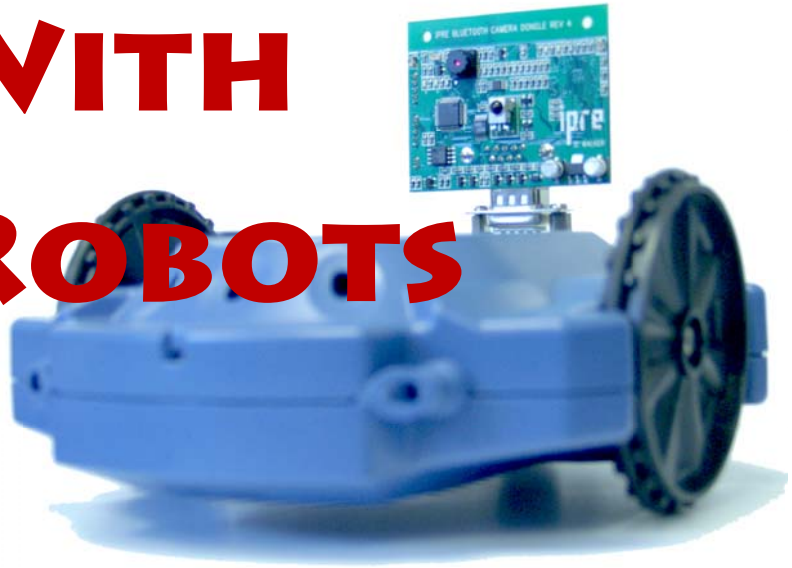
Learning Computing With Robots



Edited By

Deepak Kumar

Institute For Personal Robots in Education

www.roboteducation.org

Contributors

This text is an outcome of work done under the auspices of the Institute for Personal Robots in Education (IPRE). IPRE was created to explore the use of personal robots in education with primary funding from Microsoft Research, Georgia Institute of Technology, and Bryn Mawr College. This text would not have been possible without the hardware, software, and course materials that have led to the development of this text. Several people have directly or indirectly contributed to the material in this text. We list them below in alphabetical order.

Ben Axelrod, Georgia Institute of Technology
Tucker Balch, Georgia Institute of Technology
Douglas Blank, Bryn Mawr College
Natasha Eilbert, Bryn Mawr College
Ashley Gavin, Bryn Mawr College
Gaurav Gupta, Georgia Institute of Technology
Mansi Gupta, Bryn Mawr College
Mark Guzdial, Georgia Institute of Technology
Jared Jackson, Microsoft Research
Deepak Kumar, Bryn Mawr College
Keith O'Hara, Georgia Institute of Technology
Jay Summet, Georgia Institute of Technology
Monica Sweat, Georgia Institute of Technology
Stewart Tansley, Microsoft Research
Daniel Walker, Georgia Institute of Technology

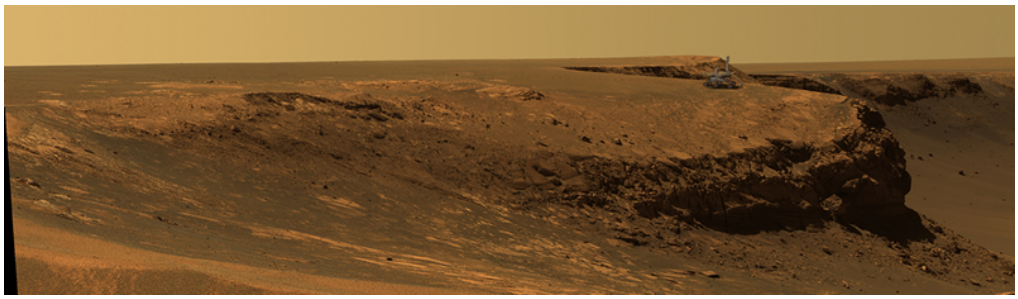
Contents

1 The World of Robots

Mars Rover.
Photo courtesy of NASA/JPL-Caltech

*I wouldn't ever want them to be brought back to Earth. We built them for Mars, and Mars is where they should stay. But **Spirit** and **Opportunity** have become more than just machines to me. The rovers are our surrogates, our robotic precursors to a world that, as humans, we're still not quite ready to visit.*

-: Steve Squyres in **Roving Mars**, Hyperion, 2005.



The rim of Victoria Crater on Mars.
The rover *Opportunity* has been superimposed on the crater rim to show scale.
Photo courtesy of JPL/NASA/Cornell University, October, 2006.

The picture above is among one of thousands sent back by *Spirit* and *Opportunity* from the surface of Mars. It goes without saying that it will probably be several years, decades or even more, before a human ever sets foot on Mars. The rovers *Spirit* and *Opportunity* landed on Mars in January 2004 as robot geologists whose mission was to analyze rocks and soils of the red planet in search for clues to past presence of water on the planet. The two robots were expected to last about 90 days. Three years later, they were still exploring the planet's surface and sending invaluable geological and pictorial data from the planet. Back on Earth, the same month as the rovers landing on Mars, the Tumbleweed robot rover traveled 40 miles across Antarctica's polar plateau transmitting local meteorological data back to its base station via satellite. Besides surviving adverse conditions on mars and Antarctica, robots are slowly becoming household consumer items. Take for instance, the Roomba from iRobot Corporation. Introduced in 2002, over 2 million Roombas have been sold to vacuum and clean floors.

One commonality among the robots mentioned above is that they are all designed for very specific tasks: analyze rocks and soils on the surface of Mars, meteorology on the polar cap, or vacuuming a room. Yet, the core of robot technology is almost as easy to use as computers. In this course you will be given a personal robot of your own. Through this personal robot, you will

learn to give it instructions to do a variety of tasks. Like the robots mentioned above your robot is also a rover. However, unlike the robots above, your personal robot does not come pre-programmed to do any specific task. It has certain basic capabilities (that you will learn about) and it can be programmed to make use of its capabilities to do various tasks. We hope that the process of learning about the capabilities of your robot and making it do different things will be exciting and fun for you. In this chapter, we introduce you to the world of robots and then introduce you to your own personal robot and some of its capabilities.

What is a robot?

The Merriam-Webster Online Dictionary gives the following definitions of the word *robot*:

1. a machine that looks like a human being and performs various complex acts (as walking or talking) of a human being; also a similar but fictional machine whose lack of capacity for human emotions is often emphasized; and also an efficient insensitive person who functions automatically
2. a device that automatically performs complicated often repetitive tasks
3. a mechanism guided by automatic controls

In today's world, the first two definitions will probably be considered archaic (the third interpretation in the first definition notwithstanding). It was generally the case that robots were initially conceived as human-like entities, real or fictional, devoid of emotions, that performed tasks that were repetitive or full or drudgery. Today's robots come in all kinds of shapes and sizes and take on all kinds of tasks (see below for some examples). While many robots are put to use for repetitive or dull tasks (including the Roomba; unless you enjoy the therapeutic side effects of vacuuming :-), robots today are capable of doing a lot more than implied by the first two definitions above. Even in fictional robots the lack of emotional capacity seems to have been overcome (see for instance Steven Spielberg's movie, *Artificial Intelligence*).

For our purposes, the third definition is more abstract and perhaps more appropriate. A *robot* is a mechanism or an artificial entity that can be guided by automatic controls. The last part of the definition, *guided by automatic controls*, is what we will focus on in this course. That is, given a mechanism capable of such guidance, what is involved in creating its controls?

Today, it is hard to imagine life without a web search engine. While there are several search engines available, the one provided by Google Inc. has become synonymous with web searching. So much so that people use it as a common phrase: "Google it!"

You may have your own personal preference for a search engine. Go ahead use it and search for the items suggested here.

A Short History of Robots

Modern robots were initially conceived as industrial robots designed to assist in automated manufacturing tasks. The first commercial robot company, Unimation, was created nearly 50 years ago. As the use of robots in industrial manufacturing grew, people also started experimenting with other uses of robots. Earlier industrial robots were mainly large arms that were attached to a fixed base. However, with the development of mobile robots people started to find uses for them in other domains. For instance, in exploring hazardous environments ranging from radioactive sites, volcanoes, finding and destroying mines, military surveillance, etc. We started this chapter by introducing you to two Mars rover robots. The first ever planetary rover landed on Mars in 1997. Increasingly in the last decade or so robots have ventured into newer and more exciting areas like medicine (Google: *robotic surgery*, *robot wheelchair*, etc.), toys and entertainment (Google: *Pleo*, *SONY Aibo*, *LEGO Mindstorms*, etc.), and even education (Google: *IPRE*). Some of the most exciting developments in robotics are still in research stages where, for example, in Artificial Intelligence research people are trying to develop intelligent robots and also using robots to understand and

explore models of human intelligence. Here we have provided some pointers (do the searches mentioned above) for examples of various robots and their uses. There are numerous web sites where you can look up more about the history of robots. We will leave that as an exercise.

Robots and Computers

A Postage stamp titled *World of Invention (The Internet)* was issued by UK's Royal Mail on March 1, 2007 honoring the development of the World Wide Web.

In the last few decades computers have become increasingly ubiquitous. Most likely you are reading this sentence on a computer. If you're reading this text online, the text itself is coming to you from another computer (located somewhere on the western banks of the Delaware River in south-eastern parts of the state of Pennsylvania in the United Stated of America). On its journey from the computer in Pennsylvania to your computer, the text has probably traveled through several computers (several dozen if you are outside the state of Pennsylvania!). What makes this journey of this text almost instantaneous is the presence of communication networks over which the internet and the World Wide Web operate. Advances in the technologies of wireless communication networks make it possible to access the internet from nearly any place on the planet. The reason that you are sitting in front of a computer and learning about robots is primarily because of the advent of these technologies. While robots are not quite as ubiquitous as computers, they are not too far behind. In fact, it is precisely the advances in computers and communications technologies that have made it feasible for you to become more familiar with the world of robots.

The relationship between robots and computers is the basis for the use of the phrase *automatic controls* in describing a robot. Automatically controlling a robot almost always implies that there is a computer involved. So, in the process of learning about and playing with robots you will also uncover the world of computers. Your robot has a computer embedded in it. You will be controlling the robot through your computer. Moreover, you will do this over a wireless communication technology called *bluetooth*. Initially, for our purposes, learning to automatically control a robot is going to be synonymous with learning to control a computer. This will become more obvious as we proceed through these lessons.

Automating control involves specifying, in advance, the set of tasks the robot or the computer is to perform. This is called *programming*. Programming involves the use of a *programming language*. Today, there are more programming languages than human languages! Perhaps you have heard of some of them: Java, C, Python, etc. In this course, we will do all our robot programming in the programming language *Python*. Python, named after the popular Monty Python TV shows, is a modern language that is very easy to learn and use.

While we are talking about computers and languages, we should also mention the *Myro* (for *My ro*bot) software system. Myro was developed by us to simplify the programming of robots. Myro provides a small set of robot commands that extend the Python language. This makes it easy, as you will see, to specify automatic controls for robots.

A Robot of Your Own: The Scribbler

The Scribbler Robot

The scribbler robot, shown here is also a rover. It can move about in its environment. The wheels, and its other functions, can be controlled through a computer via a wireless interface. Your laboratory assistants will provide you with a Scribbler and the required components to enable wireless

communication. Once configured, you will be able to control the robot's movements (and all other features) through the computer. Besides moving, your robot can also play sounds (beeps) and, with the help of a pen inserted in its pen port, it can draw a line wherever it goes (hence its name, *Scribbler*). The robot can move forward, backward, turn, spin, or perform any combination of these movements giving it adequate functionality to travel anywhere on the surface of an environment. Besides roving, the Scribbler robot can also sense certain features of its environment. For example, it is capable of sensing a wall or an obstacle, or a line on the floor. We will discuss the Scribblers sensing capabilities later.

Do This

The first few activities show you how you to set up the computer and the robot and will help you get familiarized with your Scribbler. This will involve the following four activities:

1. First things first: Setup Myro
2. Name your robot
3. Drive your robot around
4. Explore a little further

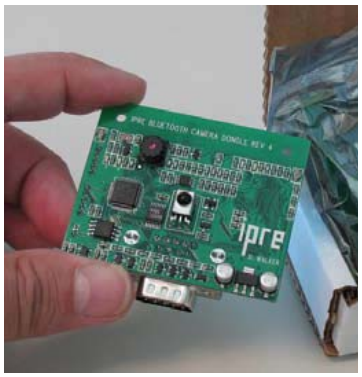
You may need the assistance of your instructor for the first activity to ensure that you know how to set up and use your robot for the remainder of the text.

Dear Student:

Every chapter in this book will include several hands-on activities. These are short learning exercises designed carefully to explore some of the concepts presented in the chapter. It is important that you do all of the activities in the chapter before moving on to the next chapter.

We would also recommend trying out some (or all) of the exercises suggested at the end to gain a better understanding.

1. First things first: Setup Myro



The Fluke Dongle adds Bluetooth and other capabilities to the Scribbler.

At the time you received your robot, its software and hardware was configured for use. The software we will be using for controlling the robot is called, **Myro** (for **My Ro**bot) which works in conjunction with the Python language. In this, the first exercise, we will start the robot and the software and ensure that the software is able to successfully communicate with the robot through your computer. If Myro has not been installed on your computer, you should go ahead and obtain a copy of it (by inserting the Myro CD into your computer or following directions from the Myro Installation Manual.

The Scribbler robot with the Fluke Dongle.

In a typical session, you will start the Python software, connect to the robot through the Myro library, and then control the robot through it. We have set up the system so that all communication between the computer and the robot occurs wirelessly over a Bluetooth connection. Bluetooth technology is a common wireless communication technology that enables electronic devices to talk to each other over short distances. For example, Bluetooth is most commonly used in cell phones to enable wireless communication between a

cell phone (which may be in your pocket) and your wireless headset. This kind of communication requires two physical devices that serve as receivers and transmitters. In the Scribbler kit you received, there is a pair of these Bluetooth devices: one connects into the scribbler (Fluke Dongle) and the other into the USB port of your computer. If your computer has a built-in Bluetooth capability, you may not need the one that goes into your computer. Go ahead and make sure that these devices are plugged in, the robot is turned on, and so is your computer.

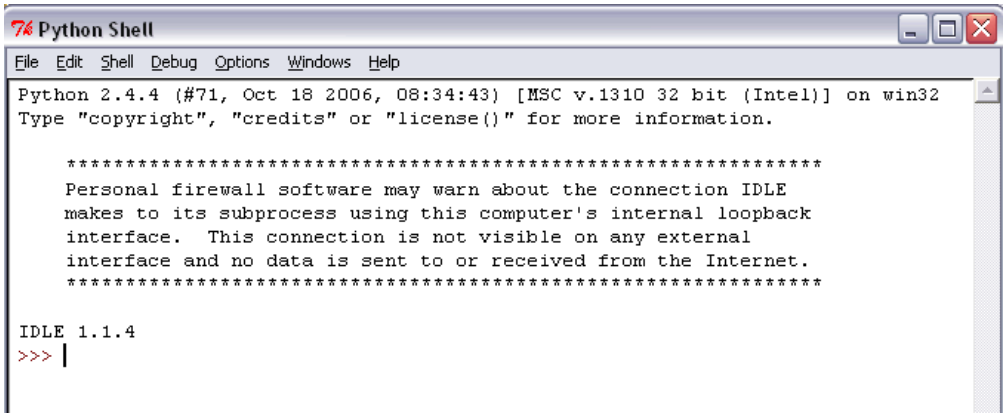
2. Name your robot

In this exercise, we will connect to the robot and make it do something simple, like make it beep. Then, we will give the robot a name to personalize it. These tasks can be performed by using the following steps:

1. Start Python
2. Connect to the robot
3. Make the robot beep
4. Give the robot a name

Since this is your very first experience with using robots, we will provide detailed instructions to accomplish the task outlined above.

1. Start Python: When you installed the software, a file called `Start Python.pyw` was created. You should copy this file into a folder where you plan to store all your robot programs. Once done, navigate to that folder and open it. In it you will find the `Start Python` icon. Go ahead and double-click on it. The following window should appear on your computer screen:

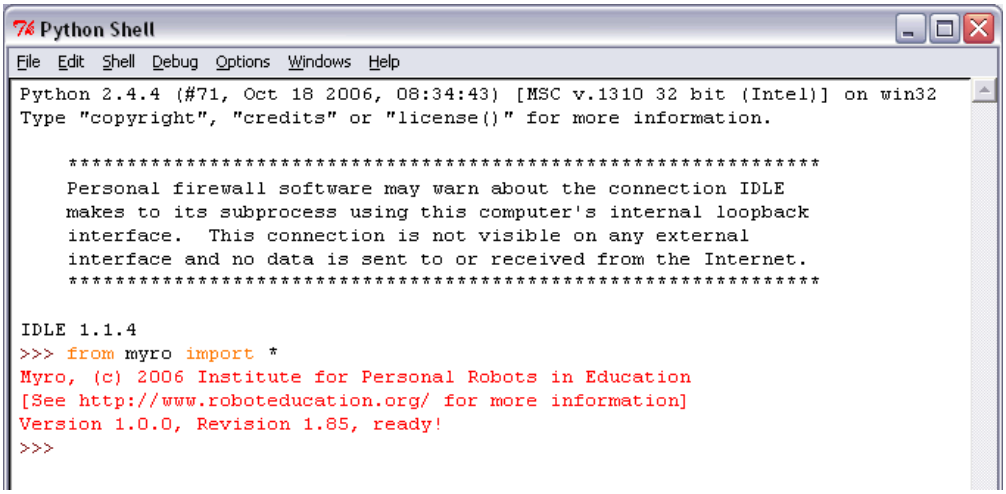


What you see above is the Python interaction window or the *Python Shell*. This particular shell is called *IDLE* (notice that it reports above that you are using `IDLE Version 1.1.4`.). You will be entering all Python commands in this IDLE window. The next step is to use Myro to coonnect to the robot.

2. Connect to the robot: Make sure your robot and the computer have their Bluetooth dongles inserted and that your robot is turned on. To connect to the robot enter the following command into the Python shell:

```
>>> from myro import *
```

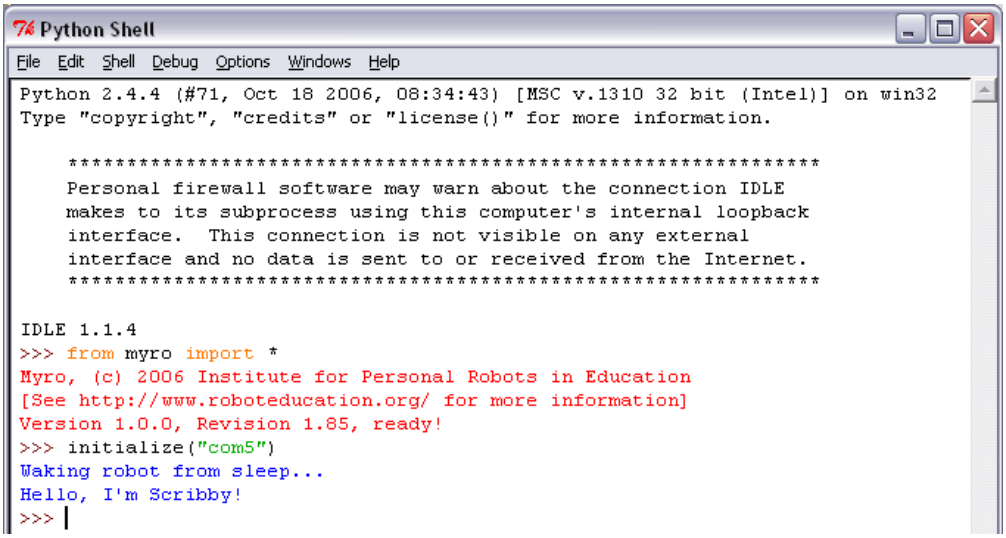
This interaction is shown below:



That is, you have now informed the Python Shell that you will be using the Myro library. The import statement/command is something you will use each time you want to control the robot. After issuing the import, some useful information is printed about Myro and then the Shell is ready for the next Python command. Now it is time to connect to the robot by issuing the following command:

```
>>> initialize("comX")
```

where `X` is the port number using which your computer is using to communicate with the robot. If you need help figuring out the port number, consult with your instructor or a TA. The example below shows how to issue the command when the port `com5` is being used:



When you issue the `initialize` command, the computer attempts to communicate with the robot. If this is successful, the robot responds with the `Hello...` line shown above. As you can see, the robot's name is `Scribby`. You can give your robot whatever name you like. We will do that later. First, let us give it a command to make a beep so that we know that we are in control of the robot:

3. Make the robot beep: In the Python Shell, enter the command:

```
>>> beep(1, 880)
```

The command above directs the robot to make a beeping sound at 880 Hertz for 1 second. Go ahead and try it. Your robot will beep for 1 second at 880 Hz. Go ahead and try the following variations to hear different beeps:

```
beep(0.5, 880)
beep(0.5, 500)
etc.
```

So now, you should realize that you are in control of the robot. By issuing simple commands like the ones above, you can make the robot perform

different behaviors. Now, we can learn the command to give the robot a new name.

4. Give the robot a name: Suppose we wanted to name the robot `Shrek`. To do this, all you have to do it give it the following command:

```
>>> setName("Shrek")
```

Whatever name you decide to give your robot, you can specify it in the command above replacing the words `Shrek`. From now on, that will be the name of the robot. How do we know this for a fact? Go ahead and try asking it its name:

```
>>> getName()
```

It will also report than name each time you connect to it using the initialize command:

```
>>> initialize("com5")
Waking robot from sleep...
Hello, I'm Shrek!
>>>
```

Congratulations! You have now completed the first Exercise and you are well on your way to more fun and exciting things with your robot. Before we move on, it would be a good idea to review what you just did. Each session with a robot begins starting the Python software (Step 1 above), followed by importing the Myro library and initializing the robot. From then on, you can issue any commands to the robot.

The Myro library contains dozens of commands that enable various kinds of robot behaviors. In the next few weeks we will be learning several robot commands and learning how to use them to program complex robot behaviors. One thing to remember at this juncture is that all commands are being issued in the Python language. Thus, as you learn more about your robot and its behaviors, you will also be learning the Python language.

One characteristic of programming languages (like Python) is that they have a very strict way of typing commands. That is, and you may already have experienced this above, the language is very precise about what you type and how you type it. Every parenthesis, quotation mark, and upper and lower case letter that makes up a command has to be typed exactly as described. While the rules are strict luckily there aren't too many of them. Soon you will get comfortable with this syntax and it will become second nature. The precision in syntax is required so that the computer can determine exactly one interpretation for the command resulting in desired action. For this reason, computer languages are often distinguished from human languages by describing them as *formal languages* (as opposed to *natural languages* that are used by humans).

3. Drive the robot around

In this exercise, we will introduce you to a way of making the robot move about in its environment manually controlled by a game pad device (see picture on right). As above, place the robot on an open floor, turn the robot on, start Python as above and connect to the robot. You may already have this from Exercise 2 above. Also, plug the game pad controller into an available USB port of your computer. At the prompt, enter the following command:

The game pad controller.

```
>>> gamepad()
```

In response to this command, you will get some help text printed in the IDLE window showing what would happen if you pressed various game pad buttons. If you look in the picture of the game pad controller above, you will notice that it has eight (8) blue buttons (numbered 1 through 8 in the picture), and an axis controller (the big blue swivel button on the left). The axis controller can be used to move the robot around. Go ahead and try it. Pressing

each of the numbered buttons will result in different behaviors, some will make the robot beep, some will make the computer speak or say things. Button#1 will result in the robot taking a picture of whatever it is currently seeing through its camera and display it on your computer screen. Button#8 will quit from the game pad control mode.

Spend some time experimenting with the various game pad control features. See how well you can navigate the robot to go to various places, or follow a wall, or go round and round something (like yourself!). You may also place the robot on a big piece of paper, insert a Sharpie pen in its pen port and then move it around to observe its scribbling. Can you scribble your name (or initials)? Try a pattern or other shapes.

Without creating a program, this is an effective remote way of controlling the movements of your robot. The next exercise asks you to try and issue commands to the robot to move.

4. Explore a little further

OK, now you are on your own. Start Python, import Myro, connect to the robot, and give commands to move forward, backward, turn left and right, and spin. Use the commands: forward(SPEED), backward(SPEED), turnLeft(SPEED), turnRight(SPEED), and rotate(SPEED)). SPEED can be any number between -1.0...1.0. These and all other robot commands are detailed in the Myro Reference Manual. This would be a good time to review the descriptions of all the commands introduced in this section.

Myro Review

```
from myro import *
```
This command imports all the robot commands available in the Myro library. We will use this whenever we intend to write programs that use the robot.

```
initialize(<PORT NAME>)
init(<PORT NAME>)
```
This command establishes a wireless communication connection with the

robot. <PORT NAME> is determined at the time you configured your software during installation. It is typically the word com followed by a number. For example, "com5". The double quotes (") are essential and required.

beep(<TIME>, <FREQUENCY>)
Makes the robot beep for <TIME> seconds at frequency specified by <FREQUENCY>.

getName()
Returns the name of the robot.

setName(<NEW NAME>)
Sets the name of the robot to <NEW NAME>. The new name should be enclosed in double quotes, no spaces, and not more than 16 characters long. For example: setName("Bender").

gamepad()
Enables manual control of several robot functions and can be used to move the robot around.

Python Review

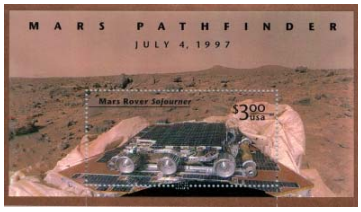
Start Python.pyw
This is the icon you double-click on to start a Python Shell (IDLE).

>>>
The Python prompt. This is where you type in a Python command.

Note: All commands you type (including the Myro commands listed above) are essentially Python commands. Later, in this section we will list those commands that are a part of the Python language.

Exercises

1. Where does the word *robot* come from? Explore the etymology of the words *robot* and *robotics* and write a short paper about it.
2. What are Asimov's Laws of robotics? Write a viewpoint essay on them.
3. Look up the Wikipedia entry on robots, as well as the section in AI Topics (see links above). Write a short essay on the current state of robots.
4. Write a short essay on a robot (real or fictional) of your choice. Based on what you have learned from your readings, evaluate its capabilities.
5. *Spirit* and *Opportunity* were not the first rovers to land on Mars. On July 4, 1997, the *Mars Pathfinder* landed on Mars with a payload that included the *Sojourner* rover. The United States Postal Service issued the stamp shown here to commemorate the landing. This is perhaps the first ever real robot to appear on a postage stamp! Find out what you can about the *Mars Pathfinder* mission and compare the *Sojourner* rover with *Spirit* and *Opportunity*.



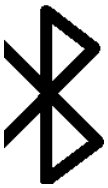
Facsimile of the Mars Pathfinder Postage Stamp

6. Through the exercises, you have experienced a subset of capabilities of the Scribbler robot. Reflect/write about the physical capabilities of the Scribbler and the kinds of tasks you could make it perform.
7. Insert a pen (provided in your kit) in the robot's pen port. Place the robot on a surface where it is OK to write/draw. Drive the robot around with the game pad controller. It will scribble on the paper as it moves. Observe its scribbles by moving it forward and then backwards. Does it trace its path exactly? Why or why not?
8. Using the game pad operation, make your robot Scribble your name on the floor. You may find this difficult for several reasons. Try to make the robot write your initials instead. Also, see if you can guide the robot to draw a five point star. This task is in some sense not too

different from controlling a robot to perform surgery. Research the capabilities of today's surgical robots and write a short paper about it.

9. Using the game pad controller draw the Bluetooth logo (see picture) using a pen inserted in the Scribbler robot. Do a web search for Harald Blåtand and read more about the runic alphabets.

Harald Blåtand Gormson
What's in a name?



The Bluetooth logo is derived from *runic* alphabet letters H and B juxtaposed together. HB for Harald Blåtand a Scandinavian King (from the 10th century AD) who was legendary in uniting Denmark and Norway. The wireless technology we use today is named in his honor (Blåtand means "Bluetooth") because the technology itself was developed by Ericsson, a Scandinavian company. The technology is designed to unite computers and telecomm devices. Bluetooth devices are most commonly found in cell phones. We're using it here to communicate between your robot and the computer.

Further Reading

1. Wikipedia entry on Robots (http://en.wikipedia.org/wiki/Robot)
2. AI Topics: Robots from the American Association for Artificial Intelligence (AAAI) (http://www.aaai.org/AITopics/html/robots.html)
3. Social Robots are robots that interact with and learn from people around them. Here is an interview with Cynthia Breazeal who heads the Robotic Life Group at MIT's Media Lab. (http://www.pbs.org/saf/1510/features/breazeal.htm)
4. Visit the online Robot Hall of Fame and find out more about the real and fictional robots that have been inducted into it. (http://www.robothalloffame.org/)

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

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 above 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 here) that is being used for therapeutic purposes for nursing home patients.

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

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.

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)
```

In the command above, `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)
backwards(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).

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.

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

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:

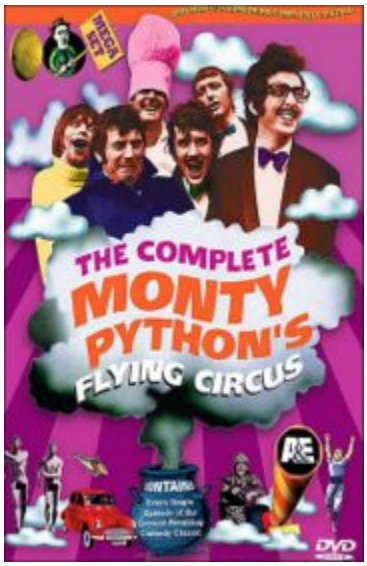
```
>>> def yoyo():
        forward(1, 1)
        backward(1, 1)
        stop()
```

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**eve**L**opment **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)
        stop()
```

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

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

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.

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 *
initialize("com5")

# 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 `initialize` commands at the top. This may or may not be necessary, check with your instructor.

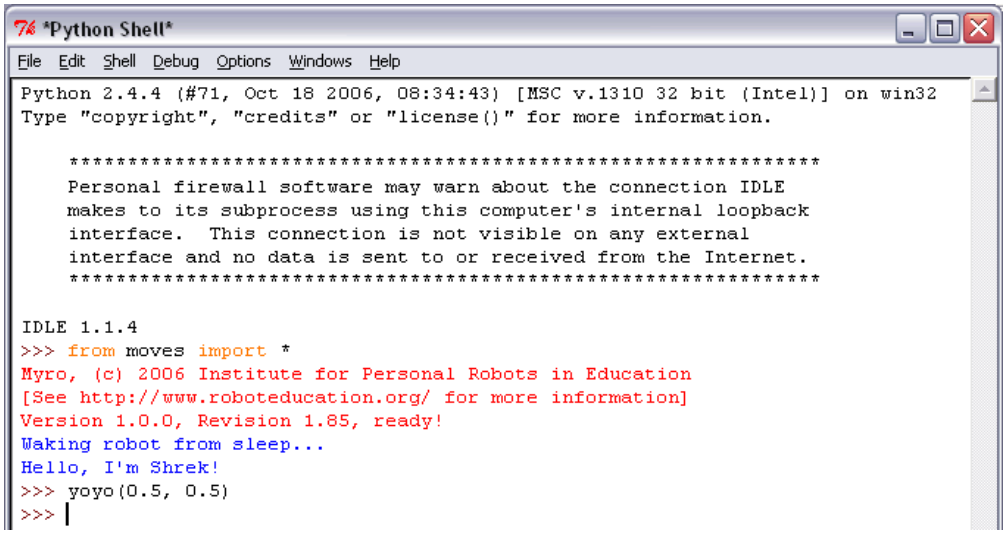
Do This: To store the `yoyo` and `wiggle` behaviors as a module in a file, you can ask IDLE for a `New Window` from the `File` menu. Next enter the text containing the two definitions and then save them in a file (let's call it `moves.py`) in your Myro folder (same place you have the `Start Python`

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:



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 (<FUCTION 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:

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

Exercise 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.

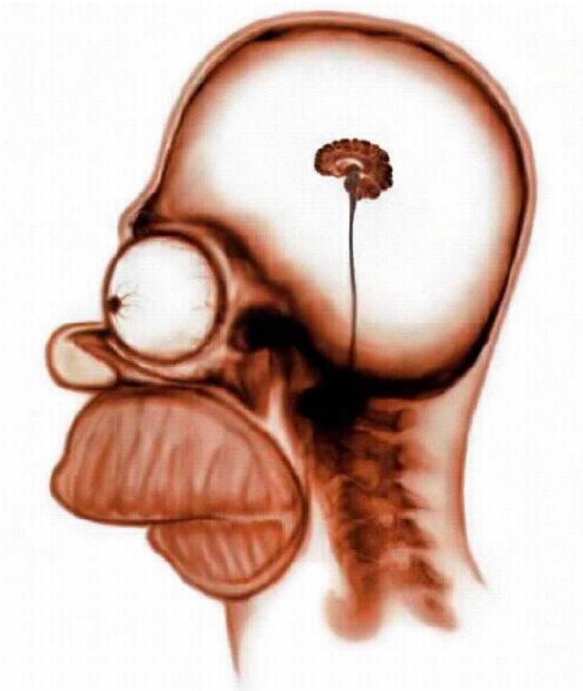
Exercise 2: Insert a pen in the scribbler's pen port and then issue it command to go forward for 1 or more seconds and then backwards for the same amount. Does the robot travel the same distance? Does it traverse the same trajectory? Record your observations.

Exercise 3: 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).

Exercise 4: 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.

Exercise 5: 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.]

3 Building Brains



What a splendid head, yet no brain.
Aesop (620 BC-560 BC)

If you think of your robot as a creature that acts in the world, then by programming it, you are essentially building the creature's brain. The power of computers lies in the fact that the same computer or the robot can be supplied a different program or brain to make it behave like a different creature. For example, a program like Firefox makes your computer behave like a web browser. But switching to your Media Player, the computer behaves as a DVD or a CD player. Similarly, your robot will behave differently depending upon the instructions in the program that you have requested to run on it. In this chapter we will learn about the structure of Python programs and how you can organize different robot behaviors as programs.

The world of robots and computers, as you have seen so far is intricately connected. You have been using a computer to connect to your robot and then controlling it by giving it commands. Most of the commands you have used so far come from the Myro library which is specially written for easily controlling robots. The programming language we are using to do the robot programming is Python. Python is a general purpose programming language. By that we mean that one can use Python to write software to control the computer or another device like a robot through that computer. Thus, by learning to write robot programs you are also learning how to program computers. Our journey into the world of robots is therefore intricately tied up with the world of computers and computing. We will continue to interweave concepts related to robots and computers throughout this journey. In this chapter, we will learn more about robot and computer programs and their structure.

Basic Structure of a Robot Brain

The basic structure of a Python program (or a robot brain) is shown below:

```
def main():
   <do something>
   <do something>
   ...
```

This is essentially the same as defining a new function. In fact, here, we are adopting a convention that all our programs that represent robot brains will be called `main`. In general, the structure of your robot programs will be as shown below (we have provided line numbers so we can refer to them):

```
Line 1: from myro import *
Line 2: initialize("comX")

Line 3: <any other imports>
Line 4: <function definitions>
Line 5: def main():
Line 6:     <do something>
Line 7:     <do something>
Line 8:     ...

Line 9: main()
```

Every robot brain program will begin with the first two lines (`Line 1` and `Line 2`). These, as you have already seen, import the Myro library and establish a connection with the robot. In case you are using any other libraries, you will then import them (this is shown in `Line 3`). This is followed by the definitions of any other functions (`Line 4`), and then the definition of the function, `main`. Finally, the last line (`Line 9`) is an invocation of the function `main`. This is placed so that when you load this program into the Python Shell the program will start executing. In order to illustrate this, let us write a robot program that makes it do a short dance using the `yoyo` and `wiggle` movements defined in the last chapter.

```
# File: dance.py
# Purpose: A simple dance routine

# First import myro and connect to the robot

from myro import *
initialize("com5")

# Define the new functions...

def yoyo(speed, waitTime):
```

```
        forward(speed, waitTime)
        backward(speed, waitTime)
        stop()

def wiggle(speed, waitTime):
        motors(-speed, speed)
        wait(waitTime)
        motors(speed, -speed)
        wait(waitTime)
        stop()

# The main dance program
def main():
        print "Running the dance routine..."
        yoyo(0.5, 0.5)
        wiggle(0.5, 0.5)
        yoyo(1, 1)
        wiggle(1, 1)
        print "...Done"

main()
```

We have used a new Python command in the definition of the `main` function: the `print` command. This command will print out the text enclosed in double quotes (") when you run the program. This program is not much different from the `dance` function defined in the previous chapter except we are using a spin motion to wiggle. However, instead of naming the function `dance` we are calling it `main`. As we mentioned earlier, this is just a naming convention that we are adopting that makes it easy to identify the main program in a program file.

Do This: In order to run this program on the robot, you can start IDLE, create a new window, enter the program in it, save it as a file (`dance.py`) and then select the `Run Module` feature in the window's `Run` menu. Alternately, to run this program, you can enter the following command in the Python Shell:

```
>>> from dance import *
```

This is essentially equivalent to the `Run Module` option described above. When you run the program you will notice that the robot carries out the dance routine specified in the `main` program. Also notice the two messages printed in the IDLE window. These are the results of the `print` command. `print` is a very useful command in Python and can be used to output essentially anything you ask it to. While you are in this session, go ahead and change the print command to the following:

```
speak(“Running the dance routine”)
```

`speak` is a Myro command that enables speech output from your computer. Go ahead and change the other `print` command also to the speak command and try your program. Once done, enter some other speak commands on the IDLE prompt. For example:

```
speak(“Dude! Pardon me, would you have any Grey Poupon?”)
```

The speech facility is built into most computers these days. Later we will see how you can find out what other voices are available and also how to change to them.

Speaking Pythonese

We have launched you into the world of computers and robots without really giving you a formal introduction to the Python language. In this section, we provide more details about the language. What you know about Python so far is that it is needed to control the robot. The robot commands you type are integrated into Python by way of the Myro library. Python comes with several other useful libraries or modules that we will try and learn in this course. If you need to access the commands provided by a library, all you have to do is import them.

The libraries themselves are largely made up of sets of functions (they can contain other entities but more on that later). Functions provide the basic building blocks for any program. Typically, a programming language (and

Python is no exception) includes a set of pre-defined functions and a mechanism for defining additional functions. In the case of Python, it is the `def` construct. You have already seen several examples of function definitions and indeed have written some of your own by now. In the `def` construct, when defining a new function, you have to give the new function a *name*. Names are a critical component of programming and Python has rules about what forms a name.

What's in a name?

A name in Python must begin with either an alphabetic letter (a-z or A-Z) or the underscore (i.e. _) and can be followed by any sequence of letters, digits, or underscore letters. For example,

```
iRobot
myRobot
jitterBug
jitterBug2
my2cents
my_2_cents
```

are all examples of valid Python names. Additionally, another important part of the syntax of names is that Python is *case sensitive*. That is the names `myRobot` and `MyRobot` and `myrobot` are distinct names as far as Python is concerned. Once you name something a particular way, you have to consistently use that exact case and spelling from then on. Well, so much about the syntax of names, the bigger question you may be asking is *what kinds of things can (or should) be named?'*

So far, you have seen that names can be used to represent functions. That is, what a robot does each time you use a function name (like `yoyo`) is specified in the definition of that function. Thus, by giving functions a name you have a way of defining new functions. Names can also be used to represent other things in a program. For instance, you may want to represent a quantity, like speed or time by a name. In fact, you did so in defining the function `yoyo` which is also shown below:

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

Functions can take parameters that help customize what they do. In the above example, you can issue the following two commands:

```
>>> yoyo(0.8, 2.5)
>>> yoyo(0.3, 1.5)
```

The first command is asking to perform the `yoyo` behavior at speed 0.8 for 2.5 seconds where as the second one is specifying 0.3 and 1.5 for speed and time, respectively. Thus, by parameterizing the function with those two values, you are able to produce similar but varying outcomes. This idea is similar to the idea of mathematical functions: *sine(x)* for example, computes the sine of whatever value you supply for *x*. However, there has to be a way of defining the function in the first place that makes it independent of specific parameter values. That is where names come in. In the definition of the function `yoyo` you have named two parameters (the order you list them is important): `speed` and `waitTime`. Then you have used those names to specify the behavior that makes up that function. That is the commands `forward`, and `backward` use the names `speed` and `waitTime` to specify whatever the speed and wait times are included in the function invocation. Thus, the names `speed` and `waitTime` represent or designate specific values in this Python program. Names in Python can represent functions as well as values. What names you use is entirely up to you. It is a good idea to pick names that are easy to read, type, and also appropriately designate the entity they represent. What name you pick to designate a function or value in your program is very important, for you. For example, it would make sense if you named a function `turnRight` so that when invoked, the robot turned right. It would not make any sense if the robot actually turned left instead, or worse yet, did the equivalent of the yoyo dance. But maintaining this kind of semantic consistency is entirely up to you.

Values

In the last section we saw that names can designate functions as well as values. While the importance of naming functions may be obvious to you by now, designating values by names is an even more important feature of programming. By naming values, we can create names that represent specific values, like the speed of a robot, or the average high temperature in the month of December on top of the Materhorn in Switzerland, or the current value of the Dow Jones Stock Index, or the name of your robot, etc. Names that designate values are also called *variables*. Python provides a simple mechanism for designating values with names:

```
speed = 0.75
aveHighTemp = 37
DowIndex = 12548.30
myFavoriteRobot = "C3PO"
```

Values can be *numbers* or *strings* (anything enclosed in double-quotes, "). The above are examples of *assignment statements* in Python. The exact syntax of an assignment statement is given below:

```
<variable name> = <expression>
```

You should read the above statement as: *Let the variable named by* `<variable name>` *be assigned the value that is the result of calculating the expression* `<expression>`. So what is an `<expression>`? Here are some examples:

```
>>> 5
5
>>> 5 + 3
8
>>> 3 * 4
12
>>> 3.2 + 4.7
7.9
>>> 10 / 2
5
```

What you actually type at the Python prompt (>>>) is actually called an expression. The simplest expression you can type is a number (as shown above). A number evaluates to itself. That is, a 5 is a 5, as it should be! And 5 + 3 is 8. As you can see when you enter an expression, Python evaluates it and then outputs the result. Also, addition (+), subtraction (-), multiplication (*), and division (/) can be used on numbers to form expressions that involve numbers. You may have also noticed that numbers can be written as whole numbers (3, 5, 10, 1655673, etc) or with decimal points (3.2, 0.5, etc) in them. Python (and most computer languages) distinguishes between them. Whole numbers are called *integers* and those with decimal points in them are called *floating point* numbers. While the arithmetic operations are defined on both kinds of numbers, there are some differences you should be aware of. Look at the examples below:

```
>>> 10.0/3.0
3.3333333333333335
>>> 10/3
3
>>> 1/2
0
>>> 1.0/2
0.5
```

When you divide a floating point number by another floating point number, you get a floating point result. However, when you divide an integer by another integer, you get an integer result. Thus, in the examples above, you get the result 3.3333333333333335 when you divide 10.0 by 3.0, but you get 3 when you divide 10 by 3. Knowing this, the result of dividing 1 by 2 (see above) is zero (0) should not be surprising. That is, while the division operation looks the same (/), it treats integers differently than floating point values. However, if at least one of the numbers in an arithmetic operation is a floating point number, Python will give you a floating point result (see last example above). You should keep this in mind. More on numbers later, before we get back to robots, let us quickly introduce you to strings.

Computers came to be called so because they excelled in doing calculations. However, these days, computers are capable of manipulating any kind of

entity: text, images, sounds, etc. Text is made of letters or *characters* and strings are simply sequences of characters. Python requires that strings be written enclosed in quotes: which could be single (`'I am a string'`), double (`"Me too!"`), or even triple quotes (`'''I'm a string as well!'''`). Treating a string as a value is a powerful feature of Python. Python also provides some operations on strings using which you can write some useful string expressions. Here are some examples:

```
>>> mySchool = "Bryn Mawr College"
>>> yourSchool = "Georgia Institute of Technology"
>>> print mySchool
Bryn Mawr College

>>> print yourSchool
Georgia Institute of Technology

>>> print mySchool, yourSchool
Bryn Mawr College Georgia Institute of Technology

>>> yourSchool+mySchool
'Georgia Institute of TechnologyBryn Mawr College'

>>> print yourSchool+mySchool
Georgia Institute of TechnologyBryn Mawr College
```

Pay special attention to the last two examples. The operation + is defined on strings and it results in concatenating the two strings. The `print` command is followed by zero or more Python expressions, separated by commas. `print` evaluates all the expressions and prints out the results on the screen. As you have also seen before, this is a convenient way to print out results or messages from your program.

A Calculating Program

Ok, set your robot aside for just a few more minutes. You have now also learned enough Python to write programs that perform simple calculations. Here is a simple problem:

On January 1, 2008 the population of the world was estimated at approximately 6.650 billion people. It is predicted that at current rates of population growth, we will have over 9 billion people by the year 2050. A gross estimate of population growth puts the annual increase at +1.14% (it has been as high as +2.2% in the past). Given this data, can you estimate by how much the world's population will increase in this year (2008)? Also, by how much will it increase each day?

In order to answer the questions, all you have to do is compute 1.14% of 6.650 billion to get the increase in population this year. If you divide that number by 365 (the number of days in 2009) you will get average daily increase. You can just use a calculator to these simple calculations. You can also use Python to do this in two ways. You can use it as a calculator as shown below:

```
>>> 6650000000*1.14/100.0
75810000.0

>>> 75810000.0/365.0
207698.63013698629
```

That is, in this year there will be an increase of 75.81 million in the world's population which implies an average daily increase of over 207 thousand people). So now you know the answer!

Also, let us try and write a program to do the above calculations. A program to do the calculation is obviously going to be a bit of overkill. Why do all the extra work when we already know the answer? Small steps are needed to get to higher places. So let's indulge and see how you would write a Python program to do this. Below, we give you one version:

```
#File: worldPop.py
# Purpose:
#     Estimate the world population growth in a year and
#     also per day.
#     Given that on Jnauray 1, 2008 the world's population was
```

```
#     estimated at 6,650,000,000 and the estimated growth is
#     at the rate of +1.14%

def main():
    population = 6650000000
    growthRate = 1.14/100.0

    growthInOneYear = population * growthRate
    growthInADay = growthInOneYear / 365

    print "World population on January 1, 2008 is", population
    print "By Jan. 1, 2009, it will grow by", growthInOneYear
    print "An average daily increase of", growthInADay

main()
```

The program follows the same structure and conventions we discussed above. In this program, we are not using any libraries (we do not need any). We have defined variables with names `population`, and `growthRate` to designate the values given in the problem description. We also defined the variables `grothInOneYear` and `growthInADay` and use them to designate the results of the calculations. First, in the `main` program we assign the values given, followed by performing the calculation. Finally, we use the print commands to print out the result of the computations.

Do This: Start Python, enter the program, and run it (just as you would run your robot programs) and observe the results. Voila! You are now well on your way to also learning the basic techniques in computing! In this simple program, we did not import anything, nor did we feel the need to define any functions. But this was a trivial program. However, it should serve to convince you that writing programs to do computation is essentially the same as controlling a robot.

Using Input

The Energy Problem

The root cause of world energy problems is growing world population and energy consumption per capita.

How many people can the earth support? Most experts estimate the limit for long-term sustainability to be between 4 and 16 billion.

From: Science, Policy & The Pursuit of Sustainability, Edited by Bent, Orr,and Baker. Illus. by Shetter. Island Press, 2002.

The program we wrote above uses specific values of the world's population and the rate of growth. Thus, this program solves only one specific problem for the given values. What if we wanted to calculate the results for a different growth rate? Or even a different estimate of the population? What if we wanted to try out the program for varying quantities of both? Such a program would be much more useful and could be used over and over again. Notice that the program begins by assigning specific values to the two variables:

```
population = 6650000000
growthRate = 1.14/100.0
```

One thing you could do is simply modify those two lines to reflect the different values. However, typical programs are much more complicated than this one and it may require a number of different values for solving a problem. When programs get larger, it is not a good idea to modify them for every specific problem instance but it is desirable to make them more useful for all problem instances. One way you can achieve this is by using the *input* facilities of Python. All computer programs typically take some input, do some computation (or something), and then produce some output. Python has a simple input command that can be used to rewrite the program above as follows:

```
#File: worldPop.py
# Purpose:
#     Estimate the world population growth in a year and
#     also per day.
#     Given that on Jnauray 1, 2008 the world's population was
#     estimated at 6,650,000,000 and the estimated growth is
#     at the rate of +1.14%

def main():
    # print out the preamble

    print "This program computes population growth figures."

    # Input the values
    population = input("Enter current world population: ")
    growthRate = input("Enter the growth rate: ")/100.0

    # Compute the results
    growthInOneYear = population * growthRate
    growthInADay = growthInOneYear / 365

    # output results
    print "World population today is", population
    print "In one year, it will grow by", growthInOneYear
    print "An average daily increase of", growthInADay

main()
```

Read the program above carefully. Notice that we have added additional comments as well as print statements. This improves the overall readability as well as the interaction of this program. Notice the use of the `input` statements above. The basic syntax of `input` is shown below:

```
<variable name> = input(<some prompt string>)
```

That is, the `input` is a function whose parameter is a string and the value it returns is the value of the expression that will be entered by the user. When executed, the computer will print out the prompt and wait for the user to enter a Python expression. The user can enter whatever expression in response to

the prompt and then hit the RETURN or ENTER key. The expression is then evaluated by Python and the resulting value is returned by the `input` function. That value is then assigned to the variable `<variable name>`. The statement above uses the same syntax as the assignment statement described above. Python has made obtaining input from a user easy by defining `input` as a function. Now, look at the use of the `input` function in the program above. With this modification, we now have a more general program which can be run again and again. Below, we show two sample runs:

```
'''
This program computes population growth figures.
Enter current world population: 6650000000
Enter the growth rate: 1.14
World population today is 6650000000
In one year, it will grow by 75810000.0
An average daily increase of 207698.630137
>>> main()
This program computes population growth figures.
Enter current world population: 6725810000
Enter the growth rate: 2.2
World population today is 6725810000
In one year, it will grow by 147967820.0
An average daily increase of 405391.287671
>>>
```

Notice how you can re-run the program by just typing the name of the `main()`function. There are other ways of obtaining input in Python. We will see those a little later.

Robot Brains

Writing programs to control your robot is therefore no different from writing a program to perform a computation. They both follow the same basic structure. The only difference is that all robot programs you will write will make use of the Myro library. In fact, there will be robot programs that will require you to obtain input from the user (see exercises below). You can then make use of the `input` function as described above.

One characteristic that will distinguish robot programs from those that just do computations is in the amount of time it will take to run a program. Typically, a program that only performs some computation will terminate as soon as the

computation is completed. However, it will be the case that most of the time your robot program will require the robot to perform some task over and over again. Here then, is an interesting question to ask:

Question How much time would it take for a vacuuming robot to vacuum a 16ft X 12ft room?

Seemingly trivial question but if you think about it a little more, you may reveal some deeper issues. If the room does not have any obstacles in it (i.e. an empty room), the robot may plan to vacuum the room by starting from one corner and then going along the entire length of the long wall, then turning around slightly away from the wall, and traveling to the other end. In this manner, it will ultimately reach the other side of the room in a systematic way and then it could stop when it reaches the last corner. This is similar to the way one would mow a flat oblong lawn, or even harvest a field of crop, or re-ice an ice hockey rink using a Zamboni machine. To answer the question posed above all you have to do is calculate the total distance travelled and the average speed of the vacuum robot and use the two to compute the estimated time it would take. However what if, as is typically the case, the room has furniture and other objects in it?

You might try and modify the approach for vacuuming outlined above but then there would be no guarantee that the floor would be completely vacuumed. You might be tempted to redesign the vacuuming strategy to allow for random movements and then estimate (based on average speed of the robot) that after some generous amount of time, you can be assured that the room would be completely cleaned. It is well known (and we will see this more formally in a later chapter) that random movements over a long period of time do end up providing uniform and almost complete coverage. Inherently this also implies that the same spot may very well end up being vacuumed several times (which is not necessarily a bad thing!). This is similar to the thinking that a herd of sheep, if left grazing on a hill, will result, after a period of time, in a nearly uniform grass height (think of the beautiful hills in Wales).

On the more practical side, iRobot's Roomba robot uses a more advanced strategy (though it is time based) to ensure that it provides complete coverage. A more interesting (and important) question one could ask would be:

Question: How does a vacuuming robot know that it is done cleaning the room?

Most robots are programmed to either detect certain terminating situations or are run based on time. For example, run around for 60 minutes and then stop. Detecting situations is a little difficult and we will return to that in the next chapter.

So far, you have programmed very simple robot behaviors. Each behavior which is defined by a function, when invoked, makes the robot do something for a fixed amount of time. For example, the `yoyo` behavior from the last chapter when invoked as:

```
>>> yoyo(0.5, 1)
```

would cause the robot to do something for about 2 seconds (1 second to go forward and then 1 second to move backward). In general, the time spent carrying out the `yoyo` behavior will depend upon the value of the second parameter supplied to the function. Thus if the invocation was:

```
>>> yoyo(0.5, 5.5)
```

the robot would move for a total of 11 seconds. Similarly, the `dance` behavior defined in the previous chapter will last a total of six seconds. Thus, the total behavior of a robot is directly dependent upon the time it would take to execute all the commands that make up the behavior. Knowing how long a behavior will take can help in pre-programming the total amount of time the overall behavior could last. For example, if you wanted the robot to perform the dance moves for 60 seconds, you can repeat the `dance` behavior ten times. You can do this by simply issuing the `dance` command 10 times. But that gets tedious for us to have to repeat the same commands so many times. Computers are designed to do repetitious tasks. In fact, repetition is one of the

key concepts in computing and all programming languages, including Python, provide simple ways to specify repetitions of all kinds.

Doing Repetition in Python

If you wanted to repeat the dance behavior 10 times, all you have to do is:

```
for i in range(10):
   dance()
```

This is a new statement in Python: the `for`-statement. It is also called a *loop statement* or simply a *loop*. It is a way of repeating something a fixed number of times. The basic syntax of a `for`-loop in Python is:

```
for <variable> in <sequence>:
   <do something>
   <do something>
   ...
```

The loop specification begins with the keyword `for` which is followed by a `<variable>` and then the keyword `in` and a `<sequence>` followed by a colon (`:`). This line sets up the number of times the repetition will be repeated. What follows is a set of statements, indented (again, indentation is important), that are called a *block* that forms the *body* of the loop (stuff that is repeated).

When executed, the `<variable>` (which is called a *loop index variable*) is assigned successive values in the <sequence> and for each of those values, the statements in the body of the loop are executed. A `<sequence>` in Python is a list of values. *Lists* are central to Python and we will see several examples of lists later. For now, look at the dance example above and notice that we have used the function `range(10)` to specify the sequence. To see what this function does you can start IDLE and enter it as an expression:

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

The result of entering the `range(10)` is a sequence (a list) of ten numbers 0..9. Notice how range returns a sequence of values starting from 0 all the way up to, but including, 10. Thus, the variable `i` in the loop:

```
for i in range(10):
   dance()
```

will take on the values 0 through 9 and for each of those values it will execute the `dance()` command.

Do This: Let us try this out on the robot. Modify the robot program from the start of this chapter to include the dance function and then write a main program to use the loop above.

```
# File: dance.py
# Purpose: A simple dance routine

# First import myro and connect to the robot

from myro import *
initialize("com5")

# Define the new functions...

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

def wiggle(speed, waitTime):
       motors(-speed, speed)
       wait(waitTime)
       motors(speed, -speed)
       wait(waitTime)
       stop()

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

```
# The main dance program
def main():
       print "Running the dance routine..."

       for danceStep in range(10):
           dance()

       print "...Done"

main()
```

IDLE Tip

You can stop a program at any time by hitting the CTRL-C keys (pronounced as Control-see). That is, pressing the CTRL-key and then at the same time pressing the c-key.

In the case of a robot program this will also stop the robot.

Notice that we have used `danceStep` (a more meaningful name than `i`) to represent the loop index variable. When you run this program, the robot should perform the dance routine ten times. Modify the value specified in the `range` command to try out some more steps. If you end up specifying a really large value, remember that for each value of `danceStep` the robot will do something for 6 seconds. Thus, if you specified 100 repetitions, the robot will run for 10 minutes.

In addition to repeating by counting, you can also specify repetition using time. For example, if you wanted the robot (or the computer) to do something for 30 seconds. You can write the following command to specify a repetition based on time:

```
while timeRemaining(10):
    <do something>
    <do something>
    ...
```

The above commands will be repeated for 10 seconds. Thus, if you wanted the computer to say “Doh!” for 5 seconds, you can write:

```
while timeRemaining(5):
    speak(“Doh!”, 0)
```

In writing robot programs there will also be times when you just want the robot to keep doing its behaviors forever! While technically by *forever* we do mean eternity in reality what is likely to happen is either it runs out of batteries, or you decide to stop it (by hitting CTRL-C). The Python command to specify this uses a different loop statement, called a `while`-loop that can be written as:

```
while True:
   <do something>
   <do something>
   ...
```

`True` is also a value in Python (along with `False`) about which we will learn more a little later. For now, it would suffice for us to say that the above loop is specifying that the body of the loop be executed forever!

Do This: Modify the `dance.py` program to use each of the `while`-loops instead of the `for`-loop. In the last case (`while TRUE:`) remember to use CTRL-C to stop the repetitions (and the robot).

As we mentioned above, repetition is one of the key concepts in computing. For example, we can use repetition to predict the world population in ten years by repeatedly computing the values for each year:

```
for year in range(10):
    population = population * (1 + growthRate)
```

That is, repeatedly add the increase in population, based on growth rate, ten times.

Do This: Modify the `worldPop.py` program to input the current population, growth rate, and the number of years to project ahead and compute the resulting total population. Run your program on several different values

(Google: "world population growth" to get latest numbers). Can you estimate when the world population will become 9 billion?

Summary

This chapter introduced the basic structure of Python (and robot) programs. We also learned about *names* and *values* in Python. Names can be used to designate functions and values. The latter are also called *variables*. Python provides several different types of values: integers, floating point numbers, strings, and also boolean values (`True` and `False`). Most values have built-in operations (like addition, subtration, etc.) that perform calculations on them. Also, one can form sequences of values using lists. Python provides simple built-in facilities for obtaining input from the user. All of these enable us to write not only robot programs but also programs that perform any kind of computation. Repetition is a central and perhaps the most useful concept in computing. In Python you can specify repetition using either a `for`-loop or a `while`-loop. The latter are useful in writing general robot brain programs. In later chapters, we will learn more about all of these Python features and also learn how to write more sophisticated robot behaviors.

Myro Review

`speak(<something>)`
The computer converts the text in `<something>` to speech and speaks it out. `<something>` is also simultaneously printed on the screen. Speech generation is done synchronously. That is, anything following the speak command is done only after the entire thing is spoken.

`speak(<something>, 0)`
The computer converts the text in `<something>` to speech and speaks it out. `<something>` is also simultaneously printed on the screen. Speech generation is done asynchronously. That is, execution of subsequent commands can be done prior to the text being spoken.

`timeRemaining(<seconds>)`
This is used to specify timed repetitions in a while-loop (see below).

Python Review

Values
Values in Python can be numbers (integers or floating point numbers) or strings. Each type of value can be used in an expression by itself or using a combination of operations defined for that type (for example, +, -, *, /, % for numbers). Strings are considered sequences of characters (or letters).

Names
A name in Python must begin with either an alphabetic letter (a-z or A-Z) or the underscore (i.e. _) and can be followed by any sequence of letters, digits, or underscore letters.

```
input(<prompt string>)
```

This function prints out <prompt string> in the IDLE window and waits for the user to enter a Python expression. The expression is evaluated and its result is returned as a value of the input function.

```
from myro import *
initialize("comX")

<any other imports>
<function definitions>
def main():
    <do something>
    <do something>
    ...

main()
```
This is the basic structure of a robot control program in Python. Without the first two lines, it is the basic structure of all Python programs.

```
print <expression1>, <expression2>, ...
```
Prints out the result of all the expressions on the screen (in the IDLE window). Zero or more expressions can be specified. When no expression is specified, it prints out an empty line.

`<variable name> = <expression>`
This is how Python assigns values to variables. The value generated by <expression> will become the new value of <variable name>.

`range(10)`
Generates a sequence, a list, of numbers from 0..9. There are other, more general, versions of this function. These are shown below.

`range(n1, n2)`
Generates a list of numbers starting from n1…(n2-1). For example, range(5, 10) will generate the list of numbers [5, 6, 7, 8, 9].

`range(n1, n2, step)`
Generates a list of numbers starting from n1…(n2-1) in steps of step. For example, range(5, 10, 2) will generate the list of numbers [5, 7, 9].

Repetition

```
for <variable> in <sequence>:
   <do something>
   <do something>
   ...

while timeRemaining(<seconds>):
    <do something>
    <do something>
    ...

while True:
   <do something>
   <do something>
   ...
```

These are different ways of doing repetition in Python. The first version will assign `<variable>` successive values in `<sequence>` and carry out the body once for each such value. The second version will carry out the body for

<seconds> amount of time. timeRemaining is a Myro function (see above). The last version specifies an un-ending repetition.

Exercises

Exercise 1: Write a Python program to convert a temperature from degrees Celsius to degrees Fahrenheit. Here is a sample interaction with such a program:

```
Enter a temperature in degrees Celsius: 5.0
That is equivalent to 41.0 degrees Fahrenheit.
```

The formula to convert a temperature from Celsius to Fahrenheit is: C/5=(F-32)/9, where C is the temperature in degrees Celsius and F is the temperature in degrees Fahrenheit.

Exercise 2: Write a Python program to convert a temperature from degrees Fahrenheit to degrees Celsius.

Exercise 3: Write a program to convert a given amount of money in US dollars to an equivalent amount in Euros. Look up the current exchange rate on the web (see xe.com, for example).

Exercise 4: Modify the version of the dance program above that uses a for-loop to use the following loop:

```
for danceStep in [1,2,3]:
    dance()
```

That is, you can actually use a list itself to specify the repetition (and successive values the loop variable will take). Try it again with the lists [3, 2, 6], or [3.5, 7.2, 1.45], or [“United”, “States”, “of”, “America”]. Also try replacing the list above with the string “ABC”. Remember, strings are also sequences in Python. We will learn more about lists later.

Exercise 5: Run the world population program (any version from the chapter) and when it prompts for input, try entering the following and observe the behavior of the program. Also, given what you have learned in this chapter, try and explain the resulting behavior.

1. Use the values 9000000000, and 1.42 as input values as above. Except, when it asks for various values, enter them in any order. What happens?

2. Using the same values as above, instead of entering the value, say 9000000000, enter 6000000000+3000000000, or 450000000*2, etc. Do you notice any differences in output?

3. For any of the values to be input, replace them with a string. For instance enter `"Al Gore"` when it prompts you for a number. What happens?

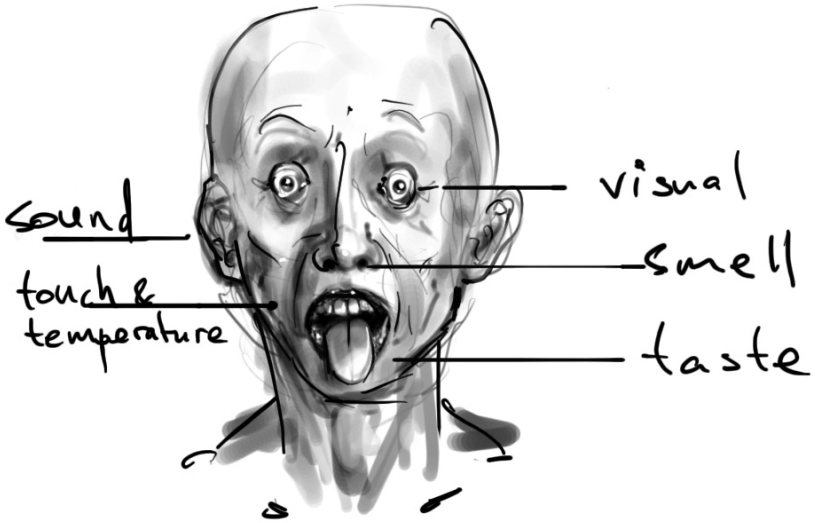
Exercise 6: Rewrite your solution to Exercise 4 from the previous chapter to use the program structure described above.

Exercise 7: You were introduced to the rules of naming in Python. You may have noticed that we have made extensive use of *mixed case* in naming some entities. For example, `waitTime`. There are several naming conventions used by programmers and that has led to an interesting culture in of itself. Look up the phrase *CamelCase controversy* in your favorite search engine to learn about naming conventions. For an interesting article on this, see The Semicolon Wars (http://www.americanscientist.org/template/AssetDetail/assetid/51982).

Exercise 8: Experiment with the `speak` function introduced in this chapter. Try giving it a number to speak (try both integers and floating point numbers). What is the largest integer value that it can speak? What happens when this limit is exceeded? Try to give the `speak` function a list of numbers, or strings, or both.

Exercise 9: Write a Python program that sings the ABC song: *ABCD…XYZ. Now I know my ABC's. Next time won't you sing with me?*

4 Sensing From Within



I see dead people.
-: Cole Sear (played by Haley Joel Osment) in Sixth Sense, M. Night Shyamalan, 1999.

I see all obstacles in my way. -: In the song I can see clearly now, Johnny Nash, 1972.

Cole Sear in Shyamalan's *Sixth Sense* is not referring to dead bodies lying in front of him (for those who have not seen the movie). The five senses that most humans relate to are: touch, vision, balance, hearing, and taste or smell. In all cases our bodies have special sensory receptors that are placed on various parts of the body to enable sensing. For example the taste receptors are concentrated mostly on the tongue; the touch receptors are most sensitive on hands and the face and least on the back and on limbs although they are present all over the body, etc. Besides the difference in the physiology of each kind of receptors there are also different neuronal pathways and thereby sensing mechanisms built into our bodies. Functionally, we can say that each type of sensory system starts with the receptors which convert the thing they sense into electrochemical signals that are transmitted over neurons. Many of these pathways lead to the cerebral cortex in the brain where they are further *processed* (like, "Whoa, that jalapeno is hot!!"). The perceptual system of an organism refers to the set of sensory receptors, the neuronal pathways, and the processing of perceptual information in the brain. The brain is capable of combining sensory information from different receptors to create richer experiences than those facilitated by the individual receptors.

The perceptual system of any organism includes a set of *external* sensors (also called exteroceptors) and some *internal* sensing mechanisms (*interoceptors* or *proprioception*). Can you touch your belly button in the dark? This is because of proprioception. Your body's sensory system also keeps track of the internal state of your body parts, how they are oriented, etc.

Proprioception
Sensing from within

Get something really delicious to eat, like a cookie, or a piece of chocolate, or candy (whatever you fancy!). Hold it in your right hand, and let your right arm hang naturally on your side. Now close your eyes, real tight, and try to eat the thing you are holding. Piece of cake! (well, whatever you picked to eat :-)

Without fail, you were able to pick it up and bring it to your mouth, right?

Give yourself a *Snickers moment* and enjoy the treat.

Sensing is an essential component of being a robot and every robot comes built with internal as well as external sensors. It is not uncommon, for example, to find sensors that are capable of sensing light, temperature, touch, distance to another object, etc. An example of internal sensing in robots is the measurement of movement relative to the robot's internal frame of reference. Sometimes also called *dead reckoning*, it can be a useful sensing mechanism that you can use to design robot behaviors.

Robots employ electromechanical sensors and there are different types of devices available for sensing the same physical quantity. For example, one common sensor found on many robots is a *proximity* sensor. It detects the distance to an object or an obstacle. Proximity sensors can be made using different technologies: infrared light, sonar, or even laser. Depending upon the type of technology used, their accuracy, performance, as well as cost vary: infrared (IR) is the cheapest, and laser is on the expensive side. Lets us take a look at the perceptual system of your Scribbler robot starting with internal sensors.

Proprioception in the Scribbler

The Scribbler has three useful internal sensory mechanisms: *stall*, *time*, and *battery level*. When your program asks the robot to move it doesn't always imply that the robot is actually physically moving. It could be stuck against a wall, for example. The stall sensor in the Scribbler enables you to detect this. You have already seen how you can use time to control behaviors using the `timeRemaining` and `wait` functions. Also, for most movement commands, you can specify how long you want that movement to take place (for example `forward(1, 2.5)` means full-speed forward for 2.5 seconds). Finally, it is also possible to detect battery power level so that you can detect when it is time to change batteries in the robot.

Time

All computers come built-in with an internal clock. In fact, clocks are so essential to the computers we use today that without them we would not have

computers at all! Your Scribbler robot can use the computer's clock to sense time. It is with the help of this clock that we are able to use time in functions like `timeRemaining`, `wait`, and other movement commands. Just with these facilities it is possible to define interesting automated behaviors.

Do This: Design a robot program for the Scribbler to draw a square (say with sides of 6 inches). To accomplish this, you will have to experiment with the movements of the robot and correlate them with time. The two movements you have to pay attention to are the rate at which the robot moves, when it moves in a straight line; and the degree of turn with respect to time. You can write a function for each of these:

```
def travelStraight(distance):
    # Travel in a straight line for distance inches
    ...

def degreeTurn(angle):
    # Spin a total of angle degrees
```

That is, figure out by experimentation on your own robot (the results will vary from robot to robot) as to what the correlation is between the distance and the time for a given type of movement above and then use that to define the two functions above. For example, if a robot (hypothetical case) seems to travel at the rate of 25 inches/minute when you issue the command `tranalate(1.0)`, then to travel 6 inches you will have to translate for a total of (6*60)/25 seconds. Try moving your robot forward for varying amounts for time at the same fixed speed. For example try moving the robot forward at speed 0.5 for 3, 4, 5, 6 seconds. Record the distance travelled by the robot for each of those times. You will notice a lot of variation in the distance even for the same set of commands. You may want to average those. Given this data, you can estimate the average amount of time it takes to travel an inch. You can then define `travelStraight` as follows:

```
def travelStraight(distance):
    # set up your robot's speed
    inchesPerSec = <Insert your robot's value here>
```

```
    # Travel in a straight line for distance inches
    forward(1, distance/inchesPerSec)
```

Similarly you can also determine the time required for turning a given number of degrees. Try turning the robot at the same speed for varying amounts of time. Experiment how long it takes the robot to turn 360 degrees, 720 degrees, etc. Again, average the data you collect to get the number of degrees per second. Once you have figured out the details use them to write the `degreeTurn` function. Then use the following main program:

```
def main():
   # Transcribe a square of sides = 6 inches

   for side in range(4):
       travelStraight(6.0)
       degreeTurn(90.0)

   print "Done."

main()
```

Run this program several times. It is unlikely that you will get a perfect square each time. This has to do with the calculations you performed as well with the variation in the robot's motors. They are not precise. Also, it generally takes more power to move from a still stop than to keep moving. Since you have no way of controlling this, at best you can only approximate this type of behavior. Over time, you will also notice that the error will aggregate. This will become evident in doing the exercise below.

Do This: Building on the ideas from the previous exercise, we could further abstract the robot's drawing behavior so that we can ask it to draw any regular polygon (given the number of sides and length of each side). Write the function:

```
def drawPolygon(SIDES, LENGTH):
    # Draw a regular polygon with SIDES number of sides
    # and each side of length LENGTH.
```

Then, we can write a regular polygon drawing robot program as follows:

```
def main():
    # Given the number of sides and the length of each side,
    # draw a regular polygon

    # First, ask the user for the number of sides and
    # side length
    print “Given # of sides and side length I will draw”
    print “a polygon for you. Specify side length in inches.”

    nSides = input("Enter # of sides in the polygon: ")
    sideLength = input("Enter the length of each side: ")

    # Draw the polygon
    drawPolygon(nSides, sidelength)

    print "Done."

main()
```

To test the program, first try drawing a square of sides 6 inches as in the previous exercise. Then try a triangle, a pentagon, hexagon, etc. Try a polygon with 30 sides of length 0.5 inches. What happens when you give 1 as the number of sides? What happens when you give zero (0) as the number of sides?

A Slight Detour: Random Walks

One way you can do interesting things with robot drawings is to inject some randomness in how long the robot does something. Python, and most programming languages, typically provide a library for generating random numbers. Generating random numbers is an interesting process in itself but we will save that discussion for a later time. Random numbers are very useful in all kinds of computer applications, especially games and in simulating real life phenomena. For example, in estimating how many cars might be entering an already crowded highway in the peak of rush hour? etc. In order to access the

random number generating functions in Python you have to import the `random` library:

```
from random import *
```

There are lots of features available in this library but we will restrict ourselves with just two functions for now: `random` and `randrange`. These are described below:

`random()` Returns a random number between 0.0 and 1.0.

`randrange(A, B)` Returns a random number in the range `[A…B-1]`.

Here is a sample interaction with these two functions:

```
>>> from random import *
>>> randrange(1, 10)
6
>>> randrange(1, 10)
5
>>> randrange(1, 10)
6
>>> randrange(1, 10)
4
>>> randrange(1, 10)
2
>>> randrange(1, 10)
7
>>> random()
0.00062680831179329211
>>> random()
0.2206784218206268
>>> random()
0.81165319999331298
>>> random()
0.1166218349355429
>>>
```

As you can see, using the random number library is easy enough, similar to using the Myro library for robot commands. Given the two functions, it is entirely up to you how you use them. Look at the program below:

```
def main():
    # generate 10 random polygons
    for poly in range(10):
        # generate a random polygon and draw it
        Print “Place a new color in the pen port and then…”
        userInput = input("Enter any number: ")
        sides = randrange(3, 8)
        size = randrange(2, 6)
        drawPolygon(sides, size)

    # generate a random walk of 20 steps
    for step in range(20):
        travelStraight(random())
        degreeTurn(randrange(0, 360))
```

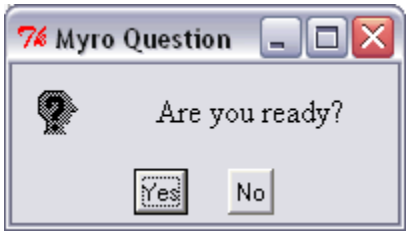
The first loop in the program draws 10 random polygons of sizes ranging from 3 to 7 sides and each side in the range 2 to 5 inches. The second loop carries out a random walk of 20 steps.

Asking Questions?

As you can see from above, it is easy to program various kinds of movements into the Scribbler. If there is a pen in the pen port, the Scribbler draws a path. As you can see in the example above, we can stop the program temporarily, pretend that we are taking some input and use that as an opportunity to change the pen and then go on. Above, we used the Python `input` command to accomplish this. There is a better way to do this and it uses a function provided in the Myro library:

```
>>> askQuestion("Are you ready?")
```

When this function is executed, a dialog window pops up as shown below:



When you press your mouse on any of the choices (Yes/No), the window disappears and the function returns the name of the key selected by the user as a string. That is, if in the above window you pressed the `Yes` key, the function will return the value:

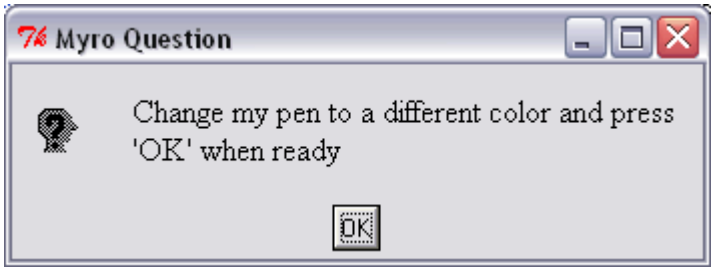
```
>>> askQuestion("Are you ready?")
'Yes'
```

The `askQuestion` command can be used in the program above as follows:

```
askQuestion("Change my pen to a different color and press
'Yes' when ready.")
```

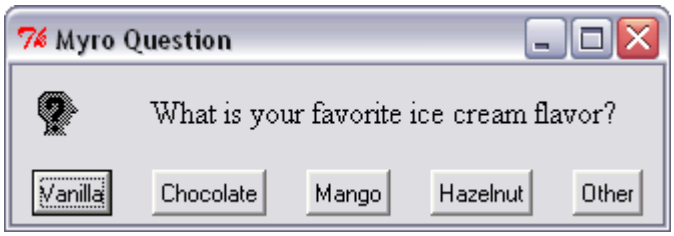
While this is definitely more functional than our previous solution, we can actually do better. For example, what happens when the user presses the `No` button in the above interaction? One thing you know for sure is that the function will return the string `'No'`. However, the way we are using this function, it really does not matter which key the user presses. `askQuestion` is designed so it can be customized by you so that you can specify how many button choices you want to have in the dialog window as well as what the names of those buttons would be. Here is an illustration of how you would write a better version of the above command:

```
askQuestion("Change my pen to a different color and press 'OK'
when ready", ["OK"])
```



Now this is certainly better. Notice that the function `askQuestion` can be used with either one parameter or two. If only one parameter is specified, then the default behavior of the function is to offer two button choices: 'Yes' and 'No'. However, using the second parameter you can specify, in a list, any number of strings that will become the choice buttons. For example,

```
askQuestion("What is your favorite ice cream flavor?",
["Vanilla", "Chocolate", "Mango", "Hazelnut", "Other"])
```



This will be a very handy function to use in many different situations. In the next exercise, try and use this function to become familiar with it.

Do This: Write a Scribbler program of your own that exploits the Scribbler's movements to make random drawings. Make sure you generate drawings with at least three or more colors. Because of random movements, your robot is likely to run into things and get stuck. Help your robot out by picking it up and placing it elsewhere when this happens.

Back to time…

Most programming languages also allow you to access the internal clock to keep track of time, or time elapsed (as in a stop watch), or in any other way you may want to make use of time (as in the case of the `wait`) function. The Myro library provides a simple function that can be used to retrieve the current time:

```
>>> currentTime()
1169237231.836
```

The value returned by `currentTime` is a number that represents the seconds elapsed since some earlier time, whatever that is. Try issuing the command several times and you will notice that the difference in the values returned by the function represents the real time in seconds. For example:

```
>>> currentTime()
1169237351.5580001
>>> 1169237351.5580001 - 1169237231.836
119.72200012207031
```

That is, 119.722 seconds had elapsed between the two commands above. This provides another way for us to write robot behaviors. So far, we have learned that if you wanted your robot to go forward for 3 seconds, you could either do:

```
forward(1.0, 3.0)
```

or

```
forward(1.0, 3.0)
wait(3.0)
```

or

```
while timeRemaining(3.0):
    forward(1.0)
```

Using the `currentTime` function, there is yet another way to do the same thing:

```
startTime = currentTime()     # record start time
while (currentTime() - startTime) < 3.0:
   forward(1.0)
```

The above solution uses the internal clock. First, it records the start time. Next it enters the loop which first gets the current time and then checks to see if the difference between the current time and start time is less than 3.0 seconds. If so, the `forward` command is repeated. As soon as the elapsed time gets over 3.0 seconds, the loop terminates. This is another way of using the `while`-loop that you learned in the previous chapter. In the last chapter, you learned that you could write a loop that executed forever as shown below:

```
while True:
   <do something>
```

The more general form of the `while`-loop is:

```
while <some condition is true>:
   <do something>
```

That is, you can specify any condition in `<some condition is true>`. The condition is *tested* and if it results in a `True` value, the step(s) specified in `<do something>` is/are performed. The condition is tested again, and so on. In the example above, we use the expression:

```
(currentTime() - startTime) < 3.0
```

If this condition is true, it implies that the elapsed time since the start is less than 3.0 seconds. If it is false, it implies that more than 3.0 seconds have elapsed and it results in a `False` value, and the loop stops. Learning about writing such conditions is essential to writing smarter robot programs.

While it may appear that the solution that specified time in the `forward` command itself seemed simple enough (and it is!), you will soon discover that

being able to use the internal clock as shown above provides more versatility and functionality in designing robot behaviors. This, for example is how one could program a vacuum cleaning robot to clean a room for 60 minutes:

```
startTime = currentTime()
while (currentTime() - startTime)/60.0 < 60.0:
  cleanRoom()
```

You have now seen how to write robot programs that have behaviors or commands that can be repeated a fixed number of times, or forever, or for a certain duration:

```
# do something N times
for step in range(N):
  do something...

# do something forever
while True:
  do something...

# so something for some duration
while timeRemaining(duration):
  do something...

# so something for some duration
duration = <some time in seconds>
startTime = currentTime()
while (currentTime() - startTime) < duration:
  do something...
```

All of the above are useful in different situations. Sometimes it becomes a matter of personal preference.

Writing Conditions

Let us spend some time here to learn about conditions you can write in `while`-loops. The first thing to realize that all conditions result in either of two values: `True` or `False` (or, alternately a 1 or a 0). These are Python values, just like numbers. And so you can use them in many ways. Simple conditions can be written using comparison (or relational) operations: < (less than), <= (less than or equal to), > (greater than), >= (greater than or equal to), == (equal to), and != (not equal to). These operations can be used to compare all kinds of values. Here are some examples:

```
>>> 42 > 23
True
>>> 42 < 23
False
>>> 42 == 23
False
>>> 42 != 23
True
>>> (42 + 23) < 100
True
>>> a, b, c = 10, 20, 10
>>> a == b
False
>>> a == c
True
>>> a == a
True
>>> True == 1
True
>>> False == 1
False
```

The last two examples above also show how the values `True` and `False` are related to 1 and 0. `True` is the same as 1 and 0 is the same as `False`. You can form many useful conditions using the comparison operations and all

conditions result in either `True` (or 1) or `False` (or 0). You can also compare other values, like strings, using these operations:

```
>>> "Hello" == "Good Bye"
False
>>> "Hello" != "Good Bye"
True
>>> "Elmore" < "Elvis"
True
>>> "New York" < "Paris"
True
>>> "A" < "B"
True
>>> "a" < "A"
False
```

Unicode

Text characters have an internal computer coding or representation that enforces lexicographic ordering. This internal encoding is very important in the design of computers and this is what enables all computers and devices like iPhones etc. to exchange information consistently. All language characters in the world have been assigned a standard computer encoding. This is called *Unicode*.

Study the above examples carefully. Two important things to notice are: strings are compared using alphabetical ordering (i.e. lexicographically). Thus `"Elmore"` is less than `"Elvis"` since `"m"` is less than `"v"` in those strings (`"El"` being equal in both). That is also why `"New York"` is less than `"Paris"` (since `"N"` is less than `"P"`). The second important thing to note is that uppercase letters are less than their equivalent lowercase counterparts (`"A"` is less than `"a"`). This is by design (see box on right).

Besides relational operations you can build more complex conditional expressions using the *logical operations* (also called *Boolean operations*): `and`, `or`, and `not`. Here are some examples:

```
>>> (5 < 7) and (8 > 3)
True
>>> not ( (5 < 7) and (8 > 3))
False
>>> (6 > 7) or (3 > 4)
```

```
False
>>> (6 > 7) or (3 > 2)
True
```

We can define the meaning of logical operators as follows:

- **<expression-1> and <expression-2>:** Such an expression will result in a value `True` only if both `<expression-1>` and `<expression-2>` are `True`. In all other cases (i.e. if either one or both of `<expression-1>` and `<expression-2>` are `False`) it results in a `False`.

- **<expression-1> or <expression-2>**: Such an expression will result in a value `True` if either `<expression-1>` or `<expression-2>` are `True` or if both are `True`. In all other cases (i.e. if both of `<expression-1>` and `<expression-2>` are `False`) it results in a `False`.

- **not <expression>**: Such an expression will result in a value `True` if `<expression>` is `False` or `False` if `<expression>` is `True`). I.e., it flips or complements the value of expression.

These operators can be combined with relational expressions to form arbitrarily complex conditional expressions. In fact, any decision making in your programs boils down to forming the appropriate conditional expressions. The logical operators were invented by the logician George Boole in the mid 19th century. Boolean algebra, named after Boole, defines some simple, yet important laws that govern the behavior of logical operators. Here are some useful ones:

- `(A or True)` is always True.
- `(not (not A))` is just `A`
- `(A or (B and C))` is the same as `((A or B) and (A or C))`
- `(A and (B or C))` is the same as `((A and B) or (A and C))`
- `(not (A or B))` is the same as `((not A) and (not B))`

- `(not (A and B))` is the same as `((not A) or (not B))`

These identities or properties can help you in simplifying conditional expressions**.** The conditional expressions can be used to write several useful conditions to control the execution of some program statements. These allow you to write conditional repetitions as:

```
while <some condition is true>:
   <do something>
```

Now you can see why the following is a way of saying, “do something forever”:

```
while True:
   <do something>
```

Since the condition is always `“True”` the statements will be repeated forever. Similarly, in the loop below:

```
while timeRemaining(duration):
   <do something>
```

As soon as the `duration` is up, the value of the `timeRemaining(duration)` expression will become `False` and the repetition will stop. Controlling the repetitions based on conditions is a powerful idea in computing. We will be using these extensively to control the behaviors of robots.

Sensing Stall

We mentioned in the beginning of this chapter that the Scribbler also has a way of sensing that it is stalled when trying to move. This is done by using the Myro function `getStall`:

`getStall()` Returns `True` if the robot has stalled, `False` otherwise.

You can use this to detect that the robot has stalled and even use it as a condition in a loop to control behavior. For example:

```
while not getStall():
   <do something>
```

That is, keep doing `<do something>` until the robot has stalled. Thus, you could write a robot behavior that goes forward until it bumps into something, say a wall, and then stops.

```
while not getStall():
    forward(1.0)
stop()
speak("Ouch! I think I bumped into something!")
```

In the above example, as long as the robot is not stalled, `getStall()` will return `False` and hence the robot will keep going forward (since `not False` is `True`). Once it does bump into something, `getStall()` will return **`True`** and then the robot will stop and speak.

Do This: Write a complete program for the Scribbler to implement the above behavior and then observe the behavior. Show this to some friends who are not in your course. Ask them for their reactions. You will notice that people will tend to ascribe some form of *intelligence* to the robot. That is, your robot is sensing that it is stuck, and when it is, it stops trying to move and even announces that it is stuck by speaking. We will return to this idea of *artificial intelligence* in a later chapter.

Sensing Battery Power Levels

Your Scribbler robot runs on 6 AA batteries. As with any other electronic device, with use, the batteries will ultimately drain and you will need to replace with fresh ones. Myro provides an internal battery-level sensing

function, called `getBattery` that returns the current voltage being supplied by the battery. When the battery levels go down, you will get lower and lower voltages causing erratic behavior. The battery voltage levels of your Scribbler will vary between 0 and 9 volts (0 being totally drained). What low means is something you will have to experiment and find out. The best way to do this is to record the battery level when you insert a fresh set of batteries. Then, over time, keep recording the battery levels as you go.

Disposing Batteries

Make sure that you dispose used batteries properly and responsibly. Batteries may contain hazardous materials like cadmium, mercury, lead, lithium, etc. which can be deadly pollutants if disposed in landfills. Find out your nearest battery recycling or disposal option to ensure proper disposal.

The Scribbler also has some built-in battery-level indicator lights. The red LED on the robot remains lit when the power levels are high (or in the good range). It starts to flash when the battery level runs low. There is also an similar LED on the Fluke dongle. Can you find it? Just wait until the battery levels run low and you will see it flashing.

You can use battery-level sensing to define behaviors for robots so that they are carried out only when there is sufficient power available. For example:

```
while (getBattery() >= 5.0) and timeRemaining(duration):
   <do something>
```

That is, as long as battery power is above 5.0 and the time limit has not exceeded `duration`, `<do something>`.

World Population, revisited

The ability to write conditional expressions also enables us to define more sophisticated computations. Recall the world population projection example

from previous chapter. Given the population growth rate and the current population, you can now write a program to predict the year when the world's population will increase to beyond a given number, say 9 billion. All you have to do is write a condition-driven repetition that has the following structure:

```
year = <current year>
population = <current population>
growthRate = <rate of growth>

# repeat as long as the population stays below 9000000000
while population < 9000000000:
    # compute the population for the next year
    year = year + 1
    population = population * (1+growthRate)

print "By the year", year, "the world's population"
print "will have exceeded 9 billion."
```

That is, add population growth in the next year if the population is below 9 billion. Keep repeating this until it exceeds 9 billion.

Do This: Complete the program above and compute the year when the world's population will exceed 9 billion. To make your program more useful make sure you ask the user to input the values of the year, population, growth rate, etc. In fact, you can even ask the user to enter the population limit so you make use the program for any kinds of predictions (8 billion? 10 billion?). How would you change the program so it prints the population projection for a given year, say 2100?

Summary

In this chapter you have learned about proprioception or internal sensory mechanisms. The Scribbler robot has three internal sensory mechanisms: time, stall, and battery-level. You have learned how to sense these quantities

and also how to use them in defining automated robot behaviors. You also learned about random number generation and used it to define unpredictable robot behaviors. Later, we will also learn how to use random numbers to write games and to simulate natural phenomena. Sensing can also be used to define conditional repetitive behaviors using conditional expressions in `while`-loops. You learned how to construct and write different kinds of conditions using *relational* and *logical* operations. These will also become valuable in defining behaviors that use external sensory mechanisms and also enable us to write more explicit decision-making behaviors. We will learn about these in the next chapter.

Myro Review

`randomNumber()`
Returns a random number in the range 0.0 and 1.0. This is an alternative Myro function that works just like the `random` function from the Python `random` library (see below).

`askQuestion(MESSAGE-STRING)`
A dialog window with `MESSAGE-STRING` is displayed with choices: `'Yes'` and `'No'`. Returns `'Yes'` or `'No'` depending on what the user selects.

`askQuestion(MESSAGE-STRING, LIST-OF-OPTIONS)`
A dialog window with `MESSAGE-STRING` is displayed with choices indicated in `LIST-OF-OPTIONS`. Returns option string depending on what the user selects.

`currentTime()`
The current time, in seconds from an arbitrary starting point in time, many years ago.

`getStall()`
Returns `True` if the robot is stalled when trying to move, `False` otherwise.

`getBattery()`
Returns the current battery power level (in volts). It can be a number between 0 and 9 with 0 indication no power and 9 being the highest. There are also LED power indicators present on the robot. The robot behavior becomes erratic when batteries run low. It is then time to replace all batteries.

Python Review

`True, False`
These are Boolean or logical values in Python. Python also defines `True` as 1 and `False` as 0 and they can be used interchangeably.

`<, <=, >, >=, ==, !=`
These are relational operations in Python. They can be used to compare values. See text for details on these operations.

`and, or not`
These are logical operations. They can be used to combine any expression that yields Boolean values.

`random()`
Returns a random number between 0.0 and 1.0. This function is a part of the `random` library in Python.

`randRange(A, B)`
Returns a random number in the range A (inclusive) and B (exclusive). This function is a part of the `random` library in Python.

Exercises

Exercise 1: Write a robot program to make your Scribbler draw a five point star. [Hint: Each vertex in the star has an interior angle of 36 degrees.]

Exercise 2: Experiment with Scribbler movement commands and learn how to make it transcribe a path of any given radius. Write a program to draw a circle of any input diameter.

Exercise 3: Write a program to draw other shapes: the outline of a house, a stadium, or create art by inserting pens of different colors. Write the program so that the robot stops and asks you for a new color.

Exercise 4: If you had an open rectangular lawn (with no trees or obstructions in it) you could use a Zanboni like strategy to mow the lawn. Start at one end of the lawn, mow the entire length of it along the longest side, turn around and mow the entire length again, next to the previously mowed area, etc. until you are done. Write a program for your Scribbler to implement this strategy (make the Scribbler draw its path as it goes).

Exercise 5: Enhance the random drawing program from this chapter to make use of speech. Make the robot, as it is carrying out random movements, to speak out what it is doing. As a result you will have a robot artist that you have created!

Exercise 6: Rewrite your program from the previous exercise so that the random behavior using each different pen is carried out for 30 seconds.

Exercise 7: The Myro library also provides a function called, `randomNumber()` that returns a random number in the range 0.0 and 1.0. This is similar to the function `random()` from the Python library `random` that was introduced in this chapter. You can use either based on your own preference. You will have to import the appropriate library depending on the function you choose to use. Experiment with both to convince yourself that these two are equivalent.

Exercise 8: In reality, you only need the function `random()` to generate random numbers in any range. For example, you can get a random number between 1 and 6 with `randRange(1,6)` or as shown below:

```
randomValue = 1 + int(random()*6)
```

The function `int()` takes any number as its parameter, truncates it to a whole number and returns an *integer*. Given that `random()` returns values between 0.0 (inclusive) and 1.0 (exclusive), the above expression will assign a random value between 1..5 (inclusive) to `randomValue`. Given this example, write a new function called `myRandRange()` that works just like `randrange()`:

```
def myRandRange(A, B):
   # generate a random number between A..B
   # (just like as defined for randrange)
```

Exercise 9: What kinds of things can your robot talk about? You have already seen how to make the robot/computer speak a given sentence or phrase. But the robot can also "talk" about other things, like the time or the weather.

One way to get the current time and date is to import another Python library called `time`:

```
>>> from time import *
```

The time module provides a function called `localtime` that works as follows:

```
>>> localtime()
(2007, 5, 29, 12, 15, 49, 1, 149, 1)
```

`localtime` returns all of the following in order:

1. year
2. month
3. day
4. hour
5. minute
6. seconds
7. weekday
8. day of the year

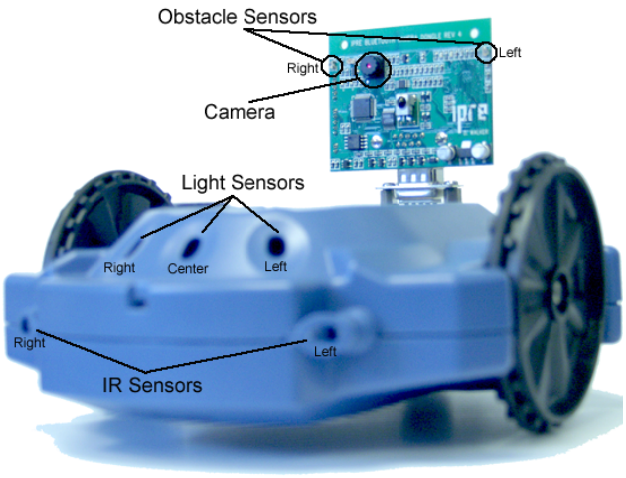
9. whether it is using daylight savings time, or not

In the example above, it is May 29, 2007 at 12:15pm and 49 seconds. It is also the 1st day of the week, 149 day of the year, and we are using daylight savings time. You can assign each of the values to named variables as shown below:

```
year, month, day,…, dayOfWeek = localtime()
```

Then, for the example above, the variable year will have the value 2007, moth will have the value 5, etc. Write a Python program that speaks out the current date and time.

5 Sensing the World



So you gotta let me know
Should I stay or should I go?
-: From the song, **Should I stay or should I go**, Mick Jones (The Clash), 1982.

In the previous chapter you learned how proprioception: sensing time, stall, and battery-level can be used in writing simple yet interesting robot behaviors. All robots also come equipped with a suite of external sensors (or exteroceptors) that can sense various things in the environment. Sensing is makes the robot *aware* of its environment and can be used to define more intelligent behaviors. Sensing is also related to another important concept in computing: *input*. Computers act on different kinds of information: numbers, text, sounds, images, etc. to produce useful applications. Acquiring information to be processed is generally referred to as input. In this chapter we will also see how other forms of input can be acquired for a program to process. First, let us focus on Scribbler's sensors.

Scribbler Sensors

The Scribbler robot can sense the amount of ambient light, the presence (or absence) of obstacles around it, and also take pictures from its camera. Several devices (or sensors) are located on the Scribbler (see picture on the previous page). Here is a short description of these:

Camera: The camera can take a still picture of whatever the robot is currently "seeing".

Light: There are three light sensors present on the robot. These are located in the three holes present on the front of the robot. These sensors can detect the levels of brightness (or darkness). These can be used to detect variations in ambience light in a room. Also, using the information acquired from the camera, the Scribbler makes available an alternative set of brightness sensors (left, center, and right).

Proximity: There are two sets of these on the Scribbler: IR Sensors (left and right) on the front; and Obstacle Sensors (left, center, and right) on the Fluke dongle. They can be used to detect objects on the front and on its sides.

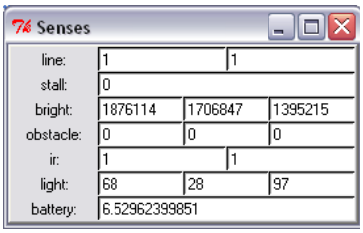
Getting to know the sensors

Sensing using the sensors provided in the Scribbler is easy. Once you are familiar with the details of sensor behaviors you will be able to use them in your programs to design interesting creature-like behaviors for your Scribbler. But first, we must spend some time getting to know these sensors; how to access the information reported by them; and what this information looks like. As for the internal sensors, Myro provides several functions that can be used to acquire data from each sensor device. Where multiple sensors are available, you also have the option of obtaining data from all the sensors or selectively from an individual sensor.

Do This: perhaps the best way to get a quick look at the overall behavior of all the sensors is to use the Myro function senses:

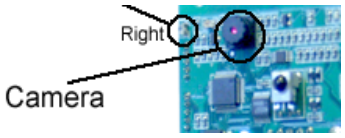
```
>>> senses()
```

Scribbler Sensors



This results in a window (see picture on right) showing all of the sensor values (except the camera) in real time. They are updated every second. You should move the robot around and see how the sensor values change. The window also displays the values of the stall sensor as well as the battery level. The leftmost value in each of the sensor sets (light, IR, obstacle, and bright) is the value of the left sensor, followed by center (if present), and then the right.

The Camera



The camera can be used to take pictures of the robot's current view. As you can see, the camera is located on the Fluke dongle. The view of the image taken by the camera

will depend on the orientation of the robot (and the dongle). To take pictures from the camera you can use the `takePicture` command:

```
takePicture()
takePicture("color")
TakePicture("gray")
```

Takes a picture and returns a picture object. By default, when no parameters are specified, the picture is in color. Using the "gray" option, you can get a grayscale picture. Example:

```
>>> p = takePicture()
>>> show(p)
```

Alternately, you can also do:

```
>>> show(takePicture())
```

Once you take a picture from the camera, you can do many things with it. For example, you may want to see if there is a laptop computer present in the picture. Image processing is a vast subfield of computer science and has applications in many areas. Understanding an image is quite complex but something we do quite naturally. For example, in the picture above, we have no problem locating the laptop, the bookcase in the background, and even a case for a badminton racket (or, if you prefer racquet). The camera on the Scribbler is its most complex sensory device that will require a lot of computational effort and energy to use it in designing behaviors. In the simplest case, the camera can serve as your remote "eyes" on the robot. We may not have mentioned this earlier, but the range of the Bluetooth wireless on the robot is 100 meters. In later chapters we will learn several ways of using the pictures. For now, if you take a picture from the camera and would like to save it for later use, you use the Myro command, `savePicture`, as in:

```
>>> savePicture(p, “office-scene.jpg”)
```

The file `office-scene.jpg` will be saved in the same folder as your `Start Python` folder. You can also use `savePicture` to save a series of pictures from the camera and turn it into an animated “movie” (an animated gif image). This is illustrated in the example below.

Do This: First try out all the commands for taking and saving pictures. Make sure that you are comfortable using them. Try taking some grayscale pictures as well. Suppose your robot has ventured into a place where you cannot see it but it is still in communication range with your computer. You would like to be able to look around to see where it is. Perhaps it is in a new room. You can ask the robot to turn around and take several pictures and show you around. You can do this using a combination of `rotate` and `takePicture` commands as shown below:

```
while timeRemaining(30):
    show(takePicture())
    turnLeft(0.5, 0.2)
```

That is, take a picture and then turn for 0.2 seconds, repeating the two steps for 30 seconds. If you watch the picture window that pops up, you will see successive pictures of the robot’s views. Try this a few times and see if you can count how many different images you are able to see. Next, change the `takePicture` command to take grayscale images. Can you count how many images it took this time? There is of course an easier way to do this:

```
N = 0
while timeRemaining(30):
    show(takePicture())
    turnLeft(0.5, 0.2)
    N = N + 1
print N
```

Now it will give you the number of images it takes. You will notice that it is able to take many more grayscale images than color ones. This is because color images have a lot more information in them than grayscale images (see text on right). A 256x192 color image requires 256x192x3 (= 147, 456) bytes of data where as a grayscale image requires only 256x192 (= 49,152) bytes. The more data you have to transfer from the robot to the computer, the longer it takes.

You can also save an animated GIF of the images generated by the robot by using the `savePicture` command by accumulating a series of images in a list. This is shown below:

```
Pics = []
while timeRemaining(30):
    pic = takePicture()
    show(pic)
    Pics = Pics.append(pic)
    turnLeft(0.5, 0.2)
savePicture(Pics, “office-movie.gif”)
```

Pixels

Each image is made up of several tiny picture elements or *pixels*. In a color image, each pixel contains color information which is made up of the amount of red, green, and blue (RGB). Each of these values is in the range 0..255 and hence it takes 3 bytes or 24-bits to store the information contained in a single pixel. A pixel that is colored pure red will have the RGB values (255, 0, 0).

A grayscale image, on the other hand only contains the level of gray in a pixel which can be represented in a single byte (or 8-bits) as a number ranging from 0..255 (where 0 is black and 255 is white).

First we create an empty list called, `Pics`. Then we append each successive picture taken by the camera into the list. Once all the images are accumulated, we use `savePicture` to store the entire set as an animated GIF. You will be able to view the animated GIF inside any web browser. Just load the file into a web browser and it will play all the images as a movie.

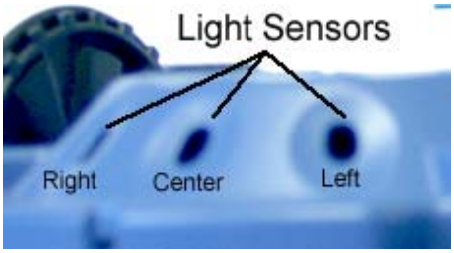
There are many more interesting ways that one can use images from the camera. In Chapter 7 we will explore further. For now, let us take a look at Scribbler's other sensors.

Light Sensing

The following functions are available to obtain values of light sensors:

`getLight()` Returns a list containing the three values of all light sensors.
`getLight(<POSITION>)` Returns the current value in the `<POSITION>` light sensor. `<POSITION>` can either be one of 'left', 'center', 'right' or one of the numbers 0, 1, 2. The positions 0, 1, and 2 correspond to the left, center, and right sensors. Examples:

```
>>> getLight()
[135, 3716, 75]
>>> getLight('left')
135
>>> getLight(0)
135
>>> getLight('center')
3716
>>> getLight(1)
3716
>>> getLight('right')
75
>>> getLight(2)
75
```



The values being reported by these sensors can be in the range [0..5000] where low values imply bright light and high values imply darkness. The above values were taken in ambient light with one finger completely covering the center sensor. Thus, the darker it is, the higher the value reported. In a way, you could even call it a darkness sensor. Later, we will see how we can easily transform these values in many different ways to affect robot behaviors.

It would be a good idea to use the `senses` function to play around with the light sensors and observe their values. Try to move the robot around to see how the values change. Turn off the lights in the room, or cover the sensors with your fingers, etc.

When you use the `getLight` function without any parameters, you get a list of three sensor values (left, center, and right). You can use assign these to individual variables in many ways:

```
>>> L, C, R = getLight()
>>> print L
135
>>> Center  = getLight(“center”)
>>> print center
3716
```

The variables can then be used in many ways to define robot behaviors. We will see several examples of these in the next chapter.

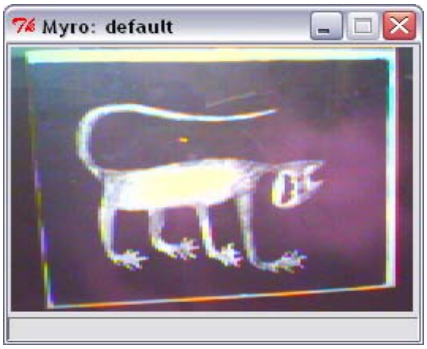
The camera present on the Fluke dongle can also be used as a kind of brightness sensor. This is done by averaging the brightness values in different zones of the camera image. In a way, you can think of it as a virtual sensor. That is, it doesn’t physically exist but is embedded in the functionality of the camera. The function `getBright` is similar to `getLight` in how it can be used to obtain brightness values:

`getbright()`Returns a list containing the three values of all light sesnors.
`getBright(<POSITION>)` Returns the current value in the `<POSITION>` light sensor. `<POSITION>` can either be one of 'left', 'center', 'right' or one of the numbers 0, 1, 2. The positions 0, 1, and 2 correspond to the left, center, and right sensors. Examples:

```
>>> getBright()
```

```
[2124047, 1819625, 1471890]
>>> getBright('left')
2124047
>>> getBright(0)
2124047
>>> getBright('center')
1819625
>>> getBright(1)
1819625
>>> getBright('right')
1471890
>>> getBright(2)
1471890
```

The above values are from the camera image of the Firefox poster (see picture above). The values being reported by these sensors can vary depending on the view of the camera and the resulting brightness levels of the image. But you will notice that higher values imply bright segments and lower values imply darkness. For example, here is another set of values based on the image shown here on the right.




```
>>> getBright()
[1590288, 1736767, 1491282]
```

As we can see, a darker image is likely to produce lower brightness values. You can clearly see that the center of the image is brighter than its left or right sections.

It is also important to note the differences in the nature of information being reported by the `getLight` and `getBright` sensors. The first one reports the amount of ambient light being sensed by the robot (including the light above the robot). The second one is an average of the brightness obtained from the image seen from the camera. These can be used in many different ways as we will see later.

Do This: The program shown below uses a normalization function to normalize light sensor values in the range [0.0..1.0] relative to the values of ambient light. Then, the normalized left and right light sensor values are used to drive the left and right motors of the robot.

```
# record average ambient light values
Ambient = sum(getLight())/3.0

# This function normalizes light sensor values to 0.0..1.0
def normalize(v):
    if v > Ambient:
        v = Ambient

    return 1.0 - v/Ambient

def main():
    # Run the robot for 60 seconds
    while timeRemaining(60):
        L, R = getLight()
        # motors run proportional to light
        motors(normalize(L), normalize(R))
```

Run the above program on your Scribbler robot and observe its behavior. You will need a flashlight to affect better reactions. When the program is running, try to shine the flashlight on one of the light sensors (left or right). Observe the behavior. Do you think the robot is behaving like an insect? Which one? Study the program above carefully. There are some new Python features used that we will discuss shortly. We will also return to the idea of making robots behave like insects in the next chapter.

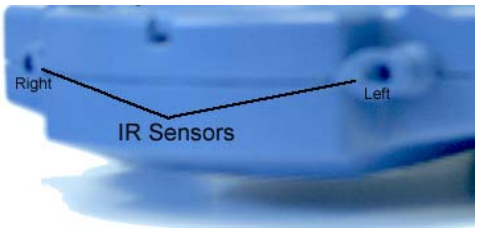
Proximity Sensing

The Scribbler has two sets of proximity detectors. There are two infrared (IR) sensors on the front of the robot and there are three additional IR obstacle sensors on the Fluke dongle. The following functions are available to obtain values of the front IR sensors:

`getIR()` Returns a list containing the two values of all IR sesnors.
`getIR(<POSITION>)` Returns the current value in the `<POSITION>` IR sensor. `<POSITION>` can either be one of 'left' or 'right' or one of the numbers 0, 1. The positions 0 and 1 correspond to the left, center, and right sensors.

Examples:

```
>>> getIR()
[1, 0]
>>> getIR('left')
1
>>> getIR(0)
1
>>> getIR('right')
0
>>> getIR(1)
0
```



IR sensors return either a 1 or a 0. A value of 1 implies that there is nothing in close proximity of the front of that sensor and a 0 implies that there is something right in front of it. These sensors can be used to detect the presence or absence of obstacles in front of the robot. The left and right IR sensors are places far enough apart that they can be used to detect individual obstacles on either side.

Do This: Run the `senses` function and observe the values of the IR sensors. Place various objects in front of the robot and look at the values of the IR proximity sensors. Take your note-book and place it in front of the robot about two feet away. Slowly move the notebook closer to the robot. Notice how the value of the IR sensor changes from a 1 to a 0 and then move the notebook away again. Can you figure out how far (near) the obstacle should be before it is detected (or cleared)? Try moving the notebook from side to side. Again notice the values of the IR sensors.

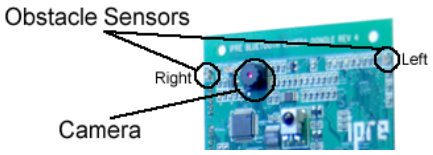
The Fluke dongle has an additional set of obstacle sensors on it. These are also IR sensors but behave very differently in terms of the kinds of values

they report. The following functions are available to obtain values of the obstacle IR sensors:

`getObstacle()`Returns a list containing the two values of all IR sesnors.
`getObstacle(<POSITION>)` Returns the current value in the `<POSITION>` IR sensor. `<POSITION>` can either be one of 'left', ‘center’, or 'right' or one of the numbers 0, 1, or 2. The positions 0, 1, and 2 correspond to the left, center, and right sensors.

Examples:

```
>>> getObstacle()
[1703, 1128, 142]
>>> getObstacle('left')
1703
>>> getObstacle(0)
1703
>>> getObstacle('center')
1128
>>> getObstacle(1)
1128
>>> getObstacle('right')
142
>>> getObstacle(2)
142
```



The values reported by these sensors range from 0 to 7000. A 0 implies there is nothing in front of the sensor where as a high number implies the presence of an object. The sensors on the sides can be used to detect the presense (or absence of walls on the sides.

Do This: Place your Scribbler on the floor, turn it on, start Python, and connect to it. Also connect the game pad controller and start the manual drive operation (gamepad()). Next, issue the `senses` command to get the realtime sensor display. Now, our objective here is to really "get into the robot's mind" and drive it around without ever looking at the robot. Also resist the temptation to take a picture. You can use the information displayed by the

sensors to navigate the robot. Try driving it to a dark spot, or the brightest spot in the room. Try driving it so it never hits any objects. Can you detect when it hits something? If it does get stuck, try to maneuver it out of the jam! This exercise will give you a pretty good idea of what the robot senses, how it can use its sensors, and to the range of behaviors it may be capable of. You will find this exercise a little hard to carry out, but it will give you a good idea as to what should go into the *brains* of such robots when you actually try to design them. We will try and revisit this scenario as we build various robot programs.

Also do this: Try out the program below. It is very similar to the program above that used the normalized light sensors.

```
def main():
    # Run the robot for 60 seconds
    while timeRemaining(60):
        L, R = getIR()
        # motors run proportional to IR values
        motors(R, L)
main()
```

Since the IR sensors report 0 or 1 values, you do not need to normalize them. Also notice that we are putting the left sensor value (`L`) into the right motor and the right sensor value (`R`) into the left motor. Run the program and observe the robot's behavior. Keep a note-book handy and try to place it in front of the robot. Also place it slightly on the left or on the right. What happens? Can you summarize what the robot is doing? What happens when you switch the `R` and `L` values to the motors?

You can see how simple programs like the ones we have seen above can result in interesting automated control strategies for robots. You can also define completely automated behaviors or even a combination of manual and automated behaviors for robots. In the next chapter we will explore several robot behaviors. First, it is time to learn about lists in Python.

Lists in Python

You have seen above that several sensor functions return lists of values. We also used lists to accumulate a series of pictures from the camera to generate an animated GIF. Lists are a very useful way of collecting a bunch of information and Python provides a whole host of useful operations and functions that enable manipulation of lists. In Python, a list is a sequence of objects. The objects could be anything: numbers, letters, strings, images, etc. The simplest list you can have is an empty list:

```
>>> []
[]
```

or

```
>>> L = []
>>> print L
[]
```

An empty list does not contain anything. Here are some lists that contain objects:

```
>>> N = [7, 14, 17, 20, 27]
>>> Cities = [“New York”, “Dar es Salaam”, “Moscow”]
>>> FamousNumbers = [3.1415, 2.718, 42]
>>> SwankyZips = [90210, 33139, 60611, 10036]
>>> MyCar = [“Mazda Protégé”, 1999, “Red”]
```

As you can see from above, a list could be a collection of any objects. Python provides several useful functions that enable manipulation of lists. Below, we will show some examples using the variables defined above:

```
>>> len(N)
5
>>>len(L)
0
>>> N + FamousNumbers
[7, 14, 17, 20, 27, 3.1415, 2.718, 42]
>>> SwankyZips[0]
90210
>>> SwankyZips[1:3]
```

```
[33139, 60611]
>>> 33139 in SwankyZips
True
>>> 19010 in SwankyZips
False
```

From the above, you can see that the function `len` takes a list and returns the length or the number of objects in the list. An empty list has zero objects in it. You can also access individual elements in a list using the indexing operation (as in `SwankyZips[0]`). The first element in a list has index 0 and the last element in a list of `n` elements will have an index `n-1`. You can concatenate two lists using the `'+'` operator to produce a new list. You can also specify a slice in the index operation (as in `SwankyZips[1:3]` to refer to the sublist containing elements from index 1 through 2 (one less than 3). You can also form `True`/`False` conditions to check if an object is in a list or not using the `in` operator. These operations are summarized in more detail at the end of the chapter.

Besides the operations above, Python also provides several other useful list operations. Here are examples of two useful list operations `sort` and `reverse`:

```
>>> SwankyZips
[90210, 33139, 60611, 10036]
>>> SwankyZips.sort()
>>> SwankyZips
[10036, 33139, 60611, 90210]
>>> SwankyZips.reverse()
>>> SwankyZips
[90210, 60611, 33139, 10036]
>>> SwankyZips.append(19010)
>>> SwankyZips
[90210, 60611, 33139, 10036, 19010]
```

`sort` rearranges elements in the list in ascending order. `reverse` reverses the order of elements in the list, and `append` appends an element to the end of the list. Some other useful list operations are listed at the end of the chapter.

Remember that lists are also sequences and hence they can be used to perform repetitions. For example:

```
>>> Cities = ["New York", "Dar es Salaam", "Moscow"]
>>> for city in Cities:
        print city

New York
Dar es Salaam
Moscow
```

The variable `city` takes on subsequent values in the list `Cities` and the statements inside the loop are executed once for each value of `city`. Recall, that we wrote counting loops as follows:

```
for I in range(5):
    <do something>
```

The function `range` returns a sequence of numbers:

```
>>> range(5)
[0, 1, 2, 3, 4]
```

This the variable `I` takes on values in the list `[0, 1, 2, 3, 4]` and, as in the example below, the loop is executed 5 times:

```
>>> for I in range(5):
     print I

0
1
2
3
4
```

Also recall that strings are sequences. That is, the string:

```
ABC = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
```

is a sequence of 26 letters. You can write a loop that runs through each individual letter in the string and speaks it out as follows:

```
>>> for letter in ABC:
        speak(letter)
```

There are also some useful functions that convert strings into lists. Say we have a string containing a sentence:

```
>>> sentence = "Would you have any Grey Poupon"
```

You can convert the string above into individual words using the `split` operation:

```
>>> sentence.split()
['Would', 'you', 'have', 'any', 'Grey', 'Poupon']
```

In light of the list operations presented above review some of the sensing examples from earlier in the chapter. We will be using lists in many examples in the remained of the text. For now, let us return to the topic of sensing.

Extrasensory Perception?

You have seen many ways of acquiring sensory information using the robot's sensors. In addition to the robot itself, you should be aware that your computer also has several "sensors" or devices to acquire all kinds of data. For example, you have already seen how, using the input function, you can input some values into your Python programs:

```
>>> N = input("Enter a number: ")
Enter a number: 42
>>> print N
42
```

Indeed, there are other ways you can acquire information into your Python programs. For example, you can input some data from a file in your folder. In Chapter 1 you also saw how you were able to control your robot using the

game pad controller. The game pad was actually plugged into your computer and was acting as an input device. Additionally, your computer is most likely connected to the internet using which you can access many web pages. It is also possible to acquire the content of any web page using the internet. Traditionally, in computer science people refer to this is a process of *input*. Using this view, getting sensory information from the robot is just a form of input. Given that we have at our disposal all of the input facilities provided by the computer, we can just as easily acquire input from any of the modalities and combine them with robot behaviors if we wish. Whether you consider this as *extra sensory perception* or not is a matter of opinion. Regardless, being able to get input from a diverse set of sources can make for some very interesting and useful computer and robot applications.

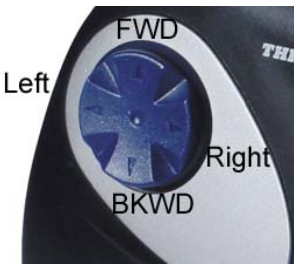
Game Pad Controllers

The game pad controller you used in Chapter 1 is a typical device that provides interaction facilities when playing computer games. These devices have been standardized enough that, just like a computer mouse or a keyboard, you can purchase one from a store and plug it into a USB port of your computer. Myro provides some very useful input functions that can be used to get input from the game pad controller. Game pads come is all kinds of flavors and configurations with varying numbers of buttons, axes, and other devices on them. In the examples below, we will restrict ourselves to a basic game pad shown in the picture on previous page.

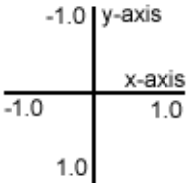
The basic game pad has eight buttons (numbered 1 through 8 in the picture) and an axis controller (see picture on right). The buttons can be pressed or released (on/off) which are represented by 1 (for on) and 0 (for off). The axis can be pressed in many different orientations represented by a pair of values

(for the x-axis and y-axis) that range from -1.0 to 1.0 with [0.0, 0.0] representing no activity on the axis. Two Myro functions are provided to access the values of the buttons and the axis:

Game Pad's Axis Control



Game Pad's Axes



`getGamepad(<device>)`
`getGamepadNow(<device>)`
returns the values indicating the status of the specified `<device>`. `<device>` can be `"axis"` or `"button"`. The `getGamepad` function returns only after `<device>` has been used by the user. That is, it waits for the user to press or use that device and then returns the values associated with the device at that instant. `getGamepadNow` does not wait and simply returns the device status right away. Here are some examples:

```
>>> getGamepadNow("axis")
[0.0, 0.0]
>>> getGamepad("axis")
[0.0, -1.0]
>>> getGamepadNow("button")
[0, 0, 0, 0, 1, 1, 0, 0]
```

Both `getGamepad` and `getGamepadNow` return the same set of values: axis values are returned as a list [x-axis, y-axis] (see picture on right for orientation) and the button values are returned as a list of 0, and 1's. The first value in the list is the status of button#1, followed by 2, 3, and so on. See picture above for button numbering.

Do This: Connect the game pad controller to your computer, start Python, and import the Myro module. Try out the game pad commands above and observe the values. Here is another way to better understand the operation of the game pad and the game pad functions:

```
while timeRemaining(30):
    print getGamepad("button")
```

Try out different button combinations. What happens when you press more than one button? Repeat the above for axis control and observe the values returned (keep the axes diagram handy for orientation purposes).

The game pad controller can be used for all kinds of interactive purposes, especially for robot control as well as in writing computer games (see Chapter X). Let us write a simple game pad based robot controller. Enter the program below and run it.

```
def main():
    # A simple game pad based robot controller
    while timeRemaining(30):
        X, Y = getGamePadNow("axis")
        motors(X, Y)
```

The program above will run for 30 seconds. In that time it will repeatedly sample the values of the axis controller and since those values are in the range -1.0..1.0, it uses them to drive the motors. When you run the above program observe how the robot moves in response to pressing various parts of the axis. Do the motions of the robot correspond to the directions shown in the game pad picture on previous page? Try changing the command from `motors` to `move` (recall that move takes two values: translate and rotate). How does it behave with respect to the axes? Try changing the command to `move(-X, -Y)`. Observe the behavior.

As you can see from the simple example above, it is easy to combine input from a game pad to control your robot. Can you expand the program above to behave exactly like the `gamepad` controller function you used in Chapter 1? (See Exercise YY).

The World Wide Web

If your computer is connected to the internet, you can also use Python facilities to access the content of any web page and use it as input to your program. Web pages are written using markup languages like HTML and so when you access the content of a web page you will get the content with the markups included. In this section we will show you how to access the content of a simple web page and print it out. Later we will see how you could use the information contained in it to do further processing.

Go to a web browser and take a look at the web page:

`http://www.fi.edu/weather/data/jan07.txt`

This web page is hosted by the Franklin Institute of Philadelphia and contains recorded daily weather data for Philadelphia for January 2007. You can navigate from the above address to other pages on the site to look at daily weather data for other dates (the data goes back to 1872!). Below, we will show you how, using a Python library called `urllib`, you can easily access the content of any web page. The `urllib` library provides a useful function called `urlopen` using which you can access any web page on the internet as follows:

```
>>> from urllib import *
>>> Data = urlopen("http://www.fi.edu/weather/data/jan07.txt")
>>> print Data.read()

January 2007
Day   Max   Min   Liquid      Snow  Depth
1     57    44    1.8   0     0
2     49    40    0     0     0
3     52    35    0     0     0
…     …     …     …     …     …
31    31    22    0     0     0
#days 31
Sum   1414  1005  4.18  1.80  1.10
```

The following two commands are important:

```
>>> Data = urlopen("http://www.fi.edu/weather/data/jan07.txt")
>>> print Data.read()
```

The first command uses the function `urlopen` (which is imported from the `urllib`) to establish a connection between your program and the web page. The second command issues a `read` to read from that connection. Whatever is read from that web page is printed out as a result of the print command.

A little more about Python functions

Before we move on, it would be good to take a little refresher on writing Python commands/functions. In Chapter 2 we learned that the basic syntax for defining new commands/functions is:

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

The Myro module provides you with several useful functions (`forward`, `turnRight`, etc.) that enable easy control of your robot's basic behaviors. Additionally, using the syntax above, you learned to combine these basic behaviors into more complex behaviors (like `wiggle`, `yoyo`, etc.). By using parameters you can further customize the behavior of functions by providing different values for the parameters (for example, `forward(1.0)` will move the robot faster than `forward(0.5)`). You should also note a crucial difference between the movement commands like `forward`, `turnLeft`, and commands that provide sensory data like `getLight` or `getStall`, etc. The sensory commands always *return* a value whenever they are issued. That is:

```
>>> getLight('left')
221
>>> getStall()
0
```

Commands that return a value when they are invoked are called *functions* since they actually behave much like mathematical functions. None of the movement commands return any value, but they are useful in other ways. For instance, they make the robot do something. In any program you typically need both kinds of functions: those that do something but do not return anything as a result; and those that do something and return a value. You can already see the utility of having these two kinds of functions from the examples you have seen so far. Functions are an integral and critical part of any program and part of learning to be a good programmer is to learn to recognize abstractions that can then be packaged into individual functions (like `drawPolygon`, or `degreeTurn`) which can be used over and over again.

Writing functions that return values

Python provides a `return`-statement that you can use inside a function to return the results of a function. For example:

```
def triple(x):
   # Returns x*3
   return x * 3
```

The function above can be used just like the ones you have been using:

```
>>> triple(3)
9
>>> triple(5000)
15000
```

The general form of a `return`-statement is:

```
return <expression>
```

That is, the function in which this statement is encountered will return the value of the `<expression>`. Thus, in the example above, the `return`-statement returns the value of the expression `3*x`, as shown in the example

invocations. By giving different values for the parameter `x`, the function simply triples it. This is the idea we used in normalizing light sensor values in the example earlier where we defined the function normalize to take in light sensor values and normalize them to the range 0.0..1.0 relative to the observed ambient light values:

```
# This function normalizes light sensor values to 0.0..1.0
def normalize(v):
    if v > Ambient:
        v = Ambient

    return 1.0 - v/Ambient
```

In defining the function above, we are also using a new Python statement: the if-statement. This statement enables simple decision making inside computer programs. The simplest form of the `if`-statement has the following structure:

```
if <CONDITION>:
   <do something>
   <do something>
   ...
```

That is, if the condition specified by `<CONDITION>` is `True` then whatever is specified in the body of the `if`-statement is carried out. In case the `<condition>` is `False`, all the statements under the `if` command are skipped over.

Functions can have zero or more `return`-statements. Some of the functions you have written, like `wiggle` do not have any. Technically, when a function does not have any return statement that returns a value, the function returns a special value called `None`. This is already defined in Python.

Functions, as you have seen, can be used to package useful computations and can be used over and over again in many situations. Before we conclude this section, let us give you another example of a function. Recall from previous chapter, the robot behavior that enables the robot to go forward until it hits a

wall. One of the program fragments we used to specify this behavior is shown below:

```
while not getStall():
   forward(1)
stop()
```

In the above example, we are using the value returned by `getStall` to help us make the decision to continue going forward or stopping. We were fortunate here that the value returned is directly usable in our decision making. Sometimes, you have to do little interpretation of sensor values to figure out what exactly the robot is sensing. You will see that in the case of light sensors. Even though the above statements are easy to read, we can make them even better, by writing a function called `stuck()` as follows:

```
def stuck():
   # Is the robot stalled?
   # Returns True if it is and False otherwise.

   return getStall() == 1
```

The function above is simple enough, since `getStall` already gives us a usable value (`0`/`False` or `1`/`True`). But now if we were to use `stuck` to write the robot behavior, it would read:

```
while not stuck():
   forward(1)
stop()
```

As you can see, it reads much better. Programming is a lot like writing in this sense. As in writing, there are several ways of expressing an idea in words. Some are better and more readable than others. Some are downright poetic. Similarly, in programming, expressing something in a program can be done in many ways, some better and more readable than others. Programming is not all about functionality, there *can* be poetry in the way you write a program.

Files

Do file input here as well…???
Exaple of baby names? Sort list using built-in sort.

Summary

In this chapter you have learned all about obtaining sensory data from the robot's perceptual system to do visual sensing (pictures), light sensing, and proximity sensing. The Scribbler provides a rich set of sensors that can be used to design interesting robot behaviors. You also learned that sensing is equivalent to the basic input operation in a computer. You also learned how to get input from a game pad, the World Wide Web, and from data files. Programs can be written to make creative use of the input modalities available to define robot behaviors, computer games, and even processing data. In the rest of the book you will learn how to write programs that make use of these input modalities in many different ways.

Myro Review

`senses()`
Displays Scribbler's sensor values in a window. The display is updated every second.

```
takePicture()
takePicture("color")
TakePicture("gray")
```
Takes a picture and returns a picture object. When no parameters are specified, the picture is in color.

`show(<picture>)`
Displays the picture in a window. You can click the left mouse anywhere in the window to display the (x, y) and (r, g, b) values of the point in the window's status bar.

`savePicture(<picture>, <file>)`
`savePicture([<picture1>, <picture2>, …], <file>)`
Saves the picture in the file specified. The extension of the file should be `".gif"` or `".jpg"`. If the first parameter is a list of pictures, the file name should have an extension `".gif"` and an animated GIF file is created using the pictures provided.

`getLight()`
Returns a list containing the three values of all light sensors.

`getLight(<POSITION>)`
Returns the current value in the `<POSITION>` light sensor. `<POSITION>` can either be one of 'left', 'center', 'right' or one of the numbers 0, 1, 2. The positions 0, 1, and 2 correspond to the left, center, and right sensors.

`getbright()`
Returns a list containing the three values of all light sensors.

`getBright(<POSITION>)`
Returns the current value in the `<POSITION>` light sensor. `<POSITION>` can either be one of 'left', 'center', 'right' or one of the numbers 0, 1, 2.

`getIR()`
Returns a list containing the two values of all IR sesnors.

`getIR(<POSITION>)`
Returns the current value in the `<POSITION>` IR sensor. `<POSITION>` can either be one of 'left' or 'right' or one of the numbers 0, 1.

`getObstacle()`
Returns a list containing the two values of all IR sesnors.

`getObstacle(<POSITION>)`
Returns the current value in the `<POSITION>` IR sensor. `<POSITION>` can either be one of 'left', ‘center’, or 'right' or one of the numbers 0, 1, or 2.

`getGamepad(<device>)`
`getGamepadNow(<device>)`
Returns the values indicating the status of the specified `<device>`. `<device>` can be `"axis"` or `"button"`. The `getGamepad` function waits for an event before returning values. `getGamepadNow` immediately returns the current status of the device.

Python review

Lists…
`<string>.split()`

`urlopen(<URL>)`
Establishes a stream connection with the `<URL>`. This function is to be imported from the Python module `urlopen`.

`<stream>.read()`
Reads the entire contents of the `<stream>` as a string.

Exercises

6 Insect-like Behaviors

Oh, Behave!
-: Austin Powers (played by Mike Myers),
Austin Powers: International Man of Mystery,
New Line Cinema, 1997.

Designing robot behaviors is a challenging, yet fun process. There isn't a formal methodology or a technique that one can follow. It involves creativity, the ability to recognize the strengths and limitations of the physical robot, the kind of environment the robot will be carrying out the behavior, and of course the knowledge of available paradigms for programming robot behaviors. Creativity is essential to the design of robot behaviors. You have already seen how even a simple robot like the Scribbler can be programmed to carry out a diverse range of behaviors. We have also spent a considerable effort so far in exploring the range of possible functions a robot can perform. Where a robot is placed when it is running can play an important role in exhibiting a

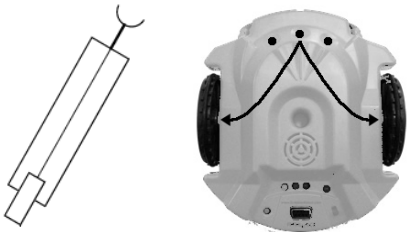
programmed behavior successfully. In this chapter, we take a different look at robot behaviors.

Braitenberg Vehicles

In 1984, Valentino Braitenberg wrote a book titled: Vehicles: Experiments in Synthetic Psychology (MIT Press). In it, he describes several thought experiments that are centered around the creation of simple vehicles with very simple control mechanisms that exhibit seemingly complex behavior. The purpose of the thought experiments was to illustrate some fundamental insights into the internal structure of animal (and human) brains. Each experiment involves a descripion of a simple vehicle that is endowed with a small suite of sensors (much like our Scribbler robot) and how the sensors can be connected to the motors of these imaginary vehicles in ways that parallel neurological connections in animals. He shows how the resulting vehicles are capable of complex behaviors which can be described as: fear, aggression, love, logic, free will, etc. One central theme underlying Braitenberg's experiments is the demonstration of what he calls the *Law of uphill analysis and downhill invention*: It is much more difficult to guess the internal structure of an entity just by observing its behavior than it is to actually create the structure that leads to the behavior. That is, trying to postulate the internal structure purely by observing certain behavior is an uphill (harder) task whereas trying to create an entity that exhibits a certain behavior is a downhill (easy) task. While all of Braitenberg's vehicles were imaginary and not really designed to be actually fabricated people have found it a fun and intellectually interesting exercise to create them. Personal robots like Scribblers make perfect platforms to do this and in what follows we will describe some of Braitenberg's (and Braitenberg-type) vehicles and design robot behaviors based on them.

Vehicle 1: Alive

Vehicle#1: Alive



The first vehicle Braitenberg describes has one sensor and one motor. The value transmitted by the sensor directly feeds into the motor. If the value being reported by the sensor is a varying quantity (say light), the vehicle will move at a speed proportional to the amount of quantity being detected by the sensor.

A schematic of the vehicle is shown on the left. In order to design this vehicle using the Scribbler, you can use the center light sensor and connect what it reports directly to both motors of the robot. This is shown on the right. That is, the same light reading is directly controlling both the motors by the same amount. As you have already seen, there are a many different ways to specify motor movement commands to the Scribbler. Suppose the value obtained from the center light sensor is `C`, you can control both motors using this value by using the command:

```
motors(C, C)
```

Alternately, you can also use the `forward` command:

```
forward(C)
```

Now that we know how the internal structure of this vehicle looks, we can start to write a program that will implement it. But, before we get there, we need to sort out a small issue of compatibility: light sensors report values in the range 0..5000 whereas motors and movement commands take values in the range -1.0 to 1.0. In this example, we are only concerned with movements that range from a complete stop to full speed forward, so the values range from 0.0 to 1.0. We have to computationally normalize, or map the light sensor values in this range. A first attempt at this would be to write a function called `normalize` that operates as follows:

```
def normalize(v):
   # normalizes v to range 0.0 to 1.0
```

Once we have this function, we can write the behavior for the vehicle as follows:

```
def main():
    # Braitenberg vehicle#1: Alive

    while True:
        L = getLight("center")
        forward(normalize(L))

main()
```

Normalizing Sensor Values

It is time now to think about the task of the `normalize` function. Given a value received from a light sensor, it has to transform it to a proportional value between 0.0 and 1.0 so that the brighter the light, the higher the value (i.e. closer to 1.0). Vehicle#1 moves in proportion to the amount of light it receives. This is a good time to revisit the `senses` function of Myro to look at the values reported by the light sensors. Go ahead and do this.

After examining the values returned by the light sensors you may notice that they report small values (less than 50) for bright light and larger values (as large as 3000) for darkness. In a way, you can say that the light sensor is really a darkness sensor; the darker it is the higher the values reported by it. The light sensors are capable of reporting values between 0 and 5000. Now, we can certainly calibrate or normalize using these values using the following definition of `normalize`

```
def normalize(v):
   # Normalize v (in the range 0..5000) to 0..1.0, inversely

   return 1.0 - v/5000.0
```

That is, we divide the value of the light sensor by its maximum value and then subtract that from 1.0 (for inverse proportionality). Thus a brighter light value, say a value of 35, will get normalized as:

```
1.0 - 35.0/5000.0 = 0.9929
```

If 0.9929 is sent to the motors (as in the above program), the robot would move full speed forward. Let us also compute the speed of the robot when it is in total darkness. When you place a finger on the center sensor, you will get values in the 2000-3000 range. For 3000, the normalization will be:

```
1.0 - 3000.0/5000.0 = 0.40
```

The robot will still be moving, although at nearly half the speed. Most likely, you will be operating the robot in a room where there is sufficient ambient light. You will notice that under ambient daylight conditions, the values reported by the light sensors are in the 150-250 range. Using the above normalization you will get:

```
1.0 - 200.0/5000.0 = 0.9599
```

That is almost full speed ahead. In order to experience the true behavior of the above vehicle, we have to use a normalization scheme that takes into account the ambient light conditions (they will vary from room to room). Further, let us assume that in ambient light conditions, we will watch the robot respond to a light source that we will control. A flashlight will work nicely. So, to make the robot appropriately sensitive to the flashlight under ambient light conditions you can write a better version of normalize as follows:

```
def normalize(v):
    if v > Ambient:
        v = Ambient

    return 1.0 - v/Ambient
```

That is, the darkest condition is represented by the ambient light value (`Ambient`) and then normalization is done with respect to that value. You can

either set the ambient value by hand, or, a better way is to have the robot sense its ambient light at the time the program is initiated. This is the same version of normalize that you saw in the previous chapter. Now you know how we arrived at it. The complete program for Vehicle#1 is shown below:

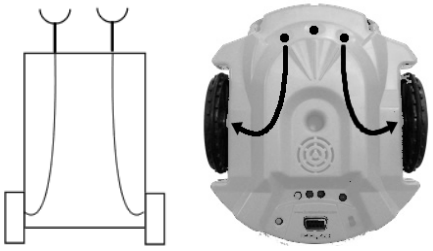
```
# Braitenberg Vehicle#1: Alive
from myro import *
initialize("com"+ask("What port?"))

Ambient = getLight("center")

def normalize(v):
    if v > Ambient:
        v = Ambient

    return 1.0 - v/Ambient

def main():
    # Braitenberg vehicle#1: Alive

    while True:
        l = getLight("center")
        forward(normalize(l))
```

Do This: Implement the program above and observe the robot's behavior. Does it respond as described above?

You may have also noticed by now that the three light sensors are not necessarily closely matched. That is, they do not report exactly the same values under the same conditions. When writing robot programs that use multiple light sensors, it is a good idea to average the values returned by all the light sensors to represent the ambient value. Modify the program above to use the average of all three values as the ambient value. There shouldn't be a noticeable difference in the robot's behavior. However, this is something you may want to use in later programs.

Vehicle 2: Coward and Aggressive

Vehicle 2a: Coward



The next set of vehicles use two sensors. Each sensor directly drives one motor. Thus the speed of the individual motor is directly proportional to the quantity being sensed by it sensor. There are two possible ways to connect the sensors. In the first case, Vehicle2a, the sensor on each side connects to the motor on the same side. In the other case, Vehicle2b, the connections are interchanged. That is, the left sensor connects to the right motor and the right sensor connects to the left motor. Let us design the control program for Vehicle 2a first:

```
# Vraitenberg Vehicle#2a
from myro import *
initialize("com"+ask("What port?"))

Ambient = sum(getLight())/3.0

def normalize(v):
    if v > Ambient:
        v = Ambient

    return 1.0 - v/Ambient

def main():
    # Braitenberg vehicle#2a: Coward

    while True:
        l = getLight("left")
        r = getLight("right")
        motors(normalize(l), normalize(r))
```

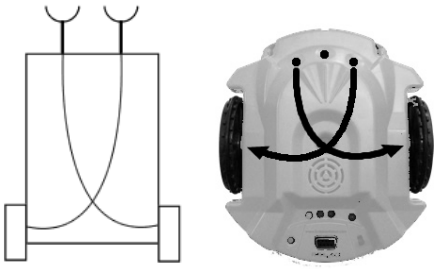
The structure of the above program is very similar to that of Vehicle1. We have modified the setting of the ambient light value to that of an average of the three light values. Also, we use the `motors` command, to drive the left and

right motors proportional to the left and right light sensor values (after normalizing).

Do This: Implement the control program for Vehicle 2a as shown above. Observe the robot's behaviors by shining the flashlight directly in front of the robot and in each of the left and right sensors.

Next, write the control program for Vehicle2b as shown here. This requires one simple change from the program of Vehicle2a: switch the parameters of the `motors` command to reflect the interchanged connections. Again observe the behaviors by shining the flashlight directly ahead of the robot and also a little to each side.

Vehicle 2b: Aggressive



You will notice that the robots behave the same way when the light is placed directly ahead of them: they are both attracted to light and hence move towards the light source. However, Vehicle 2a will move away from the light if the light source is on a side. Since the nearer sensor will get excited more, moving the corresponding motor faster, and thereby turning the robot away. In the case of Vehicle 2b, however, it will always turn towards the light source and move towards it. Braitenberg calls these behaviors *coward* (2a) and *aggressive* (2b).

Controlling Robot Responses

It is often necessary, when designing and testing robot behaviors, to properly set up the robot's environment and the orientation of the robot in it. In simple cases this is easily achieved by first placing the robot in the desired orientation and then loading and executing the program. However prior to the robot's actual behavior, the robot may need to perform some preliminary observations (for example, sensing ambient light), it becomes necessary to re-

orient the robot properly before starting the execution of the actual behavior. This can be easily accomplished by including some simple interactive commands in the robot's program. The resulting program structure is shown below:

```
# import myro library and establish connection with the robot
# define all functions here (like, normalize, etc.)
# set values of ambient conditions

def main():
    # Description of the behavior...

    # Give user the opportunity to set up the robot
    askQuestion("Press OK to begin...", ["OK"])

    # Write your robot's behavior commands here
```

Do This: Modify the programs for Vehicles 1, 2a, and 2b to include the `askquestion` command above.

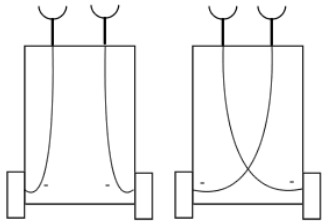
We have introduced a few basic programming patterns above that can be used in many robot programming situations. The thing to remember is that, at any point in the execution of a robot's program, you can also program appropriate interjections to perform various experimental or control functions. We will see several other examples later on.

Other Normalizations

All the normalizations of light sensor values shown above were used to normalize the values in the range 0.0..1.0 in direct proportion to the amount of light being sensed. That is, the darker it is, the closer the normalized values are to 0.0 and the brighter it gets, the closer the normalized values get to 1.0. This is just one way that one can relate the quantity being sensed to the amount of speed applied to the robot's motors. You can imagine other relationships. The most obvious of course is an inverse relationship: the darker it is the closer to 1.0 and vice versa. Braitenberg calls this *inhibitory* (as opposed to *excitatory*) relationship: the more of a quantity being sensed,

the slower the robot's motors turn. As in Vehicles 2a and 2b above, there is choice of two kinds of connections: straight and crossed. These are shown below (a plus (+) sign next to a connector indicates an excitatory connection and a minus sign (-) represents an inhibitory connection):

Vehicles 3a (Love) and 3b (Explorer)



Writing the `normalize` function for an inhibitory connection is quite straightforward:

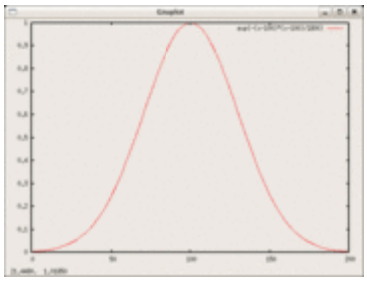
```
def normalize(v):
    if v > Ambient:
        v = Ambient

    return v/Ambient
```

Braitenberg describes the behavior of the resulting vehicles as *love* (Vehicle 3a) and *explorer* (Vehicle 3b). That is, if you were to observe the behavior of the two vehicles, you are likely to notice that Vehicle 3a will come to rest facing the light source (in its vicinity) whereas vehicle 3b will come to rest turned away from the source and may wander away depending on the presence of other light sources.

In other variations on sensor value normalizations, Braitenberg suggests using non-monotonic mathematical functions. That is, if you look at the excitatory and inhibitory normalizations, they can be described as monotonic: more light, faster motor speed; or more light, slower motor speed. But consider other kinds of relationships for normalizations. Observe the function shown on the next page. That is, the relationship is increasing in proportion to sensory input but only up to a certain point and after that it decreases. Incorporating such relationships in vehicles will lead to more complex behavior (Braitenberg describes them as vehicles having *instincts*). The following defines a normalization function, based on the curve shown:

$$f(x) = e^{-(x-100)^2/1800}$$

A Non-Monotonic Function



The above function is based on the simpler function:

$$f(x) = e^{-x^2}$$

which in the first definition is stretched to span the range 0..200 for values of x with 100 being the point where it reports the maximum value (i.e. 1.0). Mathematically this function is also known as the *bell curve* or a *Gaussian Curve* in general. A bell curve is defined in terms of a *mean (π)* and *standard deviation (σ)* as shown below:

$$f(x) = e^{-(x-\pi)^2/2*\sigma^2}$$

Thus, in the normalization function we are using 100 as mean and 30 as standard deviation. You can easily scale the curve for the range of sensor values you desire using the following `normalize` function.

```
def normalize(v):
        mean = Ambient/2.0
        stddev = Ambient/6.0
        if v >= Ambient:
                v = Ambient
        return exp(-(v - mean)**2 / 2*(stddev**2))
```

`exp(x)` is a Python function that computes the value of e^x. It is available in the Python `math` library. We will delve into the math library in more detail in the next chapter. In order to use the `exp` function as shown above you have to import the `math` library:

```
from math import *
```

There are of course several other possibilities that one could try: a step function; or a threshold; and any other mathematical combinations. The key idea is that there is a clear mapping of the range of sensor values to motor values in the range 0.0..1.0.

Robots using these normalizations and other variations are likely to exhibit very interesting and sometimes unpredictable behaviors. Observers unaware of the internal mapping mechanisms will have a hard time describing precisely the robot's behavior and will tend to use anthropomorphic terms (like, *love*, *hate*, *instincts*, etc.) to describe the behavior of robots. This is what an *uphill analysis* means.

Multiple Sensors

Adding several sensors enriches the design space for robot behaviors. As a designer, you now have a choice of different types of mappings: excitatory, inhibitory, or more complex; and connections: straight or crossed. Suddenly the resulting robot behavior will seem complex. On the Scribbler, for instance, in addition to light sensors, you also have the stall sensor, and the IR sensors. With the exception of light sensors, all of these other sensors are digital or threshold sensors that are either ON or OFF (i.e. they report values that are either 0 or 1 indicating the presence or absence or the thing they are sensing). In a way you can think that the digital sensors are already normalized, but it is still possible to invert the relationship if need be. You can design several interesting behaviors by combining two or more sensors and deciding whether to connect them straight or crossed.

Do This: In your design of vehicles 2a and 2b substitute the obstacle sensor in place of the light sensors. Describe the behavior of the resulting vehicles. Try the same for Vehicles 3a and 3b. Next, combine the behavior of the resulting vehicles with the light sensors. Try out all combinations of connections, as well as inhibitory and excitatory mappings. Which vehicles exhibit the most interesting behaviors?

More Vehicles

Here are descriptions of several vehicles that are in the spirit of Braitenberg's designs and also exhibit interesting behaviors. Using the concepts and programming techniques from above, try to implement these on the Scribbler robot. Once completed, you should invite some friends to observe the behaviors of these creatures and record their reactions.

Timid

Timid is capable of moving forward in a straight line. It has one threshold light sensor, pointing up. When the light sensor detects light, the creature moves forward, otherwise, it stays still. The threshold of the light sensor should be set to ambient light. That way, when the creature can "see" the light, it will move. When it enters a shadow (which can be cast by a hand or another object) it stops. If whatever is casting the shadow is moved, the creature will move again. Therefore, timid is a shadow seeker.

Indecisive

Indecisive is similar to Timid, except, it never stops: its motors are always running, either in forward direction, or in reverse direction, controlled by the threshold light sensor. When the light sensor detects light, it moves forward, otherwise, it moves backwards. When you run this creature, you will notice that it tends to oscillate back and forth at shadow edges. Thus, Indecisive is a shadow edge seeker.

Paranoid

Paranoid is capable of turning. This is accomplished by moving the right motor forward and moving the left motor in reverse direction at the same time. It has a single threshold light sensor. When the sensor detects light, it moves forward. When the sensor enters a shadow, it reverses the direction of its left motor, thus turning right. Soon the sensor will swing around, out of the

shadow. When that happens, it resumes its forward motion. Paranoid, is a shadow fearing creature.

This, That, or the Other

The `if`-statement introduced earlier is a way of making simple decisions (also called one-way decisions). That is, you can conditionally control the execution of a set of commands based on a single condition. The if-statement is Python is quite versatile and can be used to make two-way or even multi-way decisions. Here is how you would use it to choose among two sets of commands:

```
if <condition>:
   <this>
else:
   <that>
```

That is, if the `<condition>` is true it will do the commands specified in `<this>`. If, however, the `<condition>` is false, it will do `<that>`. Similarly, you can extend the `if`-statement to help specify multiple options:

```
if <condition-1>:
   <this>
elif <condition-2>:
   <that>
elif <condition-3>:
   <something else>
...
...
else:
   <other>
```

Notice the use of the word `elif` (yes, it is spelled that way!) to designate "else if". Thus, depending upon whichever condition is true, the corresponding `<this>`, `<that>`, or `<something else>` will be carried out. If all else fails, the `<other>` will be carried out.

Simple Reactive Behaviors

Using the three light sensors the robot can detect varying light conditions in its environment. Let us write a robot program that makes it detect and orient towards bright light. Recall from Chapter 5 that light sensors report low values in bright light conditions and high values in low light. To accomplish this task, we only need to look at the values reported by left and right light sensors. The following describes the robot's behavior:

```
do for a given amount of time
   if left light is brighter than right light
      turn left
   else
      turn right
```

Thus, by making use of the if-else statement, we can refine the above into the following:

```
while timeRemaining(30):
   if left light is brighter that right light:
      turnLeft(1.0)
   else:
      turnRight(1.0)
```

The only thing remaining in the commands above is to write the condition to detect the difference between the two light sensors. This can be done using the expression:

```
getLight('left') < getLight('right')
```

Do This: Write a complete program that implements the above behavior and test it on your robot.

You may have noticed that even in uniform lighting conditions sensors tend to report different values. It is generally a good idea to threshold the difference when making the decision above. Say we set the threshold to a difference of at least 50. That is, if the left and right sensors differ by at least 50 then turn

towards the brighter sensor. What happens if the difference is less than the threshold? Let us decide that in that case the robot will stay still. This behavior can be captured by the following:

```
thresh = 50

while timeRemaining(30):
   # Get sensor values for left and right light sensors
   L = getLight('left')
   R = getLight('right')

   # decide how to act based on sensors values
   if (L - R) > thresh:
      # left is seeing less light than right so turn right
      turnRight(1.0)
   elif (L - R) < thresh:
      # right is seeing less light than left, so turn left
      turnLeft(1.0)
   else:
      # the difference is less than the threshold, stay put
      stop()
```

Notice how we have used the variable `thresh` to represent the threshold value. This is good programming practice. Since the performance of sensors varies under different light conditions, this allows you to adjust the threshold by simply changing that one value. By using the name `thresh` instead of a fixed value, say 50, you only have to make such changes in one place of your program.

In the statements above, there is a pattern that you will find recurring in many programs that define robot behaviors using simple decisions:

```
while timeRemaing(<seconds>):
   <sense>
   <decide and then act>
```

Such behaviors are called *reactive* behaviors. That is, a robot is reacting to the change in its environment by deciding how to act based on its sensor values. A wide range of robot behaviors can be written using this program structure.

Below, we present descriptions of several interesting, yet simple automated robot behaviors. Feel free to implement some of them on your robot.

Simple Reactive Behaviors

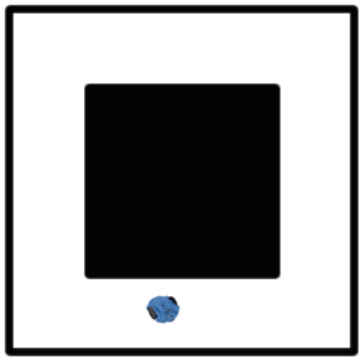
Most of the behaviors described below require selection among alternatives using conditional expressions and `if`-statements.

Refrigerator Detective: As a child did you ever wonder if that refrigerator light was always on? Or did it shut off when you closed the door? Well, here is a way to find out. Build a refrigerator detective robot that sits inside the fridge and tells you if the light is on or off!

Burglar Alarm Robot: Design a robot that watches your dorm door. As soon as the door opens, it sounds an alarm (beeps).

Wall Detector: Write a robot program that goes straight and then stops when it detects a wall in front. You will be using the IR sensors for this task.

Hallway Cruiser: Imagine your robot in an environment that has a walled corridor going around a 2 ft by 2 ft square box (see picture on right). Write a program that will enable the robot to go around this box. One strategy you can use is to have the robot go forward in a straight line until it bumps into a wall. After a bump it will proceed to make a 90 degree turn (you may need to have it go backwards a little to enable turning room) and then continue again in a straight line.



Measuring Device: You have calibrated your robot with regards to how far it travels in a given amount of time. You can use that to design a robot that

measures space. Write a program that enables a robot to measure the width of a hallway.

Follower: Write a robot program to exhibit the following behavior: The robot prefers to stay near a wall (in front). If it does not have a wall in front of it, it moves forward until it finds it. Test your program first by placing the robot in a play pen. Ensure that your program behaves as described. Next, place it on a floor and hold a blank piece of paper in front of it (close enough so the robot can detect it). Now, slowly move the paper away from the robot. What happens?

Designing Reactive Behaviors

Most of the robot behaviors that are implemented using the Braitenberg style rely on a few simple things: selecting one or more sensors; choosing the kind of wiring (straight or crossed); and selecting normalization functions for each sensor. While you can guess the behavior that may result from these designs the only way to confirm this is by actually watching the robot carry out the behavior. You also saw how, using if-statements you can design simple, yet interesting robot. In this section we will design additional reactive behaviors.

Light Following

To begin, it will be fairly straightforward to extend the behavior of the light orienting behavior from above into one that results in a light follower robot. That is, with a flashlight you will be able to guide the robot to follow you around. Again, you have to start by observing the range of values reported by the robot under various lighting conditions. If a flashlight is going to be the bright light source, you will observe that the light sensors report very low values when a light is shining directly on them (typically in the 0..50 range). Thus, deciding which way to go (forward, turn left, or turn right) can be decided based on the sensor readings from the three light sensors. The structure of the program appears as follows:

```
# Light follower

from myro import *
initialize(ask("What port?"))

# program settings...

thresh = 50
fwdSpeed = 0.8
cruiseSpeed = 0.5
turnSpeed = 0.7     # this is a left turn, -0.7 will be right
turn

def main():
    while True:
       # get light sensor values for left, center, and right
       L, C, R = getLight()

       # decide how to act based on sensor values
       if C < thresh:
          # bright light from straight ahead, go forward
          move(fwdSpeed, 0)
       elif L < thresh:
          # bright light at left, turn left
          move(cruiseSpeed, turnSpeed)
       elif R < thresh:
          # bright light on right side, turn right
          move(cruiseSpeed, -turnSpeed)
       else:
          # no bright light, move forward slowly (or stop?)
          move(cruiseSpeed/2, 0)
main()
```

Notice that, in the program above, we have decided to set values for light threshold (`thresh`) as well as movements to specific values. Also, in all cases, we are using the `move` command to specify robot movement. This is because the `move` command allows us to blend translation and rotation movement. Additionally, notice that regardless of the sensor values, the robot is always moving forward some amount even while turning. This is essential since the robot has to follow the light and not just orient towards it. In the case where

there is no bright light present, the robot is still moving forward (at half the cruise speed).

Do This: Implement the light following program as described above and observe the robot's behavior. Try adjusting the value settings (for threshold as well as motor speeds) and note the changes in the robot's behaviors. Also, do you observe that this behavior is similar to any of the Braitenberg vehicles described above? Which one?

In the design of the light following robot above, we used a threshold value for detecting the presence of bright light. Sometimes it is more interesting to use differential thresholds for sensor values. That is, is the light sensor's value different from the ambient light by a certain threshold amount? You can use the `senses` function again observe the differences from ambient light and modify the program above to use the differential instead of the fixed threshold.

Here is another idea. Get several of your classmates together in a room with their robots, all running the same program. Make sure the room has plenty of floor space and a large window with a curtain. Draw close the curtains so the outside light is temporarily blocked. Place the robots all over the room and start the program. The robots will scurry around, cruising in the direction of their initial orientation. Now, slowly draw the curtains open to let in more light. What happens?

Avoiding Obstacles

Obstacles in the path of a robot can be detected using the IR sensors in front of the robot. Then, based on the values obtained, the robot can decide to turn away from an approaching obstacle using the following algorithm:

```
if obstacle straight ahead, turn (left or right?)
if obstacle on left, turn right
if obstacle on right, turn left
otherwise cruise
```

This can be implemented using the program below:

```
# Avoiding Obstacles

from myro import *
initialize(ask("What port?"))

# program settings...

cruiseSpeed = 0.6
turnSpeed = 0.5     # this is a left turn, -0.5 will be right
turn

def main():
    while True:
       # get sensor values for left and right IR sensors
       L, R = getIR()
       L = 1 - L
       R = 1 - R

       # decide how to act based on sensors values
       if L and R:
          # obstacle straight ahead, turn (randomly)
          move(0, turnSpeed)
       elif L:
          # obstacle on left, turn right
          move(cruiseSpeed, -turnSpeed)
       elif R:
          # obstacle on right, turn left
          move(cruiseSpeed, turnSpeed)
       else:
          # no obstacles
          move(cruiseSpeed, 0)
main()
```

As in the case of the light follower, observe that we begin by setting values for movements. Additionally, we have flipped the values of the IR sensors so that the conditions in the `if`-statements look more natural. Recall that the IR sensors report a 1 value in the absence of any obstacle and a 0 in the presence of one. By flipping them (using `1 - value`) the value is 1 for an obstacle

present and 0 otherwise. These values make it more natural to write the conditions in the program above. Rremember in Python, a 0 is equivalent to `False` and a 1 is equivalent to `True`. Read the program above carefully and make sure you understand these subtleties. Other than that, the program structure is very similar to the light follower program.

Another way to write a similar robot behavior is to use the value of the stall sensor. Recall that the stall sensor detects if the robots has bumped against something. Thus, you can write a behavior that doesn't necessarily avoid obstacles, but navigates itself around by bumping into things. This is very similar to a person entering a dark room and then trying to feel their way by touching or bumping slowly into things. In the case of the robot, there is no way to tell if the bump was on its left or right. Nevertheless, if you use the program (shown below) you will observe fairly robust behavior from the robot.

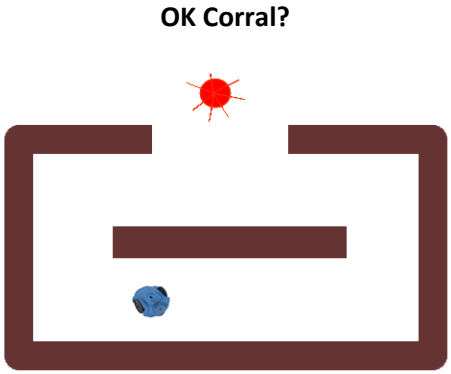
```
# Avoiding Obstacles by bumping

from myro import *
initialize(ask("What port?"))

# program settings...

cruiseSpeed = 1.0
turnSpeed = 0.5     # this is a left turn, -0.5 will be right
turn

def main():
    while True:

        if getStall():
           # I am stalled, turn (randomly?)
           move(0, turnSpeed)
        else:
           # I am not stalled, cruise on
            move(cruiseSpeed, 0)

main()
```

At times, you may notice that the robot gets stuck even when trying to turn. One remedy for this is to stop the robot, back up a little, and then turn.

Do This: Implement the program above, observe the robot behavior. Next, modify the program as suggested above (when stalled stop, backup, then turn).

Maze Solver: Create a simple maze for your robot. Place the robot at one end of the maze and use the obstacle avoidance programs from above (both versions). Does you robot solve the maze? If not, note if your maze is right handed or left handed (i.e. every turn is a right turn or left turn in the maze), or both. Modify the obstacle avoidance programs to solve the right-handed, left-handed mazes. How would you enable the robot to solve a maze that has both right and left turns?

Corral Exiting



Given that a simple obstacle avoidance program can enable a robot to solve simple mazes, we can also design more interesting behaviors on top of that. Imagine a corral: an enclosed area with maze like partitions and an entrance, with a light source at the entrance (see picture on right). Given the robot's position, can we design a behavior that will enable the robot to exit the corral?

One can design a solution for the specific corral shown here: follow a wall (any wall) until it sees bright light then switch to light seeking. Can the Scribbler be designed to follow a wall? Remember the Fluke dongle has left and right obstacle sensors that are pointing to its sides. Another approach will be to combine the obstacle avoidance behavior from above with the light

seeker behavior. That is, in the absence of any bright light, the robot moves around the corral avoiding obstacles and when it sees a bright light, it heads towards it. The hard part here will be to detect that it has exited the corral and needs to stop.

Summary

Braitenberg uses very simple ideas to enable people to think about the way animal and human brains and bodies are wired. For example, in humans, the optic nerves (as do some others) have crossed connections inside the brain. That is, the nerves from the left eye are connected to the right side of the brain and vice versa. Actually they cross over and some information from either side is also represented on the same side (that is there are straight as well as crossed connections). However, it is still a puzzle among scientists as to why this is the case and what, if any, are the advantages or disadvantages of this scheme. Similarly, observing the behaviors of Vehicles 2a and 2b one can easily see in them parallels in the behavior of several animals, like flies orienting towards light/heat sources. Simple robot behaviors can provide deep insights into complex behavior: that the observation and analysis of something is an uphill task if one doesn't know the internal structure. And, by constructing simple internal structures one can arrive at seemingly complex behaviors. These seemingly complex behaviors have also been shown to influence group behavior is insects (see the picture of article on next page). That is, robots that do not look anything like insects, and not too different in size than the Scribbler, can be used to influence insect behavior in many situations.

In this chapter, we have attempted to give you a flavor for the idea of synthetic psychology. At the same time you have also learned how to program internal structures in a robot brain and learned several techniques for robot control.

Story from *Science* Magazine, January 10, 2008

BEHAVIOR

Robot Cockroach Tests Insect Decision-Making Behavior

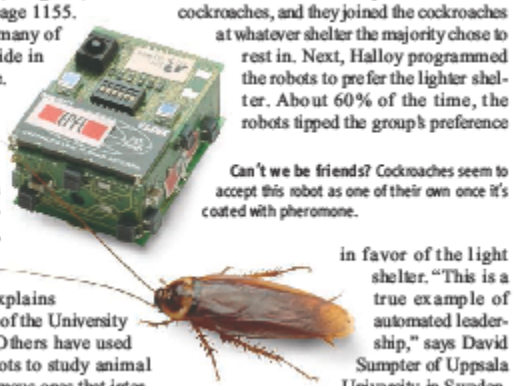
Science-fiction writers have long envisioned societies in which the boundaries between humans and lifelike droids blur and man and machine freely intermingle. José Halloy has taken the first steps toward creating that world, at least for insects. His tiny, autonomous robots lack legs, wings, and antennae, but they nonetheless pass muster with cockroaches. Indeed, these wheeled machines are so well accepted by the household pests that the robots become part of the insects' collective decision-making process, Halloy, a theoretical biologist at the Free University of Brussels, Belgium, and his colleagues report on page 1155. The robots persuaded many of their insect "peers" to hide in an unconventional place.

Halloy's innovative approach puts theories of collective behavior among insects into practice. "We can manipulate these behaviors very easily in a model, but doing so in experiments is often challenging," explains ethologist Jerome Buhl of the University of Sydney, Australia. Others have used remote-controlled robots to study animal

level of darkness of the shelter and the number and activity of its fellow roaches. Halloy's group then used this model to program robots designed by him and Francesco Mondada and other engineers at the École Polytechnique Fédérale de Lausanne, Switzerland.

The roaches usually ran away from the robots but not if the machines smelled like the insects. For the experiments, Halloy and Sempo covered the robots with a filter paper containing the pheromone equivalent of one cockroach.

Halloy initially programmed the robots to have the same darkness preference as the cockroaches, and they joined the cockroaches at whatever shelter the majority chose to rest in. Next, Halloy programmed the robots to prefer the lighter shelter. About 60% of the time, the robots tipped the group's preference



Can't we be friends? Cockroaches seem to accept this robot as one of their own once it's coated with pheromone.

in favor of the light shelter. "This is a true example of automated leadership," says David Sumpter of Uppsala

Background

All the numbered vehicles described here were developed in a set of thought experiments designed by Valentino Braitenberg in his book, Vehicles: Experiments in Synthetic Psychology, MIT Press, 1984.

Some of the other vehicles described here were designed by David Hogg, Fred Martin, and Mitchel Resnick of the MIT Media Laboratory. Hogg et al

used specialized electronic LEGO bricks to build these vehicles. For more details, see their paper titled, *Braitenberg Creatures*.

To read more about robots influencing insect behavior see the November 16, 2007 issue of *Science* magazine. The primary article that is discussed in the picture above is by Halloy *et al*, *Social Integration of Robots into Groups of Cockroaches to Control Self-Organized Choices, Science*, November 16, 2007. Volume 318, pp 1155-1158.

Myro Review

There were no new Myro features introduced in this chapter.

Python Review

The if-statement in Python has the following forms:

```
if <condition>:
   <this>

if <condition>:
   <this>
else:
   <that>

if <condition-1>:
   <this>
elif <condition-2>:
   <that>
elif <condition-3>:
   <something else>
...
...
else:
   <other>
```

The conditions can be any expression that results in a True, False, 1, or 0 value. Review Chapter 4 for details on writing conditional expressions.

Exercises

Exercise 1: An even better way of averaging the ambient light conditions for purposes of normalization is to have the robot sample ambient light all around it. That is, turn around a full circle and sample the different light sensor values. The ambient value can then be set to the average of all the light values. Write a function called, `setAmbient` that rotates the robot for a full circle (or you could use time), samples light sensor values as it rotates, and then returns the average of all light values. Change the line:

```
Ambient = sum(getLight())/3.0
```

to the line:

```
Ambient = setAmbient()
```

Try out all of the earlier behaviors described in this chapter to see how this new mechanism affects the robot's behavior.

Exercise 2: Design and implement a program that exhibits the corral exiting behavior described in this chapter.

7 Control Paradigms

What, no quote?

Writing programs is all about exercising control. In the case of a robot's brain your program directs the operations of a robot. However, it is also important to realize that the program itself is really controlling the computer. That is, when you write Python programs you are controlling the computer that is then communicating with the robot. Your program is directing the computer to control the robot. If you take Myro out of the picture you are writing programs to control the computer. This is the sense in which learning with robots also leads to learning computing. Every program you write is doing computation. Contrary to popular misconceptions computing is not just about doing calculations with numbers. Controlling your robot is also computing, as is predicting the world's population, or composing an image, etc. This is one aspect of control.

When writing robot control programs, the structure you use to organize the program itself is a control strategy. Programming a robot is specifying automated control. As a programmer or behavior designer you structure your program to accomplish the goals of the behavior: how the sensors are used to decide what to do next. This is another aspect of control. So far, you have seen how to write control programs using Braitenberg style sensor-motor *wiring*. You have also seen how to specify reactive control. These are examples of two robot control paradigms.

In this chapter we delve further into the world of computation and robot control paradigms. We will learn how to write robot control programs for more complex and more robust robot tasks. We will also see how, using the concepts learned so far, we can write useful and interesting computer applications.

Behavior-based Control

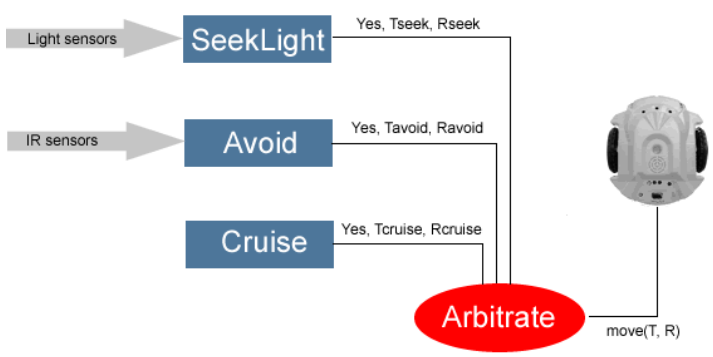
When writing robot control programs, so far, you have used a very basic technique in designing control programs:

```
def main():
    # do forever or for some time
    # or until a certain condition is satisfied

       # sense and transform sensor values
       # reason or decide what to do next
       # do it
```

As you have seen, such a control program works well for many simple tasks. However, you may have already run into many situations where, once the task gets a little complex, it becomes difficult to structure a program in terms of a single stream of control as shown above. For example, the corral exiting behavior from the last chapter requires you to combine two simple behaviors: solve a maze (avoid obstacles) and seek light to get out of the corral. As you have seen before, it is fairly easy to program each of the individual behaviors: obstacle avoidance; light following. But, when you combine these behaviors to accomplish the corral exiting behavior two things happen: you are forced to amalgamate the two control strategies into a single one and it may become difficult to decide which way to combine them; additionally, the resulting program is not very pretty and hard to read. In reality, hardly any robot programs are written that way. In this section, we will look at a different way of structuring robot programs that makes designing behaviors easy, and yet, the resulting structure of the overall program is also clean and straightforward. You can design some very sophisticated behaviors using these ideas.

People in the robotics community call the style of programming shown above as *reactive control* or *direct control*. Also referred to as *sensor fusion*, the resulting programs are purely sensor driven and hence appear to be too bottom-up. That is, the values of the sensors drive the logic of control as opposed to the goals of the robot tasks themselves. In *behavior-based control* you get away from sensors and focus the design of your robot programs based on the number and kinds of behaviors your robot has to carry out.

Let us look at how behavior-based control enables us to design the corral exiting behavior. The robot essentially has to carry out three kinds of behaviors: cruise (in the absence of any obstacles and/or light), avoid obstacles (if present), and seek light (if present). In a behavior-based style of writing programs, you will define each of these behaviors as an individual decision unit. Thus, each is quite simple and straightforward to write. Next, the control program has to fuse the behaviors recommended by each individual behavior unit. Look at the picture shown below:



In the diagram above, we have shown the three basic behaviors that we are trying to combine: `Cruise`, `Avoid`, `SeekLight`. Each of these behaviors outputs a triple: `Yes`/`No`, Translate Speed, Rotate Speed. A `Yes` implies that the behavior module has a recommendation. `No` implies that it doesn't. That is, it

allows the possibility of a behavior having no recommendation. For example, in the corral exiting situation, in the absence of a light source being sensed by the robot, the `SeekLight` module will not have any recommendation. It then becomes the task of the arbitrator (or the decision module) to decide which of the available recommendations to use to drive the robot. Notice that in the end to control the robot, all one has to do is decide how much to translate and rotate. Many different arbitration schemes can be incorporated. We will use a simple but effective one: Assign a priority to each behavior module. Then the arbitrator always chooses the highest priority recommendation. This style of control architecture is also called *subsumption architecture*. In the figure above, we have actually drawn the modules in the order of their priority: the higher the module is in the figure, the higher its priority. The lowest behavior, cruise does not require any sensors and is always present: it wants the robot to always go forward.

Arranging a control based on combining simple behaviors has several advantages: you can design each individual behavior very easily; you can also test each individual behavior but only adding that behavior and seeing how well it performs; you can incrementally add any number of behaviors on top of each other. In the scheme above, the control regime implies that the robot will always cruise forward. But, if there is an obstacle present, it will override the cruise behavior and hence try to avoid the obstacle. However, if there is a light source detected, it will supersede all behaviors and engage in light seeking behavior. Ideally, you can imagine that all the behaviors will be running simultaneously (asynchronously). In that situation, the arbitrator will always have one or more recommendations to adopt based on priority.

Let us develop the program that implements behavior-based control. First, we define each behavior:

```
cruiseSpeed = 0.8
turnSpeed = 0.8
lightThresh = 80

def cruise():
   # is always ON, just move forward
   return [True, cruiseSpeed, 0]
```

```
def avoid():
   # see if there are any obstacles
   L, R = getIR()
   L = 1 - L
   R = 1 - R

   if L:
      return [True, 0, -turnSpeed]
   elif R:
      return [True, 0, turnSpeed]
   else:
      return [False, 0, 0]

def seekLight():
   L, C, R = getLight()

   if L < lightThresh:
      return True, cruiseSpeed/2.0, turnSpeed
   elif R < lightThresh:
      return True, cruiseSpeed/2.0, -turnSpeed
   else:
      return [False, 0, 0]
```

In the above, you can see that each individual behavior is simple and is easy to read (and write). There are several ways to incorporate these into a behavior-based program. Here is one:

```
# list of behaviors, ordered by priority (left is highest)
behaviors = [seekLight, avoid, cruise]

def main():

   while True:
      T, R = arbitrate()
      move(T, R)

main()
```

The main program calls the arbitrate function that returns the chosen translate and rotate commands which are then applied to the robot. The function `arbitrate` is simple enough:

```
# Decide which behavior, in order of priority
# has a recommendation for the robot
def arbitrate():

   for behavior in behaviors:
      output, T, R = behavior()
      if output:
         return [T, R]
```

That is, it queries each behavior in order of priority to see if it has a recommendation. If it does, that is the set of motor commands returned.

Do This: Implement the program above and see how well the robot behaves in navigating around and exiting the corral. What happens if you change the priority (ordering in the list) of behaviors? In writing the program above, we have used two new Python features. We will review these next.

Names and Return values

In Chapter 3 you learned that in Python names can be used to represent functions as well as numbers and strings. You also saw in Chapter 5 that lists, and even pictures or images could be represented using names. A name is an important programming concept. In Python a name can represent anything as its value: a number, a picture, a function, etc. In the program above, when we defined the variable behaviors as:

```
behaviors = [seekLight, avoid, cruise]
```

we used the names `seekLight`, `avoid`, and `cruise` to denote the functions that they represented. We named these functions earlier in the program (using `def`). Thus, the list named behaviors is a list of function names each of which denote the actual function as its value. Next, look at the way we used the variable `behaviors` in the function arbitrate:

```
for behavior in behaviors:
      output, T, R = behavior()
      …
```

Since `behaviors` is a list it can be used in the loop to represent the sequence of objects (in this case functions). Thus, in each iteration of the loop, the variable `behavior` takes on successive values from this list: `seekLight`, `avoid`, and `cruise`. When the value of the variable is `seekLight`, the function `seekLight` is called in the statement:

```
output, T, R = behavior()
```

There is no function named `behavior()` defined anywhere in the program. However, since the value of the variable name `behavior` is set to one of the functions in the list `behaviors`, the corresponding function is invoked.

The other new aspect of Python that we have used in the program above is the fact that a function can return any object as its value. Thus the functions `cruise`, `avoid`, `seekLight`, and `arbitrate` all return lists as their values.

Both of these are subtle features of Python. Many other programming languages have restrictions on the kinds of things one can name as variables and also on the kinds of values a function can return. Python gives uniform treatment to all objects.

The Python Math Library

Most programming languages, Python included, provide a healthy selection of libraries of useful functions so you do not have to write them. Such libraries are often termed as an *application programming interface* or an *API*. Earlier you were introduced to the `random` library provided by Python. It contains several useful functions that provide facilities for generating random numbers. Similarly, Python also provides a `math` library that provides often used mathematical functions. In Chapter 6, we used the function `exp` from the `math`

library to normalize sensor values. Some of the commonly used functions in the `math` library are listed below:

Commonly used functions in the `math` module

`ceil(x)` Returns the ceiling of x as a float, the smallest integer value greater than or equal to x.

`floor(x)` Returns the floor of x as a float, the largest integer value less than or equal to x.

`exp(x)` Returns e**x.

`log(x[, base])` Returns the logarithm of x to the given base. If the base is not specified, return the natural logarithm of x (i.e., the logarithm to base e).

`log10(x)` Returns the base-10 logarithm of x.

`pow(x, y)` Returns x**y.

`sqrt(x)` Returns the square root of x.

Some of the other functions available in the `math` module are listed at the end of the chapter. In order to use any of these all you have to do is import the `math` module:

```
import math

>>> math.ceil(5.34)
6.0
>>> math.floor(5.34)
5.0

>>> math.exp(3)
20.085536923187668

>>> math.log10(1000)
```

```
3.0
>>> math.log(1024, 2)
10.0
>>> math.pow(2, 10)
1024.0

>>> math.sqrt(7.0)
2.6457513110645907
```

Alternately, you can import the `math` module as shown below:

```
>>> from math import *
>>> ceil(5.34)
6.0
>>> floor(5.34)
5.0
>>> exp(3)
20.085536923187668
>>> log10(1000)
3.0
>>> log(1024,2)
10.0
>>> sqrt(7.0)
2.6457513110645907
```

In Python, when you import a module using the command:

```
import <module>
```

You have to prefix all its commands with the module name (as the case in the first set of examples above). We have also been using the form

```
from <module> import *
```

This imports all functions and other objects provided in the module which can then be used without the prefix. You can also individually import specific functions from a module using:

```
from <module> import <this>, <that>, …
```

Which version you use may depend on the context in which you use the imported functions. You may run into a situation where two different modules define functions with the same name but to do different things (or to do things in a different way). In order to use both functions in the same module you will have to use the module prefix to make clear which version you are using.

Doing Computations

Lets us weave our way back to traditional style computing for now. You will see that the concepts you have learned so far will enable you to write lots of different and more interesting computer applications. It will also give you a clear sense of the structure of typical computer programs. Later, in Chapter 10, we will also return to the larger issue of the design of general computer programs.

A Loan Calculator

Your current car, an adorable 1992 SAAB 93 was bought used and, for past several months, you have had nothing but trouble keeping the car on the road. Last night the ignition key broke off in the key slot when you were trying to start it and now the broken piece would not come out (this used to happen a lot with older SAAB's). The mechanic has to dismantle the entire ignition assembly to get the broken key out and it could cost you upwards of $500. The car's engine, which has done over 185,000 miles, has left you stranded on the road many times. You have decided that this is it, you are going to go out and get yourself a brand new reliable car. You have been moonlighting at a restaurant to make extra money and have managed to save $5500.00 for exactly this situation. You are now wondering what kind of new car you can buy. Obviously, you will have to take a loan from a bank to finance the rest of the cost but you are not sure how big a loan, and therefore what kind of car, you can afford. You can write a small Python program to help you try out various scenarios.

You can either dream (realistically) of the car you would like to buy, or you can go to any of the helpful car buying sites on the web (www.edmunds.com

is a good place to start). Let us say that you have spent hours looking at features and options and have finally narrowed your desires to a couple of choices. Your first choice is going to cost you $22,000.00 and your second choice is priced at $18,995.00. Now you have to decide which of these you can actually afford to purchase.

First, you go talk to a couple of banks and also look at some loan offers on the web. For example, go to bankrate.com and look for current rates for new car loans.

Suppose the loan rates quoted to you are: 6.9% for 36 months, 7.25% for 48 months, and 7.15 for 60 months.

You can see that there is a fair bit of variation in the rates. Given all this information, you are now ready to write a program that can help you figure out which of the two choices you may be able to make. In order to secure the loan, you have to ensure that you have enough money to pay the local sales tax (a 6% sales tax on a $20,000 car will add up to a hefty $1200!). After paying the sales tax you can use the remainder of the money you have saved up towards the down payment. The remainder of the money is the amount that you would borrow. Depending on the type of loan you choose, your monthly payments and how long you will make those payments will vary. There is a simple formula that you can use to estimate your monthly payment:

$$MonthlyPayment = \frac{LoanAmount * MonthlyInterestRate}{1 - e^{-LoanTerm * \log(1 + MonthlyInterestRate)}}$$

Whoa! That seems complicated. However, given the formula, you can see that it really requires two mathematical functions: $\log(x)$ and e^x, both of which are available in the Python `math` module. Suddenly, the problem seems not too hard.

Let us try and outline the steps needed to write the program: First, note the cost of the car, the amount of money you have saved, and the sales tax rate Also, note the financials: the interest rate, and the term of the loan. The

interest rate quoted is generally the annual percentage rate (APR) convert it to monthly rate (by dividing it by 12) Next, compute the sales tax you will pay Use the money left to make a down payment Then determine the amount you will borrow Plug in all of the values in the formula and compute the monthly payment Also, compute the total cost of the car. Output all the results Next, we can take each of the above steps and start to encode them into a program. Here is a first order refinement:

```
def main():
   # First, note the cost of the car (Cost),
   # the amount of money you have saved (Cash),
   # and the sales tax rate (TaxRate)

   # Also, note the financials: the interest rate (APR),
   # and the term of the loan (Term)
   # The interest rate quoted is generally the annual
   # percentage rate (APR)
   # Convert it to monthly rate (by dividing it by 12) (MR)

   # Next, compute the sales tax you will pay (SalesTax)
   # Use the money left to make a down payment (DownPayment)
   # Then determine the amount you will borrow (LoanAmount)

   # Plug in all of the values in the formula and compute
   # the monthly payment (MP)

   # Also, compute the total cost of the car. (TotalCost)

   # Output all the results

main()
```

Above, we have taken the steps and converted them into a skeletal Python program. All the steps are converted to Python comments and where needed, we have decided the names of variables that will hold the values that will be needed for the calculations. This is useful because this also helps determine how the formula will be encoded and also helps determine what values can be programmed in and which ones you will have to supply as input. Making the program require inputs will easily enable you to enter the different parameters

and then based on the outputs you get, you can decide which car to buy. Let us encode all the inputs first:

```
def main():
    # First, note the cost of the car (Cost),
    Cost = input("Enter the cost of the car: $")

    # the amount of money you have saved (Cash),
    Cash = input("Enter the amount of money you saved: $")

    # and the sales tax rate (TaxRate) (6% e.g.)
    SalesTaxRate = 6.0

    # Also, note the financials: the interest rate (APR),
    # and the term of the loan (Term)
    # The interest rate quoted is generally the annual
    # percentage rate (APR)
    APR = input("Enter the APR for the loan (in %): ")

    # Convert it to monthly rate (by dividing it by 12) (MR)

    # Next, compute the sales tax you will pay (SalesTax)
    # Use the money left to make a down payment (DownPayment)
    # Then determine the amount you will borrow (LoanAmount)

    # Plug in all of the values in the formula and compute
    # the monthly payment (MP)

    # Also, compute the total cost of the car. (TotalCost)

    # Output all the results

main()
```

We have refined the program to include the inputs that will be needed for each run of the program. Notice that we chose not to input the sales tax rate and instead just assigned it to the variable `SalesTaxRate`. If you wanted, you could also have that be entered as input. What you choose to have as input to your program is your design decision. Sometimes the problem may be framed so it explicitly specifies the inputs; sometimes you have to figure that out. In general, whatever you need to make your program more versatile is what you

have to base your decisions on. For instance, fixing the sales tax rate to 6.0 will make the program usable only in places where that rate applies. If, for example, you wanted your friend in another part of the country to use the program, you should choose to make that also an input value. Let us go on to the next steps in the program and encode them in Python. These are mainly computations. The first few are simple. Perhaps the most complicated computation to encode is the formula for computing the monthly payment. All of these are shown in the version below.

```
From math import *

def main():
    # First, note the cost of the car (Cost),
    Cost = input("Enter the cost of the car: $")

    # the amount of money you have saved (Cash),
    Cash = input("Enter the amount of money you saved: $")

    # and the sales tax rate (TaxRate) (6% e.g.)
    SalesTaxRate = 6.0

    # Also, note the financials: the interest rate (APR),
    # and the term of the loan (Term)
    # The interest rate quoted is generally the annual
    # percentage rate (APR)
    APR = input("Enter the APR for the loan (in %): ")

    # Input the term of the loan (Term)
    term = input("Enter length of loan term (in months): ")

    # Convert it (APR) to monthly rate (divide it by 12) (MR)
    # also divide it by 100 since the value input is in %
    MR = APR/12.0/100.0

    # Next, compute the sales tax you will pay (SalesTax)
    SalesTax = Cost * SalesTaxRate / 100.0

    # Use the money left to make a down payment (DownPayment)
    DownPayment = Cash - SalesTax

    # Then determine the amount you will borrow (LoanAmount)
```

```
    LoanAmount = Cost - DownPayment

    # Plug in all of the values in the formula and compute
    # the monthly payment (MP)
    MR = (LoanAmount * MR) / (1.0 - exp(-term * log(1.0+MR)))

    # Also, compute the total cost of the car. (TotalCost)
    TotalCost = SalesTax + DownPayment + MR * term

    # Output all the results
    print "Here are the details about your new car..."
    print "----------------------------------------"
    print
    print "Money you have saved $", Cash
    print "Cost of the car $", Cost
    print "Sales Tax rate is", SalesTaxRate, "%"
    print "Sales Tax on the car $", SalesTax
    print "Your down payment will be $", DownPayment
    print "You will be borrowing $", LoanAmount
    print "A", term, "month loan at", APR, "% APR"
    print "Your monthly payment will be $", MP
    print "Total cost will be $", TotalCost
    print

main()
```

Do This: When you enter the above program and run it in Python, you can enter the data about your car. Here is a sample run:

```
Enter the cost of the car: $20000.00
Enter the amount of money you saved: $5500.00
Enter the APR for the loan (in %): 6.9
Enter the length of loan term (in months): 36
Here are the details about your new car...
----------------------------------------

Money you have saved $ 5500.0
Cost of the car $ 20000.0
Sales Tax rate is 6.0 %
Sales Tax on the car $ 1200.0
Your down payment will be $ 4300.0
You will be borrowing $ 15700.0
```

```
A 36 month loan at 6.9 % APR
Your monthly payment will be $ 484.052914723
Total cost will be $ 22925.90493
```

It appears that at for the $20000.00 car, for a 36 month 6.9% loan you will end up paying $484.05 (or $484.06 depending upon how your loan company round pennies!).

When you need to restrict your output values to specific decimal places (two in the case of dollars and cents, for example), you can use the string formatting features built into Python. For example, in a string, you can specify how to include a floating point value as follows:

```
"Value of PI %5.3f in 3 places after the decimal" % (math.pi)
```

The above is a Python expression that has the syntax:

```
<string> % <expression>
```

Inside the `<string>` there is a format specification beginning with a `%`-sign and ending in an `f`. What follows the `%`-sign is a numerical specification of the value to be inserted at that point in the string. The `f` in the specification refers to the fact it is for a floating point value. Between the `%`-sign and the `f` is a number, 5.3. This specifies that the floating point number to be inserted will take up at least 5 spaces, 3 of which will be after the decimal. One of the spaces is always occupied by the decimal. Thus, the specification `%5.3f` specifies a value to be inserted in the following manner:

```
"This is the value of PI -.--- expressed in three places after
the decimal"
```

You can see the result of executing this in Python below:

```
>>> "Value of PI %5.3f in 3 places after the decimal" %
(math.pi)
'Value of PI 3.142 in 3 places after the decimal'
```

```
>>> "Value of PI %7.4f in 4 places after the decimal" %
(math.pi)
'Value of PI  3.1416 in 4 places after the decimal'
```

In the second example above, we replaced the specification with `%7.4f`. Notice that the resulting string allocates seven spaces to print that value. If there are more spaces than needed they get padded by blanks on the leading edge (notice the extra space before 3 in the second example above). If the space specified is less, it will always be expanded to accommodate the number. For example:

```
>>> "Value of PI %1.4f in 4 places after the decimal" %
(math.pi)
'Value of PI 3.1416 in 4 places after the decimal'
```

We deliberately specified that the value be 1 space wide with 4 spaces after the decimal (i.e. `%1.4f`). As you can see, the space was expanded to accommodate the value. What is assured is that the value is always printed using the exact number of spaces after the decimal. Here is another example:

```
>>> "5 is also %1.3f with 3 places after the decimal." % 5
'5 is also 5.000 with 3 places after the decimal.'
```

Thus, the value is printed as 5.000 (i.e. the three places after the decimal are always considered relevant in a specification like `%1.3f)`. Similarly, for specifying whole number or integer values you can use the letter-`d`, and for strings you can use the letter-`s`:

```
>>> "Hello %10s, how are you?" % "Arnold"
'Hello     Arnold, how are you?'
```

By default, longer specifications are right-justified. You can use a `%-` specification to left-justify. For example:

```
>>> "Hello %-10s, how are you?" % "Arnold"
'Hello Arnold    , how are you?'
```

Having such control over printed values is important when you are trying to output tables of aligned values. Let us modify our program from above to use these formatting features:

```
From math import *

def main():
    # First, note the cost of the car (Cost),
    Cost = input("Enter the cost of the car: $")

    # the amount of money you have saved (Cash),
    Cash = input("Enter the amount of money you saved: $")

    # and the sales tax rate (TaxRate) (6% e.g.)
    SalesTaxRate = 6.0

    # Also, note the financials: the interest rate (APR),
    # and the term of the loan (Term)
    # The interest rate quoted is generally the annual
    # percentage rate (APR)
    APR = input("Enter the APR for the loan (in %): ")

    # and the term of the loan (Term)
    term = input("Enter length of loan term (in months): ")

    # Convert it (APR) to monthly rate (divide it by 12) (MR)
    # also divide it by 100 since the value input is in %
    MR = APR/12.0/100.0

    # Next, compute the sales tax you will pay (SalesTax)
    SalesTax = Cost * SalesTaxRate / 100.0

    # Use the money left to make a down payment (DownPayment)
    DownPayment = Cash - SalesTax

    # Then determine the amount you will borrow (LoanAmount)
    LoanAmount = Cost - DownPayment

    # Plug in all of the values in the formula and compute
    # the monthly payment (MP)
    MP = (LoanAmount * MR) / (1.0 - exp(-term * log(1.0+MR)))
```

```
    # Also, compute the total cost of the car. (TotalCost)
    TotalCost = SalesTax + DownPayment + MP * term

    # Output all the results
    print "Here are the details about your new car..."
    print "-----------------------------------------"
    print
    print "Money you have saved $%1.2f" % Cash
    print "Cost of the car $%1.2f" % Cost
    print "Sales Tax rate is %1.2f" % SalesTaxRate
    print "Sales Tax on the car $", SalesTax, "%"
    print "Your down payment will be $%1.2f" % DownPayment
    print "You will be borrowing $%1.2f" % LoanAmount
    print "A %2d month loan at %1.2f APR" % (term, APR)
    print "Your monthly payment will be $%1.2f" % MP
    print "Total cost will be $%1.2f" % TotalCost
    print

main()
```

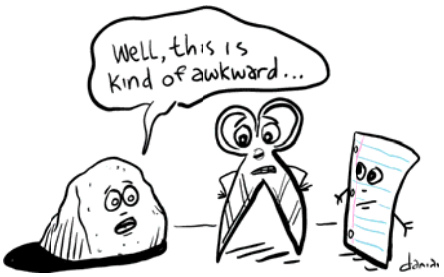
When you run it again (say for a slightly different loan term), you get:

```
Enter the cost of the car: $20000.00
Enter the amount of money you saved: $5500.00
Enter the APR for the loan (in %): 7.25
Enter the length of loan term (in months): 48
Here are the details about your new car...
-----------------------------------------

Money you have saved $5500.00
Cost of the car $20000.00
Sales Tax rate is 6.00
Sales Tax on the car $ 1200.0 %
Your down payment will be $4300.00
You will be borrowing $15700.00
A 48 month loan at 7.25 APR
Your monthly payment will be $377.78
Total cost will be $23633.43
```

You can see that for the same amount if you borrow it for a longer period you can reduce your monthly payments by over $100 (but you pay about $700 more in the end).

Decision Making in Computer Programs

Decision making is central to all computer programs. In fact there is a famous theorem in computer science that says that if any programmable device is capable of being programmed to do *sequential execution*, *decision making*, and *repetition*, it is capable of expressing any computable algorithm. This is a very powerful idea that is couched in terms of very simple ideas. Given a problem, if you can describe its solution in terms of a combination of the three execution models then you can get any computer to solve that problem. In this section, we will look at decision making in a classic game playing situation.



The exact name of the game can vary, with the three components appearing in a different order, or with "stone" in place of "rock". Non-English speakers may know the game by their local words for "rock, paper, scissors", although it is also known as ***Jankenpon*** in Japan, ***Rochambeau*** in France, and in South Africa as ***Ching-Chong-Cha***...
From:talkingtotan.wordpress.com/2007/08/

Computers have been extensively used in game playing: for playing games against people and other computers. The impact of computers on game playing is so tremendous that these days several manufacturers make game playing consoles with computers in them that are completely dedicated to game playing.

Let us design a simple game that you have most likely played while growing up: Paper, Scissors, Rock!

In this game, two players play against each other. At the count of three each player makes a hand gesture indicating that they have selected one of the three items: paper (the hand is held out flat), scissors (two fingers are extended to designate scissors, or rock (indicated by making a fist). The rules of deciding

the winner are simple: if both players pick the same object it is a draw; otherwise, paper beats rock, scissors beat paper, and rock beats scissors. Let us write a computer program to pay this game against the computer. Here is an outline for playing this game once:

```
# Computer and player make a selection
# Decide outcome of selections (draw or who won)
# Inform player of decision
```

If we represent the three items in a list, we can have the computer pick one of them at random by using the random number generation facilities provided in Python. If the items are represented as:

```
items = ["Paper", "Scissors", "Rock"]
```

Then we can select any of the items above as the computer's choice using the expression:

```
# Computer makes a selection
myChoice = items[<0 or 1 or 2 selected randomly>]
```

That is `items[0]` represents the choice `"Paper"`, `items[1]` represents `"Scissors"`, and `items[2]` is `"Rock"`. We saw, in Chapter 4, that we can generate a random number in any range using the `randrange` function from the `random` library module in Python. Thus we can model the computer making a random selection using the following statement:

```
from random import *

# Computer makes a selection
myChoice = items[randrange(0, 3)]
```

Recall that `randrange(n, m)` returns a random numbers in the range `[n..m-1]`. Thus, `randrange(0, 3)` will return either 0, 1, or 2. We can use the `askQuestion` command in Pyro to ask the player to indicate their selection (see Chapter 3).

```
# Player makes a selection
yourChoice = askQuestion("Pick an item by clicking on it.",
items)
```

Now that we know how to the computer and player make their selection, we need to think about deciding the outcome. Here is an outline:

```
if both picked the same item then it is a draw
if computer wins then inform the player
if player wins then inform the player
```

Rewriting the above in Python using `if`-statements we get the following first draft:

```
if myChoice == yourChoice:
    print "It is a draw."

if <myChoice beats yourChoice>:
    print "I win."
else:
    print "You win."
```

All we need to do is figure out how to write the condition `<myChoice beats yourChoice>`. The condition has to capture the rules of the game mentioned above. We can encode all the rules in a conditional expression as follows:

```
   if (myChoice == "Paper" and yourChoice == "Rock")
       or (myChoice == "Scissors" and yourChoice == "Paper")
       or (myChoice == "Rock" and yourChoice == "Scissors"):
      print "I win."
   else:
      print "You win."
```

The conditional expression above captures all of the possibilities that should be examined in order to make the decision. Another way of writing the above decision would be to use the following:

```
   if myChoice == "Paper" and yourChoice == "Rock":
      print "I win."
```

```
   elif myChoice == "Scissors" and yourChoice == "Paper":
      print "I win."
   elif myChoice == "Rock" and yourChoice == "Scissors":
      print "I win."
   else:
      print "You win."
```

That is each condition is examined in turn until one is found that confirms that the computer wins. If none such condition is true, the `else`-part of the `if`-statement will be reached to indicate that the player won.

Another alternative to writing the decision above is to encode the decision in a function. Let us assume that there is a function `beats` that returns `True` or `False` depending on the choices. We could then rewrite the above as follows:

```
if myChoice == yourChoice:
    print "It is a draw."
if beats(myChoice, yourChoice):
    print "I win."
else:
    print "You win."
```

Let us take a closer look at how we could define the `beats` function. It needs to return `True` if `myChoice` beats `yourChoice`. So all we need to do is encode the rules of the game described above. Here is a draft of the function:

```
def beats(me, you):
   # Does me beat you? If so, return True, False otherwise.

   if me == "Paper" and you == "Rock":
      # Paper beats rock
      return True
   elif me == "Scissors" and you == "Paper":
      # Scissors beat paper
      return True
   elif me == "Rock" and you == "Scissors":
      # Rock beats scissors
      return True
   else:
      return False
```

Once again, we have used the `if`-statements in Python to encode the rules of the game. Now that we have completely fleshed out all the critical parts of the program, we can put them all together as shown below:

```
# A program that plays the game of Paper, Scissors, Rock!

from myro import *
from random import randrange

def beats(me, you):
   # Does me beat you? If so, return True, False otherwise.

   if me == "Paper" and you == "Rock":
      # Paper beats rock
      return True
   elif me == "Scissors" and you == "Paper":
      # Scissors beat paper
      return True
   elif me == "Rock" and you == "Scissors":
      # Rock beats scissors
      return True
   else:
      return False

def main():
    # Play a round of Paper, Scissors, Rock!
    print "Lets play Paper, Scissors, Rock!"
    print "In the window that pops up, make your selection>"

    items = ["Paper", "Scissors", "Rock"]

    # Computer and Player make their selection...
    # Computer makes a selection
    myChoice = items[randrange(0, 3)]

    # Player makes a selection
    yourChoice = askQuestion("Pick an item by clicking on
it.", items)

    # inform Player of choices
    print
    print "I picked", myChoice
```

```
    print "You picked", yourChoice

    # Decide if it is a draw or a win for someone
    if myChoice == yourChoice:
        print "We both picked the same thing."
        print "It is a draw."
    elif beats(myChoice, yourChoice):
        print "Since", myChoice, "beats", yourChoice, "..."
        print "I win."
    else:
        print "Since", yourChoice, "beats", myChoice, "..."
        print "You win."

    print "Thank you for playing. Bye!"

main()
```

A few more print commands were added to make the interaction more natural.

Do This: Implement the Paper, Scissors, Rock program from above and play it several times to make sure you understand it completely. Modify the program above to play several rounds. Also, incorporate a scoring system that keeps track of the number of times each player won and also the number of draws.

Summary

In this chapter you have seen a variety of control paradigms: behavior-based control for robots, writing a simple, yet useful computational program, and writing a simple computer game. Behavior-based control is a powerful and effective way of describing sophisticated robot control programs. This is the paradigm used in many commercial robot applications. You should try out some of the exercises suggested below to get comfortable with this control technique. The other two programs illustrate how, using the concepts you have learned, you can design other useful computer applications. In the next several chapters, we will explore several other dimensions of computing: media computation, writing games, etc.

Myro Review

There was nothing new from Myro in this chapter.

Python Review

The math library module provides several useful mathematics functions. Some of the commonly used functions are listed below:

`ceil(x)` Returns the ceiling of x as a float, the smallest integer value greater than or equal to x.

`floor(x)` Returns the floor of x as a float, the largest integer value less than or equal to x.

`exp(x)` Returns e**x.

`log(x[, base])` Returns the logarithm of x to the given base. If the base is not specified, return the natural logarithm of x (i.e., the logarithm to base e).

`log10(x)` Returns the base-10 logarithm of x.

`pow(x, y)` Returns x**y.

`sqrt(x)` Returns the square root of x.

Trigonometric functions

`acos(x)` Returns the arc cosine of x, in radians.

`asin(x)` Returns the arc sine of x, in radians.

`atan(x)` Returns the arc tangent of x, in radians.

`cos(x)` Returns the cosine of x radians.

`sin(x)` Returns the sine of x radians.

`tan(x)` Returns the tangent of x radians.

`degrees(x)` Converts angle x from radians to degrees.

`radians(x)` Converts angle x from degrees to radians.

The module also defines two mathematical constants:

`pi` The mathematical constant pi.

`e` The mathematical constant e.

Exercises

8 Sights & Sounds

If a tree falls in the forest and there is no one to hear it, will it make a sound?
-: Old philosophical question.

If a tree falls in the forest and there is only a robot around to hear it, does that count?
-: New philosophical question.

Don't make music for some vast, unseen audience or market or ratings share or even for something as tangible as money. Though it's crucial to make a living, that shouldn't be your inspiration. Do it for yourself.
-Billy Joel

We mentioned earlier that the notion of computation these days extends far beyond simple numerical calculations. Writing robot control programs is computation, as is making world population projections. Using devices like iPods you are able to enjoy music, videos, and radio and television shows. Manipulating sounds and images is also the realm of computation and in this

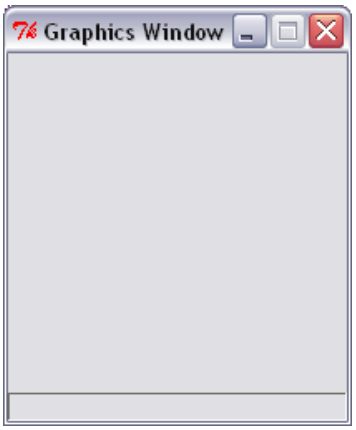
chapter we will introduce you to these. You have already seen how, using your robot, you can take pictures of various scenes. You can also take similar images from your digital camera. Using basic computational techniques you have learned so far you will see, in this chapter, how you can do computation on images. You will also see how you can create images using computation. The images you create can be used for visualizing data or even for aesthetic purposes to explore your creative side. We will also present some fundamentals of sound and music and show you how you can also do similar forms of computation using music.

Sights: Drawing

If you have used a computer for any amount of time you must have used some kind of a drawing application. Using a typical drawing application you can draw various shapes, color them etc. You can also generate drawings using drawing commands provided in the Myro library module. In order to draw anything you first need a place to draw it: a canvas or a window. You can create such a window using the command:

```
myCanvas = GraphWin()
```

A Graphics window

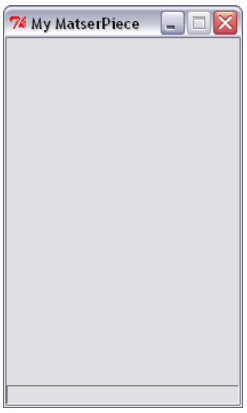


If you entered the command above in IDLE, you will immediately see a small gray window pop up (see picture on right). This window will be serving as our canvas for creating drawings. By default, the window created by the GraphWin command is 200 pixels high and 200 pixels wide and its name is “Graphics Window”. Not a very inspiring way to start, but the GraphWin command has some other variations as well. First, in order to make the window go away, you can use the command:

```
myCanvas.close()
```

To create a graphics window of any size and a name that you specify, you can use the command below:

My Masterpiece 200x300




```
myCanvas = GraphWin("My Masterpiece",
200, 300)
```

The command above creates a window named `"My Masterpiece"` that will be 200 pixels wide and 300 pixels tall (see picture on right). You can change the background color of a graphics window as shown below:

```
myCanvas.setBackground("white")
```

You can name any of a number of colors in the command above, ranging from mundane ones like `"red"`, `"blue"`, `"gray"`, `"yellow"`, to more exotic colors ranging from `"AntiqueWhite"` to `"LavenderBlush"` to `"WhiteSmoke"`. Colors can be created in many ways as we will see below. Several thousand color names have been pre-assigned (Google: color names list) that can be used in the command above.

Now that you know how to create a canvas (and make it go away) and even set a background color, it is time to look at what we can draw in it. You can create and draw all kinds of geometrical objects: points, lines, circles, rectangle, and even text. Depending on the type of object you wish to draw, you have to first *create* it and then draw it. Before you can do this though, you should also know the coordinate system of the graphics window.

In a graphics window with width, W and height H (i.e WxH pixels) the pixel (0, 0) is at the top right corner and the pixel (199, 299) will be at the bottom right corner. That is, x-coordinates increase as you go right and y-coordinates increase as you go left.

The simplest object that you can create is a *point*. This is done as follows:

```
p = Point(100, 50)
```

That is, p is an object that is a Point whose x-coordinate is at 100 and y-coordinate is at 50. So far, you have created a Point object. In order to draw it, you have to issue the command:

```
p.draw(myCanvas)
```

The syntax of the above command may seem a little strange at first. Certainly it is different from what you have seen so far. But if you think about the objects you have seen so far: numbers, strings, etc. Most of them have standard operations defined on them (like +, *, /, etc.). But when you think about geometrical objects, there is no standard notation. Programming languages like Python provide facilities for modeling any kind of object and the syntax we are using here is standard syntax that can be applied to all kinds of objects. The general form of commands issued on objects is:

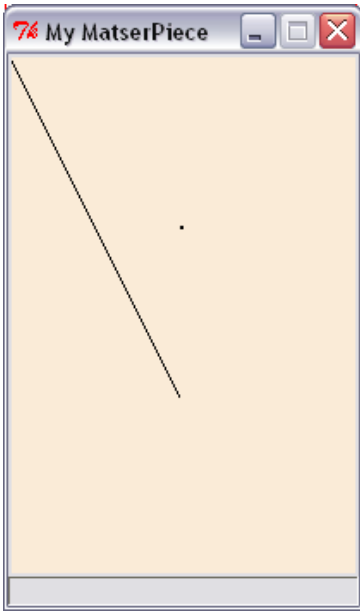
```
<object>.<function>(<parameters>)
```

Thus, in the example above, <object> is the name p which was earlier defined to be a Point object, <function> is draw, and <parameters> is myCanvas. The draw function requires the graphics window as the parameter. That is, you are asking the point represented by p to be drawn in the window specified as its parameter. The Point objects have other functions available:

```
>>> p.getX()
100
>>> p.getY()
50
```

That is, given a Point object, you can get its x- and y-coordinates. Objects are created using their *constructors* like the Point(x, y) constructor above. We will use lots of constructors in this section for creating the graphics objects. A line object can be created similar to point objects. A line requires the two end points to be specified. Thus a line from (0, 0) to (100, 200) can be created as:

```
L = Line(Point(0,0), Point(100,200))
```



And you can draw the line using the same draw command as above:

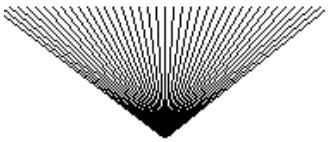
```
L.draw(myCanvas)
```

The picture on the right shows the two objects we have created and drawn so far. Like a Point, you can get the values of a line's end points:

```
>>> start = L.getP1()
>>> start.getX
0
>>> end = L.getP2()
>>> end.getY()
200
```

Here is a small Python loop that can be used to create and draw several lines:

```
for n in range(0, 200, 5):
    L=Line(Point(n,25),Point(100,100))
    L.draw(myCanvas)
```



In the loop above (the results are shown on the right), the value of `n` starts at 0 and increases by 5 after each iteration all the way up to but not including 200 (i.e. 195). For each value of `n` a new `Line` object is created with starting co-ordinates `(n, 25)` and end point at `(100, 100)`.

Do This: Try out all the commands introduced so far. Then observe the effects produced by the loop above. Change the increment 5 in the loop above to different values (1, 3, etc.) and observe the effect. Next, try out the following loop:

```
for n in range(0, 200, 5):
    L = Line(Point(n, 25), Point(100, 100))
```

```
    L.draw(myCanvas)
    wait(0.3)
    L.undraw()
```

The `undraw` function does exactly as the name implies. In the loop above, for each value that `n` takes, a line is created (as above), drawn, and then, after a wait of 0.3 seconds, it is erased. Again, modify the value of the increment and observe the effect. Try also changing the amount of time in the `wait` command.

You can also draw several geometrical shapes: circles, rectangles, ovals, and polygons. To draw a circle, (or any geometrical shape), you first create it and then draw it:

```
C = Circle(centerPoint, radius)
c.draw(myCanvas)
```

`centerPoint` is a `Point` object and radius is specified in pixels. Thus, to draw a circle centered at (100, 150) with a radius of 30, you would do the following commands:

```
C = Circle(Point(100, 150), 30)
c.draw(myCanvas)
```

Rectangles and ovals are drawn similarly (see details at the end of the chapter). All geometrical objects have many functions in common. For example, you can get the center point of a circle, a rectangle, or an oval by using the command:

```
centerPoint = C.getCenter()
```

By default, all objects are drawn in black. There are several ways to modify or specify colors for objects. For each object you can specify a color for its outline as well as a color to fill it with. For example, to draw a circle centered at (100, 150), radius 30, and outline color red, and fill color yellow:

```
C = Circle(Point(100, 150), 30)
C.draw(myCanvas)
C.setOutline("red")
C.setFill("yellow")
```

By the way, `setFill` and `setOutline` have the same effect on `Point` and `Line` objects (since there is no place to fill any color). Also, the line or the outline drawn is always 1 pixel thick. You can change the thickness by using the command `setWidth(<pixels>)`:

```
C.setWidth(5)
```

The command above changes the width of the circle's outline to 5 pixels.

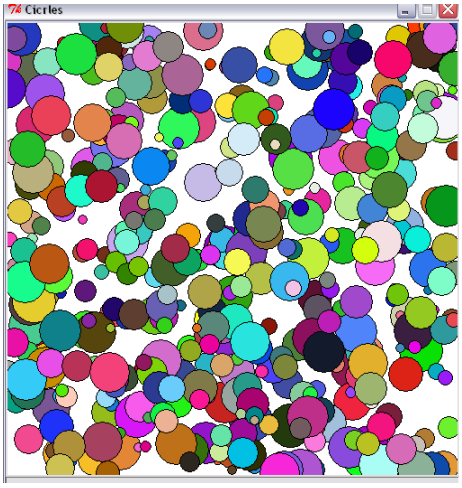
Do This: Try out all the command introduced here. Also, look at the end of the chapter for details on drawing other shapes.

Earlier, we mentioned that several colors have been assigned names that can be used to select colors. You can also create colors of your own choosing by specifying their red, green, and blue values. In Chapter 5 we mentioned that each color is made up of three values: RGB or red, green and blue color values. Each of the these values can be in the range 0..255 and is called a 24-bit color value. In this way of specifying colors, the color with values (255, 255, 255) (that is red = 255, green = 255, and blue = 255) is white; (255, 0, 0) is pure red, (0, 255, 0), is pure blue, (0, 0, 0) is black, (255, 175, 175) is pink, etc. You can have as many as 256x256x256 colors (i.e. over 16 million colors!). Given specific RGB values, you can create a new color by using the command, color_rgb:

```
myColor = color_rgb(255, 175, 175)
```

Do This: The program below draws several circles of random sizes with random colors. Try it out and see its outcome. A sample output screen is show on the right. Modify the program to input a number for the number of circles to be drawn.

```
# Program to draw a bunch of # random colored circles
from myro import *
from random import *

def makeCircle(x, y, r):
    # creates a Circle centered at point (x, y) of radius r
    return Circle(Point(x, y), r)

def makeColor():
    # creates a new color using random RGB values
    red = randrange(0, 256)
    green = randrange(0, 256)
    blue = randrange(0, 256)
    return color_rgb(red, green,blue)

def main():
    # Create and display a graphics window
    width = 500
    height = 500
    myCanvas = GraphWin("Cicrles", width, height)
    myCanvas.setBackground("white")

    # draw a bunch of random circles with random colors.
    N = 500
    for i in range(N):
        # pick random center
        # point and radius
        # in the window
        x = randrange(0,width)
        y = randrange(0,height)
        r = randrange(5, 25)
        c = makeCircle(x, y, r)
        # select a random color
        c.setFill(makeColor())

        c.draw(myCanvas)

main()
```



Notice our use of functions to organize the program. From a design perspective, the two functions makeCircle and makeColor are written differently. This is just for illustration purposes. You could, for instance, define makeCircle just like makeColro so it doesn't take any parameters and generates the values of x, y, and radius as follows:

```
def makeCircle():
    # creates a Circle centered at point (x, y) of radius r
    x = randrange(0,width)
    y = randrange(0,height)
    r = randrange(5, 25)
    c = makeCircle(x, y, r)

    return Circle(Point(x, y), r)
```

Unfortunately, as simple as this change seems, the function is not going to work. In order to generate the values of x, and y it needs to know the width and height of the graphics window. But width and height are defined in the function main and are not available or accessible in the function above. This is an issue of *scope* of names in a Python program: what is the scope of accessibility of a name in a program?

Python defines the scope of a name in a program textually or lexically. That is, any name is visible in the text of the program/function *after* it has been defined. Note that the notion of *after* is a textual notion. Moreover, Python restricts the accessibility of a name to the text of the function in which it is defined. That is, the names width and height are defined inside the function main and hence they are not visible anywhere outside of main. Similarly, the variables `red`, `green`, and `blue` are considered local to the definition of `makeColor` and are not accessible outside of the function, `makeColor`.

So how can `makeCircle`, if you decided it would generate the `x` and `y` values relative to the window size, get access to the width and height of the window? There are two solutions to this. First, you can pass them as parameters. In that case, the definition of `makeCircle` will be:

```
def makeCircle(w, h):
    # creates a Circle centered at point (x, y) of radius r
    # such that (x, y) lies within width, w and height, h
    x = randrange(0,w)
    y = randrange(0,h)
    r = randrange(5, 25)
    c = makeCircle(x, y, r)

    return Circle(Point(x, y), r)
```

Then the way you would use the above function in the main program would be using the command:

```
C = makeCircle(width, height)
```

That is, you pass the values of `width` and `height` to `makeCircle` as parameters. The other way to define `makeCircle` would be exactly as shown in the first instance:

```
def makeCircle():
    # creates a Circle centered at point (x, y) of radius r
    x = randrange(0,width)
    y = randrange(0,height)
    r = randrange(5, 25)
    c = makeCircle(x, y, r)

    return Circle(Point(x, y), r)
```

However, you would move the definitions of width and height outside and before the definitions of all the functions: function:

```
from myro import *
from random import *

width = 500
height = 500

def makeCircle(x, y, r):
    …
```

```
def makeColor():
   …

def main():
   …
```

Since the variables are defined outside of any function and before the definitions of the functions that use them, you can access their values. You may be wondering at this point, which version is better? Or even, why bother? The first version was just as good. The answer to these questions is similar in a way to a paragraph is an essay. You can write a paragraph in many ways. Some versions will be more preferable than others. In programming, the rule of thumb one uses when it comes to the scope of names is: ensure that only the parts of the program that are supposed to have access to a name are allowed access. This is similar to the reason you would not share your password with anyone, or your bank card code, etc. In our second solution, we made the names `width` and `height` *globally* visible to the entire program that follows. This implies that even `makeColor` can have access to them whether it makes it needs it or not.

You may want to argue at this point: what difference does it make if you make those variables visible to `makeColor` as long as you take care not to use them in that function? You are correct, it doesn't. But it puts an extra responsibility on your part to ensure that you will not do so. But what is the guarantee that someone who is modifying your program chooses to?

We used the simple program here to illustrate simple yet potentially hazardous decisions that dot the landscape of programming like land mines. Programs can crash if some of these names are *mishandled* in a program. Worse still, programs do not crash but lead to incorrect results. However, at the level of deciding which variables to make local and which ones to make global, the decisions are very simple and one just needs to exercise safe programming practices. We will conclude this section of graphics with some examples of creating animations.

Any object drawn in the graphics window can be moved using the command `move(dx, dy)`. For example, you move the circle 10 pixels to the right and 5 pixels down you can use the command:

```
C.move(10, 5)
```

Do This: Let us write a program that moves a circle about (randomly) in the graphics window. First, enter this following program and try it out.

```
# Moving circle; Animate a circle...
from myro import *
from random import *

def main():
    # create and draw the graphics window
    w = GraphWin("Moving Circle", 500, 500)
    w.setBackground("white")

    # Create a red circle
    c = Circle(Point(250, 250), 50)
    c.setFill("red")
    c.draw(w)

    # Do a simple animation for 200 steps
    for i in range(200):
        c.move(randrange(-4, 5), randrange(-4, 5))
        wait(0.2)

main()
```

Notice, in the above program, that we are moving the circle around randomly in the x- and y directions. Try changing the range of movements and observe the behavior. Try changing the values so that the circle moves only in the horizontal direction or only in the vertical direction. Also notice that we had to *slow down* the animation by inserting the wait command after every move. Comment the wait command and see what happens. It may appear that nothing did happen but in fact the 200 moves went so quickly that your eyes

couldn't even register a single move! Using this as a basis, we can now write a more interesting program. Look at the program below:

```
# Moving circle; Animate a circle...
from myro import *
from random import *

def main():
    # create and draw the graphics window
    winWidth = winHeight = 500
    w = GraphWin("Bouncing Circle", winWidth, winHeight)
    w.setBackground("white")

    # Create a red circle
    radius = 25
    c = Circle(Point(53, 250), radius)
    c.setFill("red")
    c.draw(w)

    # Animate it
    dx = dy = 3
    while timeRemaining(15):
        # move the circle
        c.move(dx, dy)

        # make sure it is within bounds
        center = c.getCenter()
        cx, cy = center.getX(), center.getY()

        if (cx+radius >= winWidth) or (cx-radius <= 0):
            dx = -dx

        if (cy+radius >= winHeight) or (cy-radius <= 0):
            dy = -dy

        wait(0.01)

main()
```

For 15 seconds, you will see a red circle bouncing around the window. Study the program above to see how we keep the circle inside the window at all times and how the direction of the ball bounce is being changed. Each time you change the direction, make the computer beep:

```
computer.beep(0.005, 440)
```

If you are excited about the possibility of animating a circle, imagine what you can do if you have many circles, and other shapes animated. Also, plug in the game pad controller and see if you can control the circle (or any other object) using the game pad controls. This is very similar to controlling your robot. Design an interactive computer game that takes advantage of this new input modality. You can also design multi-user games since you can connect multiple game pad controllers to your computer. See the Reference documentation for details on how to get input from several game pad controllers.

Sights: Images & Text

Like shapes, you can also place text in a graphics window. The idea is the same. You first create the text (using the `Text(<anchor point>, <string>)` command) and then draw it. You can specify the type face, size, and style of the text. We will not detail these here. They are summarized in the reference at the end of the text. When we get an opportunity, we will use these features below in other examples.

Images in this system are treated just like any other objects. You can create an image using the Image command:

```
myPhoto = Image(<centerPoint>, <file name>)
```

You have to have an already prepared image in one of the common image formats (like JPEG, GIF, etc.) and stored in a file (`<file name>`). Once the image object is created, it can be drawn, undrawn, or moved just like other shapes.

What you do with all this new found functionality depends on your creativity. You can use graphics to visualize some data: plotting the growth of world population, for example; or create some art, an interactive game, or even an animated story. You can combine your robot, the game pad controller, or even sound to enrich the multi-media experience. The exercises at the end of the chapter present some ideas to explore further. Below, we delve into sounds.

Sound

As we have seen, you can have your robot make *beeps* by calling the `beep()` function, like so:

```
beep(1, 440)
```

This command instructs the robot to play a tone at 440 Hz for 1 second. Let us first try and analyze what is in the **440 Hz** tone. First, the letters **Hz** are an abbreviation for **Hertz**. The name itself comes from a German physicist, Heinrich Rudolph Hertz who did pioneering work in the production of electromagnetic waves in the late 19th century. Today, we use **Hertz** (or *Hz*) as a unit for specifying frequencies.

$$1 Hertz = 1 cycle \ / \ second$$

The most common use of frequencies these days is in specifying the clock speeds of computer CPU's. For example, a typical PC today runs at clock speeds of a few GigaHertz (or GHz).

$$1 GigaHertz = 10^9 cycles \ / \ second$$

Frequencies are related to periodic (or repetitive) motions or vibrations. The time it takes for a motion or vibration to repeat is called its *time period*. Frequency and time period are inversely related. That is the number of cycles or repetitions in a second is called the frequency. Thus 1 Hertz refers to any motion or vibration that repeats every 1 second. In the case of computer clock frequencies then, a computer running at 4 Gigahertz is repeating 4 billion

times a second! Other examples of periodic motions include: the earth's rotation on its axis (1 cycle every 24 * 60 * 60 = 86400 seconds or at a frequency of 0.00001157 cycles/second), a typical audio CD spins 400 times a second, a CD drive on your computer rated at 52x spins the CD at 52 * 400 = 20800 times per second, hummingbirds can flap their wings at frequencies ranging from 20-78 times/second (some can go even as high as 200!). Sound is a periodic compression and refraction (or return to its original state) of air (for simplicity, let us assume that the medium is air). One Cycle of a sound comprises one compression and one refraction. Thus, producing a beep at 440 Hz represents 440 complete cycles of compression and refraction. generally, a human ear is capable for hearing frequencies in the 20 Hz to 20000 Hz (or 20 Kilo Hertz) range. However the capability varies from person to person. Also, many electronic devices are not capable for producing frequencies in that entire range. 20-20KHz is considered hi-fidelity for stereo or home theater audio components. Let us first examine the range of audible sounds the Scribbler can produce. To make a sound out of the Scribbler, you have to give a frequency and the duration (in seconds) that the sound should be played. For example, to play a 440 Hz sound for 0.75 seconds:

```
beep(0.75, 440)
```

The human ear is capable of distinguishing sounds that differ only by a few Hertz (as little as 1 Hz) however this ability varies from person to person. Try the commands:

```
beep(1, 440)
beep(1, 450)
```

Can you distinguish between the two tones? Sometimes it is helpful to place these in a loop so you can repeatedly hear the alternating tones to be able to distinguish between them. The next exercise can help you determine what frequencies you are able to distinguish.

Exercise: Using the example above, try to see how close you can get in distinguishing close frequencies. As suggested, you may want to play the tones alternating for about 5-10 seconds. Start with 440 Hz. Can you hear the difference between 440 and 441? 442? etc. Once you have established your range, try another frequency, say 800. Is the distance that you can distinguish the same?

Do This: You can program the Scribbler to create a siren by repeating two different tones (much like in the example above). You will have to experiment with different pairs of frequencies (they may be close together or far apart) to produce a realistic sounding siren. Write your program to play the siren for 15 seconds. The louder the better!

You can also have Myro make a beep directly out of your computer, rather than the robot, with the command:

```
computer.beep(1, 440)
```

Unfortunately, you can't really have the robot and computer play a duet. Why not? Try this:

```
beep(1, 440)
computer.beep(1, 440)
beep(1, 880)
computer.beep(1, 880)
beep(1, 440)
computer.beep(1, 440)
```

What happens? Try your solutions to the above exercises by making the sounds on the computer instead of the Scribbler.

Musical Scales

In western music, a *scale* is divided into 12 notes (from 7 major notes: ABCDEFG). Further there are *octaves*. An octave in C comprises of the 12 notes shown below:

```
C C#/Db D D#/Eb E F F#/Gb G G#/Ab A A#/Bb B
```

C# (pronounced "C sharp") is the same tone as Db (pronounced "D flat").

Frequencies corresponding to a specific note, say C, are multiplied (or divided) by 2 to achieve the same note in a higher (or lower) octave. On a piano there are several octaves available on a spread of keys. What is the relationship between these two tones?

```
beep(1, 440)
beep(1, 880)
```

The second tone is exactly one octave above middle C. To raise a tone by an octave, you simply multiply the frequency by 2. Likewise, to make a tone an octave lower, you divide by 2. Notes indicating an octave can be denoted as follows:

```
C0 C1 C2 C3 C4 C5 C6 C7 C8
```

That is, C0 is the note for C in the lowest (or 0) octave. The fifth octave (numbered 4) is commonly referred to as a middle octave. Thus C4 is the C note in the middle octave. The frequency corresponding to C4 is 261.63 Hz. Try playing it on the Scribbler. Also try C5 (523.25) which is twice the frequency of C4 and C3 (130.815). In common tuning (*equal temperament*) the 12 notes are equidistant. Thus, if the frequency doubles every octave, each successive note is $2^{1/12}$ apart. That is, if C4 is 261.63 Hz, C# (or Db) will be:

$$C\#4/Db4 = 261.63 * 2^{1/12} = 277.18$$

Thus, we can compute all successive note frequencies:

$$D4 = 277.18 * 2^{1/12} = 293.66$$

$$D\#4/Eb = 293.66 * 2^{1/12} = 311.13$$

etc.

The lowest tone that the Scribbler can play is A0 and the highest tone is C8. A0 has a frequency of 27.5 Hz, and C8 has a frequency of 4186 Hz. That's quite a range! Can you hear the entire range?

```
beep(1, 27.5)
beep(1, 4186)
```

Exercise: Write a Scribbler program to play all the 12 notes in an octave using the above computation. You may assume in your program that C0 is 16.35 and then use that to compute all frequencies in a given octave (C4 is $16.35 * 2^4$). Your program should input an octave (a number from 0 through 8), produce all the notes in that octave and also printout a frequency chart for each note in that octave.

Making Music

Playing songs by frequency is a bit of a pain. Myro contains a set of functions to make this task a bit more abstract. A Myro song is a string of characters composed like so:

```
NOTE1 [NOTE2] WHOLEPART
```

where `[]` means optional. Each of these notes/chords is composed on its own line, or separated by semicolons where:

```
    NOTE1 is either a frequency or a NOTENAME
    NOTE2 is the same, and optional. Use for Chords.
    WHOLEPART is a number representing how much of
              a whole note to play.
```

`NOTENAMES` are case-insensitive strings. Here is an entire scale of `NOTENAMES`:

```
C C#/Db D D#/Eb E F F#/Gb G G#/Ab A A#/Bb B C
```

This is the default octave. It is also the 5th octave, which can also be written as:

```
C5 C#5/Db5 D5 D#5/Eb5 E5 F5 F#5/Gb5 G5 G#5/Ab5 A5 A#5/Bb5 B5
C5
```

The Myro Song Format replicates the keys on the piano, and so goes from A0 to C8. The middle octave on a keyboard is number 4, but we use 5 as the default octave. See http://en.wikipedia.org/wiki/Piano_key_frequencies for additional details. Here is a scale:

```
"C 1; C# 1; D 1; D# 1; E 1; F 1; F# 1; G 1; G# 1; A 1; A# 1; B
1; C 1;"
```

The scale, one octave lower, and played as a polka:

```
"C4 1; C#4 1/2; D4 1/2; D#4 1; E4 1/2; F4 1/2; F#4 1; G4 1/2;
G#4 1/2; A4 1; A#4 1/2; B4 1/2; C4 1;"
```

There are also a few other special note names, including `PAUSE`, `REST`, you can leave the octave number off of the default octave notes if you wish. Use "#" for sharp, and "b" for flat.

`WHOLEPART` can either be a decimal notation, or division. For example:

```
Ab2 .125
```

or

```
Ab2 1/8
```

represents the A flat in the second octave (two below middle).

As an example, try playing the following:

```
c 1
c .5
```

```
c .5
c 1
c .5
c .5
e 1
c .5
c .5
c 2
e 1
e .5
e .5
e 1
e .5
e .5
g 1
e .5
e .5
e 2
```

Do you recognize it??

You may leave blank lines, and comments should begin with a # sign. Lines can also be separated with a semicolon.

Using a song

For the following exercises, you will need to have an object to play the song. You will need to initialize the robot in a slightly different way. Rather than:

```
initialize()
```

do:

```
robot = Scribbler()
```

Now that you have a song, you probably will want to play it. If your song is in a file, you can read it:

```
s = readSong(filename)
```

and play it on the robot:

```
robot.playSong(s)
```

or on the computer:

```
computer.playSong(s)
```

You can also use `makeSong(text)` to make a song. For example:

```
s = makeSong("c 1; d 1; e 1; f 1; g 1; a 1; b 1; c7 1;")
```

and then play it as above.

If you want to make it play faster or slower, you could change all of the `WHOLENOTE` numbers. But, if we just want to change the tempo, there is an easier way:

```
robot.playSong(s, .75)
```

The second argument to `playSong` is the duration of a whole note in seconds. Standard tempo plays a whole note in about .5 seconds. Larger numbers will play slower, and smaller numbers play faster.

Summary

You can use the graphics window as a way of visualizing anything. In the graphics window you can draw all kinds of shapes: points, line, circles, rectangles, ovals, polygons, text, and even images. You can also animate these shapes as you please. What you can do with these basic drawing capabilities is limited only by your creativity and your programming ability. You can combine the sights you create with sounds and other interfaces, like the game pad controller, or even your robot. The multimedia functionalities introduced in this chapter can be used in all kinds of situations for creating interesting programs.

Myro Reference

Below, we summarize all of the graphics commands mentioned in this chapter. You are urged to review the reference manual for more graphics functionality.

`GraphWin()`
`GraphWin(<title>, <width>, <height>)`
Returns a graphics window object. It creates a graphics window with title, `<title>` and dimensions `<width>` x `<height>`. If no parameters are specified, the window created is 200x200 pixels.

`<window>.close()`
`Closes the displayed graphics window <window>.`

`<window>.setBackground(<color>)`
Sets the background color of the window to be the specified color. `<color>` can be a named color (Google: color names list), or a new color created using the `color_rgb` command (see below)

`color_rgb(<red>, <green>, <blue>)`
Creates a new color using the specified `<red>`, `<green>`, and `<blue>` values. The values can be in the range 0..255.

`Point(<x>, <y>)`
Creates a point object at (`<x>`, `<y>`) location in the window.

`<point>.getX()`
`<point>.getY()`
Returns the x and y coordinates of the point object `<point>`.

`Line(<start point>, <end point>)`
Creates a line object starting at `<start point>` and ending at `<end point>`.

`Circle(<center point>, <radius>)`
Creates a circle object centered at `<center point>` with radius `<radius>` pixels.

`Rectangle(<point1>, <point2>)`
Creates a rectangle object with opposite corners located at `<point1>` and `<point2>`.

`Oval(<point1>, <point2>)`
Creates an oval object in the bounding box defined by the corner points `<point1>` and `<point2>`.

`Polygon(<point1>, <point2>, <point3>,…)`
`Polygon([<point1>, <point2>, …])`
Creates a polygon with the given points as vertices.

`Text(<anchor point>, <string>)`
Creates a text anchored (bottom-left corner of text) at `<anchor point>`. The text itself is defined by `<string>`.

`Image(<centerPoint>, <file name>)`
Creates an image centered at `<center point>` from the image file `<file name>`. The image can be in GIF, JPEG, or PNG format.

All of the graphics objects respond to the following commands:

`<object>.draw(<window>)`
Draws the `<object>` in the specified graphics window `<window>`.

`<object>.undraw()`
Undraws `<object>`.

`<object>.getCenter()`
Returns the center point of the `<object>`.

`<object>.setOutline(<color>)`
`<object>.setFill(<color>)`
Sets the outline and the fill color of the `<object>` to the specified `<color>`.

`<object>.setWidth(<pixels>)`
Sets the thickness of the outline of the `<object>` to `<pixels>`.

`<object>.move(<dx>, <dy>)`
Moves the object `<dx>`, `<dy>` from its current position.

The following sound-related functions were presented in this chapter.

`beep(<seconds>, <frequency>)`
`beep(<seconds>, <f1>, <f2>)`
Makes the robot beep for `<seconds>` time at frequency specified. You can either specify a single frequency `<frequency>` or a mix of two: `<f1>` and `<f2>`.

`<robot/computer object>.beep(<seconds>, <frequency>)`
`<robot/computer object>.beep(<seconds>, <f1>, <f2>)`
Makes the robot or computer beep for `<seconds>` time at frequency specified. You can either specify a single frequency `<frequency>` or a mix of two: `<f1>` and `<f2>`.

`robot.playSong(<song>)`
Plays the <song> on the robot.

`readSong(<filename>)`
`Reads a song file from <filename>.`

`song2text(song)`
Converts a <song> to text format.

`makeSong(<text>)`
`text2song(<text>)`
Converts `<text>` to a song format.

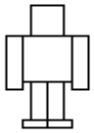
Python reference

In this chapter we presented informal scope rules for names in Python programs. While these can get fairly complicated, for our purposes you need to know the distinction between a *local variable name* that is local within the scope of a function versus a *global name* defined outside of the function. The text ordering defines what is accessible.

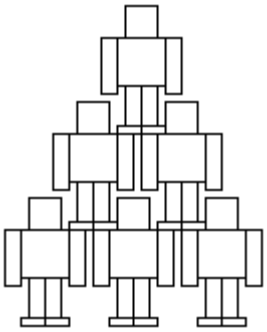
Exercises

Exercise: Using the graphics commands learned in this chapter, write a program to generate a seasonal drawing: winter, summer, beach, or a festival scene.

Exercise: Write a program that has a function `drawRobot(x, y, w)` such that it draws the figure of a robot shown on the right. The robot is anchored at (`x`, `y`) and is drawn in the window, `w`. Also notice that the robot is entirely made up of rectangles (8 of them).



Exercise: Using the `drawRobot` function from the previous exercise, draw a pyramid of robot as shown on the right.



Exercise: Suppose that each robot from the previous exercise is colored using one of the colors: red, blue, green, and yellow. Each time a robot is drawn, it uses the next color in the sequence. Once the sequence is exhausted, it recycles. Write a program to draw the above pyramid so robots appear in these colors as they are drawn. You may decide to modify `drawRobot` to have an additional parameter: `drawRobot(x, y, c, w)` or you can write it so that `drawRobot` decides which color it chooses. Complete both versions and compare the resulting programs. Discuss the merits and/or pitfalls of these versions with friends. [Hint: Use a list of color names.]

9 Robot Vision

Seeing is believing.
-: Proverb

Your robot has a small camera that can be used to take pictures. Myro includes several functions that allow you to capture and modify images. Lets start with the simplest, capturing a picture, assigning it to a variable, and then showing it on the screen:

```
pic = takePicture()
show(pic)
```

You can get some information about the size of the picture with the getWidth() and getHeight() functions:

```
picWidth = getWidth(pic)
picHeight = getHeight(pic)
print "The Picture is", picWidth , "pixels wide and",
picHeight, "pixels high."
```

A normal Myro picture is full color, which means that each pixel is made up of a blend of three primary colors (Red, Green, and Blue). Each pixel has a separate value (from 0 to 255) for each of the three primary colors. Note that the computer is using 8 bits (one byte) to store each color value. One byte is made up of 8 bits, and with 8 bits you can represent 2^8 (256) values, or a number between zero and 255.

So a completely red pixel would be represented as (255,0,0), or full red, and zero green and blue. Similarly a green pixel would be represented as (0,255,0) and a blue pixel would be represented as (0,0,255). Other colors are made by mixing various values of red, green, and blue. For example, white is made by combining all of the colors (255,255,255), and black is made by an absence of any color (0,0,0). If you combine green and blue with no red (0,255,255) you would get a teal color, while a combination of red and green with no blue (255,255,0) would produce yellow.

Because each pixel in the picture needs three bytes (one for the Red, Green, and Blue values), each pixel requires three bytes (or 24 bits) to store. This is why full color pictures are sometimes referred to as "24 bit".

Do This: If our picture is 256 pixels wide, and 192 pixels high, how many total pixels does it have? How many bytes does it take to store this picture?

Robot Explorer

If you do not need a full color picture, you can tell myro to capture a gray-scale image by giving the `takePicture()` function the `"gray"` parameter.

```
grayPic = takePicture("gray")
show(grayPic)
```

Did you notice that taking the gray-scale picture took less time than taking the color picture? This is because the gray scale picture only uses one byte per pixel, instead of three. The gray-scale picture does not have Red, Green, and Blue values for each pixel. Instead, it stores a single number (again, one bye, or a number between zero and 255) for each pixel that represents how dark or light the pixel is (zero is full black, and 255 is full white).

Because gray-scale images can be transferred from the robot to your computer more quickly than full color images, they can be useful if you want the images to update quickly. For example, you can use the `joyStick()` function combined with a loop that quickly takes and displays pictures to turn your robot into a remotely piloted explorer, similar to the mars rovers.

```
joyStick()
for i in range(25):
  pic = takePicture("gray")
  show(pic)
```

The above code will open a joy stick window so that you can control your robot, and then capture and show 25 pictures, one after the other. While the pictures are being captured and displayed like a movie, you can use the joystick to drive your robot around, using the pictures to guide it. Of course, if you removed the "gray" parameter from the `takePicture()` function call, you would get color pictures instead of gray-scale pictures, but they would take much longer to transfer from the robot to your computer, and make it more difficult to control the robot.

Saving and Loading Pictures

If you take a picture that you would like to save, you can save it as a JPEG file on your computer's hard drive with the `savePicture()` function.

```
myPicture = takePicture()
```

```
savePicture(myPicture,"myPic.jpg")
```

You can use any name you want for your picture, as long as it ends with `.jpg` to tell Myro (and your computer) that it is a JPEG image file. Later, you can load the picture from disk with the `makePicture()` function:

```
mySavedPicture = makePicture("myPic.jpg")
show(mySavedPicture)
```

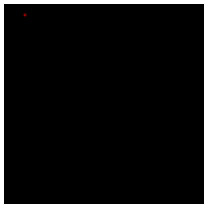
Manipulating Pixels

Myro provides functions that allow you to get the individual red, green, and blue values from a pixel, as well as set the values to numbers of your choosing. But before you get or set the value of a pixel, you need to select a specific pixel to change. To demonstrate, we will start out by making a blank picture that is 100 pixels high by 100 pixels wide, using the `makePicture()` function:

```
newPic = makePicture(100,100)
show(newPic)
```

Note that the picture we made starts out will all pixels colored pure black. We can select a specific pixel (say, the one at X location 10 and Y location 5) using the `getPixel()` function. After we select a specific pixel, we can set its red value to full on (255) with the `setRed()` function.

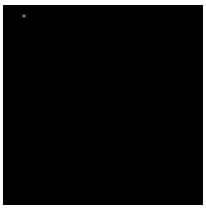
```
onePixel = getPixel(newPic,10,5)
print getRed(onePixel)     #initial red value
setRed(onePixel,255)
print getRed(onePixel)     #new red value
show(newPic)
```



Note that the red value of the pixel at (10,5) was initially zero, but after our call to setRed() the pixel value has changed to 255. Notice also that we had to call show() again before our change is displayed on the screen. If you look at the picture very carefully, you will see that one pixel

near the top left corner is now red, instead of black. If you want to make that pixel white, you have to assign 255 to the green and blue colors as well:

```
setGreen(onePixel,255)
setBlue(onePixel,255)
show(newPic)
```



Now the pixel at location (10,5) is pure white.

Drawing a Line

If we want to change a lot of pixels all at once, we can use a loop. For example, the following loop will change all pixels that have an X value of 10 and a Y value anywhere between 0 and 100 (but not including 100) to be red:

```
for yValue in range(0,100):
  aPixel = getPixel(newPic, 10, yValue)
  setRed(aPixel,255)
  setGreen(aPixel,0)
  setBlue(aPixel,0)

show(newPic)
```

The result is a vertical red line (at X position 10). Note that this piece of code will only work correctly for pictures that are exactly 100 pixels high (because we loop from zero to 100). But we can generalize this code to work on pictures of any size by replacing the 100 with a function call that tells us the actual height of the picture as follows:

```
for yValue in range(0, getHeight(newPic) ):
  aPixel = getPixel(newPic, 10, yValue)
  setRed(aPixel,255)
  setGreen(aPixel,0)
  setBlue(aPixel,0)
```

Drawing a line could be a useful function to use later, so we can prepare to re-use the code above by putting it into a function. But since we don't know the

exact X position that the user will want to draw their line, we should make that into a parameter that the user can specify. Also, since we don't know the name of the variable that will hold the picture, that should also be a parameter:

```
def drawVerticalRedLine( picture, xPos ):
  for yPos in range(0, getHeight(picture) ):
     aPixel = getPixel( picture, xPos, yPos)
     setRed(aPixel,255)
     setGreen(aPixel,0)
     setBlue(aPixel,0)
```

Now we have a function that will draw a vertical red line on a picture of any height. Note that our function does NOT show the image, so a user would have to call our function, and then call the show() function to display the line on screen:

```
pic = takePicture()
show(pic)
wait(1)
drawVerticalRedLine(pic,
25)
show(pic)
```

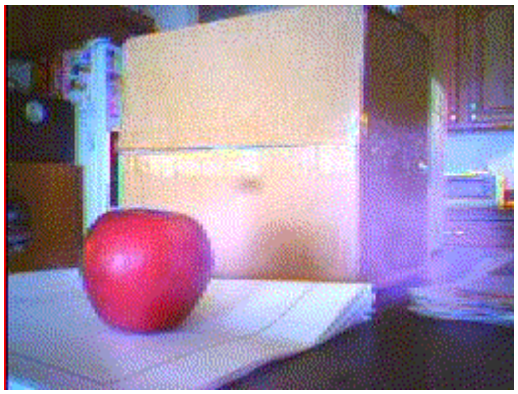
Do This: Write a function to draw a horizontal red line.

Moving a Line

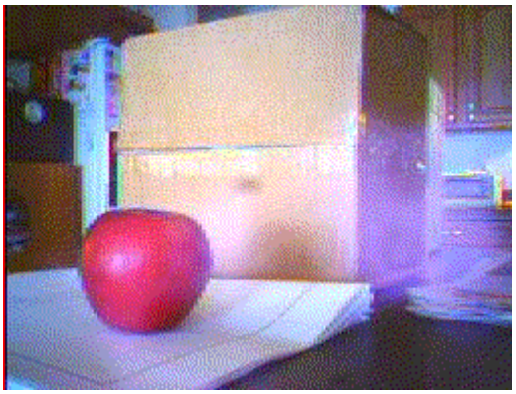
You can make a special effect by drawing the red line moving across the picture one X position at a time with another loop:

```
myPic = takePicture()

for xPos in range(0, getWidth(myPic) ):
  print "Now Drawing at xPos: ", xPos
  drawVerticalRedLine(myPic, xPos)
```

```
  show(myPic)
```



Of course, this slowly covers up the existing picture with nothing but red pixels. If you want to only cover one line at a time, you have to draw on a copy of the original picture to keep from losing your picture data. The copyPicture() function will return a copy of a picture so that you can draw on a copy without covering up pixels from the original.



```
originalPic = takePicture()

for xPos in range(0,
getWidth(originalPic) ):
  #Make a copy of the
original to draw on
  copyPic =
copyPicture(originalPic)
  print "Now Drawing at
xPos:", xPos

drawVerticalRedLine(copyPic,xPos)
  show(copyPic)
```

Operating on all pixels in an image

The `getPixel()` function will return a pixel at a specific location in the picture. However, sometimes you want to do something to all pixels in the picture. The `getPixels()` method will return a generator that is similar to a list of all pixels. You can use the `getPixels()` method in a for loop to perform an operation on all pixels in the image. For example, we can turn the red color of all pixels all the way on with the following code:

```
myPic = takePicture()
```

```
show(myPic)

for eachPixel in getPixels(myPic):
  setRed(eachPixel,255)

show(myPic)
```

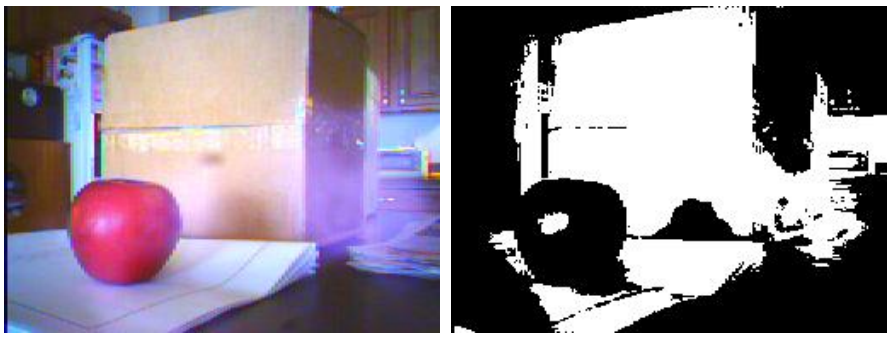
Because the above code does not change the green or blue values of the pixels, the picture is still recognizable, but all pixels have a reddish tint.

Robot Vision

By making a decision about each pixel in an image, you can locate specific areas in an image. For example, by looking for pixels that have a large value, you can locate bright areas in the image. You can even modify the image to outline bright areas. For example:

```
myPicture = takePicture()
show(myPicture)

for pixel in getPixels(myPicture):
  redValue = getRed(pixel)
  greenValue = getGreen(pixel)
  blueValue = getBlue(pixel)
  averageValue = ( redValue + greenValue + blueValue) / 3.0

  if averageValue > 175 :
    #Turn the pixel white
```

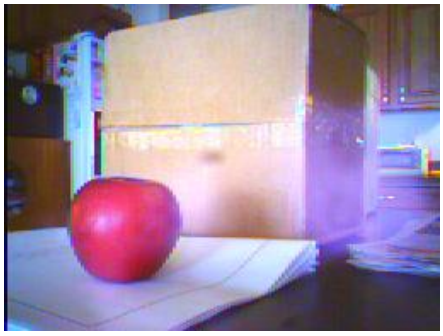
```
    setRed(pixel,255)
    setGreen(pixel,255)
    setBlue(pixel,255)
  else:
    #Otherwise, turn it black.
    setRed(pixel,0)
    setGreen(pixel,0)
    setBlue(pixel,0)

show(myPicture)
```



By changing the conditional test, we can instead look for areas that have a large amount of the color red. Red areas are characterized by having large red values, but smaller blue and green values.

```
myPicture = takePicture()
show(myPicture)

for pixel in getPixels(myPicture):
  redValue = getRed(pixel)
  greenValue = getGreen(pixel)
  blueValue = getBlue(pixel)

  if redValue > 175 and greenValue < 175 and blueValue < 175 :
    #Turn the pixel white
    setRed(pixel,255)
    setGreen(pixel,255)
    setBlue(pixel,255)
```

```
  else:
    #Otherwise, turn it black.
    setRed(pixel,0)
    setGreen(pixel,0)
    setBlue(pixel,0)

show(myPicture)
```



Original Picture

Pixels with Red value larger than 175 highlighted in white

Pixels with green values larger than 175 highlighted in white.

We can make this code to find red pixels into a function that will return a black and white picture with white pixels representing "red" areas.

Pixels that have both red values larger than 175 and green and blue values lower than 175 highlighted in white.

```
def findRedAreas(picture):

  for pixel in getPixels(picture):
    redValue = getRed(pixel)
    greenValue = getGreen(pixel)
    blueValue = getBlue(pixel)

    if redValue > 175 and greenValue < 175 and blueValue <
175 :
      #Turn the pixel white
      setRed(pixel,255)
      setGreen(pixel,255)
      setBlue(pixel,255)
    else:
      #Otherwise, turn it black.
      setRed(pixel,0)
      setGreen(pixel,0)
      setBlue(pixel,0)

  return picture
```

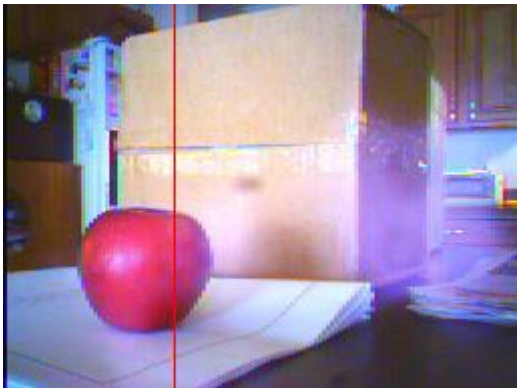
The result of testing for "red areas" is an image where most of the apple has been detected, but a lot of other pixels scattered around the image are also somewhat "reddish". If we want the robot to turn towards the direction with the most red pixels, we need to calculate the average X (horizontal) location of the pixels that have been marked (by turning them white).

If we examine every pixel in the image, and average the X coordinates of all the "detected" (or white) pixels, we can determine the middle of the red areas. To do this, we use two loops (one for the Y positions, and one for the X positions) to look at every pixel, and add up the X positions of all pixels that are turned "on" (or white). Because a white pixel has values of (255,255,255) and an "off" or black pixel has values of (0,0,0) we can take a shortcut and only test the value of one of the three colors.

```
def AverageXofWhitePixels(picture):
   sumX = 0.0           # We use floating point values.
   counter = 0.0        # We use floating point values.
   for xPos in range(0, getWidth(picture) ):
      for yPos in range(0, getHeight(picture) ):
          pixel = getPixel(picture,xPos,yPos)
          value = getGreen(pixel)
          if value > 0 :
             sumX = sumX + xPos
             counter = counter + 1

   averageX = sumX / counter
   return int(averageX)         #Return an Integer
```

Now, we can use the functions we have defined so far to locate the average X location of red pixels, and draw a line at that position:

```
myPicture = takePicture()
picture = copyPicture(myPicture)
picture = findRedAreas(picture)
XposAvg = AverageXofWhitePixels(picture)
drawVerticalRedLine(myPicture,XposAvg)
show(myPicture)
```



Because of all the other "red" pixels detected that were not on the apple, the red line is not exactly centered on the apple, but it is close enough that we could use the value in the XposAvg variable to figure out which way to turn the robot to face the apple.

Do This: Write a program that will turn your robot towards red objects.

Animated GIF movies

The `savePicture()` function will also allow you to make an animated GIF, which is a special type of picture that in a web browser will show several pictures one after another in an animation. To save an animated GIF, you must give the `savePicture()` function a list of pictures (instead of a single picture) and a filename that ends in `".gif"`. Here is an example:

```
pic1 = takePicture()
turnLeft(0.5,0.25)
pic2 = takePicture()
turnLeft(0.5,0.25)
pic3 = takePicture()
turnLeft(0.5,0.25)
pic4 = takePicture()
```

```
listOfPictures = [pic1, pic2, pic3, pic4]

savePicture(listOfPictures, "turningMovie.gif")
```

The best way to view an animated GIF file is to use a web browser. If you are using Firefox, you can use the `FILE->Open File` menu, and then pick the turningMovie.gif file. The web browser will show all frames in the movie, but then stop on the last frame. To see the movie again, press the `"Reload"` button.

You can also use code in a loop to make a longer movie with more images:

```
pictureList = []  #Start with an empty list.
for i in range(15):
  pic = takePicture()
  pictureList = pictureList + [pic]   #Append the new picture
  turnLeft(0.5,0.1)

savePicture(pictureList,"rotatingMovie.gif")
```

Summary

In this chapter, you have seen various Myro facilities that can be used for image processing and vision. Combined with robot behaviors these faciltities can be used for modeling smarter behaviors. This chapter is still in its

preliminary form and with time it will evolve into a more substantial chapter on vision, perception, and illustrate several useful vision paradigms.

Myro review

Once the final version of this chapter is written, we will summarize the new Myro features here. Several functions introduced here have already been covered in earlier chapters.

Python Review

There were no new Python features introduced in this chapter.

Exercises

10 Artificial Intelligence

David: Martin is Mommy and Henry's real son. After I find the Blue Fairy then I can go home. Mommy will love a real boy. The Blue Fairy will make me into one.
Gigolo Joe: Is Blue Fairy Mecha, Orga, man or woman?
David: Woman.
Gigolo Joe: Woman? I know women! They sometimes ask for me by name. I know all about women. About as much as there is to know. No two are ever alike, And after they've met me, no two are ever the same. And I know where most of them can be found.
David: Where?
Gigolo Joe: Rouge City. Across the Delaware.

-: Dialog between two Artificial Intelligence entities: Gigolo Joe (played by Jude Law) and David (played by Haley Joel Osment) in the movie, Artificial Intelligence (2001), Directed by Steven Speilberg, Warner Bros.

The Question of Intelligence

The quest for the understanding of intelligence probably forms the oldest and yet to be fully understood human inquiry. With the advent of computers and robots the question of whether robots and computers can be as intelligent as humans has driven the scientific pursuits in the field of *Artificial Intelligence (AI)*. Whether a computer can be intelligent was lucidly discusses by Professor Alan Turing in 1950. To illustrate the issues underlying machine intelligence, Turing devised a thought experiment in the form of an *imitation game*. It is played with three people, a man, a woman, and an interrogator. They are all in separate rooms and interact with each other by typing text into a computer (much like the way people interact with each other over IM or other instant messaging services). The interrogator's task is to identify which person is a man (or woman). To make the game interesting, either player can try and be deceptive in giving their answers. Turing argues that a computer should be considered intelligent if it could be made to play the role of either player in the game without giving itself away. This *test* of intelligence has come to be called the *Turing Test* and has generated much activity in the community of AI researchers (see exercises below). The dialog shown above, from the movie *Artificial Intelligence*, depicts an aspect of the test of intelligence designed by Alan Turing. Based on the exchange between Gigolo Joe and David, can you conclude that they are both intelligent? Human?

After over five decades of AI research, the field has matured, and evolved in many ways. For one, the focus on intelligence is no longer limited to humans: insects and other forms of animals depict varying degrees and kinds of intelligence have been the subject of study within AI. There has also been a fruitful exchange of ideas and models between AI scientists, biologists, psychologists, cognitive scientists, neuro-scientists, linguists and philosophers. You saw examples of such an influence in the models of Braitenberg vehicles introduced earlier. Given the diversity of researchers involved in AI there has also been an evolution of what AI itself is really about. We will return to this later in the chapter. First, we will give you a few examples of models that could be considered *intelligent* that are commonly used by many AI scientists.

Language Understanding

One aspect of intelligence acknowledged by many people is the use of language. People communicate with each other using a language. There are many (several thousand) languages in use on this planet. Such languages are called *natural languages*. Many interesting theories have been put forward about the origins of language itself. An interesting question to consider is: Can people communicate with computers using human (natural) languages? In other words, can a computer be made to understand language? Think about that for a minute and see if you can come up with a possible answer.

To make the question of language understanding more concrete, think of your Scribbler robot. So far, you have controlled the behavior of the robot by writing Python programs for it. Is it possible to make the Scribbler understand English so that you could interact with it in it? What would an interaction with Scribbler look like? Obviously, you would not expect to have a conversation with the Scribbler about the dinner you ate last night. However, it would probably make sense to as it to move in a certain way? Or to ask whether it is seeing an obstacle ahead?

Researchers working in the field of *computational linguistics* (or *natural language understanding*) have proposed many theories of language processing that can form the basis of a computational model for a Scribbler to understand a small subset of the English language. In the section, we will examine one such model which is based on the processing of syntax and semantics of language interaction. Imagine, interacting with the Scribbler using the following set of sentences:

```
You: do you see a wall?
Scribbler: No

You: Beep whenever you see a wall.
You: Turn right whenever you see a wall to your left.
You: Turn left whenever you see a wall to your right.
You: Move for 60 seconds.
```

```
[The Scribbler robot moves around for 60 seconds turning
whenever it sees a wall.
 It also beeps whenever it sees a wall.]
```

Earlier, you have written Python programs that perform similar behaviors. However, now imagine interacting with the robot in the fashion described. From a physical perspective, imagine that you are sitting in front of a computer, and you have a Bluetooth connection to the robot. The first question then becomes: Are you actually speaking or typing the above commands? From an AI perspective, both modalities are possible: You could be sitting in front of the computer and speaking into a microphone; or you could be typing those commands on the keyboard. In the first instance, you would need a speech understanding capability. Today, you can obtain software (commercial as well as freeware) that will enable you to do this. Some of these systems are capable of distinguishing accents, intonations, male or female voices etc. Indeed, speech and spoken language understanding is an fascinating field of study that combines knowledge from linguistics, signal processing, phonology, etc. You can imagine that the end result of speaking into a computer is a piece of text that transcribes what you said. So, the question posed to the Scribbler above: *Do you see a wall?* will have to be processed and then transcribed into text. Once you have the text, that is, a string "Do you see a wall?" it can be further processed or analyzed to understand the *meaning* or the content of the text. The field of *computational linguistics* provides many ways of syntactic parsing, analyzing, and extracting meaning from texts. Researchers in AI itself have developed ways of representing knowledge in a computer using symbolic notations (e.g. *formal logic*). In the end, the analysis of the text will result in a `getIR()` or `getObstacle()` command to the Scribbler robot and will produce in a response shown above.

Our goal of bringing up the above scenario here is to illustrate to you various dimensions of AI research that can involve people from many different disciplines. These days, it is entirely possible even for you to design and build computer programs or systems that are capable of interacting with robots using language.

Game Playing

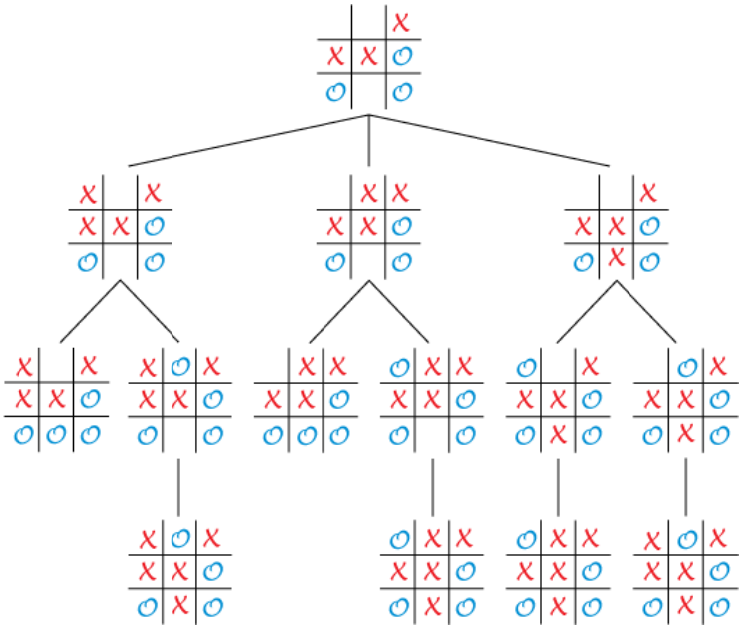
In the early history of AI, scientists posed several challenging tasks which if performed by computers could be used as a way of demonstrating the feasibility of machine intelligence. It was common practice to think of games in this realm. For example, if a computer could play a game, like chess, or checkers, at the same level or better that humans we would we convinced into thinking that it was indeed feasible to think of a computer as a possible candidate for machine intelligence. Some of the earliest demonstrations of AI research included attempts at computer models for playing various games. Checkers and chess seemed to be the most popular choices, but researchers have indulged themselves into examining computer models of many popular games: poker, bridge, scrabble, backgammon, etc. In many of these games, it is now possible for computer models to play at the highest levels of human performance. In Chess, for example, even though the earliest programs handily beat novices in the 1960's, it wasn't until 1996 when an IBM computer chess program, named Deep Blue, beat the world champion Gary Kasparov at a tournament-level game, though Kasparov did manage to win the match 4-2. A year later, in New York, Deep Blue beat Kasparov in a 6 game match representing the very first time a computer beat the best human player in a classical style game of chess. While these accomplishments are worthy of praise, it also now clear that the quest for machine intelligence is not necessarily answered by computer game playing. This has resulted in much progress in game playing systems and game playing technology which now stands in its own right as a multi-billion dollar industry.

It turns out that in many chess-like games the general algorithm for a computer to play the game is very similar. Such games are classified as two-person zero-sum games: two people/computers play against each other and the result of the game is either a win for one player and loss for the other, or it is a draw (which makes it a zero-sum end result). In many such games, the basic strategy for making the next move is simple: look at all the possible moves I have and for each of them all the possible moves the other player might have and so on until the very end. Then, trace back from wins (or draws) and make

the next move based on those desirable outcomes. You can see this easily in a simple Tic-Tac-Toe game (see picture below):

When you play against an opponent, you are anticipating possible moves down the road and then playing your own moves with those in mind. Good players are able to mentally picture the game several moves ahead. In many games, like Chess, certain recognizable situations lead to well determined outcomes and so a great part of playing a successful game also relies on the ability to recognize those situations. Looking ahead several moves in a

A Tic Tac Toe Game Tree to look for possible next moves for X



systematic manner is something computers are quite capable of doing and hence anyone (even you!) can turn them into fairly good players. The challenge lies in the number of moves you can look ahead and in the limited capacity, if time to make the next move is limited, how to choose among the best available options? These decisions lend interesting character to computer game programs and continue to be a constant source of fascination for many

people. For example, a computer program to play Tic-Tac-Toe can easily look at all the possible moves all the way to the end of game in determining its next move (which, in most situations leads to a draw, given the simplicity of the game). However, if you consider a typical game of Chess, in which each player makes an average of 32 moves and the number of feasible moves available at any time averages around 10, you would soon realize that the computer would have to examine something of the order of 10^65 board positions before making a move! This, even for the fastest computers available today, will take several gazillion years! More on that later. But, to play an interesting two-person zero-sum game, it is not essential to look so far ahead.

In Chapter 7, you saw an example of a program that played the game of Paper-Scissors-Rock against a human user. In that version, the program's choice strategy for picking an object was completely random. We reproduce that section of the program here:

```
…
items = ["Paper", "Scissors", "Rock"]
…
# Computer makes a selection
myChoice = items[randrange(0, 3)]
…
```

In the above program segment, `myChoice` is the program's choice. As you can see, the program uses a random number to select its object. That is, the likelihood of picking any of the three objects is 0.33 or 33%. The game and winning strategies for this game have been extensively studied. Some strategies rely on detecting patterns in human choice behavior. Even though we may not realize it there are patterns in our seemingly random behavior. Computer programs can easily track such behavior patterns by keeping long histories or player's choices, detect them, and then design strategies to beat those patterns. This has been shown to work quite effectively. It involves recording player's choices and searching through them (see Exercises). Another strategy is to study human choice statistics in this game. Before we present you with some data, do the exercise suggested below:

Do This: Play the game against a few people, Play several dozen rounds. Record the choices made by each player (just write a P/S/R in two columns). Once done, compute the percentages of each object picked. Now read on.

It turns out that most casual human players are more prone towards picking Rock than Paper or Scissors. In fact, various analyses suggest that 36% of the time people tend to pick Rock, 30% Paper, and 34% Scissors. This suggests that RPS is not merely a game of chance there is room for some strategies at winning. Believe it or not, there are world championships of PSR held each year. Even a simple game like this has numerous possibilities. We can use some of this information, for instance, to make our program smarter or better adept at playing the game. All we have to do is instead of using a fair 33% chance of selecting each object we can skew the chances of selection based on people's preferences. Thus, if 36% of the time people tend to pick Rock, it would be better for our program to pick Paper 36% of the time since Paper beats Rock. Similarly, our program should pick Scissors 30% of the time to match the chance of beating Paper, and pick Rock 34% of the time to match the chances of beating Paper. We can bias the random number generator using these percentages as follows:

First generate a random number in the range 0..99
If the number generated is in the range 0..29, select Scissors (30%)
If the number generated is in the range 30..63, select Rock (34%)
If the number generated is in the range 64..99, select Paper (36%)

The above strategy of biasing the random selection can be implemented as follows:

```
def mySelection():

    # First generate a random number in the range 0..99
    n = randrange(0, 100)

    # If the n is in range 0..29, select Scissors
    if n <= 29:
        return "Scissors"
    elif n <= 63:
```

```
        # if n in range 30..63, select Rock
        return "Rock"
    else:
        return "Paper"
```

Do This: Modify your RPS program from Chapter 7 to use this strategy. Play the game several times. Does it perform much better that the previous version? You will have to test this by collecting data from both versions against several people (make sure they are novices!).

Another strategy that people use is based upon the following observation:

After many rounds, people tend to make the move that would have beaten their own previous move.

That is, if say a player picks Paper. Their next pick will be Scissors. A computer program or a player playing against this player should then pick Rock to beat Scissors. Since the relationship between the choices is cyclical the strategy can be implemented by picking the thing that beats the opponent's pervious move beats. Paper beats Rock. Therefore since the player's previous move was Paper, your program can pick Rock in anticipation of the player's pick of Scissors. Try to think over this carefully and make sure your head is not spinning by the end of it. If a player can spot this they can use this as a winning strategy. We will leave the implementation the of last strategy as an exercise. The exercises also suggest another strategy.

The point of the above examples is that using strategies in your programs you can make your programs smarter or more intelligent. Deliberately, we have started to use the term intelligence a little more loosely than what Alan Turing implied in his famous essay. Many people would argue that these programs are not intelligent in the ultimate sense of the word. We agree. However, writing smarter programs is a natural activity. If the programs incorporate strategies or heuristics that people would use when they are doing the same activity, then the programs have some form of artificial intelligence in them. Even if the strategy used by the program is nothing like what people would use, but it would make the program smarter or better, we would call it

artificial intelligence. Many people would disagree with this latter claim. To some, the quest for figuring out intelligence is limited to the understanding of intelligence in humans (and other animals). In AI both points of view are quite prevalent and make for some passionate debates among scholars.

Learning

Here we will give you an overview of machine learning, introduce you to the idea of computational neural networks, and then show you how using the Myro modules for neural networks, you can design a learning program for your Scribbler robot to learn how to avoid obstacles.

Discussion

The very idea of considering a computer as an intelligent device has its foundations in the general purpose nature of computers. By changing the program the same computer can be made to behave in many different ways. At the core of it a computer is just a symbol manipulator: manipulating encodings for numbers, or letters, or images, etc. It is postulated that the human brain is also a symbol manipulator. The foundations of AI lie in the fact that most intelligent systems are physical symbol systems and since a computer is a general purpose symbol manipulator, it can be used for studying or simulating intelligence.

11 Computers & Computation

Home computers are being called upon to perform many new functions, including the consumption of homework formerly eaten by the dog.
-: Doug Larson

Computer Science is no more about computers than astronomy is about telescopes.
-: Edsger W. Dijkstra

What is the central core of computer science? What is it that distinguishes it from the separate subjects with which it is related? What is the linking thread which gathers these disparate branches into a single discipline. My answer to these questions is simple -it is the art of programming a computer. It is the art of designing efficient and elegant methods of getting a computer to solve problems, theoretical or practical, small or large, simple or complex. It is the art of translating this design into an effective and accurate computer program.
-: C.A.R. Hoare

Today, there are more computers than people on most college campuses in the United States. The laptop computer shown on the previous page is the XO laptop developed by the One Laptop Per Child Project (OLPC). It is a low cost computer designed for kids. The OLPC project aims to reach out to over 2 billion children all over the world. They aim to get the XO computers in their hands to help in their education. Their mission is to bring about a radical change in the global education system by way of computers. Of course a project with such a grandiose mission is not without controversy. It has had a bumpy ride since its inception. The technological issues were the least of their problems. They have had to convince governments (i.e. politicians) to buy-in into the mission. Several countries have agreed to buy-in and then reneged for all kinds of reasons. The laptops are not available for open purchase within the United States which has led to other socio-political controversies. Regardless, in the first year of its release the project aims to have over 1 million XO's in the hands of children in many developing countries. The thing about technology that has always been true is that a good idea is infective. Other players have emerged who are now developing ultra-low cost computers for kids. It will not be long before other competitive products will be available to all. The bigger questions are: What kind of change will this bring about in the world? How will these computers be used in teaching elementary and middle school children? Etc. You may also be wondering if your Scribbler robot can be controlled by the XO. It can.

You have been writing programs to control your robot through a computer. Along the way you have seen many different ways to control a robot and also ways of organizing your Python programs. Essentially you have been engaging in the activity commonly understood as *computer programming* or *computer-based problem solving*. We have deliberately refrained from using those terms because the general perception of these conjures up images of geeks with pocket-protectors and complex mathematical equations that seem to be avoidable or out of reach for most people. Hopefully you have discovered by now that solving problems using a computer can be quite exciting and engaging. Your robot may have been the real reason that motivated you into this activity and that was by design. We are confessing to you, in a way, that robots were used to attract you to pay some attention to

computing. You have to agree, if you are reading this, that the ploy worked! But remember that the reason you are holding a robot of your own in your hand is also because of computers. Your robot itself is a computer. Whether it was a ploy or not you have assimilated many key ideas in computing and computer science. In this chapter, we will make some of these ideas more explicit and also give you a flavor for what *computer science* is really all about. As Dijkstra puts it, *computer science is no more about computers than astronomy is about telescopes.*

Computers are dumb

Say you are hosting an international exchange student in your home. Soon after her arrival you teach her the virtues of PB&J (Peanut Butter & Jelly) sandwiches. After keenly listening to you, her mouth starts to water and she politely asks you if you can share the recipe with her. You write it down on a piece of paper and hand it to her.

Do This: Go ahead; write down the recipe to make a PB&J sandwich.

Seriously, do try to write it down. We insist!

OK, now you have a recipe for making PB&J sandwiches.

Do This: Go ahead, use your recipe from above to make yourself a PB&J sandwich. Try to follow the instructions *exactly* as *literally* as you can. If you successfully managed to make a PB&J sandwich, congratulations! Go ahead and enjoy it. Do you think your friend will be able to follow your recipe and enjoy PB&J sandwiches?

You have to agree, writing a recipe for PB&J sandwiches seemed trivial at first, but when you sit down to write it you have no choice but to question several assumptions: does she know what peanut butter is? Should I recommend a specific brand? Ditto for jelly? By the way, did you forget to mention it was *grape jelly*? What kind of bread to use? Will it be pre-sliced? If not, you need a knife for slicing the loaf? Did you specify how thick the

slices should be? Should she use the same knife for spreading the peanut butter and the jelly? Etc. The thing is, at such a level of detail you can go on and on...does your friend know how to use a knife to spread butter or jelly on a slice? Suddenly, a seemingly trivial task becomes a daunting exercise. In reality, in writing down your recipe, you make several assumptions: she knows what a sandwich is, it involves slices of bread, spreading things on the slices, and slapping them together. There, you have a sandwich!

Think of the number of recipes that have been published in cookbooks all over the world. Most good cookbook authors start with a dish, write down a recipe, try it a number of times and refine it in different ways. Prior to publication, the recipe is tested by others. After all, the recipe is for others to follow and recreate a dish. The recipe is revised and adjusted based on feedback by recipe testers. The assumption that cookbook authors make is that you will have the competence to follow the recipe and recreate the dish to your satisfaction. It may never result in the same dish that the author prepared. But it will give you a base to improvise upon. Wouldn't it be nice if there was an exact way of following a recipe so that you end up with the exact same result as the cookbook author every time you made that dish? Would it? That may depend on your own tastes and preferences. Just for fun, here is a recipe for some yummy Saffron Chicken Kabobs.

Saffron Chicken Kabobs

Ingredients

1 lb boneless chicken breast, cubed into 1-2 inch pieces
1 medium onion, sliced
1 tsp saffron threads
1 lime
1 tbsp olive oil
Salt and black pepper to taste

Preparation

1. Mix the chicken and onions in a non-reactive bowl.
2. With your fingers crush and add saffron threads.
3. Add the juice of the lime, olive oil, and salt and pepper.
4. Marinade in the refrigerator for at least 30 min (or overnight).
5. Preheat a grill (or oven to 400 degrees).
6. Skewer kabobs, discarding the onion slices. Or place everything in a lined baking sheet if using oven.
7. Grill/bake for 12-15 min until done.

In cooking recipes, like the ones above, you can assume many things: they will be used by people (like you and me); they will be able to follow them; perhaps even improvise. For instance, in the recipe above, we do not specify that one will need a knife to cut the chicken, onions, or the lime; or that you will need a grill or an oven; etc. Most recipes assume that you will be able to *interpret* and follow the recipe as written.

Computer programs are also like recipes, to some extent. Think of the program you wrote for choreographing a robot dance, for instance. We have reproduced the version from Chapter 3 here:

```
# File: dance.py
# Purpose: A simple dance routine
# First import myro and connect to the robot

from myro import *
initialize("com5")

# Define the new functions...

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

def wiggle(speed, waitTime):
       motors(-speed, speed)
       wait(waitTime)
       motors(speed, -speed)
       wait(waitTime)
```

```
        stop()

# The main dance program
def main():
        print "Running the dance routine..."
        yoyo(0.5, 0.5)
        wiggle(0.5, 0.5)
        yoyo(1, 1)
        wiggle(1, 1)
        print "...Done"

main()
```

In many ways, this program above is like a recipe:

To do a robot dance

Ingredients

1 function `yoyo` for the robot to go back and forth at a given speed
1 function `wiggle` that enables the robot to wiggle at a given speed

Preparation

1. yoyo at speed 0.5, wait 0.5
2. wiggle at speed 0.5, wait 0.5
3. yoyo at speed 1, wait 1
4. wiggle at speed 1, wait 1

Further, you could similarly specify the steps involved in doing the `yoyo` and `wiggle` motions as a recipe. This may seem like a trivial example, but it makes two very important points: a computer program is like a recipe in that it lays out the list of ingredients and a method or steps for accomplishing the given task; and, like a recipe, its ingredients and the steps require careful pre-planning and thought. Importantly, computer programs are different from recipes in one aspect: they are designed to be followed by a computer!

A computer is a dumb device designed to follow instructions/recipes. We will save the technical details of how a computer does what it does for a later course. But it is almost common knowledge that everything *inside* is represented as 0's and 1's. Starting from 0's and 1's one can design encoding schemes to represent numbers, letters of the alphabet, documents, images, movies, music, etc. and whatever other abstract entities you would like to manipulate using a computer. A computer program is ultimately also represented as a sequence of 0's and 1's and it is in this form that most computers like to follow recipes. However limiting or degenerate this might sound it is the key to the power of computers. Especially when you realize that it is this simplification that enables a computer to manipulate hundreds of millions of pieces of information every second. The price we have to pay for all this power is that we have to specify our recipes as computer programs in a rather formal and precise manner. So much so that there is no room for improvisation: no pinch of salt vagaries, as in cooking recipes, is acceptable. This is where *programming languages* come in. Computer scientists specify their computational recipes using *programming languages*. You have been using the programming language Python to write your robot programs. Other examples of programming languages are Java, C++ (pron.: sea plus plus), C# (pron.: sea sharp), etc. There are well over 2000 programming languages in existence!

Do This: Can you find out how many programming languages there are? What are the ten most commonly used programming languages?

Prior to the existence of programming languages computers were programmed using long sequences of 0's and 1's. Needless to say it drove several people crazy! Programming languages, like Python, enable a friendlier way for programmers to write programs. Programming languages provide easy access to encodings that represent the kinds of things we, humans, relate to. For example, the Python statement:

```
meaningOfLife = 42
```

is a command for the computer to associate the value, 42 with the name `meaningOfLife`. This way, we can ask the computer to check that it is indeed 42:

```
if meaningOfLife == 42:
     speak("Eureka!")
else:
     speak("What do we do now?")
```

Once again, it would be good to remind you that the choice of the name, `meaningOfLife`, doesn't really mean that we are talking about *the meaning of life*. We could as well have called it `timbuktoo`, as in:

```
timbuktoo = 42
```

You see, computers are truly dumb!

It is really up to us, the programmer, to ensure that we use our names consistently and choose them, in the first place, carefully. But, by creating a language like Python, we have created a formal notation so that when translated into 0's and 1's each statement will mean only one thing, no other interpretations. This makes them different from a cooking recipe.

Robot goes to buy fresh eggs

Recipes, however, form a good conceptual basis for starting to think about a program to solve a problem. Say, you have in mind to make your favorite *Apple Strudel*. You know you will need apples. Perhaps it is the apple season that prompted the thought in the first place. And pastry. But when to get down to it, you will need that recipe you got from your grandma.

Whenever we are asked to solve a problem using a computer, we begin by laying out a rough plan for solving the problem. That is, sketch out a strategy. This is further refined into specific steps, perhaps even some variables are identified and named, etc. Once you convince yourself that you have a way of solving the problem, what you have is an *algorithm*

The idea of an algorithm is central to computer science so we will spend some time here developing this notion. Perhaps the best way to relate to it is by an example. Assume that a robot goes into a grocery store to buy a dozen fresh eggs. Assuming it is capable of doing this, how will it ensure that it has selected the freshest eggs available?

Your personal robot is probably not up to this kind of task but imagine that it was. Better yet, leave the mechanics aside, let us figure out how *you* would go and buy the freshest eggs. Well, you would somehow need to know what today's date is. Assume it is September 15, 2007 (why this date? it'll become clear soon!). Now you also know that egg cartons typically carry a freshness date on them. In fact, USDA (the United States Department of Agriculture) offers voluntary, no cost, certification programs for egg farms. An egg farmer can volunteer to participate in USDA's egg certification program whereby the USDA does regular inspections and also provides help in categorizing eggs by various sizes. For example, eggs are generally classified as Grade AA, Grade A, or Grade B. Most grocery stores carry Grade A eggs. They can also come in various sizes: Extra Large, Large, Small, etc. What is more interesting is that the carton labeling system has some very useful information encoded on it.

Egg Carton Labeling

Every USDA certified egg carton has at least three pieces of information (see picture on right): a "sell by" date (or a "use by date" or a "best by" date), a code identifying the specific farm the eggs came from, and a date on which the eggs were packed in that carton. Most people buy eggs by looking at the "sell by" date or the "best by" date.

However the freshness information is really encoded in the packed on date. To make things more confusing, this date is encoded as the day of the year.

For example, take a look at the top carton shown on the right. Its "sell by" date is October 4. "P1107" is the farm code. This carton was packed on the 248th day of the year. Further, USDA requires that all certified eggs be packed within 7 days of being laid. Thus, the eggs in the top carton were laid somewhere between day 241 and day 248 of 2007. What dates correspond to those dates?

Next, look at the bottom carton. Those eggs have a later "sell by" date (October 18) but an earlier packed date: 233. That is those eggs were laid somewhere between day 226 and day 233 of 2007.

Which eggs are fresher?

Even though the "sell by" date on the second carton is two weeks later, the first carton contains fresher eggs. In fact, the eggs in the upper carton were laid at least two weeks later!

The packed on date is encoded as a 3-digit number. Thus eggs packed on January 1 will be labeled: 001; eggs packed on December 31, 2007 will be labeled: 365.

Do This: Go to the USDA web site (www.usda.gov) and see if you can find out which farm the two eggs cartons came from.

For a robot, the problem of buying the freshest eggs becomes that of figuring out, given a packed on date, what the date was when the eggs were packed?

Fasten your seatbelts, we are about to embark on a unique computational voyage...

Designing an algorithm

So far, we have narrowed the problem down to the following specifications:

Input

3-digit packed on date encoding

Output

Date the eggs were packed

For example, if the packed on date was encoded as 248, what will be the actual date?

Well, that depends. It could be September 4 or September 5 depending on whether the year was a leap year or not. Thus, it turns out, that the problem above also requires that we know which year we were talking about. Working out one or two sample problems is always a good idea because it helps identify missing information that may be critical to solving the problem. Given that we do need to know the year, we can ask the user to enter that at the same time the 3-digit code is entered. The problem specification then becomes:

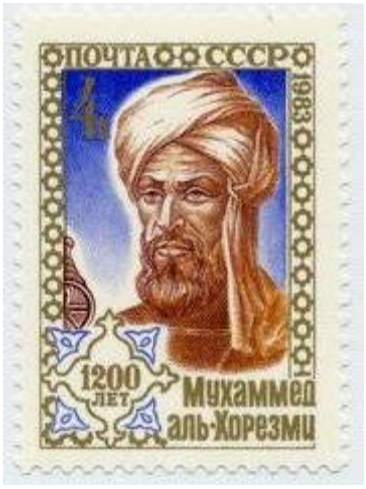
Input

3-digit packed on date encoding

Current year

The Etymology of Algorithm

The word algorithm, an anagram of logarithm, is believed to have been derived from Al-Khowarizmi, a mathematician who lived from 780-850 AD. His full name was Abu Ja'far Muḥammad ibn Mūsā al-Khwārizmī, (Mohammad, father of Jafar, son of Moses, a Khwarizmian). Much of the mathematical knowledge of medieval Europe was derived from Latin translations of his works



In 1983, The Soviet Union issued the stamp shown below in honor of his 1200th anniversary.

Output

Date the eggs were packed

Example:

Input: 248, 2007

Output: The eggs were packed on September 5, 2007

Any ideas as to how you would solve this problem? It always helps to try and do it yourself, with pencil and paper. Take the example above, and see how you would arrive at the output date. While you are working it out, try to write down your problem solving process. Your algorithm or recipe will be very similar.

Suppose we are trying to decode the input 248, 2007. If you were to do this by hand, using a pen and paper, the process might go something like this:

```
The date is not in January because it has 31 days and 248 is
much larger than 31.
Lets us subtract 31 out of 248: 248 - 31 = 217

217 is also larger than 28, the number of days in February,
2007.
So, let us subtract 28 from 217: 217 - 28 = 189

189 is larger than 31, the number of days in March.
Subtract 31 from 189: 189 - 31 = 158

158 is larger than 30, the number of days in April.
So: 158 - 30 = 128

128 is larger than 31, the number of days in May.
Hence: 128 - 31 = 97

97 is larger than 30, the number of days in June.
97 - 30 = 67

67 is larger than 31, the number of days in July.
67 - 31 = 36
```

```
36 is larger than the number of days in August (31).
36 - 31 = 5

5 is smaller than the number of days in September.
Therefore it must be the 5th day of September.

The answer is: 248th day of 2007 is September 5, 2007.
```

That was obviously too repetitious and tedious. But that is where computers come in. Take a look at the process above and see if there is a pattern to the steps performed. Sometimes, it is helpful to try another example.

Do This: Suppose the input day and year are: 56, 2007. What is the date?

When you look at the sample computations you have performed, you will see many patterns. Identifying these is the key to designing an algorithm. Sometimes, in order to make this easier, it is helpful to identify or name the key pieces of information being manipulated. Generally, this begins with the inputs and outputs identified in the problem specification. For example, in this problem, the inputs are: day of the year, current year. Begin by assigning these values to specific variable names. That is, let us assign the name `day` to represent the day of the year (248 in this example), and `year` as the name to store the current year (2007). Notice that we didn't choose to name any of these variables `timbuktu` or `meaningOfLife`!

Also, notice that you have to repeatedly subtract the number of days in a month, starting from January. Let us assign a variable named, `month` to keep track of the month under consideration.

Next, you can substitute the names `day` and `year` in the sample computation:

```
Input:
   day = 248
   year = 2007

# Start by considering January
month = 1
```

```
The date is not in month = 1 because it has 31 days and 248 is
much larger than 31.
day = day - 31

# next month
month = 2

day (= 217) is also larger than 28, the # of days in month = 2
day = day - 28

# next month
month = 3
day (= 189) is larger than 31, the # of days in month = 3.
day = day - 31

# next month
month = 4
day (= 158) is larger than 30, the # of days in month = 4.
day = day - 30

# next month
month = 5
day (= 128) is larger than 31, the # of days in month = 5.
day = day - 31

# next month
month = 6
day (= 97) is larger than 30, the # of days in month = 6.
day = day = 30

# next month
month = 7
day (= 67) is larger than 31, the # of days in month = 7.
day = day - 31

# next month
month = 8
day (= 36) is larger than the # of days in month = 8.
day = day - 31

# next month
month = 9
day (= 5) is smaller than the # of days in month = 9.
```

```
Therefore the it must be the 5th day of September.

The answer is: 9/5/2007
```

Notice now how repetitious the above process is. The repetition can be expressed more concisely as shown below:

```
Input:
   day
   year

# start with month = 1, for January
month = 1
repeat
   if day is less than #days in month
      day = day - #days in month
      # next month
      month = month + 1
   else
      done

Output: day/month/year
```

It is now starting to look like a recipe or an algorithm. Go ahead and try it with the sample inputs from above and ensure that you get correct results. Additionally, make sure that this algorithm will work for boundary cases: 001, 365, etc.

Thirty days hath September

We can refine the algorithm above further: one thing we have left unspecified above is the computation of the number of days in a month. This information has to be made explicit for a computer to be able to follow the recipe. So, how do we compute the number of days in a month? The answer may seem simple. Many of you may remember the following poem:

Thirty days hath September
April, June, and November
All the rest have thirty-one
Except for February alone
Which hath twenty-eight days clear
And twenty-nine in each leap year

From a design perspective, we can assume that we have an ingredient, a function in this case, called `daysInMonth` that, given a month and a year will compute and return the number of days in the month. That is, we can refine our algorithm above to the following:

```
Ingredients:
   1 function daysInMotnh(m, y): returns the # days in month,
m in year y.

Input:
   day
   year

# start with month = 1, for January
month = 1
repeat
   if day is less than #days in month
      day = day - daysInMonth(month, year)
      # next month
      month = month + 1
   else
      done

Output: day/month/year
```

Now, we do have to solve the secondary problem:

```
Input
  month, M
  year, Y
```

Output

Number of days in month, M in year, Y

On the surface this seems easy, the poem above specifies that April, June, September, and November have 30 days, and the rest, with the exception of February have 31. February has 28 or 29 days depending upon whether it falls in a leap year or not. Thus, we easily elaborate a recipe or an algorithm for this as follows:

```
Input:
   m, y

if m is April (4), June(6), September(9), or November (11)
   days = 30
else if m is February
         if y is a leap year
              days = 29
         else
              days = 28
else
   (m is January, March, May, July, August, October, December)
    days = 31

Output:
   days
```

This still leaves out one more detail: how do we tell if y is a leap year?

First, try and answer the question, *what is a leap year?*

Again, we can refine the algorithm above by assuming that we have another ingredient, a function: `leapYear`, that determines if a given year is a leap year or not. Then we can write the algorithm above as:

```
Ingredients:
   1 function leapYear(y)
     returns True if y is a leap year, false otherwise
```

```
Input:
   m, y

if m is April (4), June(6), September(9), or November (11)
   days = 30
else if m is February
         if leapYear(y)
              days = 29
         else
              days = 28
else
   (m is January, March, May, July, August, October, December)
    days = 31

Output:
   days
```

Most of us have been taught that a leap year is a year that is divisible by 4. That is the year 2007 is not a leap year, since 2007 is not divisible by 4, but 2008 is a leap year, since it is divisible by 4.

Do This: How do you determine if something is divisible by 4? Try your solution on the year 1996, 2000, 1900, 2006.

Leap Years: Papal Bull

To design a recipe or an algorithm that determines if a number corresponding to a year is a leap year or not is straightforward if you accept the definition from the last section. Thus, we can write:

Input

y, a year

Output

True if y is a leap year, false otherwise

Method

```
if y is divisible by 4
  it is a leap year, or True
else
  it is not a leap year, or False
```

However, this is not the complete story. The western calendar that we follow is called the *Gregorian Calendar* which was adopted in 1582 by a Papal Bull issued by Pope Gregory XIII. The Gregorian Calendar defines a leap year, by adding an extra day, every fourth year. However, there is a 100-year correction applied to it that makes the situation a little more complicated: Century years are not leap years except when they are divisible by 400. That is the years 1700, 1800, 1900, 2100 are not leap years even though they are divisible by 4. However, the years 1600, 2000, 2400 are leap years. For more information on this, see the exercises at the end of the chapter. Our algorithm for determining if a year is a leap year can be refined as shown below:

```
Input
   y, a year

   if y is divisible by 400
      it is a leap year, or True
   else if y is divisible by 100
      it is not a leap year, or False
   else if y is divisible by 4
      it is a leap year, or True
   else
      it is not a leap year, or False
```

Finally, we have managed to design all the algorithms or recipes required to solve the problem. You may have noticed that we used some familiar constructs to elaborate our recipes or algorithms. Next, let us take a quick look at the essential constructs that are used in expressing algorithms.

Essential components of an algorithm

Computer scientists express solutions to problems in terms of algorithms, which are basically more detailed recipes. Algorithms can be used to express any solution and yet are comprised of some very basic elements:

1. Algorithms are step-by-step recipes that clearly identify the inputs and outputs
2. Algorithms name the entities that are manipulated or used: variables, functions, etc.
3. Steps in the algorithm are followed in the order they are written (from top to bottom)
4. Some steps can specify decisions (if-then) over the choice of some steps
5. Some steps can specify repetitions (loops) of steps
6. All of the above can be combined in any fashion.

Computer scientists claim that solutions/algorithms to *any* problem can be expressed using the above constructs. You do not need any more! This is a powerful idea and it is what makes computers so versatile. From a larger perspective, if this is true, then these can be used as tools for thinking about any problem in the universe. We will return to this later in the chapter.

Programming Languages

Additionally, as you have seen earlier, in writing Python programs, programming languages (Python, for example) provide formal ways of specifying the essential components of algorithms. For example, the Python language provides a way for you to associate values to variables that you name, it provides a sequential way of encoding the steps, it provides the if-then conditional statements, and also provides the while-loop and for-loop constructs for expressing repetitions. Python also provides means for defining functions and also ways of organizing groups of related functions into libraries or modules which you can import and use as needed. As an example,

we provide below, the Python program that encodes the `leapYear` algorithm shown above:

```
def leapYear(y):
   '''Returns true if y is a leap year, false otherwise.'''

   if y %400 == 0:
      return True
   elif y % 100 == 0:
      return False
   elif y % 4 == 0:
      return True
   else:
      return False
```

The same algorithm, when expressed in C++ (or Java) will look like this:

```
bool leapYear(int y) {
   // Returns true if y is a leap year, false otherwise.
   if (y % 400 == 0)
      return true
   else if (y % 100 == 0)
      return false
   else if (y % 4 == 0)
      return true
   else
      return false
}
```

As you can see, there are definite syntactic variations among programming languages. But, at least in the above examples, the coding of the same algorithm looks very similar. Just to give a different flavor, here is the same function expressed in the programming language CommonLisp.

```
(defun leapYear (y)
   (cond
     ((zerop (mod y 400)) t)
     ((zerop (mod y 100)) nil)
     ((zerop (mod y 4)) t)
     (t nil)))
```

Again, this may look weird, but it is still expressing the same algorithm.

What is more interesting is that given an algorithm, there can be many ways to encode it, even in the same programming language. For example, here is another way to write the `leapYear` function in Python:

```
def leapYear(y):
   '''Returns true if y is a leap year, false otherwise.'''

   if ((y % 4 == 0) and (y % 100 != 0)) or (y % 400 == 0):
      return True
   else:
      return False
```

Again, this is the same exact algorithm. However, it combines all the tests into a single condition: y is divisible by 4 or by 400 but not by 400. The same condition can be used to write an even more succinct version:

```
def leapYear(y):
   '''Returns true if y is a leap year, false otherwise.'''

  return ((y % 4 == 0) and (y % 100 != 0)) or (y % 400 == 0)
```

That is, return whatever the result is (`True`/`False`) of the test for `y` being a leap year. In a way, expressing algorithms into a program is much like expressing a thought or a set of ideas in a paragraph or a narrative. There can be many ways of encoding an algorithm in a programming language. Some seem more natural, and some more poetic, or both, and, like in writing, some can be downright obfuscated. As in good writing, good programming ability comes from practice and, more importantly, learning from reading well written programs.

From algorithms to a working program

To be able to solve the fresh eggs problem, you have to encode all the algorithms into Python functions and then put them together as a working program. Below, we present one version:

```
# File: fresheggs.py

def leapYear(y):
   '''Returns true if y is a leap year, false otherwise.'''

   return ((y % 4 == 0) and (y % 100 != 0)) or (y % 400 == 0)

def daysInMonth(m, y):
   '''Returns the number of days in month, m (1-12)
      in year, y.'''

   if (m == 4) or (m == 6) or (m == 9) or (m == 11):
      return 30
   elif m == 2:
      if leapYear(y):
         return 29
      else:
         return 28
   else:
      return 31

def main():
   '''Given a day of the year (e.g. 248, 2007),
      convert it to the date (i.e. 9/5/2007)'''

   #Input: day, year
   day, year = input("Enter the day, year: ")

   # start with month = 1, for January
   month = 1

   while day > daysInMonth(month, year):
      day = day - daysInMonth(month, year)

      # next month
      month = month + 1

   # done, Output: month/day/year
   print "The date is: %1d/%1d/%4d" % (month, day, year)

main()
```

If you save this program in a file, `freshheggs.py`, you will be able to run it and test it for various dates. Go ahead and do this. Here are some sample outputs:

```
Enter the day, year: 248, 2007
The date is: 9/5/2007

>>> main()
Enter the day, year: 12, 2007
The date is: 1/12/2007

>>> main()
Enter the day, year: 248, 2008
The date is: 9/4/2008

>>> main()
Enter the day, year: 365, 2007
The date is: 12/31/2007

>>> main()
Enter the day, year: 31, 2007
The date is: 1/31/2007
```

All seems to be good. Notice how we tested the program for different input values to confirm that our program is producing correct results. It is very important to test your program for a varied set of input, taking care to include all the *boundary* conditions: first and last day of the year, month, etc. Testing programs is a fine art in itself and several books have been written about the topic. One has to ensure that all possible inputs are tested to ensure that the behavior of the program is acceptable and correct. You did this with your robot programs by repeatedly running the program and observing the robot's behavior. Same applies to computation.

Testing and Error Checking

What happens, if the above program receives inputs that are outside the range? What if the user enters the values backwards (e.g. 2007, 248 instead of 248, 2007)? What if the user enters her name instead (e.g. Paris, Hilton)? Now

is the time to try all this out. Go ahead and run the program and observe its behavior on some of these inputs.

Ensuring that a program provides acceptable results for all inputs is critical in most applications. While there is no way to avoid what happens when a user enters his name instead of entering a day and a year, you should still be able to safeguard your programs from such situations. For example:

```
>>> main()
Enter the day, year: 400, 2007
That corresponds to the date: 14/4/2007
```

Obviously, we do not have a month numbered 14!

The thing that comes to rescue here is the realization that, it is your program and the computer will only carry out what you have expressed in the program. That is, you can include *error checking* facilities in your program to account for such conditions. In this case, any input value for day that is outside the range 1..365 (or 1..366 for leap years) will not be acceptable. Additionally, you can also ensure that the program only accepts years greater than 1582 for the second input value. Here is the modified program (we'll only show the `main` function):

```
def main():
   '''Given a day of the year (e.g. 248, 2007), convert
      it to the date (i.e. 9/5/2007)'''

   #Input: day, year
   day, year = input("Enter the day, year: ")

   # Validate input values...
   if year <= 1582:
       print "I'm sorry. You must enter a valid year
              (one after 1582). Please try again."
       return
   if day < 1:
       print "I'm sorry. You must enter a valid day
             (1..365/366). Please try again."
```

```
        return
    if leapYear(year):
        if day > 366:
            print "I'm sorry. You must enter a valid day
                  (1..365/366). Please try again."
            return
    elif day > 365:
        print "I'm sorry. You must enter a valid day
              (1..365/366). Please try again."
        return

    # input values are safe, proceed...
    # start with month = 1, for January
    month = 1

    while day > daysInMonth(month, year):
       day = day - daysInMonth(month, year)

       # next month
       month = month + 1

    # done, Output: month/day/year
    print "The date is: %1d/%1d/%4d" % (month, day, year)

main()
```

Here are the results of some more tests on the above program.

```
Enter the day, year: 248, 2007
The date is: 9/5/2007

>>> main()
Enter the day, year: 0, 2007
I'm sorry. You must enter a valid day (1..365/366). Please try
again.

>>> main()
Enter the day, year: 366, 2007
I'm sorry. You must enter a valid day (1..365/366). Please try
again.

>>> main()
Enter the day, year: 400, 2007
```

```
I'm sorry. You must enter a valid day (1..365/366). Please try
again.

>>> main()
Enter the day, year: 248, 1492
I'm sorry. You must enter a valid year (one after 1582).
Please try again.

>>> main()
Enter the day, year: 366, 2008
The date is: 12/31/2008
```

Starting from a problem description it is a long and carefully planned journey that involves the development of the algorithm, the encoding of the algorithm in a program, and finally testing and improving the program. In the end you are rewarded not just by a useful program, you have also honed your general problem solving skills. Programming forces you to anticipate unexpected situations and to account for them prior to encountering them which itself can be a wonderful life lesson.

Modules to organize components

Often, in the course of designing a program, you end up designing components or functions that can be used in many other situations. For example, in the problem above, we wrote functions `leapYear` and `daysInMonth` to assist in solving the problem. You will no doubt agree that there are many situations where these two functions could come in handy (see Exercises below). Python provides the *module* facility to help organize related useful functions into a single file that you can then use over and over whenever they are needed. For example, you can take the definitions of the two functions and put them separately in a file called, `calendar.py`. Then, you can *import* these functions whenever you need them. You have used the Python `import` statement to import functionality from several different modules: `myro`, `random`, etc. Well, now you know how to create your own. Once you create the `calendar.py` file, you can import it in the `fresheggs.py` program as shown below:

```
from calendar import *

def main():
   '''Given a day of the year (e.g. 248, 2007), convert
      it to the date (i.e. 9/5/2007)'''

   #Input: day, year
   day, year = input("Enter the day, year: ")

   # Validate input values...
   if year <= 1582:
       print "I'm sorry. You must enter a valid year
             (one after 1582). Please try again."
       return
   if day < 1:
       print "I'm sorry. You must enter a valid day
             (1..365/366). Please try again."
       return
   if leapYear(year):
       if day > 366:
           print "I'm sorry. You must enter a valid day
                 (1..365/366). Please try again."
           return
   elif day > 365:
       print "I'm sorry. You must enter a valid day
             (1..365/366). Please try again."
       return
   # input values are safe, proceed...
   # start with month = 1, for January
   month = 1

   while day > daysInMonth(month, year):
      day = day - daysInMonth(month, year)

      # next month
      month = month + 1

   # done, Output: month/day/year
   print "The date is: %1d/%1d/%4d" % (month, day, year)

main()
```

Homer contemplates P = NP?

The second equation on the right is Euler's Equation.

It may have also occurred to you by now that for any given problem there may be many different solutions or algorithms. In the presence of several alternative algorithms how do you decide which one to choose? Computer Scientists have made it their primary business to develop, analyze, and classify algorithms to help make these decisions. The decision could be based on ease of programming, efficiency, or the number of resources it takes for a given algorithm. This has also led computer scientists to create a classification of problems: from easy to hard, but in a more formal sense. Some of the hardest open questions in the realm of problems and computing lie in this domain of research. We will not go into details here, but these questions have even shown up in several popular TV shows (see picture here).

Summary

OK, so this chapter is not quite complete…

Myro review

No new Myro freatures were introduced in this chapter.

Python Review

The only new Python feature introduced in this chapter was the creation of modules. Every program you create can be used as a library module from which you can import useful facilities.

Exercises