unless you really concentrate hard on walking in a straight line, you are most likely to display similar variability as your Scribbler. Walking in a straight line requires constant feedback and adjustment, something humans are quite adept at doing. This is hard for robots to do. Luckily, roving does not require such precise moments anyway.

Do humans walk straight?

Find a long empty hallway and make sure you have a friend with you to help with this. Stand in the center of the hallway and mark your spot. Looking straight ahead, walk about 10-15 paces without looking at the floor. Stop, mark your spot and see if you walked in a straight line.

Next, go back to the original starting spot and do the same exercise with your eyes closed. Make sure your friend is there to warn you in case you are about to run into an object or a wall. Again, note your spot and see if you walked in a straight line.

Do This: Review all of the other movement commands listed above and try them out on your Scribbler. Again, note the behavior of the robot from each of these commands. In doing this activity, you may find yourself repeatedly

entering the same commands (or simple variations). IDLE provides a convenient way to repeat previous commands (see the Tip in the box on the right).

Defining New Commands

Trying out simple commands interactively in IDLE is a nice way to get to know your robot's basic features. We will continue to use this each time we want to try out something new. However, making a robot carry out more complex behaviors requires several series of commands. Having to type these over and over interactively while the robot is operating can get tedious. Python provides a convenient way to package a series of commands into a brand new command called a *function*. For example, if we wanted the Scribbler to move forward and then move backward (like a yoyo), we can define a new command (function) called `yoyo` as follows:

```
>>> def yoyo():
        forward(1)
        backward(1)
        stop()
```

The first line defines the name of the new command/function to be `yoyo`. The lines that follow are slightly indented and contain the commands that make up the `yoyo` behavior. That is, to act like a yoyo, move forward and then backward and then stop. The indentation is important and is part of the Python

IDLE Tip

You can repeat a previous command by using IDLE's command history feature:

ALT-p retrieves previous command
ALT-n retrieves next
(Use CTRL-p and CTRL-n on MACs)

Pressing ALT-p again will give the previous command from that one and so on. You can also move forward in the command history by pressing ALT-n repeatedly. You can also click your cursor on any previous command and press ALT-ENTER to repeat that command.

syntax. It ensures that all indented commands are part of the definition of the new command. We will have more to say about this later.

Once the new command has been defined, you can try it by entering the command into IDLE as shown below:

```
>>> yoyo()
```

Do This: If you have your Scribbler ready, go ahead and try out the new definition above by first connecting to the robot, and then entering the definition above. You will notice that as soon as you type the first line, IDLE automatically indents the next line(s). After entering the last line hit an extra RETURN to end the definition. This defines the new command in Python.

Observe the robot's behavior when you give it the `yoyo()` command. You may need to repeat the command several times. The robot momentarily moves and then stops. If you look closely, you will notice that it does move forward and backwards.

In Python, you can define new functions by using the `def` syntax as shown above. Note also that defining a new function doesn't mean that the commands that make up the function get carried out. You have to explicitly issue the command to do this. This is useful because it gives you the ability to use the function over and over again (as you did above). Issuing the new function like this in Python is called, *invocation*. Upon invocation, all the commands that make up the function's definition are executed in the sequence in which they are listed in the definition.

How can we make the robot's `yoyo` behavior more pronounced? That is, make it move forward for, say 1 second, and then backwards for 1 second, and then stop? You can use the `SECONDS` option in `forward` and `backward` movement commands as shown below:

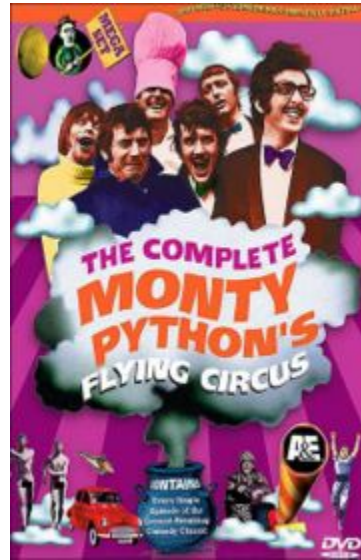
```
>>> def yoyo():
        forward(1, 1)
        backward(1, 1)
        stop()
```

The same behavior can also be accomplished by using the command, `wait` which is used as shown below:

```
wait(SECONDS)
```

where `SECONDS` specifies the amount of time the robot waits before moving on to the next command. In effect, the robot continues to do whatever it had been asked to do just prior to the `wait` command for the amount of time specified in the `wait` command. That is, if the robot was asked to move forward and then asked to wait for 1 second, it will move forward for 1 second before applying the command that follows the wait. Here is the complete definition of `yoyo` that uses the `wait` command:

And now for something completely different



DVD Cover, from <http://Wikipedia.com>

IDLE is the name of the editing and Python shell program. When you double-click **Start Python** you are really starting up IDLE. Python is the name of the language that we will be using, and gets its name from *Monty Python's Flying Circus*. IDLE supposedly stands for **I**nteractive **D**evelopment **E**nvironment, but do you know to whom else it might be homage?

```
>>> def yoyo():
    forward(1)
    wait(1)
    backward(1)
    wait(1)
    stop()
```

Do This: Go ahead and try out the new definitions exactly as above and issue the command to the scribbler. What do you observe? In both cases you should see the robot move forward for 1 second followed by a backward movement for 1 second and then stop.

Scribbler Tip:

Remember that your Scribbler runs on batteries and with time they will get drained. When the batteries start to run low, the Scribbler may exhibit erratic movements. Eventually it stops responding. When the batteries start to run low, the Scribbler's red LED light starts to blink. This is your signal to replace the batteries.

Adding Parameters to Commands

Take a look at the definition of the `yoyo` function above and you will notice the use of parentheses, `()`, both when defining the function as well as when using it. You have also used other functions earlier with parentheses in them and probably can guess their purpose. Commands or functions can specify certain *parameters* (or values) by placing them within parentheses. For example, all of the movement commands, with the exception of `stop` have one or more numbers that you specify to indicate the speed of the movement. The number of seconds you want the robot to wait can be specified as a parameter in the invocation of the `wait` command. Similarly, you could have chosen to specify the speed of the forward and backward movement in the `yoyo` command, or the amount of time to wait. Below, we show three definitions of the `yoyo` command that make use of parameters:

```
>>> def yoyo1(speed):
    forward(speed, 1)
    backward(speed, 1)
```

```
>>> def yoyo2(waitTime):
    forward(1, waitTime)
    backward(1, waitTime)

>>> def yoyo3(speed, waitTime):
    forward(speed, waitTime)
    backward, waitTime)
```

In the first definition, `yoyo1`, we specify the speed of the forward or backward movement as a parameter. Using this definition, you can control the speed of movement with each invocation. For example, if you wanted to move at half speed, you can issue the command:

```
>>> yoyo1(0.5)
```

Similarly, in the definition of `yoyo2` we have parameterized the wait time. In the last case, we have parameterized both speed and wait time. For example, if we wanted the robot to move at half speed and for 1 ½ seconds each time, we would use the command:

```
>>> yoyo3(0.5, 1.5)
```

This way, we can customize individual commands with different values resulting in different variations on the yoyo behavior. Notice in all of the definitions above that we did not have to use the `stop()` command at all. Why?

Saving New Commands in Modules

As you can imagine, while working with different behaviors for the robot, you are likely to end up with a large collection of new functions. It would make sense then that you do not have to type in the definitions over and over again. Python enables you to define new functions and store them in files in a folder on your computer. Each such file is called a *module* and can then be easily used over and over again. Let us illustrate this by defining two behaviors: a parameterized yoyo behavior and a wiggle behavior that makes the robot wiggle left and right. The two definitions are given below:

```
# File: moves.py
# Purpose: Two useful robot commands to try out as a module.

# First import myro and connect to the robot

from myro import *
init()

# Define the new functions...

def yoyo(speed, waitTime):
    forward(speed)
    wait(waitTime)
    backward(speed)
    wait(waitTime)
    stop()

def wiggle(speed, waitTime):
    rotate(-speed)
    wait(waitTime)
    rotate(speed)
    wait(waitTime)
    stop()
```

All lines beginning with a '#' sign are called comments. These are simply annotations that help us understand and document the programs in Python. You can place these comments anywhere, including right after a command. The # sign clearly marks the beginning of the comment and anything following it on that line is not interpreted as a command by the computer. This is quite useful and we will make liberal use of comments in all our programs.

Notice that we have added the `import` and the `init` commands at the top. The `init` command will always prompt you to enter the com-port number.

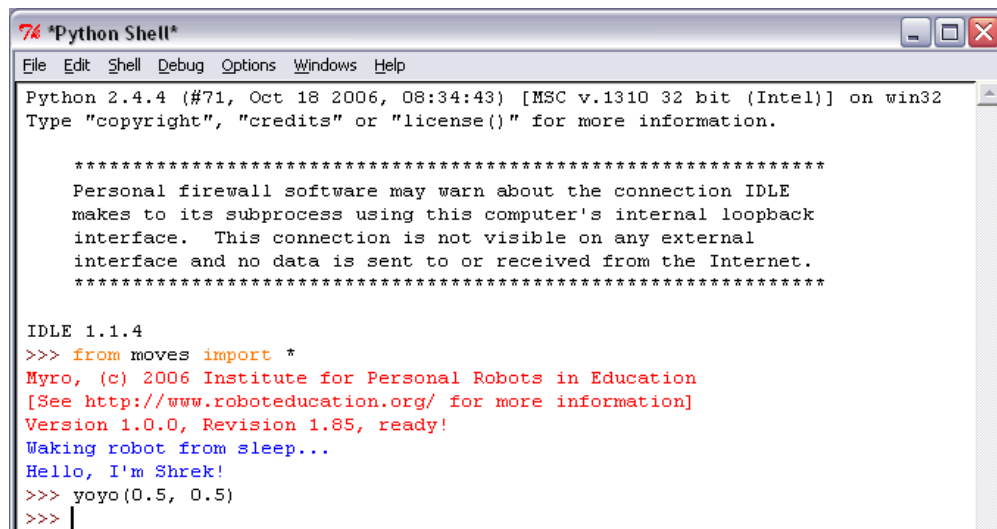
Do This: To store the `yoyo` and `wiggle` behaviors as a module in a file, you can ask IDLE for a New Window from the File menu. Next enter the text containing the two definitions and then save them in a file (let's call it `moves.py`) in your Myro folder (same place you have the `Start Python`

icon). All Python modules end with the filename extension `.py` and you should make sure they are always saved in the same folder as the `Start Python.pyw` file. This will make it easy for you as well as IDLE to locate your modules when you use them.

Once you have created the file, there are two ways you can use it. In IDLE, just enter the command:

```
>>> from moves import *
```

and then try out any of the two commands. For example, the following shows how to use the `yoyo` function after importing the `moves` module:



```
Python 2.4.4 (#71, Oct 18 2006, 08:34:43) [MSC v.1310 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.

*****
Personal firewall software may warn about the connection IDLE
makes to its subprocess using this computer's internal loopback
interface. This connection is not visible on any external
interface and no data is sent to or received from the Internet.
*****

IDLE 1.1.4
>>> from moves import *
Myro, (c) 2006 Institute for Personal Robots in Education
[See http://www.roboteducation.org/ for more information]
Version 1.0.0, Revision 1.85, ready!
Waking robot from sleep...
Hello, I'm Shrek!
>>> yoyo(0.5, 0.5)
>>> |
```

As you can see from above, accessing the commands defined in a module is similar to accessing the capabilities of the `myro` module. This is a nice feature of Python. In Python, you are encouraged to extend the capabilities of any system by defining your own functions, storing them in modules and then using them by importing them. Thus importing from the `moves` module is no different that importing from the `myro` module. In general, the Python `import`

command has two features that it specifies: the module name; and what is being imported from it. The precise syntax is described below:

```
from <MODULE NAME> import <SOMETHING>
```

where <MODULE NAME> is the name of the module you are importing from, and <SOMETHING> specifies the commands/capabilities you are importing. By specifying a * for <SOMETHING> you are importing everything defined in the module. We will return to this a little later in the course. But at the moment, realize that by saying:

```
from myro import *
```

you are importing everything defined in the `myro` module. Everything defined in this module is listed and documented in the Myro Reference Manual. The nice thing that this facility provides is that you can now define your own set of commands that extend the basic commands available in Myro to customize the behavior of your robot. We will be making use of this over and over again in this course.

Functions as Building Blocks

Now that you have learned how to define new commands using existing ones, it is time to discuss a little more Python. The basic syntax for defining a Python function takes the form:

```
def <FUNCTION NAME>(<PARAMETERS>):  
    <SOMETHING>  
    . . .  
    <SOMETHING>
```

That is, to define a new function, start by using the word `def` followed by the name of the function (<FUNCTION NAME>) followed by <PARAMETERS> enclosed in parenthesis followed by a colon (:). This line is followed by the commands that make up the function definition (<SOMETHING> . . . <SOMETHING>). Each command is to be placed on a separate line, and all lines that make up the

definition should be indented (aligned) the same amount. The number of spaces that make up the indentation is not that important as long as they are all the same. This may seem a bit awkward and too restricting at first, but you will soon see the value of it. First, it makes the definition(s) more readable. For example, look at the following definitions for the `yoyo` function:

```
def yoyo(speed, waitTime):
    forward(speed)
    wait(waitTime)
    backward(speed)
    wait(waitTime)
    stop()
```

```
def yoyo(speed, waitTime):
    forward(speed); wait(waitTime)
    backward(speed); wait(waitTime)
    stop()
```

The first definition will not be accepted by Python, as shown below:

```
>>> def yoyo(speed, waitTime):
      forward(speed)
      wait(waitTime)
      backward(speed)
      wait(waitTime)
      stop()
SyntaxError: invalid syntax
>>> |
```

It reports that there is a syntax error and it highlights the error location by placing the thick red cursor (see the third line of the definition). This is because Python strictly enforces the indentation rule described above. The second definition, however, is acceptable. For two reasons: indentation is consistent; and commands on the same line can be entered separated by a semi-colon (;). We would recommend that you continue to enter each command on a separate line and defer from using the semi-colon as a separator until you are more comfortable with Python. More importantly, you

will notice that IDLE helps you in making your indentations consistent by automatically indenting the next line, if needed.

Another feature built into IDLE that enables readability of Python programs is the use of color highlighting. Notice in the above examples (where we use screen shots from IDLE) that pieces of your program appear in different colors. For example, the word `def` in a function definition appears in red, the name of your function, `yoyo` appears in blue. Other colors are also used in different situations, look out for them. IDLE displays all Python words (like `def`) in red and all names defined by you (like `yoyo`) in blue.

The idea of defining new functions by using existing functions is very powerful and central to computing. By defining the function `yoyo` as a new function using the existing functions (`forward`, `backward`, `wait`, `stop`) you have *abstracted* a new behavior for your robot. You can define further higher-level functions that use `yoyo` if you want. Thus, functions serve as basic building blocks in defining various robot behaviors, much like the idea of using building blocks to build bigger structures. As an example, consider defining a new behavior for your robot: one that makes it behave like a yoyo twice, followed by wiggling twice. You can do this by defining a new function as follows:

```
>>> def dance():
    yoyo(0.5, 0.5)
    yoyo(0.5, 0.5)
    wiggle(0.5, 1)
    wiggle(0.5, 1)

>>> dance()
```

Do This: Go ahead and add the `dance` function to your `moves.py` module. Try the `dance` command on the robot. Now you have a very simple behavior that makes the robot do a little shuffle dance.

Guided by Automated Controls

Earlier we agreed that a robot is a “mechanism guided by automated controls”. You can see that by defining functions that carry out more complex movements, you can create modules for many different kinds of behaviors. The modules make up the programs you write, and when they are invoked on the robot, the robot carries out the specified behavior. This is the beginning of being able to define automated controls for a robot. As you learn more about the robot’s capabilities and how to access them via functions, you can design and define many kinds of automated behaviors.

Summary

In this chapter, you have learned several commands that make a robot move in different ways. You also learned how to define new commands by defining new Python functions. Functions serve as basic building blocks in computing and defining new and more complex robot behaviors. Python has specific syntax rules for writing definitions. You also learned how to save all your function definitions in a file and then using them as a module by importing from it. While you have learned some very simple robot commands, you have also learned some important concepts in computing that enable the building of more complex behaviors. While the concepts themselves are simple enough, they represent a very powerful and fundamental mechanism employed in almost all software development. In later chapters, we will provide more details about writing functions and also how to structure parameters that customize individual function invocations. Make sure you do some or all of the exercises in this chapter to review these concepts.

Myro Review

`backward(SPEED)`

Move backwards at `SPEED` (value in the range -1.0...1.0).

`backward(SPEED, SECONDS)`

Move backwards at `SPEED` (value in the range -1.0...1.0) for a time given in `SECONDS`, then stop.

`forward(SPEED)`

Move forward at `SPEED` (value in the range -1.0..1.0).

`forward(SPEED, TIME)`

Move forward at `SPEED` (value in the range -1.0...1.0) for a time given in seconds, then stop.

`motors(LEFT, RIGHT)`

Turn the left motor at `LEFT` speed and right motor at `RIGHT` speed (value in the range -1.0...1.0).

`move(TRANSLATE, ROTATE)`

Move at the `TRANSLATE` and `ROTATE` speeds (value in the range -1.0...1.0).

`rotate(SPEED)`

Rotates at `SPEED` (value in the range -1.0...1.0). Negative values rotate right (clockwise) and positive values rotate left (counter-clockwise).

`stop()`

Stops the robot.

`translate(SPEED)`

Move in a straight line at `SPEED` (value in the range -1.0...1.0). Negative values specify backward movement and positive values specify forward movement.

`turnLeft(SPEED)`

Turn left at `SPEED` (value in the range -1.0...1.0)

```
turnLeft(SPEED, SECONDS)
```

Turn left at `SPEED` (value in the range -1.0..1.0) for a time given in seconds, then stops.

```
turnRight(SPEED)
```

Turn right at `SPEED` (value in the range -1.0..1.0)

```
turnRight(SPEED, SECONDS)
```

Turn right at `SPEED` (value in the range -1.0..1.0) for a time given in seconds, then stops.

```
wait(TIME)
```

Pause for the given amount of `TIME` seconds. `TIME` can be a decimal number.

Python Review

```
def <FUNCTION NAME>(<PARAMETERS>):  
    <SOMETHING>  
    ...  
    <SOMETHING>
```

Defines a new function named `<FUNCTION NAME>`. A function name should always begin with a letter and can be followed by any sequence of letters, numbers, or underscores (`_`), and not contain any spaces. Try to choose names that appropriately describe the function being defined.

Exercises

1. Compare the robot's movements in the commands `turnLeft(1)`, `turnRight(1)` and `rotate(1)` and `rotate(-1)`. Closely observe the robot's behavior and then also try the motor commands:

```
>>> motors(-0.5, 0.5)  
>>> motors(0.5, -0.5)  
>>> motors(0, 0.5)  
>>> motors(0.5, 0)
```

Do you notice any difference in the turning behaviors? The `rotate` commands make the robot turn with a radius equivalent to the width of the robot (distance between the two left and right wheels). The `turn` command causes the robot to spin in the same place.

2. Insert a pen in the scribbler's pen port and then issue it command to go forward for 1 or more seconds and then backwards for the same amount. Does the robot travel the same distance? Does it traverse the same trajectory? Record your observations.

3. Measure the length of the line drawn by the robot in Exercise 2. Write a function `travel(DISTANCE)` to make the robot travel the given `DISTANCE`. You may use inches or centimeters as your units. Test the function on the robot a few times to see how accurate the line is.

4. Suppose you wanted to turn/spin your robot a given amount, say 90 degrees. Before you try this on your robot, do it yourself. That is, stand in one spot, draw a line dividing your two feet, and then turn 90 degrees. If you have no way of measuring, your turns will only be approximate. You can study the behavior of your robot similarly by issuing it turn/spin commands and making them wait a certain amount. Try and estimate the wait time required to turn 90 degrees (you will have to fix the speed) and write a function to turn that amount. Using this function, write a behavior for your robot to transcribe a square on the floor (you can insert a pen to see how the square turns out).

5. Generalize the wait time obtained in Exercise 3 and write a function called `degreeTurn(DEGREES)`. Each time it is called, it will make the robot turn the specified degrees. Use this function to write a set of instructions to draw a square.

6. Using the functions `travel` and `degreeTurn`, write a function to draw the Bluetooth logo (See Chapter 1, Exercise 9).

7. Choreograph a simple dance routine for your robot and define functions to carry it out. Make sure you divide the tasks into re-usable moves and as much

as possible parameterize the moves so they can be used in customized ways in different steps. Use the building block idea to build more and more complex series of dance moves. Make sure the routine lasts for at least several seconds and it includes at least two repetitions of the entire sequence. You may also make use of the beep command you learned from the last section to incorporate some sounds in your choreography.

8. Record a video of your robot dance and then dub it with a soundtrack of your choosing. Use whatever video editing software accessible to you. Post the video online on sites like YouTube to share with friends.

9. Lawn mower robots and even vacuuming robots can use specific *choreographed* movements to ensure that they provide full coverage of the area to be serviced. Assuming that the area to be mowed or cleaned is rectangular and without any obstructions, can you design a behavior for your Scribbler to provide full coverage of the area? Describe it in writing. [Hint: Think about how you would mow/vacuum yourself.]