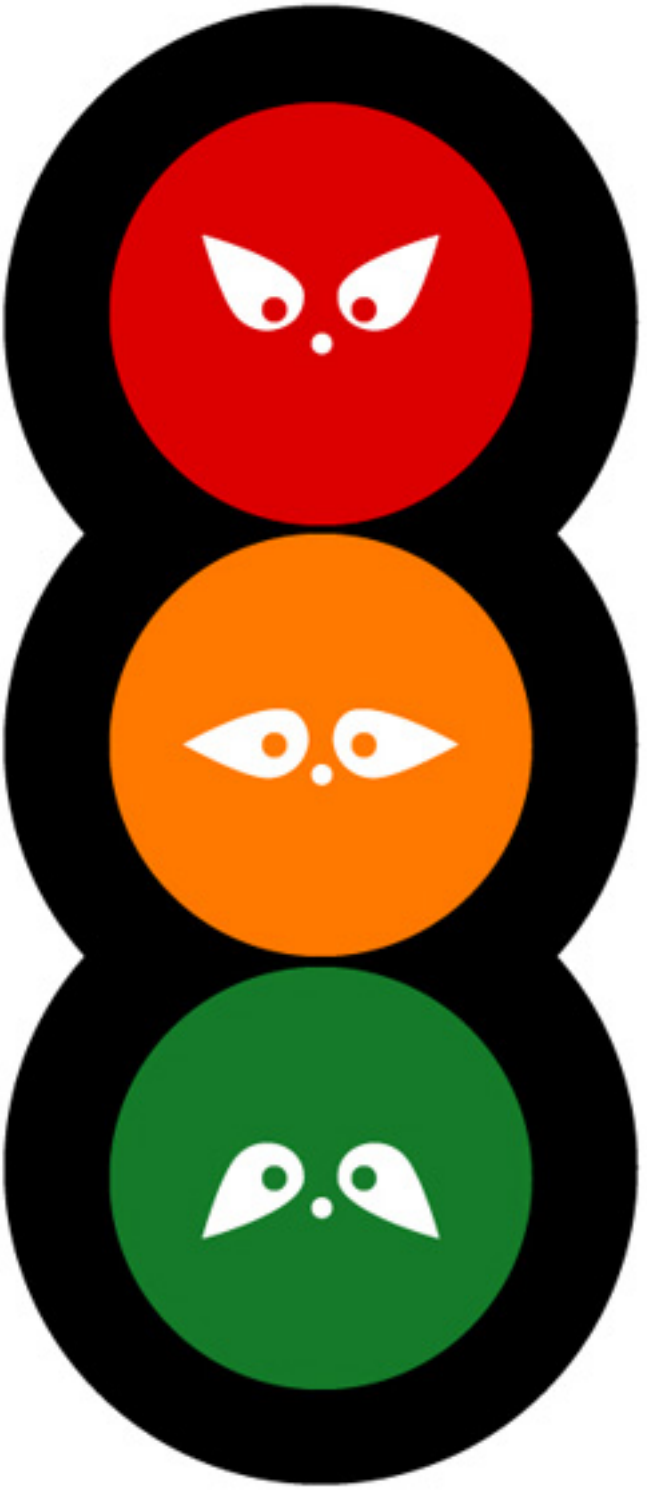


# 7

# Behavior Control

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

Opposite page: My Traffic Lights
Illustration by Yingshun Wong (yingshun.co.uk)

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:

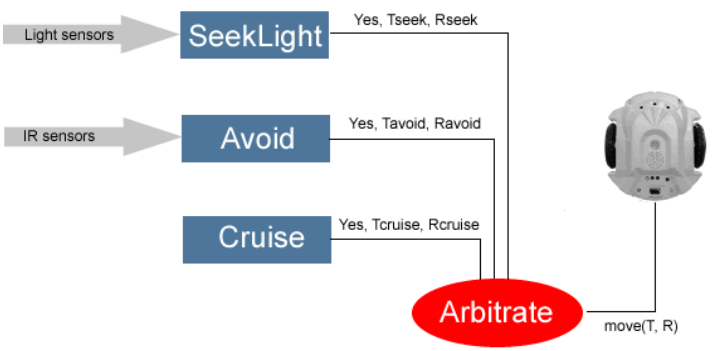
```
def main():
    # do forever or for some time
    # or until a certain condition is satisfied

       # 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 by 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., $\log_e x$).

**`log10(x)`** Returns the base-10 logarithm of x (i.e. $\log_{10} x$).

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

**`sqrt(x)`** Returns the square root of x ($\sqrt{x}$).

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 11, 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 that 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

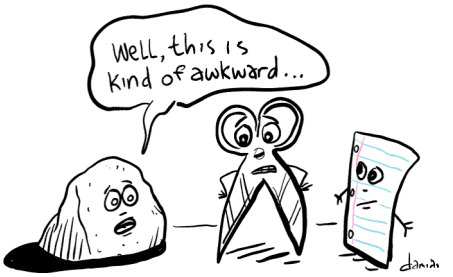
Decision making is central to all computer programs. In fact there is a famous theorem in computer science that says that if any programmable device is capable of being programmed to do *sequential execution*, *decision making*, and *repetition*, it is capable of expressing any computable algorithm. 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.

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 (indicating 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:




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/

```
# 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 `randint` 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[randint(0,2)]
```

Recall that `randint(n, m)` returns a random numbers in the range `[n..m]`. Thus, `randint(0, 2)` 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 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.", 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., $\log_e x$).

**`log10(x)`** Returns the base-10 logarithm of x (i.e. $\log_{10} x$).

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

**`sqrt(x)`** Returns the square root of x ($\sqrt{x}$).

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 π.

**`e`** The mathematical constant *e*.

## Exercises

**1.** Replace the Avoid module in the behavior-based control program with a module called: follow. The follow module tries to detect a wall to the robot's right (or left) and then tries to follow it at all times. Observe the resulting behavior.

**2.** Do Population growth by modeling it as a difference equation.

**3.** Do population growth by modeling it as a continuous growth function.

**4.** Instead of using the statement:

```
yourChoice = items[randint(0,2)]
```

use the command:

```
yourChoice = choice(items)
```

`choice(<list>)` is another function defined in the random module. It randomly selects an element from the supplied list.

**5.** Can you make the program better?

**6.** Implement the HiLo game: the computer picks a random number between 0 and 1000 and the user has to try and guess it. With every guess the computer informs the user f their guess was high (Hi), low (Lo), or they got it.

**7.** Reverse the roles in the HiLo game from above. This time, the user guesses a number between 0 and 1000 and computer program has to guess it. Think about a good strategy for guessing the number based on high and low clues, and then implement that strategy.