Python for Fun

Purpose of this Collection

This collection is a presentation of several small Python programs. They are aimed at intermediate programmers; people who have studied Python and are fairly comfortable with basic recursion and object oriented techniques. Each program is very short, never more than a couple of pages and accompanied with a write-up.

I have found Python to be an excellent language to express algorithms clearly. Some of the ideas here originated in other programs in other languages. But in most cases I developed code from scratch from just an outline of an idea. However "Lisp in Python" was almost a translation exercise from John McCarthy's original "Evalquote in Lisp".

From many years of programming these are some of my favorite programs. I hope you enjoy them as much as I do. I look forward to hearing from readers, especially folks with suggestions for improvements, ideas for new projects, or people who are doing similar things. You can email me at mailme.html

Many thanks to Paul Carduner and Jeff Elkner for their work on this page, especially for Paul's graphic of "Psyltherin" (apologies to Harry Potter) and to the Twisted developement team for their Lore documentation generator to which all the other web pages in this collection have been recently adapted.

Chris Meyers

Python for Fun

The Collection

Items with a "*" have been recently added or updated

Table of Content

Programming a video game

This study concerns the development of a little video game. It's modeled on a game my family enjoyed in the 1980's called Lode Runner. We would sometimes play it by the hour, having a good time and wearing out keyboards. But Lode Runner, at least the version we had, assumed the cpu ran at a certain speed and once we had processors faster than about 8Mhz, the game was too fast to be played.

Lode runner consists of a vertical "board" containing ladders and "catwalks" between the ladders. In the original game there were about 100 different boards or levels, each one just a bit harder than the last. "You" were represented by a little person going up and down the ladders and running across the catwalks gathering the "lodes" of treasure. You used the arrow keys to indicate a change of direction. While doing this you were pursued by what I always thought of as robots. You won a level by retrieving all the lodes and getting to the top of the board before being tagged by a robot. You had one weapon at your disposal. You could burn a hole in a catwalk that a robot would fall into and get stuck. The robot would take a couple of seconds to get back out. And in a few seconds more the catwalk would self repair. If you fell into the hole you couldn't get out and with the repair tragedy was the result.

We are going to develop a game quite similar, using simplified graphics and keyboard input. The code only works for Linux and Python 2.0 or newer. This is because of the ansii escape sequences we'll use to display the game on the screen and the special termio module for monitoring the keyboard.

The nature of game programs

Game programs, actually games in general whether Lode Runner or Chess, have a certain structure. A game is a set of moves. There must be a way to visualize the game in progress (screen), a way to get input from the player(s), rules that determine what moves are possible and to determine when the game is over.

At the center of any game program is the "game loop" that repeatably performs the above steps until the game is terminated, that is, won, lost or otherwise interrupted.

Moves in chess require each player to examine the board, decide on the best next move and take it. Moves in Lode Runner are a little more dynamic. Each player (you and the robots) must be advanced one position and each player must have the opportunity to change direction. A certain time should elapse between moves so that you, the human, can keep up. Finally, it needs to be determined when the game is over. This will happen if a robot tags you or, as we'll see, if you fall off the board. You will have won if you gathered all the lodes before this happens.

We will build the program in stages so we can concentrate one aspect at a time. We'll start with some unusual I/O to the terminal.

Terminal Escape Sequences

Each character in the Ascii character set has a numeric value. You can use the "ord" function to find out what that value is.

```
>>> ord('A')
65
```

The reverse of the "ord" function is "chr". It will turn a numeric value from 0 to 255 into the corresponding character.

```
>>> chr(65)
'A'
```

Some characters do not display and in order to represent them in a Python string they must be quoted with the "\" and an octal (base 8) number. The escape key (ESC) has a numeric value of 27 decimal. In a Python string it looks like

```
>>> chr(27)
'\033'
```

You may be used to using ESC in Vim editor to slip out of insert mode. It is also the basis for escape sequences that do special things. Try this.

```
>>> print "\033[1;1H\033[JHello World"
```

Your screen or window should have erased and then printed "Hello World" in the top left corner, and leaving a new ">>>" prompt on the second line.

There are two escape sequences in the above string. "\033[1;1H" positions the cursor to row one and column one. The "1;1" may be replaced with other numbers. "\033[J" will erase the screen from the current cursor position.

Escape sequences are also used from the keyboard for function keys and the four arrow keys. The up arrow generates a string "\033[A", the down key "\033[B", the right key "\033[C" and the left key "\033[D". However you need to be in a special terminal mode to see the character sequence faithfully reproduced.

Using the termios module

We have two problems with the keyboard in a arcade-like game. One is to input the arrow keys so that the program can change your direction on the fly. The second is a little more subtle. While waiting for you to type a key the game still needs to keep going. If you use input functions like `input` or `raw_input` then you know that the program stops until you input a string followed by the return key. That won't work here.

The solution, which is specific to Unix implementations, consists of 3 parts. The first is to put the keyboard input into a mode called "non-canonical". Canonical mode means that you must input an entire line before the program sees any of it. That allows you to edit the line as you are typing without your program having to worry about seeing backspace characters followed by corrections. In non-canonical input your program gets characters just as soon as they are typed.

The second part of the solution is to input keystrokes without the system trying to "echo" them onto the screen. An up arrow will often show on the screen like "^[[A" which we don't want to see. Our program just wants to deal with the fact that you pressed the up arrow.

The final part of the solution lets the program keep going whether or not you press a key on each move. And at 10 moves a second this is of course a must. The secret is to use the timeout function, but set the timeout to zero. This will allow us to read what is in the look ahead buffer, that is anything typed since the last move 1/10th second earlier.

These ideas are encapsulated in the module ttyLinux.py which you should now examine.

There are 3 functions for the keyboard input. The function "setSpecial" puts the keyboard into our non-canonical, non-echo, lookahead mode. But first it saves the prior settings so that the function "setNormal" can restore them. I don't want to go into detail on the "termios" module but, if you are interested, "man termios" will give you lots of information. Suffice it to say that setSpecial turns individual binary bits off in a flag word and sets a byte value for the timeout function. The function "readLookAhead" simply reads a maximum of three characters, enough to hold an arrow key escape sequence.

You should check at this time whether termios is available in your Python.

```
>>> import termios
>>>
```

If you don't get any error message then everything is ok. Otherwise you will have to rebuild and reinstall Python to include it. Briefly, you must edit Modules/Setup and uncomment the line containing "termios". Then do a "make" from the top directory followed by a "make install". If this means nothing to you, check with your system administrator.

Let's look at a little program to use ttyLinux.py. The program key1.py will set the terminal to our special mode, retrieve look ahead for 10 one second intervals and then restore the terminal to normal. Here is a sample run. After the import I typed the up arrow, down arrow and then the string "this is a test of everything". You can see how it reads 3 characters at a time.

```
>>> import key1
>>> key1.test()
Got ['\033[A']
Got ['\033[B']
Got ['thi']
Got ['s i']
Got ['s a']
Got [' te']
Got ['st ']
Got ['of ']
Got ['eve']
Got ['ryt']
>>>
```

There is a problem with key1.py which is annoying. If the loop doesn't complete for any reason and ttyLinux.setNormal() is not called (say you typed a control-C) then your terminal is left in the special state even though control has returned to Python or to the shell. You can still issue commands, they just don't echo. Nor can you correct mistakes. One way out is to get to the shell and "exit" to the login prompt. A new login will reset the terminal characteristics.

But a better way to do this can be found in key2.py and this is the model we'll use in the game program. Here we use a try/finally clause (no except) to guarentee that setNormal() is called whether or not the call to loop() completes normally. Without an "except" clause we still get any traceback messages.

```
>>> import key2
>>> key2.test()
Got ['\033[A']
Got ['\033[B']   (Control-C typed here)
Traceback (most recent call last):
File "stdin", line 1, in ?
File "key2.py", line 14, in test
loop()
File "key2.py", line 7, in loop
time.sleep(1)
KeyboardInterrupt
>>> # The keyboard still works fine!□
```

Dealing with the board and screen.

The module boards.py contains our board as a list of python strings, each string representing one row on the screen. This format is convenient for two reasons. One, it is easy to edit the board since the rows are aligned vertically and, two, reading the board into the program is done by simply importing the module. If you want to edit the board with "vi" or "vim", use the "R" command to get into "replace" mode. Within the strings '_' are catwalks, '|' are ladders and '*' are the lodes of treasure. The players are not represented on the initial board.

The list "boards" references board0 in a single item. This is meant to leave room for expansion to allow you to have several boards and let the game switch between them. In the traditional Lode Runner game winning at one level automatically advanced you to the next.

The module util.py has some functions to automatically handle the board and the screen so that the main game program does not have to worry about petty details. The function setBoard sets the global "board" to a fresh copy of the nth board in the module boards. The next two functions let us read and write a single spot in the board. The try/except clauses keep out of bounds references from crashing the program.

```
>>> import util
>>> util.getSpot(13,38)
'|'
>>> util.setSpot(13,38,'X')
>>> util.writeBoard()
                      ___*____|_
                  |        |
                  |        |
                 _|__*__________|_______*___
```

```
             |      |
         ___*____|__________|_
             |      |
             |      |
             |      |
 ____*__________________|___________X______*____
>>>
```

Our call to util.setSpot changed a single character on the board.

Notice that the function writeBoard uses the escape sequence to erase the screen followed by a function writeScreen to position the cursor at each line and then write the characters for that line. This is followed by a "flush" call which guarentees the output is not cached in memory but rather output immediately. Without the flush the game can be quite jerky.

The function writeScreen will also be used in the game program to move the players on the screen. For example, if you, (represented by "^"), are on a catwalk and are to move one column to the right, we use writeScreen once to place a "_" where you currently are and then again to place a "^" at your new location, one position to the right.

The game loop

(lode1.py) is the first of five programs on our way to making a real computer game.

We first import everything from the util module which, in turn, imports the board. We also import the time module for its sleep function. We'll have a single player represented by the '^' character which simply moves according to where it is sitting on the board. In open air (' ') it falls down (and to a higher row), on a catwalk ('_') it moves to the right and finally, on a ladder ('|') it climbs up. In between each move the program pauses for .1 second. Just before pausing the program calls writeScreen(20,0,'') to simply get the cursor out of the picture.

Make sure you have ttyLinux.py, util.py and lode1.py. Run the program with

```
>>> import lode1
>>> lode1.main()
```

The board appears on the screen and a couple of seconds later "^" appears falling to the catwalk, moving to the right, and climbing the ladder. You will need to press contol-C to stop the program.

Adding keyboard control

One problem with lode1.py is that information about You is contained in variables 'row' and 'col'. If we have multiple players we need to do better.

A great approach is to have an object (instance of a class) represent each player. Each object can remember not only row and column but also lots of other information and with Python this can be very open-ended.

With lode2.py we define a class "You" which has attributes for the above as well as the direction you want to travel. A method "setDirection" uses the latest keyboard input available (keys) and sets your direction accordingly. Notice that direction is contained in a

2 value tuple with horizontal and vertical components.

A separate method "move" is used to change your location on the screen. You can't always move in the direction you want. In free space you fall. On a ladder you can only go up or down, on a catwalk left or right. Later we'll fix that so that you can transfer onto ladders from catwalks and vice versa.

The main function is the try/finally clause discussed earlier in the program key2.py. The game loop is in the function playGame.

Run the program. You will fall to the catwalk and stop. A right arrow starts you to the right but you stop at the ladder. An up arrow climbs and finally a control-C will exit.

```
>>> import lode2
>>> lode2.main()
```

Adding more players.

The next version lode3.py adds a robot to play against you. Since it makes sense to have "you" as a class instance it makes just as much sense to do the same for the robot. In fact both you and the robot can share the same "move" method. So let's use inheritance to have a superclass "Player" and subclasses "You" and "Robot".

This game, even though not complete, is fun to play. Let's look closely at the robots setDirection method which is where most of the new code resides.

The global "inPlay" is true if the game is still going. If the robot sees that he has tagged you then "inPlay" is set to false. Next the robot checks to see you are on the same level and runs towards you if so. If the robot finds itself on a ladder, it attempts to match your vertical location.

This strategy works pretty well on this board, even though it is quite simple. If you make more complex boards you will find the robots overly challenged and may want to program in some additional "smarts".

The move method in the Player class also has an addition. The use of variables lspot and rspot (for left/right) enable both "you" and the robot to transfer to and from ladders and catwalks.

Finally there is a variable "clock" which just keeps track of howmany moves have taken place. It is used in this version to give you a 4 second (40 moves) head start before the robot is created and comes after you. However the availability of a clock can greatly enhance what you can do with the game.

Keeping score. Burning holes.

Our next version lode4.py adds score keeping and a defense mechanism against the robot. Score is kept as an attribute of "you" and ten points are added each time you pass over a lode ('*'). The lode is then erased from both the screen and the board. Just before taking its .1 second nap the program writes the score to the screen in lower left corner where it moved to get the cursor out of the way.

The defense mechanism lets you burn a hole in the catwalk, either to your left ("a" key) or right ("s" key). The idea is that you'll work the arrow keys with your right hand and the 'a' and 's' keys with your left. These holes don't repair themselves and the robot and you fall through them to the next level down or off the screen. Your falling off the screen ends the game.

The final version for now

Our last version of the game will demonstrate a couple of possible ways to extend the game.

lode5.py will create 2 robots to chase you. The list "players" keeps track of "you" and whenever its length is under 3 it adds a robot. You can kill a robot by making it fall through the lowest catwalk. When a robot sees that it is dead it politely removes itself from the "players" list. But once that happens the list length is less than 3 and a new robot drops from the top at a random column (look for "random" in the code).

Because this game is a little too challenging (for me anyway) I also made the robots move at half speed. This is done by having a Robot.move method that calls Player.move every other time, when "clock%2 == 0"

Some ideas for further exploration.

The best thing about programing your own games is that you can extend them any way you want, at least if you can figure out how. Here are a just a few ideas.

> Play around with the speed of the game (time.sleep) and the number of robots. Let the user control some of this on starting the game. Level=easy
>
> Make more boards and add them to the boards list. Create a mechanism to have the program advance a level when you win or let you choose a level. Level=easy
>
> Change the rules. Perhaps instead of burning holes drop a "stun bomb" that stuns a robot for a few seconds keeping it from moving. Level=medium
>
> Make the robots more intelligent, keeping track of the last ladder used by "you" and heading toward it when other options fail. Level=medium
>
> The game "PacMan" is actually quite similar. Use this code as a basis for a PacMan like game. Level=medium
>
> Make "you" an automatic player like the robots. You must avoid the robots and try to reach treasure. Level=hard
>
> Make the game for two "You" players running on autopilot. Each may use a different strategy in their setDirection method. Level=hard
>
> Make the robots work cooperatively, anticipating moves of the other robots and your responses. Level=PhD Thesis

If you are interesting in the more challenging possibilities take a look at the following website.

http://icfpcontest.cse.ogi.edu/task.html

Index

Queues, Trees and Water Buckets

In this study, we'll look at a specific problem that is actually an example of a wide class of problems, all of which may be addressed in the same manner.

Water Bucket Problems

The problem is as follows. You go to the well with a 7 liter bucket and an 11 liter bucket. You wish to fill one of them with exactly X liters of water. How can you do this in the fewest number of operations, where each operation either fills or empties one of the two buckets.

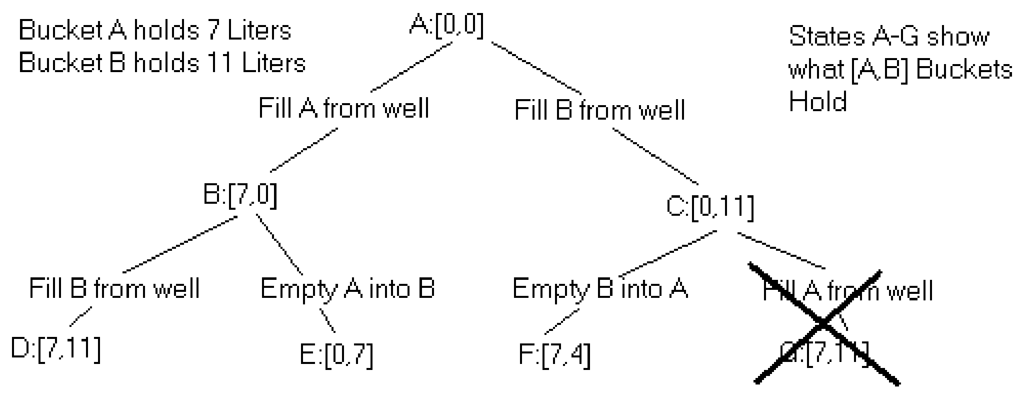
Well if X is 7 or 11 then the answer is easy. One operation is all that is required. Suppose that X is 4. Here we can fill the 11 liter bucket from the well and then use it to fill the 7 liter bucket. That will leave 4 liters of water in the 11 liter bucket.

What if X is 6? Before proceeding, try to solve this on your own.

We are going to develop a Python program that will search for the solution in a brute force (considering all possibilites) but fairly efficient manner.

State Search

It is useful to view this search as a tree of "states", where each state is the amount of water in each bucket and a filling or emptying operation generates a new state. Look at the following diagram.



We start with the state [0,0], meaning that both buckets are empty (contain zero liters of water). We labeled this state "A". Filling bucket A advances us to state [7,0] which we label B.

At each state we may generate up to 6 new states. (What are they?) But states that have already been encountered are uninteresting for 2 reasons. First they may make the program loop and secondly, we know there is a shorter path to that particular state. Notice how we crossed out state G because it is a duplicate of D. Some rejected states were not shown. For example, at state B we could empty bucket A into the well taking us back to [0,0]. Again, that is simply not an operation of interest.

So the operation of the program will be as follows. We start with state "A" and from it generate states "B" and "C". Then we work on state "B" generating states D and E.

Two Types of Searches

Now we have a choice. Do we do a depth first or breadth first search? Depth first would mean working next on D and its descendants, then coming back to state E, and possibly waiting a while to get to state B. Breadth first means putting the new states D and E on hold, and working next on state B. With breadth first we process the tree "row by row".

This works out to be the best strategy for finding the optimum solution in the shortest time. You might want to give some thought as to why this is so.

But how do we put states D and E on hold? Well, there is a handy data structure called a FIFO (first in, first out) queue which works like a line of people at a ticket counter. Each time a new state is generated we will add it to the back of the queue. When we're ready to resume work on an older state, we'll remove the one at the front of the queue and work on it.

Queues in Python

A FIFO queue is easily implemented with a Python list. The list append method adds new states to the back of the queue. The first state is accessed as "queue[0]" and this state may be popped off with either the "remove" method or a slice operation.

This strategy is quite general and can used for lots of search problems. One common example is the tile game where, for example we have 8 numbered tiles an 3x3 square. One tile may be moved at a time into the empty square. The goal is to arrange the 8 tiles in order. The more complex game found in toy stores has 15 tiles in a 4x4 square.

The Program

The program divides the water bucket problem into 2 parts and defines a class for each. Click here to view buckets.py I suggest printing it out to follow along.

The manager object handles the queue, adding new states as they are generated, first checking that they have not been encountered already. For this it uses a dictionary `seen`. This dictionary also serves a second purpose. It keeps track of each states "parent" state. There are 3 methods for the manager, adding a new state (with parent), getting the next state to work on, and finally generating a solution from the last state added by chaining back parent by parent to the beginning state.

The bucketPlayer object is specific for this game but interacts with the manager is a fixed pattern. Other classes such as tilePlayer could be written as well. It would even be possible to subclass each of these from a general "Player" class.

A "player" object gets states from the manager after first "seeding" an initial state. It works on the state to generate new states. Each is tested to see if we have won. The state is then submitted to the manager for queuing. Once the winning state has been found, the manager is queried for the full solution.

In this game, states are represented by a 2 element list. The 1st element is the number of liters of water in bucket A, and the 2nd the amount in bucket B. When generating new states notice the use of the `min` function to determine how much water to pour from A to B, or from B to A. You can't pour more water than is in the container, nor more than the other container has room for.

Here is a sample run of the program. It is the problem given at the beginning.

```
>>> from buckets import *
>>> m = manager()
>>> p = bucketPlayer(m)
>>> # A=7 liters, B=11 liters, Want=6
...
>>> p.playGame(7,11,6)
Solution is
[0, 0]
[7, 0]
[0, 7]
[7, 7]
[3, 11]
[3, 0]
[0, 3]
[7, 3]
[0, 10]
[7, 10]
[6, 11]
>>>
```

We separated the manager functions from the game playing functions into 2 classes so that the manager class could be used with multiple games. It doesn't need to know how a state is represented, as long as it can turn it into a unique string to use it as a key value in the `seen` dictionary.

There is a second advantage also in the separation of manager and player functions. Imagine several player objects working in parallel, all connected to the same manager. The manager and each player are separate processes, perhaps running on separate computers. Now we have parallel programming. It wouldn't make a lot of sense on this problem because the communication overhead is apt to be much greater than the cost of computing new states. But with more complex problem (like chess?) there could be tremendous gains in speed.

Ideas for further Development

Place some print statements in the code to watch it carefully as the queue grows and shrinks.

Find a way to test if there are any amounts of water between 0 and 11 liters that cannot be obtained with a 7 liter and 11 liter bucket. (whole liters only, of course)

Change the program to generate bucket problems for your friends requiring a certain number of moves.

What is special about 7 and 11 as bucket sizes? What happens when other combinations are used? Yes, Kirby, this is for you.

What happens when new states are added to the front of the queue instead of the back? Try it.

For the the more ambitious.

Write a player class for the tile game. Use 8 tiles on 9 squares. How would you represent each state? Hint: any 9 digit number will fit in a Python integer.

If you have access to Twisted or other CGI environment, consider implementing a parallel version of this program.

Index

The Tower of Hanoi Revisted with Objects

The Tower of Hanoi is a classic game that is often emulated on computers to demonstrate recursion.

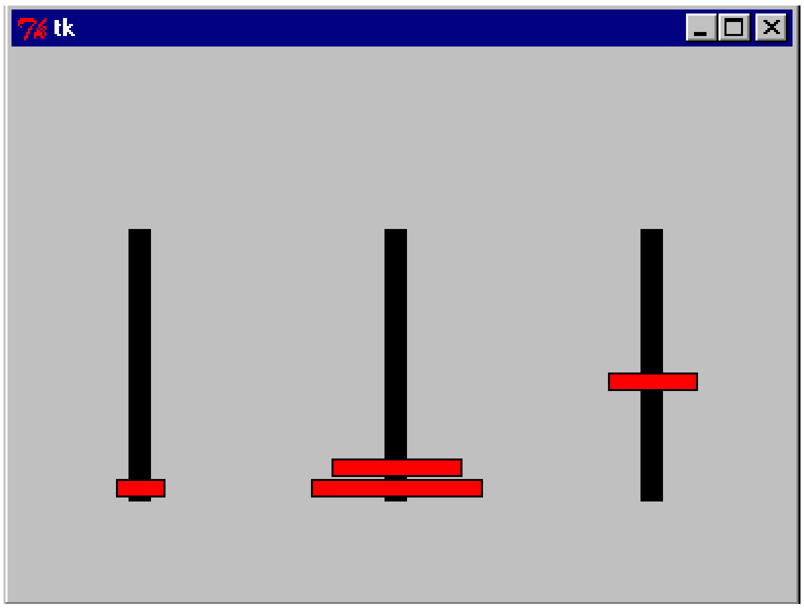
The game runs as follows. There are a number of discs each with a hole in the center. Each disc can fit on any of 3 pegs and each peg is high enough to hold all the discs in a stack. In the initial configuration all the discs are stacked on the first peg with the largest disc on the bottom and the smallest on top. A disc is never stacked on top of a smaller disc. The problem is to move the discs, one at a time from peg to peg in such a way that this is always true and to finally end up with all of the discs on peg 3 in the original order.

The solution is elegant. Let's call the three pegs A, B, and C. If you don't know how to move 5 discs from A to C (using B for intermediate storage), then just move 4 discs from A to B, one disk from A to C (this is really easy) and then move 4 discs from B to C. Can't do 4 discs? Well then, move just 3 ...

Guido's hanoi.py in the Python Demo area is a nice demonstration of this recursive process. The heart of the program is just this.

```
def hanoi(n, a, b, c, report):
  if n <= 0: return
  hanoi(n-1, a, c, b, report)
  report(n, a, b)
  hanoi(n-1, c, b, a, report)
```

The other 99% of the program involves doing the TK graphics to make it a real demo. Here is a screen shot of the game in progress.



I've tried a couple of times to teach the Hanoi puzzle as a good example of recursion. It is much better than the usual factorial example which does nothing more than use recursion to replace a simple for loop. But students have a hard time getting it. And I think the problem is the difficulty in visualizing the nested context of the recursive calls.

The recursion can be avoided, or at least made more intuative by using an object oriented approach. Then the discs, as objects, can hold the information normally held in the call stack.

Lets think of the discs as being animated and that we can request one of them to move itself, along with all of the smaller brethen above it, to another peg. We'll have each disc use the following strategy. If there is no smaller disc on top, simply move to the requested peg. But if there is, pass the buck. First ask the smaller disc above you to move to the alternate peg (along with its brethren above, if any), make your move, and finally ask the same disc to now move to your peg. When that is done, declare success. Each disc will need only to talk to the one just smaller

It might prove instructive to have the game played in a classroom with a different student playing the role of each disc. It might be a good idea to choose students by height to represent larger or smaller discs so that it is easy to see that the rules are being followed. Have 3 lines (A,B,C) that the students can stand in and initially line them up on A, tallest to shortest. Finally ask the tallest student representing the largest disc to move to line C.

Each student must follow these instructions exactly. It's probably a good idea that each have their own copy and use a pencil to keep track of exactly where they are at all times.

```
If you are requested to move to another line then
  If no one is in front, move to that line and say "OK".
Otherwise
  Ask the person in front of you to move to the alternate line
    (not the one you will move to).
  Wait for that person to say "OK".
  Move to line you were requested to.
  Ask the same person to move to the line you are now on.
  Wait for that person to say "OK"
Say "OK"
```

Play this game with 1, 2, 3, and 4 players. How many total moves are made in each case? How many moves would you expect for 5 players? 10 players? The monks of Hanoi were said to play the game with 64 discs and when finished, the universe would end. How well does this jive with current cosmology? ;)

Click here to view the source code of tower.py. It is a small python program that essentially follows these instructions. Here is a sample run when we import it.

```
>>> import tower
>>> tower.test()
A : I have been requested to move to peg 3
A : Asking B to get out of my way and move to peg 2
B : I have been requested to move to peg 2
B : Asking C to get out of my way and move to peg 3
C : I have been requested to move to peg 3
C : Moving to 3
B : Moving to 2
B : Asking C to rejoin me on peg 2
C : I have been requested to move to peg 2
C : Moving to 2
A : Moving to 3
A : Asking B to rejoin me on peg 3
B : I have been requested to move to peg 3
B : Asking C to get out of my way and move to peg 1
C : I have been requested to move to peg 1
```

```
C : Moving to 1
B : Moving to 3
B : Asking C to rejoin me on peg 3
C : I have been requested to move to peg 3
C : Moving to 3
```

Index

Animals and the Tree of Knowledge

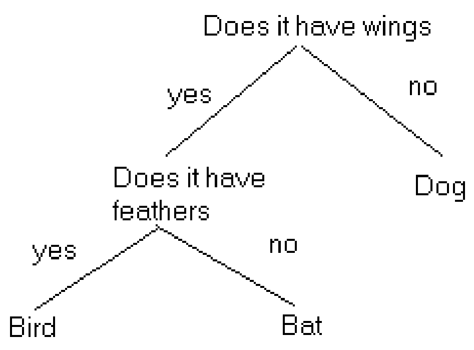
What it Does

This little program builds a classification tree by interacting with the user. It starts off knowing only that a bird is an animal and bit by bit allows learns more as it interacts with you. Here is an example run. You would type everything that follows a '?'.

```
>>> import animal
>>> animal.main()

Are you thinking of an animal? y
Is it a bird? n
What is the animals name? Dog
What question would distinguish a Dog from a bird? Does it have wings
If the animal were Dog the answer would be? n

Are you thinking of an animal? y
Does it have wings? y
Is it a bird? n
What is the animals name? Bat
What question would distinguish a Bat from a bird? Does it have feathers
If the animal were Bat the answer would be? n

Are you thinking of an animal? y
Does it have wings? y
Does it have feathers? y
Is it a bird? y

Are you thinking of an animal? n
>>>
```

The Data Structure

This is a schematic of the knowledge tree built from the above dialog.



The Code

The program is very simple. It starts with the top question and depending on the users yes or no response, chooses either the left or right fork down the tree. At the last element (the "leaf" node), it makes its guess. If the guess is wrong then it has the user input the name of a new animal and a question to distinguish the new animal from the guess. Then the tree is grown by one more node to accomodate the question and the new animal.

The code should be very straight forward. An object class is used for nodes in the tree. Each node contains a question plus pointers to left and right nodes. Leaf nodes simply contain the name of an animal and the value None in the pointer attributes.

The function "yes" is a convenience. It will recognize "Yes", "YES", "Y", "y", etc. all correctly, as well as similar variations of "No". Otherwise it keeps asking the question.

I first saw this program in very old fashioned Basic about 20 years ago. The code was much more complex requiring separate arrays of strings and pointers. I was starting my new job in Holland and took it along. It was something of a hit. After several people played with it for an hour or so, it had developed a nice biological hierarchy, half in English and half in Dutch, of everyone who worked in the office!

Ideas for further Development

Enable the program to save its knowledge tree between runs. You might use the pickle mechanism. Or write a function that saves all the user responses so that they can be played back as a script via the UserInput module.

Make a command to display the knowledge tree on the screen using indenting. You might need to consider recursion for this. Alternatively map the tree with a graphics package like TKinter.

Index

Building a GUI Application with Tkinter

In this tutorial we will use Tkinter to build a graphical user interface that we can use to maintain a simple phone list. On the way we'll play around with several Tkinter widgets thru the Python interactive prompt, hopefully getting a good feel for how all the pieces work together. Then we'll study a functional event driven GUI program.

It's best if you follow along with Python in another window and watch the widgets come and go and move around. Make sure you are using the ordinary python.exe executable and not one of the IDE like IDLE or PythonWin. Sometimes the graphics they are doing will interfere with the graphics we are creating resulting in some odd behaviour.

The amount of Tk that is introduced in this tutorial is small but surprisingly effective for many applications. However, you will also want to have a good reference to find out about other widgets such as check boxes and radio buttons as well as how to extend the functionality of the widgets being introduced here.

Playing with buttons

The first thing to do is to import the Tkinter module. Generally this is done by importing the module into our local namespace so that we can use the classes and constants by their names (like Label, Button, TOP) instead of having to constantly qualify everything (like Tkinter.Label, Tkinter.Button, Tkinter.TOP). So lets do that first.

```
>>> from Tkinter import *
```

Now if this produced an error message it means that either Tk/Tcl is not installed on your system or that Python is not linked to it. You will need to fix that before going on and that will probably involve your system administrator. Otherwise, if all is quiet, try the first command which will create a window and assign it to the variable "win".

```
>>> win=Tk()
```

You should now have a small window on your screen with "tk" in the title bar. Let's create a couple of buttons for this window

```
>>> b1 = Button(win,text="One")
>>> b2 = Button(win,text="Two")
```

The class Button takes the parent window as the first argument. As we will see later other objects, such as frames, may also act as parents. The rest of the arguments are passed by keyword and are all optional.

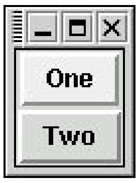
You might be surprised that the buttons did not appear in the window. They must first be placed with one of the so called geometry managers. The two most common ones are "pack" and "grid".

Using the Pack Manager

With "pack" you tell your widget to pack itself into its parent. You may specify a side (TOP, LEFT, RIGHT, BOTTOM) and your widget will be packed against either the parents wall or a previous widget with the same packing. If you don't specify a side the default is TOP. Do the following.

```
>>> b1.pack()
>>> b2.pack()
```

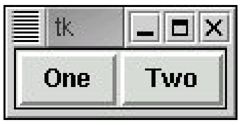
Notice that after the first command the button is placed in the window and the window itself is shrunk to the size of the button. When the second button is packed the window is expanded to accomodate it. The default TOP stacked them vertically in the order they were packed.



Now try the following two commands.

```
>>> b2.pack(side=LEFT)
>>> b1.pack(side=LEFT)
```

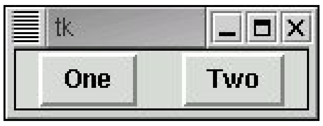
Now the buttons look like



In practice the pack geometry manager is generally used in one of these two modes to place a set of widgets in either a vertical column or horizontal row.

Our buttons look a little squished. We can fix that by packing them with a little padding. "padx" adds pixels to the left and right and "pady" adds them to the top and bottom.

```
>>> b1.pack(side=LEFT,padx=10)
>>> b2.pack(side=LEFT,padx=10)
```



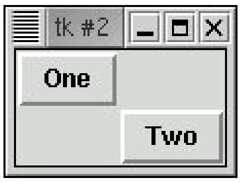
I suggest you create a couple more buttons and play with BOTTOM and RIGHT to get a good feel for the "pack" manager.

Using the Grid Manager

Another way to place widgets (buttons, labels and whatnot) is in a table or grid. Here the parent window is divided into rows and columns and each widget is placed in a given cell. The grid manager keeps track of how many row and columns are actually needed and fills out the window accordingly. It also keeps track of how wide each column, and how tall each row must be to accomodate the largest widget in that row or column. Rows do not all have to be the same height and columns do not have to all be the same width.

Let's make a new window with the same buttons but this time lay them out in a two by two grid.

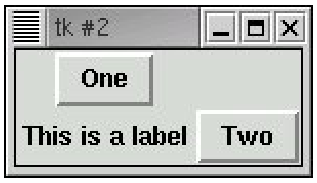
```
>>> win = Tk()
>>> b1 = Button(win,text="One")
>>> b2 = Button(win,text="Two")
>>> b1.grid(row=0, column=0)
>>> b2.grid(row=1, column=1)
```



You can see that some empty space is left since nothing was put into row 0, column 1 or into row1, column 0. Let's use this as an oppurtunity to look at a new widget type.

A label widget is used to place text into the window and is very simple.

```
>>> l = Label(win, text="This is a label")
>>> l.grid(row=1,column=0)
```

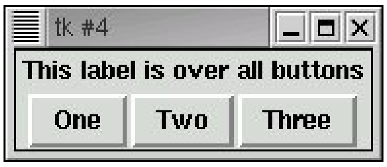


Notice how the label pushed the width of column 0 out to accomodate the text.

More complex layouts.

A frame is a widget whose sole purpose is to contain other widgets. Groups of widgets, whether packed or placed in a grid, may be combined into a single Frame. Frames may then be packed with other widgets and frames. This feature lets us create just about any kind of layout. As an example let's place a lable over 3 buttons in a row. We'll first pack the buttons into a frame horizontally and then pack the label and frame vertically in the window.

```
>>> win = Tk()
>>> f = Frame(win)
>>> b1 = Button(f, "One")
>>> b2 = Button(f, "Two")
>>> b3 = Button(f, "Three")
>>> b1.pack(side=LEFT)
>>> b2.pack(side=LEFT)
>>> b3.pack(side=LEFT)
>>> l = Label(win,"This label is over all buttons")
>>> l.pack()
>>> f.pack()
```



In addition to pack and grid there is a place method to position a widget at a precise location within a frame or window. It is not often used because it is frankly easier to let pack and grid just spread things out as needed, especially if you use the mouse to shrink or expand a window.

There are other keyword arguments that are common when using either pack or grid. We saw padx and pady above. With grids there is a "sticky" parameter which takes a map coordinate like N, E, S, W, NE, etc. If the grid cell is larger than your widget because a larger widget is in the same row or column, sticky helps you put the widget where you want it in the cell.

At this point you might want to check out your reference guide and play with other keyword parameters from the interactive prompt in order to get a good feel for how they function.

Bringing the buttons to life.

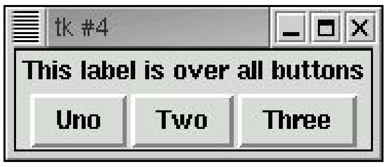
You may have tried clicking the buttons. If so, you noticed that they highlight and depress fine but they just don't do anything. Let's fix that.

As we've seen, widgets are objects and have methods. We've been using their pack and grid methods. Now we'll use a new method, "configure".

Any keyword argument that we can pass when creating a widget may also be passed to its "configure" method. For example, if we do the following

```
>>> b1.configure(text="Uno")
```

suddenly our window looks like



Buttons are tied to callback functions using the parameter "command" either when the button is created or with configure. Let's start by defining a function that simply prints a message

```
>>> def but1() : print "Button one was pushed"
...
>>> b1.configure(command=but1)
```

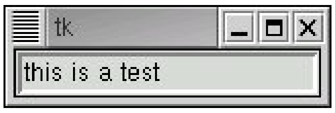
Now when we click button "Uno" the message is printed.

Entry widgets

To input text from the user we use an entry widget. Just as in the case of buttons we need some way to communicate with the entry widget, in this case to set and retrieve text. This is done with a special Tkinter object called a StringVar that simply holds a string of text and allows us to set its contents and read it (with get). Let's start with a clean window.

```
>>> win = Tk()
>>> v = StringVar()
>>> e = Entry(win,textvariable=v)
>>> e.pack()
```

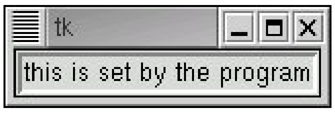
Now let's type "this is a test" into the entry and then retrieve it from our linked StringVar object



```
>>> v.get()
"this is a test"
```

We can also set text into our StringVar object and have it appear in the entry widget.

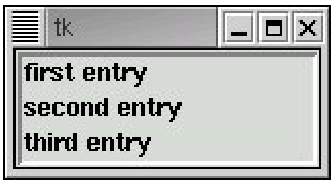
```
>>> v.set("this is set from the program")
```



The Listbox widget

Our last widget in the project will let us have a menu of items to choose from. A listbox is created with the following command (after opening a window). The "height" parameter limits how many lines will show.

```
>>> win = Tk()
>>> lb = Listbox(win, height=3)
>>> lb.pack()
>>> lb.insert(END,"first entry")
>>> lb.insert(END,"second entry")
>>> lb.insert(END,"third entry")
>>> lb.insert(END,"fourth entry")
```

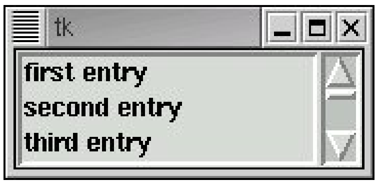


The fourth entry doesn't show since the listbox is set to just 3 lines.

Items in the listbox may be also inserted not only at the end (END) but also at the begining or even the middle. They may also be deleted. In fact we'll use the command "lb.delete(0,END)" later to clear the listbox.

A listbox may be used in conjunction with a scroll bar. Let's start by making a scroll bar and packing it next to the list box.

```
>>> sb = Scrollbar(win,orient=VERTICAL)
>>> sb.pack(side=LEFT,fill=Y)
```

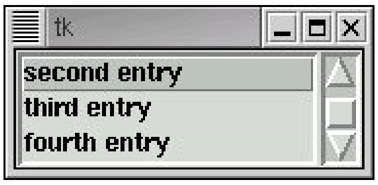


```
first entry
second entry
third entry
```

This looks good but if you operate the scroll bar you'll see that it doesn't do anything yet. The scroll bar and the list box need to know about each other. This is done in a manner similar to how we tied buttons to call back functions. Two calls are needed, one to tell each about the other.

```
>>> sb.configure(command=lb.yview)
>>> lb.configure(yscrollcommand=sb.set)
```

Now manipulate the scroll bar and see the listbox respond.



```
second entry
third entry
fourth entry
```

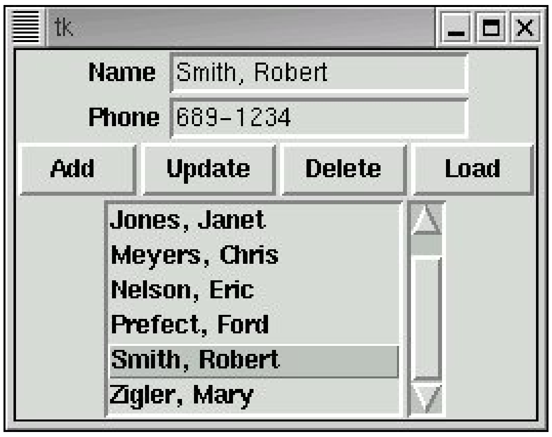
If you have selected an item in the listbox, the method curselection will return it for you. Actually it returns a tuple of items selected. It is possible to configure the listbox to allow multiple items to be selected together. An empty tuple is returned if no item is selected. Otherwise the tuple contains the index(es) of the selected items. (but as strings!)

For example, click on the 3rd item and do the following. In typical Python fashion indexes start at zero.

```
>>> lb.curselection()
('2',)
```

Putting it all together.

Our phone list editor uses all of the features discussed so far plus a few more that we'll touch on. Here is a view of the running application.



```
Name  Smith, Robert
Phone 689-1234

Jones, Janet
Meyers, Chris
Nelson, Eric
Prefect, Ford
Smith, Robert
Zigler, Mary
```

Now would be a good time to bring to bring up the python source in another window or make a printout. Click here to view the source. Click here to see the initial phone list.

The variable "phonelist" is a list of name/phone number pairs. As we work with the application pair entries will be added, modified and deleted.

The first thing the program does is call makeWindow, returning a reference to it which is stored in the variable "win". Looking at the code in the function "makeWindow" we can see all the widgets talked about above. The widgets are set into three frames which are then packed vertically.

The top frame is a 2x2 grid for the name and phone entry fields and labels to the left of them. Notice the rather strange form of the call

```
Label(frame1, text="Name").grid(row=0, column=0, sticky=W)
```

Here we create a label and immediately "grid" it into row=0, column=0. We loose all reference to it but frame1 has it tucked away. This is a very common way to define widgets when once they are created and set in their parent, we need no further contact with them.

Notice that two globals, "nameVar" and "phoneVar" reference StringVar objects that are tied to the Entry widgets "name" and "phone".

The second frame contains the 4 buttons packed left to right. Each is tied to a callback function which we'll discuss shortly. Here we've assigned each button to a variable (like "b1") but since we don't access them later we could have done the same thing that we did with the label above.

```
Button(frame2,text=" Add  ",command=addEntry).pack(side=LEFT)
```

Finally the listbox and its scrollbar are packed into their own frame3. The "fill" parameters guarentee that the scrollbar and listbox will be the same height and that the listbox (fill=BOTH) will expand to the full width of the parent window.

The function setSelect first sorts the phonelist (it may get out of sort order during modifcations to it) and then essentially writes it to the listbox.

Once the window is built and the listbox initially populated a new function win.mailoop() is called. This puts the program into the event driven mode where everything that happens, until the program exits, is due to callback functions, initiated in this program by clicking the buttons.

Run the program and click one of the names in the list box. Then click the Load button. The name and phone number of your selection should appear in the Entry widgets. Clicking the Load button activated the callback function LoadEntry which first accessed the index of your listbox selection (via function whichSelected()) and then accessed the data from the list "phonelist". Now modify the Entry widgets contents and click Update or Add. Play around also with the Delete function.

A little careful study should make this program completely understandable. But it is incomplete is several respects. Most glaringly, there is no way to save your changes when the window is closed! As an exercise, add a Save button and make it work. One possiblity is to have Save's callback write a new "phones.py" to import the next time the program is run. Another is to use the "pickle" module to store the contents of "phonelist" between runs.

In the followup to this study, we'll extend this program to use a MySQL database for storing the phone list. This will enable multiple people to modify the list concurrently.

Index

Using an SQL database with our GUI

In the previous tutorial we kept our phone list data in a python module that was simply imported. The phone list was a list of lists where each sublist contained the name and phone number.

This was very convenient when keeping the code as small as possible is the primary concern. The Python compiler becomes the parser for the input data and the "str" function can format the entire phonelist in order to write it back to the disc.

Another simple way to store the phone list might be in a text file with one entry per line. This would require more code of our own to convert this file to (and from) the internal format but would be useful in other ways. For one it would be easier to edit with a standard text editor like emacs or vi. For another, simple utilities like "grep" could be used for rapid searching from a command line. That is unless the phone list grows to the size of a phone book.

For more sophisticated data storage we need more. If several people are using our GUI at the same time to update phone numbers, we would like to see those changes immediately and have the changes made by one person not interfere with those of another. If two people pull up the phone list and make changes to seperate records, the second person to save the phone list back to the disc will wipe out the change made by the first person.

SQL databases will synchornize update and offer many other features besides.

We'll look at a very simple example of using Mysql with our phone list. Another tutorial is available oriented around PostSql

Just enough SQL

Mysql, like other database systems, is a client/server application. The server program (or daemon) is called "mysqld" and it actually does all of reading and writing to the disc. Client programs then request actions from the daemon, such as inserting, updating, deleting or just searching for data.

A standard client, the program "mysql", lets you interact directly with the database daemon thru the keyboard or other `stdin` input.

A database consists of tables which in turn consists of rows and columns. Columns in a table are fields such as "name" and "phone". Each column has a datatype such as varchar (roughly equivalent to a Python string but with a maximum length) or integer. For our gui we are going to make a table called "phones" with 2 varchar columns called "name" and "phone". Our table will be part of the database "test" which comes built-in.

Unless the mysql daemon is already running you may have to start it. If you get an error message when running mysql, check with your system administrator.

```
mysql>use test;
mysql>show tables;
Empty Set (0.00 sec)
```

The commands requested the daemon to use the "test" database and then to show tables in the test database. Since we haven't created any (nor has anyone else) it is currently empty.

Creating our table

Creating a table involves listing the name and type of each column.

```
mysql> create table phones (id int, name varchar(20), phone varchar(12));
```

Our table "phones" has 3 fields; id, name, and phone. The integer "id" field will be used as a handle on a row. All rows will be assigned a unique id, just increasing numbers. It will become clearer why this is a good idea as we proceed. Once our table is created the "describe" command shows off its structure. Don't worry about the last 4 columns.

```
mysql> describe phones;

+-------+-------------+------+-----+---------+-------+
| Field | Type        | Null | Key | Default | Extra |
+-------+-------------+------+-----+---------+-------+
| id    | int(11)     | YES  |     | NULL    |       |
| name  | varchar(20) | YES  |     | NULL    |       |
| phone | varchar(12) | YES  |     | NULL    |       |
+-------+-------------+------+-----+---------+-------+
3 rows in set (0.11 sec)
```

Inserting data into the table

The SQL insert command lets us populate the table with rows. Here is an example.

```
insert into  phones values (1,'Meyers, Chris',  '343-4349');
```

We supply values for id, name and phone. Incidentally there is another format for the insert command where only designated columns are set and other columns are set to a default value or null (equivalent to None in Python).

The files phones.sql contains commands to create the database and populate it with insert commands. It may be piped to the client program "mysql" to initialize the database table.

```
mysql <phones.sql
```

Accessing data in the table

The SQL select command lets us find data we are interested in. The simplest format will dump the entire table.

```
mysql> select * from phones;
+------+-----------------+----------+
| id   | name            | phone    |
+------+-----------------+----------+
|    1 | Meyers, Chris   | 343-4349 |
|    2 | Smith, Robert   | 689-1234 |
|    3 | Jones, Janet    | 483-5432 |
|    4 | Barnhart, Ralph | 683-2341 |
|    5 | Nelson, Eric    | 485-2689 |
|    6 | Prefect, Ford   | 987-6543 |
|    7 | Zigler, Mary    | 567-8901 |
|    8 | Smith, Bob      | 689-1234 |
+------+-----------------+----------+
8 rows in set (0.11 sec)
```

But we can be more restrictive by using a "where" clause. For example.

```
mysql> select * from phones where id=6;
+------+---------------+----------+
| id   | name          | phone    |
+------+---------------+----------+
|    6 | Prefect, Ford | 987-6543 |
+------+---------------+----------+
1 row in set (0.05 sec)
```

We can also request just certain columns.

```
mysql> select name,phone from phones where name like "Smith%";
+---------------+----------+
| name          | phone    |
+---------------+----------+
| Smith, Robert | 689-1234 |
| Smith, Bob    | 689-1234 |
+---------------+----------+
2 rows in set (0.05 sec)
```

Here the "%" is the wildcard character.

Updating data in the table

The SQL update command is used to change column values in rows specified by a "where" clause. Here is an example.

```
mysql> update phones set name='Chase, Chevy' where id=6;
Query OK, 1 row affected (0.06 sec)

mysql> select * from phones where id=6;
+------+--------------+----------+
| id   | name         | phone    |
+------+--------------+----------+
|    6 | Chase, Chevy | 987-6543 |
+------+--------------+----------+
1 row in set (0.06 sec)
```

Deleting rows from the table

Finally, the SQL delete command will erase any rows matching the "where" clause. If there is no where clause, all rows are deleted.

```
mysql> delete from phones where id=6;
```

```
Query OK, 1 row affected (0.05 sec)

mysql> select * from phones;
+------+-----------------+----------+
| id   | name            | phone    |
+------+-----------------+----------+
|    1 | Meyers, Chris   | 343-4349 |
|    2 | Smith, Robert   | 689-1234 |
|    3 | Jones, Janet    | 483-5432 |
|    4 | Barnhart, Ralph | 683-2341 |
|    5 | Nelson, Eric    | 485-2689 |
|    7 | Zigler, Mary    | 567-8901 |
|    8 | Smith, Bob      | 689-1234 |
+------+-----------------+----------+
7 rows in set (0.00 sec)
```

It should now be clear why the id column is important. If we want to be sure we are modifying or deleting a single row, this provides a mechanism. We may have a given name in the table more than once (maybe she has a cellphone) and the phone number may linked to multiple people (a house phone).

Python and MySQL

A Python program may be an sql client as well. In fact, the interface has much the same look and feel as the standard mysql client. A few examples will demonstate. We'll start by creating a connection object called db and telling it to use the test database.

```
>>> import MySQL
>>> db = MySQL.connect('')
>>> db.selectdb("test")
>>>
```

Next let's do a query. The variable "c" is set to a cursor object which can fetch rows for us in a list of lists. The data is returned in the same format as in the previous program.

```
>>> c = db.query("select * from phones")
>>> rows = c.fetchrows()
>>> for row in rows : print row
...
[1, 'Meyers, Chris', '343-4349']
[2, 'Smith, Robert', '689-1234']
[3, 'Jones, Janet', '483-5432']
[4, 'Barnhart, Ralph', '683-2341']
[5, 'Nelson, Eric', '485-2689']
[7, 'Zigler, Mary', '567-8901']
[8, 'Smith, Bob', '689-1234']
>>>
```

Although we won't use it in the program, the cursor object may also retrieve a list of dictionaries. This can be more convenient since we don't have worry about which column is in which position.

```
>>> d = c.fetchdict()
>>> for row in d : print row
...
{'phones.name': 'Meyers, Chris', 'phones.phone': '343-4349', 'phones.id': 1}
{'phones.name': 'Smith, Robert', 'phones.phone': '689-1234', 'phones.id': 2}
{'phones.name': 'Jones, Janet', 'phones.phone': '483-5432', 'phones.id': 3}
{'phones.name': 'Barnhart, Ralph', 'phones.phone': '683-2341', 'phones.id': 4}
{'phones.name': 'Nelson, Eric', 'phones.phone': '485-2689', 'phones.id': 5}
{'phones.name': 'Zigler, Mary', 'phones.phone': '567-8901', 'phones.id': 7}
{'phones.name': 'Smith, Bob', 'phones.phone': '689-1234', 'phones.id': 8}
>>>
```

Inserting, updating and deleting rows is, by comparsion, quite simple. We just pass the command to db.query and, voila, it happens.

```
>>> db.query("update phones set phone='338-1233' where id=1")
>>> c = db.query("select * from phones where id=1")
>>> c.fetchrows()
[[1, 'Meyers, Chris', '338-1233']]
>>>
```

Adapting our GUI for SQL

The changes required to use mysql with our GUI are actually fairly minor. Click here to see the full code. Let's look at the changes one at a time.

At the top of the program we import MySQL and set up a database connection to the test database.

```
import MySQL
db = MySQL.connect('')
db.selectdb("test")
```

Let's look next at the function "setSelect" which fills in our list control. Here, instead of importing the phone list, we simply use fetchrows to get the same list of lists.

```
def setSelect () :
global phoneList
c = db.query("select id,name,phone from phones order by name")
phoneList = c.fetchrows()
select.delete(0,END)
for id,name,phone in phoneList :
select.insert (END, name)
```

All other SQL commands are channeled to the function "dosql" which makes sure setSelect is called after the update, delete or insert happens. This also catch any changes made in the meantime by other users. For learner feedback "dosql" also prints the sql command to the launch window.

About the only other feature worth remarking on is the generation of new id numbers as new rows are inserted.

```
c = db.query("select max(id)+1 from phones")
id = c.fetchdict()[0].values()[0]
```

The SQL max function does what you would expect. Adding one gives us a new unique id to be used in the insert immediately following.

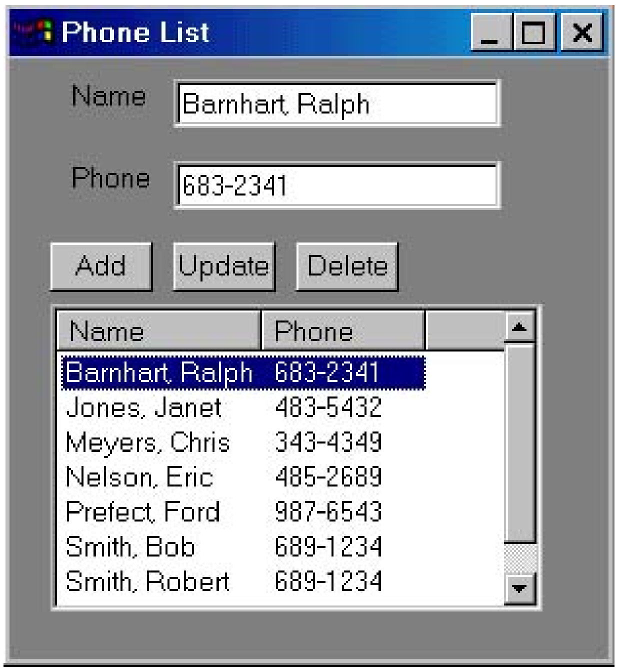
There is a potential problem with this however. If hundreds of users were using the program at the same time, two might inadvertantly fetch the same "max(id)+1" before either does their insert. Then we would have two rows in the table with the same id; something we definitely don't want. Databases have ways of dealing with these "racing" conditions, from providing automatic id columns to transaction processing, where multiple SQL statements can be guarenteed concurrent (and non-interrupted) execution.

Building the GUI with wxPython

What is wxPython

As an alternative to Tkinter, there is an interface available to adapt the wxWindows package to Python. The interface, appropriately enough, is called wxPython.

wxWindows is a C++ package that runs on both Windows and Unix and lets you build GUI programs with a very native look and feel on each platform. Here is what our program will look like under Windows 98.



Name: Barnhart, Ralph
Phone: 683-2341

Name	Phone
Barnhart, Ralph	683-2341
Jones, Janet	483-5432
Meyers, Chris	343-4349
Nelson, Eric	485-2689
Prefect, Ford	987-6543
Smith, Bob	689-1234
Smith, Robert	689-1234

Tkinter is certainly very object oriented with windows and widget objects created, customized and positioned by your code. Your code itself, however, can be completely procedural. In fact this is exactly how the previous program was constructed. The function makeWindow assembled all the pieces including callback functions for the buttons. Then the mainloop method of the window was called passing control to the window itself.

With wxPython programs are built using inheritance. Your application program is a subclass of a generic application class (wxApp) and windows (or frames) are subclassed from a generic wxFrame. Our makeWindow function essentially becomes the __init__ method of our Frame class. Button callback functions become methods within our class.

In my opinion we get a more organized program but at a cost. It seems a good deal harder to play with the pieces at the interactive prompt than it was with Tkinter. Instead, one starts building a bare-bones application and frame class and then testing it. Then, as it is working correctly, one adds more and more to the frame class filling it in.

Comparing the code

At this point it is probably a good idea to get a printout of the wxPython code. in order to compare it to the Tkinter GUI program.

A basic application class is pretty much boilerplate code. Each application subclass must have an OnInit method which instantiates one or more windows (frames) and selects one to be active and shown on top. If only one frame is created this code is very straight forward.

```
class MyApp (wxApp) :
  def OnInit (self) :
    frame = MyFrame(NULL, -1, "Phone List")
    frame.Show(true)
    frame.setSelect()
    self.SetTopWindow (frame)
    return true
```

Within MyApp we create an instance of a MyFrame object and then show it, select it for focus and make sure its the top window. Notice that we have no __init__ method in MyApp. The __init__ method of the parent class, wxApp, is used instead and it calls OnInit (among other things).

Our skeletal frame (window) is built with the following.

```
class MyFrame (wxFrame) :
def __init__ (self, parent, ID, title) :
  wxFrame.__init__(self, parent, ID, title,
          wxDefaultPosition, wxSize(300,325))
```

The __init__ method takes 3 arguments. Looking at call above in OnInit, we set parent to NULL (meaning no parent), ID to -1 (use a default), and the title to "Phone List". These 3 arguments are passed to the frame initializer along with size and position.

Size and position arguments are passed as 2 element tuples in wxPython, specifying x and y pixel or "dialog" values. The value -1 tells wxPython to use a default value. wxPython has functions that let us use nicer notation that just translates back to simple tuples. We can observe this at the command prompt.

```
>>> print wxDefaultPosition
(-1, -1)
>>> print wxSize(300,325)
(300, 325)
>>> f = wxFrame(NULL,-1,"test")
>>> print wxDLG_SZE(f,100,100)
(200,200)
```

The function wxSize simply passes the tuple through. But wxDLG_SZE is a bit different. It translates an x/y size depending on the frame passed. wxDLG_PNT is similar. Using these functions lets your program behave the same way even with different screen resolutions.

The Controls

Our window will contain lables, buttons (and their callbacks), text entry boxes, and a list control. We saw each of these in Tkinter and now we'll look at their counterparts in wxPython.

To keep things simple we are going to use absolute positioning in our wxPython version of the phone list program. This is done by simply giving the position (with wxDLG_PNT) and the size (with wxDLG_SZE), although wxPython has facilities similar to "pack" and "grid" in Tkinter.

Labels in Tkinter were simply a bit of geometry and text and the same is true with wxPython. Only the name has been changed. The frame class contains a method to add a label. Being a method it takes the first argument "self". The second argument is id number which can be ignored (defaulted). Then comes the text in the label and its position in the frame.

```
wxStaticText(self,-1,"Name",wxDLG_PNT(self,15,5))
```

Entry widgets need a both a position and a size and a way to get text into or out of them. With Tkinter the latter is done by tying the widget to a StringVar object and then getting and setting its value. Here wxPython is a bit more straight forward.

```
name  = wxTextCtrl(self,12,"",wxDLG_PNT(self,40, 5),wxDLG_SZE(self,80,12))
name.SetValue("Muggs, J. Fred")
print name.GetValue()
```

Here we used the id value of 12 and would use this id to tie events in the text control to another method. We'll see how this works with buttons.

Button widgets need a name (title), size and position, and a callback function (or method) to invoke when clicked. Here wxPython is a little less straight forward than Tkinter. It is necessary to give each button an id number and then that id number is linked to a method in the class. In the following case, the id is 11.

```
wxButton(self,11,"Add",    wxDLG_PNT(self,10,45),wxDLG_SZE(self,25,12))
EVT_BUTTON(self, 11, self.addEntry)
```

Finally, we come to the list box and its scrollbar. We need functionality to delete the contents, insert rows with one or more columns, and access a user selection. With wxPython this is pretty straight forward. The scroll bar incidentally comes for free here whereas in Tkinter it was a seperate widget "married" to the list box.

```
self.lc = wxListCtrl(self,15,wxDLG_PNT(self,10,60),wxDLG_SZE(self,120,75),
style=wxLC_REPORT)
EVT_LIST_ITEM_SELECTED(self,15,self.getSelect)
```

Here our list control is given an id of 15 and positioned and sized. The style parameter means we'll have columns with headings in the control. The second statement ties the callback method "getSelect" to the list control. It is called whenever an entry is clicked.

```
self.lc.InsertColumn(0,"Name")
self.lc.InsertColumn(1,"Phone")
```

Columns in the list control are set up just once. They are given a column number starting with zero and text for the heading.

```
self.lc.InsertStringItem(row, name)
self.lc.SetStringItem(row, 1, phone)
```

Two calls are used to set information in a row. InsertStringItem inserts a new row and fills column 0. Other columns in the same row are filled with SetStringItem.

Finally, when the call back method is called with an event object, the attribute m_itemIndex contains the row clicked.

```
def getSelect (self, event) :
self.currentSel = event.m_itemIndex
```

The complete program

Armed with the above you are ready to tackle the code and by comparing it with the Tkinter version have everything come together. The last thing needed is to create an instance of our application class and set it running by calling its MainLoop method.

```
app = MyApp(0)
app.MainLoop()
```

Index

Erlang for Python Programers

Introduction

There has been a lot of interest in the Erlang language in the last few years. Its model of programming relies on concurrent processes that communicate only by sending and receiving messages to each other. These processes are built using a fairly simple functional language that requires a different mind-set than the one we are used to when programming in imperative languages like Python, Ruby or Java.

In this project, we will explore some simple examples of this functional way of programming in both Erlang and in Python. Erlang, like other functional languages such as ML or Haskell, is quite restrictive in constructs we can use. We can write equivalent Python programs with the same restrictions, and in the process leverage our knowledge of Python to a better understanding of Erlang and functional programming in general.

In a second part of this project, we'll explore Erlang's concurrency and message passing features. We'll adapt an example from the logic circuits project, building a composite logic gate using instances of a single simple Nand gate. Instead of each gate being a Python object instance, in Erlang they will each be a seperate concurrent process. Message passing will connect the circuits.

In what may be a third part of the project, we will look at more sophisticated techniques of functional programming such as higher-order functions.

Here are links to the Python and Erlang code.

Replacing Iteration with Recursion

Let's look at a simple factorial function using a "while" loop.

```
def factorialOld(n) :
    ans = 1
    while n > 1 :
        ans = ans * n
        n   = n - 1
    return ans
```

```
>>> import samples
>>> samples.factorialOld(5)
120
```

Now, in Erlang, such an approach will simply not do. Interation using a "while" or "for" keyword is not allowed. Secondly, a variable may not take multiple values. The reasons for these restrictions will be clear in a bit.

So let's rewrite the factorial function using recursion.

```
def factorial(n) :
   if n == 0 : return 1
   else      : return n * factorial(n-1)
```

And run it

```
>>> import samples
>>> samples.factorial(5)
120
```

Now that should be pretty straightforward. You might be complaining that the variable "n" really does take on different values, a different one at each level of recursion. But, actually "n" is a different variable at each level. All variables are "assigned" only as a new level of recursion takes place. Within a level, the single value property is intact.

But why all the fuss?

Well, not being able to change a variables "binding" (a term more accurate than "value") means that, within a level of recursion, the relationship between the variables is constant and will not "swim around". The computation is much simpler to analyze and far less error prone. Many subtle errors, maybe most, arise from variables that interact with other with different values at different times. The timing of when each variable is set relative to the others leads to surprising complexity. In the bad old days of GOTO spaghetti code (before even structured programming) most code modifications would introduce new bugs.

Once the variables become fixed, giving up loops within a specific recursion level is actually no longer a big deal because the reason we wanted the loops was to change the value of one or more variables and their relationship with each other.

So now let's look at the factorial function in Erlang.

```
 factorial(0) -> 1;
 factorial(N) -> N * factorial(N-1).
```

Now this may seem strange if you are not used to pattern matching. Basically, there are two cases that in Python we addressed with an "if/else" inside a single function definition. Here pattern matching happens on the outside, instead. If the argument to the factorial call is zero, then a one is returned, no explicit "return" keyword is required. Otherwise, the variable "N" is bound to the argument and the result, from evaluating "N * factorial(N-1)", is returned. It is basically the same logic as in the Python version.

And here is how we can test this erlang version.

```
chris@ubuntu:~/projects/erlang$ erl
Erlang (BEAM) emulator version 5.6.3
```

```
Eshell V5.6.3  (abort with ^G)
1> c(samples).
{ok,samples}
2> samples:factorial(20).
2432902008176640000
```

Line 1 "c(samples)." compiles "samples.erl" and will return error messages if there are problems. Basically the same as a Python "import". Line 2 runs the function "factorial" in the module "samples". Notice the ":" seperates the module name from the function name, where Python uses a ".". Also notice the ending "." after each statement.

Some Basics of Erlang Syntax.

Just a few things to keep in mind. Once you are used to it, Erlang is actually a surprisingly simple language. This is not at all complete, but enough for what we are working with right now.

> "->" sets up a conditional and in Python we would always find a ":" in its place.
>
> "." ends a statement. It will consist of one or more clauses seperated by a ";". Within a statement only one clause will be chosen, the first whose pattern matches the input.
>
> Within a clause there may be multiple expressions seperated by ",". They will be evaluated sequentially. The value last expression evaluated in a statement is returned to the caller.
>
> Variables in Erlang begin with an uppercase character. For convenience we'll use the same variable names in our Python equivalent programs.
>
> Words starting with a lower case letter represent symbols in Erlang that simply stand for themselves. In Python, we generally use strings for this purpose. We won't use symbols (or tuples) until part 2.

Watching Recursion

Let's modify our earlier Python version of the factorial function to watch it in action. To make it easier to compare the Python and Erlang versions, I'm going to start capitilizing the Python variable names.

```
def factorialD(N) :
   print "Entering", N
   if N == 0 : Ans = 1
   else      : Ans = N * factorialD(N-1)
   print "Returning", N, Ans
   return Ans
```

```
>>> import samples
>>> samples.factorialD(5)
Entering 5
Entering 4
Entering 3
Entering 2
Entering 1
Entering 0
Returning 0 1
Returning 1 1
Returning 2 2
Returning 3 6
Returning 4 24
Returning 5 120
120
>>>
```

Notice that we progress down the recursive rabbit hole, and finally reaching the bottom, and then on the way back up actually do the computation.

Accumulators and Tail Recursion

Now let's try another version of the factorial function. Again, we'll place a print statement stratigically so we can follow the action.

```
def factorial2(N, ACC=1) :
   print "Entering with", N, ACC
   if N == 0 : return ACC
   else      : return factorial2(N-1, ACC*N)
```

```
>>> import samples
>>> samples.factorial2(5)
Entering with 5 1
Entering with 4 5
Entering with 3 20
Entering with 2 60
Entering with 1 120
Entering with 0 120
120
>>>
```

Now the computation is done on the way down through the recursion, carrying the partial result along in ACC. The final result is simply popped back up through the nested returns. Notice that by using a named parameter for ACC in our Python version, it can be omitted on the initial call and will be automatically assigned the correct initial value.

Now, if the Erlang compiler (not Python) can detect that for all clauses in a function, no actual computation takes place after each recursive return, it will simply not push the call stack down for new invocations, but rather reuse the stack space of the previous one. This is called "tail recursion". It has two big advantages. It is more efficient, just a single return instead of many redundant ones, and it makes infinite recursion possible without overflowing the stack. And infinite recursion is the only way in Erlang to have an infinite loop.

Here is the Erlang version of our tail recursive "factorial".

```
factorial2(N)     -> factorial2(N,1).
factorial2(0,ACC) -> ACC;
factorial2(N,ACC) -> factorial2(N-1, ACC*N).
```

Notice that there are two function definitions, each ending with a period. The first takes a single argument and is the called from the outside. The definition with two arguments carries the accumulated result and finally returns it. This second definition satisfies the conditions for tail recursion. Remember that we made ACC a named argument in the Python version to get roughly the same effect. Here is a sample run of the Erlang code.

```
6> c(samples).
{ok,samples}
7> samples:factorial2(6).
720
```

List Processing

Consider the following dialog with the Erlang interactive shell.

```
Eshell V5.6.3  (abort with ^G)
1> A = [1,2,3,4].
[1,2,3,4]
2> [HIT] = A.
[1,2,3,4]
3> H.
1
4> T.
[2,3,4]
```

An Erlang list looks very much like a Python one. In line 1 the variable A is bound to the list [1,2,3,4]. In line 2 we can really see that "=" is no simple assignment operator. It rather tries to unify the left and right hand sides, assigning values to unbound variables as needed. In this case the unbound variable H is set to the head of the list, "1" and T is to the tail. The pipe character "|" has a special meaning. As in Python, commas in Erlang seperate items in the list but "|" seperates the first item from all the rest.

In Erlang, this syntax can also be used on the right hand side to build lists. Consider.

```
2> [4 I [5,6,7]].
[4,5,6,7]
```

Here, we are supplying the head and tail and the "|" operator combines them to a single list.

Python does not have anything like the "|" operator, but we can emulate the action easily.

"[HIT] = L" in Erlang becomes "H=L[0]; T=L[1:]" in Python.

"L = [HIT]" in Erlang becomes "L = [H]+T" in Python.

Both Python and Erlang can concatenate lists. Python simply uses the "+" operator. In Erlang the operator is "++".

In Python

```
>>> [1,2,3] + [5,6,7]
[1, 2, 3, 5, 6, 7]
>>>

And in Erlang

Eshell V5.6.3  (abort with ^G)
1> [1,2,3] ++ [6,7,8].
[1,2,3,6,7,8]
2>
```

Let's look at a simple example using lists. We will sum the elements which are assumed to be numbers. Here's two Python versions, the second one is tail recursive.

```
def sum(L) :
   if not L : return 0
   else     : return L[0] + sum(L[1:])

def suma(L, Acc=0) :
   if not L : return Acc
   else     : return suma(L[1:], Acc+L[0])
```

Let's test them quickly

```
>>> import samples
>>> samples.sum([1,2,3,4,5])
15
>>> samples.suma([1,2,3,4,5])
15
>>>
```

And the Erlang version are basically the same.

```
sum([]) -> 0;
sum([H|T]) ->  H + sum(T).

suma(L) -> suma(L,0).
suma([],Acc)    -> Acc;
suma([H|T],Acc) -> suma(T, Acc+H).
```

Let's run it in the Erlang shell

```
Eshell V5.6.3  (abort with ^G)
1> c(samples.erl).
{ok,samples}
2> samples:sum([1,2,3,4,5,6]).
21
3> samples:suma([1,2,3,4,5,6]).
21
4>
```

Quicksort in Python and Erlang

Finally, let's look at the classic Quicksort algorithm in both Python and Erlang.

The algorithm is beautifull in its simple recursion and may remind you of the "Tower of Hanoi", another project on this site. Basically, a list of items is seperated into two lists based on picking a random element from the list, which we call the pivot. Items greater than the pivot go to one list and those less than to the other. Those equal to the pivot, if any, are assigned uniformily to one of the two lists. Here is a Python version of the split function, using only recursion. (no while loop)

```
def split(P, L, A=[], B=[]) :
   if  not L : return [A,B]
   H = L[0]              # H and T assigned only once
   T = L[1:]
   if H <= P : return split(P, T, [H]+A, B    )
   else      : return split(P, T, A,     [H]+B)
```

Take a deep breath. This is the trickiest bit of code you'll see here. The recursion is replacing what would normally be a while loop. Each recursive call operates on the tail of the previous call, assigning the head to one of the two output lists. The output lists are carried the recursion and the whole thing is nicely tail recursive.

```
A sample run

>>> samples.split(5,[1,2,3,4,5,6,7,8,9])
[[5, 4, 3, 2, 1], [9, 8, 7, 6]]
>>>
```

Once we have the function to split lists, the sort itself is not difficult. To sort a list, including the recursive sub-lists, we just use the head of the list as the pivot, split the tail into two lists, sort each of them and finally recombine everthing with Python list concatenation. Here is the code.

```
def sort(L) :
   if not L : return []
   H = L[0]              # H and T assigned only once
   T = L[1:]
   [A,B] = split(H,T)
   print "Pivot %s: %s --> %s %s" % (H,T,A,B)
   return sort(A) + [H] + sort(B)
```

To make it a little more interesting, we print the results of each split; the pivot value, the input list and the outputs.

```
>>> samples.sort([5,4,3,6,7,8,4,3])
Pivot 5: [4, 3, 6, 7, 8, 4, 3] --> [3, 4, 3, 4] [8, 7, 6]
Pivot 3: [4, 3, 4] --> [3] [4, 4]
Pivot 3: [] --> [] []
Pivot 4: [4] --> [4] []
Pivot 4: [] --> [] []
Pivot 8: [7, 6] --> [6, 7] []
Pivot 6: [7] --> [] [7]
Pivot 7: [] --> [] []
[3, 3, 4, 4, 5, 6, 7, 8]
>>>
```

Finally, let's see the whole program in Erlang.

```
split(P,L) -> split(P,L,[],[]).

split(_,[],A,B) -> [A,B];
split(P,[H|T],A,B) when H =< P -> split(P,T,[H|A],  B);
split(P,[H|T],A,B)             -> split(P,T,   A,[H|B]).


sort( []   ) -> [];
sort([H|T])  ->
        [A,B] = split(H,T),
        io:format("Pivot ~p: ~p ~p ~p~n",[H,T,A,B]),
        sort(A) ++ [H] ++ sort(B).
```

And here's it in action.

```
Eshell V5.6.3  (abort with ^G)
1> samples:sort([5,4,3,6,7,8,4,3]).
Pivot 5: [4,3,6,7,8,4,3] [3,4,3,4] [8,7,6]
Pivot 3: [4,3,4] [3] [4,4]
Pivot 3: [] [] []
Pivot 4: [4] [4] []
Pivot 4: [] [] []
Pivot 8: [7,6] [6,7] []
Pivot 6: [7] [] [7]
Pivot 7: [] [] []
[3,3,4,4,5,6,7,8]
2>
```

Conclusion

Of course, there is much more to Erlang what's shown here. But in my experience, getting very familiar with this particular pattern of programming was the necessary first step in working with functional programs. If you are new to this, I would suggest that you give yourself some challenges, for example, zip two lists together or append two lists without using the "+" or "++" operators. You'll make lots of mistakes (and find them) but that is often a necessary part of the learning process. Good Luck and have fun.

Index

Erlang for Logic Gates

Introduction

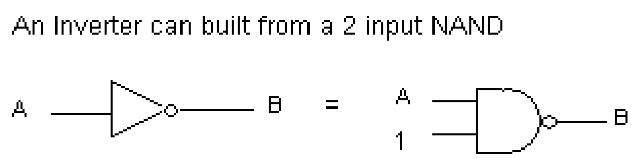
In this section we will explore using Erlangs concurrency capabilities to simulate logic gates and circuits. I'm going to assume some familiarity with the Logic Circuits project, one of the first projects on this site.

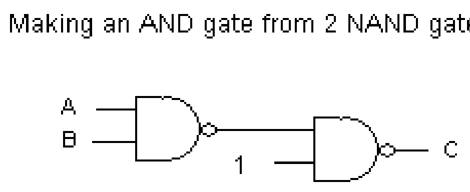
In the earlier project basic logic gates are built as Python objects. Each basic gate has one or more inputs and a single output. As input values change (0 or 1) the output value is recalculated. In addition, the output of any gate is connected to a set of inputs and sends any change to them. A network of simple gates is built into circuits that add and even perform multiplication. The stuff of computers simulated on a computer.

In this project, we'll take a similar route, but with a few twists. Instead of Python objects as basic gates, we'll spawn independent Erlang processes. These processes communicate with one another only through message passing which is exactly what we need for connecting them together.

The basic NAND gate

We actually need only to build a single basic gate, the 2 input NAND gate. All other gates can be built as circuits starting with it alone. The output of NAND is 0 if and only if both inputs are 1. It is a mirror of an AND gate. The NAND gate followed by an INVERTER becomes an AND gate. An INVERTER can, in turn be made from a single NAND by either connecting the 2 inputs together (treating them as one) or leaving one input always set to 1.





Concurrency in Erlang

Erlang has a simple and elegant of spawning processes and of message passing. We won't explore it in full, just enough for our purposes here.

Since processes communicate only through message passing, the top level structure of a process always looks the same. It resembles a server program in a client/server system. That is, it runs in a loop waiting for a message and responding to it.

Let's look at a simple example. A ticket dispenser gives tickets with ascending numbers each time it is used. Here's a little Erlang model in the file ticket.erl.

```
ticket(N) ->
  receive
    _ ->
      io:format("Next ticket is ~p~n", [N]),
      ticket(N+1)
  end.
```

The variable N holds the next ticket number. A message is received in the receive/end block. The pattern "_" will match any message. The next ticket number is printed and the process repeated through tail recursion. We can launch a ticekt dispenser with the following.

```
1> c(ticket).
{ok,ticket}
2> T1  = spawn(fun() -> ticket:ticket(100) end).
```

The "spawn" function takes a function of no arguments as its argument, creates a new process and starts the function passed running within that process. Typically, this function is created on the fly using a fun/end block which is bascially the equivalent of the Python lambda expression. Here, the function in turn calls our "ticket" function passing in its initial ticket number.

It is very convenient to have a helper function to do this spawn for us. For example.

```
makeTicket(N) -> spawn(fun() -> ticket(N) end).
```

Now we can do the following.

```
makeTicket(N) -> spawn(fun() -> ticket(N) end).
```

Now let's watch the following.

```
Eshell V5.6.3  (abort with ^G)
1>  T1  = spawn(fun() -> ticket:ticket(100) end).
<0.33.0>
2> T2 = ticket:makeTicket(200).
<0.35.0>
3> T1 ! 0.
```

```
Next ticket is 100
0
4> T1 ! 0.
Next ticket is 101
0
5> T2 ! 0.
Next ticket is 200
0
6> T2 ! 0.
Next ticket is 201
0
7> T1 ! 0.
Next ticket is 102
0
8>
```

Here we've spawned two ticket dispensers, the first directly and the second using the convenience function "makeTicket". T1 and T2 are set to the "pid" (process id) of each processes. The "!" operator is used to send a message to pid. We're just sending the number zero, but as mentioned above it could be anything since we're matching with "_". As we send messages to these processes they print the next ticket number and then increment. The value returned from the spawn is simply the message sent.

NAND gate as an Erlang process

Let's look at the basic function for the emulation of out NAND gate.

```
nandG(Tag,A,B,Prev,Con) ->
    C = 1-(A band B),
    if not (Prev==C) ->
      propogate(Tag, Prev, C, Con), nandG(Tag,A,B,C,Con);
    true ->
      receive
        {a,V} -> nandG(Tag,V,B,C,Con);
        {b,V} -> nandG(Tag,A,V,C,Con);
        {connect, X} ->
            propogate(Tag,Prev,C,X),
            nandG(Tag,A,B,C,X)
      end
    end.
```

The NAND gate has 2 inputs (A and B) and an output C. The function is passed a Tag for identification and the initial values of A and B. We'll get to the parameters Prev and Con in a moment.

Let's first look at the "receive" portion of the function. A message of the form {a,V} or {b,V} is used to set the input, either A or B to the value of V (0 or 1). Getting such a message results in a tail recursive call, resetting the state of gate. As the function is reentered, the output C is computed. If it is different from the previous output, the new value is propogated by messages to inputs of other gates that are connected to this output. Then the function is reentered once more to basically reset the variable "Prev" and enter the receive loop to wait for more messages.

Connections to other inputs is formatted as a list of tuples. Each tuple consists of a gate identifier, basically its Pid, and a symbol for the input itself. Let's look at the propogate function.

```
propogate(Who, Prev, Val, Con) ->
    io:format("~p was ~p xmits ~p to ~p~n", [Who, Prev, Val, Con]),
    prop2 (Con, Val).

prop2([],Val) -> Val;
prop2([{Gate,Input}|T],Val) -> Gate ! {Input, Val}, prop2(T,Val).
```

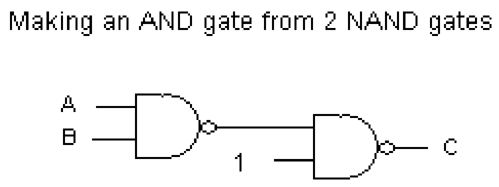
The "propogate" function outputs a console message showing which gate is outputting, what value, to what inputs. The function "prop2" does the real work, extracting the Gate and Input from each tuple, and then sending a message to the receiver gates input with the new value (0 or 1). Notice the variable "Prev" is printed just for information purposes.

Finally, the "nandG" function with a single "Tag" parameter (nandG/1) is the one actually used from the outside. It spawns the nandG/5 function as a new process setting the inputs initially to "1", their natural state when unconnected, and then returns the pid of the new process.

```
nandG(Tag) -> spawn(fun() -> nandG(Tag,1,1,99,[]) end).
```

Making a composite gate

A composite gate combines basic gates at one level, connects them together and then provides inputs and an output from the entire structure. Let's make an AND gate using two NAND gates as shown here.



Again we'll break the system into 2 parts. The first (andG/1) will assemble the parts, wire them together a new process for the composite gate.

```
andG(Tag) ->
 G1 = logic:nandG(g1),
 G2 = logic:nandG(g2),
 G1 ! {connect, [{G2,a}]},    % internal connection
    spawn(fun() -> andG(Tag,G1,G2) end).
```

As you can see, the inner gates (actually, their process ids) are held in the variables G1 and G2. Then the internal connection is made from the output of G1 to input A of G2. This connection is not visible to the outside. Finally, we spawn yet another process to handle the composite gate itself. This process receives message to set its inputs and make connections to its output in exactly the same way a basic gate does.

```
andG(Tag, G1, G2) ->
   receive
     {a,V}        -> G1 ! {a,V}, andG(Tag,G1,G2);
```

```
      {b,V}       -> G1 ! {b,V}, andG(Tag,G1,G2);
    {connect, X} -> G2 ! {connect, X}, andG(Tag,G1,G2)
  end.
```

Let's play with this a bit

```
Eshell V5.6.3  (abort with ^G)
1> And = logic:andG("A1").
g2 was 99 xmits 0 to []
g1 was 99 xmits 0 to []
<0.35.0>
g1 was 0 xmits 0 to [{<0.34.0>,a}]
g2 was 0 xmits 1 to []
2> And ! {a,0}.
g1 was 0 xmits 1 to [{<0.34.0>,a}]
{a,0}
g2 was 1 xmits 0 to []
3> And ! {a,1}.
g1 was 1 xmits 0 to [{<0.34.0>,a}]
{a,1}
g2 was 0 xmits 1 to []
4>
```

In line one we create a composite AND gate giving it the name "A1". Since the io.format call is still in the propogate function we can see the 2 internal gates trying to transmit an output to empty recipricant lists. The pid of our AND gate (0.35.0) is printed as its process is spawned. Then we see more propogation and the interior gates are connected.

On line two we set input A of our composite AND to 0, which brings the composite output to zero as well. In line three we set it back to one.

A monitor process

The output spewed to our screen from the propogate function is really more verbose than we want, at least once we trust the basic logic of our gates. Let's comment out that particular statement leaving us with

```
propogate(Who, Prev, Val, Con) ->
 % io:format("~p was ~p xmits ~p to ~p~n", [Who, Prev, Val, Con]),
  prop2 (Con, Val).

prop2([],Val) -> Val;
prop2([{Gate,Input}|T],Val) -> Gate ! {Input, Val}, prop2(T,Val).
```

and in its place have a function "monitor" that accepts a single input. This will be connected to an output that we wish to monitor. We could have several monitors in a circuit. By providing a tag for each we can keep track of who's talking. Here's the code

```
monitor(Tag) -> spawn(fun() -> monitor(Tag,0) end).
monitor(Tag,_) ->
   receive
      {a,V} -> io:format("Monitor ~p reads ~p~n", [Tag,V]), monitor(Tag,V)
  end.
```

Now let's play with this a bit. I have deleted the values echoed from the message passing commands.

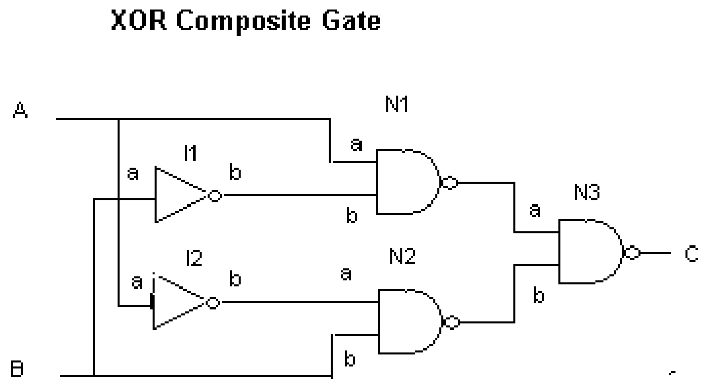
```
3> M1 = logic:monitor(m1).
4> M1 ! {a,1}.
Monitor m1 reads 1
5> M1 ! {a,1}.
Monitor m1 reads 1
6> M1 ! {a,0}.
Monitor m1 reads 0
```

We created a monitor "m1" whose pid is in M1 (line 3). Then we set its input to one resulting in a readout. Setting it back to zero gives another readout. Now let's connect the monitor to the output of a composite gate.

```
6> A1 = logic:andG(a1).
7> A1 ! {connect, [{M1,a}]}.
Monitor m1 reads 1
8> A1 ! {a,0}.
Monitor m1 reads 0
9> A1 ! {a,1}.
Monitor m1 reads 1
```

A more complex example

The following is composite gate for an exclusive OR (XOR) gate. This gate outputs 1 if either input is 1 but not both. Or to put it another way, it outputs 1 if the inputs are different. It is the basis of the half-adder circuit.



See if you can walk through the logic on your own.

Without too much surprise, here is the erlang code for the XOR gate. Though more elaborate than the AND gate, the ideas are still the same.

```
xorG(Tag) ->
 I1 = logic:nandG(i1),     % two inverters (just use one input of nand)
 I2 = logic:nandG(i2),
 N1 = logic:nandG(n1),     % three 2 input nand gates
 N2 = logic:nandG(n2),
 N3 = logic:nandG(n3),

 I1 ! {connect, [{N1,b}]}, % internal connections
 I2 ! {connect, [{N2,a}]},
 N1 ! {connect, [{N3,a}]},
 N2 ! {connect, [{N3,b}]},
 spawn(fun() -> xorG(Tag,N1,N2,N3,I1,I2) end).

xorG(Tag,N1,N2,N3,I1,I2) ->
   receive
     {a,V}       -> N1 ! {a,V}, I2 ! {a,V}, xorG(Tag,N1,N2,N3,I1,I2);
     {b,V}       -> N2 ! {b,V}, I1 ! {a,V}, xorG(Tag,N1,N2,N3,I1,I2);
   {connect, X} -> N3 ! {connect, X}     , xorG(Tag,N1,N2,N3,I1,I2)
 end.
```

If you didn't walk through the logic of the XOR gate on your own, here's my rendition. N3 outputs 1 if either of its inputs is zero. N1 outputs zero if A is 1 and B is 0. N2 outputs zero if B is 1 and A is 0.

And here is an interaction with an Xor gate and a monitor on its output. Again extra echoing of values by the erlang shell have been removed.

```
11> X1 = logic:xorG(x1).
12> M2 = logic:monitor(x1_monitor).
13> X1 ! {connect, [{M2,a}]}.
Monitor x1_monitor reads 0
14> X1 ! {b,0}.
Monitor x1_monitor reads 1
15> X1 ! {a,0}.
Monitor x1_monitor reads 0
```

Of course we could extend all this to emulate adders, registers, and other circuits built in the earlier Python project. It would quickly swell to hunderds of Erlang processes giving the system quite a nice workout.

It would also be interesting to extend these ideas into the realm of analog electronics building units for resisters, capacitors, coils, diodes and transistors. Well, maybe something for another rainy day in Oregon. :)

Links to the Erlang code

Finally, here are links to ticket.erl and logic.erl code.

Index

FORTH - A simple stack oriented language

What is Forth?

In a nutshell, Forth is a combination compiler and interpreter. The compiler translate source code not to machine code like we saw in the previous chapter, but into instructions for a "virtual" machine, which we'll refer to as "pcode". The same idea is used in Java and Python and is very in modern dynamic languages.

However, in Forth, we program at this pcode level, at what we might almost call an assembly language for the virtual machine. But it is language that can extended in interesting ways that give it quite a dynamic character.

With Forth you play what I would like to call "snippets of computation", referred to as "words" that manipulate a push down data stack directly or get strung together to build (compile) new "words", which in turn do the same things just one level higher.

Although no longer a commonly used, Forth has features that make it rewarding to study. The language has a wonderful minimalism that can be appreciated only with the study of example code.

Forth was a good compromise for building programs relatively easily, compared with assembler, that would use surprisely little memory and could be run surprisely fast. I built a system with a resident compiler, interpreter and Forth appliction code with concurrent threads all sitting in a few kilobytes of computer memory and running at about half the speed of the same program written in assembler.

According to Wikipedia, Forth is still used for such things as boot loaders, embedded systems and space applications (refs). There is an active implementation by the GNU Project.

In this chapter we'll build a small Forth compiler/interpreter system in Python and examine it in detail. Our implementation is very basic and small. Numbers (floats and ints) are the only data type, and I/O is only through the terminal.

Later on, we'll briefly look at the application mentioned above, a process control system for a newspaper mailroom. Sit back and enjoy the ride.

A word about quoting convention. If I use a word in all caps it is a Forth word. Our Forth is actually case insensitive, as you'll see, but traditionally Forth programs were in upper case. Words with mixed case are entities in the Python code. TOS is top-of-stack. Other words will be quoted when they stand for themselves.

Using the Data Stack

The "data stack" is the central feature, like the stove in your kitchen. And because we manipulate it directly, we have to compromise a little for the language. We have to get comfortable working in postfix.

Let's jump into an example. Click Here to access forth.py

```
$ python forth.py
Forth> 5 6 + 7 8 + * .
165
```

Here we are evaluating the product of two sums "(5+6)*(7+8)" which in normal infix notation requires that the sums be placed in parenthesis to ensure the multiplication follows the additions. Now, in a prefix notation, like that used in the Lisp language, one more pair of parans are needed "(* (+ 5 6) (+ 7 8))". But with Forth we use postfix notation and the need for parens simply disappears. Operators are simply placed where they are used in the computation.

(A side note. If, by chance, you know some Japanese then you are aware that thinking in postfix is not that unnatural)

We can see what is going on a little more clear by using the Forth word "dump" to display the data stack at any point in time. Let's go a little wild and show the data stack after each single operation.

```
Forth> 5 dump 6 dump + dump 7 dump 8 dump + dump * dump
ds =  [5]
ds =  [5, 6]
ds =  [11]
ds =  [11, 7]
ds =  [11, 7, 8]
ds =  [11, 15]
ds =  [165]
Forth>
```

Hopefully that will make it crystal clear.

Built in primitive words for data stack manipulation

Along with add, subtract, multiply and divide there are other basic builtin words that manipulate the stack as well.

- DUP duplicates the top of stack (TOS)
- SWAP interchanges the top two elements
- DROP pops the TOS discarding it
- "." pops and prints the TOS
- "=" pops the top two elements, compares them, and push 1 onto the stack if there are equal, and 0 if not

These can all be combined in clever ways. Here are two examples.

```
Forth> # Lets square and print the TOS
Forth> 25 dup * .
625
Forth> # Lets negate the TOS and print it
Forth> 42 0 swap - .
-42
Forth>
```

Here is the Python code for some of the basic runtime functions in forth.py. Our data stack is simply called "ds" and is a python list. We use the list "append" method to push items onto the stack and its "pop" method to get (and drop) the top element. Each basic word is a tiny function with two arguments. These functions do not use these arguments but we'll later see other runtime functions that will.

```python
def rAdd (cod,p) : b=ds.pop(); a=ds.pop(); ds.append(a+b)
def rMul (cod,p) : b=ds.pop(); a=ds.pop(); ds.append(a*b)
def rSub (cod,p) : b=ds.pop(); a=ds.pop(); ds.append(a-b)
def rDiv (cod,p) : b=ds.pop(); a=ds.pop(); ds.append(a/b)
def rEq  (cod,p) : b=ds.pop(); a=ds.pop(); ds.append(int(a==b))
def rGt  (cod,p) : b=ds.pop(); a=ds.pop(); ds.append(int(a>b))
def rLt  (cod,p) : b=ds.pop(); a=ds.pop(); ds.append(int(a<b))
def rSwap(cod,p) : a=ds.pop(); b=ds.pop(); ds.append(a); ds.append(b)
def rDup (cod,p) : ds.append(ds[-1])
def rDrop(cod,p) : ds.pop()
def rOver(cod,p) : ds.append(ds[-2])
def rDump(cod,p) : print "ds = ", ds
def rDot (cod,p) : print ds.pop()
```

Some of these could, of course, be done more efficiently but we'll opt for clarity. Notice the order of operations, especially with divide and subtract. Also, with the divide operator we are using the Python "/" so if at least one argument is a float, the result will be float. If both are integers, then integer division is done.

A reference to each of these functions is in the lookup table "rDict" which matches operators like "+" to their function (rAdd). These dictionary entries are straightforward.

Defining new words

Let's define a word NEGATE to replace the TOS with its negative. Every word definition starts with a ":" immediately followed by the word we want to define. Then everything following up to the closing ";" becomes the body of the definition. So let's define NEGATE and test it and then do the same for SQR.

```
Forth> : negate 0 swap - ;
Forth> 5 negate .
-5
Forth> : sqr dup * ;
Forth> 6 sqr .
36
Forth>
```

Compiling Words to Pcode

The first step in compilation is lexical parsing, which in Forth is really simple. The "tokenize" function first strips comments, which in this little compiler is anything from "#" to the end of a line. Then it simply splits the text to a list of words, breaking on whitespace. Let's watch this from the Python prompt.

```
>>> import forth
>>> forth.tokenizeWords("5 6 + .  # this is a comment")
>>> forth.words
['5', '6', '+', '.']
>>>
```

When the compiler is not in the middle of defining a new word, or, as we'll see later, building a control structure, it operates in what is called "immediate mode". This means that a single word will be compiled and returned.

```
>>> forth.compile()
Forth> 5 6 + .
[<function rPush at 0x9bdf5a4>, 5]
>>> forth.compile()
[<function rPush at 0x9bdf5a4>, 6]
>>> forth.compile()
[<function rAdd at 0x9bdf1ec>]
>>> forth.compile()
[<function rDot at 0x9bdf48c>]
>>> forth.words
[]
>>>
```

Notice how running the compile function automatically prompts us for input.

Compiled code comes in lists. A number, either a integer or float compiles to an "rPush" of the number onto the data stack. "+" and "." simply compile to their corresponding runtime functions. Let's look at the Python "compile" function.

```
def compile() :
    pcode = []; prompt = "Forth> "
    while 1 :
        word = getWord(prompt)  # get next word
        cAct = cDict.get(word)  # Is there a compile time action ?
        rAct = rDict.get(word)  # Is there a runtime action ?

        if cAct : cAct(pcode)   # run at compile time
        elif rAct :
            if type(rAct) == type([]) :
                pcode.append(rRun)     # Compiled word.
                pcode.append(word)     # for now do dynamic lookup
            else : pcode.append(rAct)  # push builtin for runtime
        else :
            # Number to be pushed onto ds at runtime
            pcode.append(rPush)
            try : pcode.append(int(word))
            except :
                try: pcode.append(float(word))
                except :
```

```
            pcode[-1] = rRun     # Change rPush to rRun
            pcode.append(word)  # Assume word will be defined
    if not cStack : return pcode
    prompt = "... "
```

Part of this may be a bit obscure. Basically, we get the next word using "getWord", which if it needs more input, will prompt for it. This "compile" function controls whether the prompt is "Forth> " or "... ". We'll see how that is used in a bit. If the word names a basic runtime action in "rDict", then it is translated to a call of that function. If the word can be made into an integer or float then "rPush" (also a builtin) becomes the translation followed by the actual number to be pushed onto the data stack. In immediate mode a single input word will be compiled and returned.

Next, let's compile a new word.

```
>>> forth.compile()
Forth> : negate 0 swap - ;
[]
```

No pcode is returned! But there is an interesting side effect

```
>>> forth.rDict['negate']
[<function rPush at 0x9fbe5a4>, 0, <function rSwap at 0x9fbe374>,
 <function rSub at 0x9fbe25c>]
>>>
```

We have a new entry in the rDict which can now be used just like a builtin

```
>>> forth.compile()
Forth> 6 negate
[<function rPush at 0x9fbe5a4>, 6]
>>> forth.compile()
[<function rRun at 0x9fbe56c>, 'negate']
>>>
```

Notice in the "compile" function above the check for cAct in cDict. This is looking for a compile time function. These functions are helper functions for the compilation process. Both ":" and ";" are compile time words. Let's look at their corresponding python functions

```
def cColon (pcode) :
   if cStack : fatal(": inside Control stack: %s" % cStack)
   label = getWord()
   cStack.append(("COLON",label))  # flag for following ";"

def cSemi (pcode) :
   if not cStack : fatal("No : for ; to match")
   code,label = cStack.pop()
   if code != "COLON" : fatal(": not balanced with ;")
   rDict[label] = pcode[:]      # Save word definition in rDict
   while pcode : pcode.pop()
```

You can see that an entry is pushed onto the cStack with the ":" and later popped by the

";". The ":" word gets a label from the input, and if it were not available you would be prompted for it with "...". This label is saved for the ";" word to connect it with the compiled code and make an entry in rDict. Once this happens the code is then erased. Now, since cStack is not empty during this time, compilation once started by ":" must proceed to the matching ";". Then we say that the compiler is running in the "deferred" mode.

So now let's look at other word groups that also use the cStack, forcing deferred compilation.

BEGIN ... UNTIL Control structure

The BEGIN and UNTIL words set up an iterative loop. UNTIL will pop the TOS and, if it is zero, will return control to the word following BEGIN. Here is an example

```
>>> import forth
>>> forth.main()
Forth> 5 begin dup . 1 - dup 0 = until
5
4
3
2
1
Forth>
```

We start by pushing 5 onto the stack, print it, subtract 1, and repeat until it is zero. Notice the use of the word DUP twice, needed to preserve our number as we are not only counting it down, but also printing it and checking for zero.

Both BEGIN and UNTIL are compile time words. Here are their corresponding Python functions

```
def cBegin (pcode) :
   cStack.append(("BEGIN",len(pcode)))  # flag for following UNTIL□

def cUntil (pcode) :
   if not cStack : fatal("No BEGIN for UNTIL to match")
   code,slot = cStack.pop()
   if code != "BEGIN" : fatal("UNTIL preceded by %s (not BEGIN)" % code)
   pcode.append(rJz)
   pcode.append(slot)
```

BEGIN generates no code. It simply pushes the address (len(pcode)) of the next word to come onto the control stack and, thereby, puts the compiler into deferred mode. UNTIL checks that it has a matching BEGIN and generates an rJZ call (Jump if Zero) back to the address saved. At runtim rJz will pop the TOS and do the jump if it is zero.

Here is a nice "Forth-like" word definition for computing the factorial of the TOS. This code has been put into a file "fact1.4th"

```
# fact1.4th

: fact                        #  n --- n!  replace TOS with its factorial
  0 swap                        # place a zero below n
  begin dup 1 - dup  1 = until    # make stack like 0 n ... 4 3 2 1
  begin dump *  over 0 = until     # multiply till see the zero below answer
```

```
   swap drop ;                    # delete the zero
```

Notice the DUMP in the middle for debugging. Let's run this.

```
>>> import forth
>>> forth.main()
Forth> @fact1.4th
Forth> 5 fact .
ds =  [0, 5, 4, 3, 2, 1]
ds =  [0, 5, 4, 3, 2]
ds =  [0, 5, 4, 6]
ds =  [0, 5, 24]
120
Forth>
```

IF, ELSE, THEN Control Structure

The format of this control structure is

```
condition IF true-clause THEN

  or

condition IF true-clause ELSE false-clause THEN
```

Once again, if you are somewhat familiar with Japanese grammer, this ordering will not seem terribly unnatural. Let's look at the compile time helpers for these words.

```
def cIf (pcode) :
    pcode.append(rJz)
    cStack.append(("IF",len(pcode)))  # flag for following Then or Else
    pcode.append(0)                   # slot to be filled in

def cElse (pcode) :
    if not cStack : fatal("No IF for ELSE to match")
    code,slot = cStack.pop()
    if code != "IF" : fatal("ELSE preceded by %s (not IF)" % code)
    pcode.append(rJmp)
    cStack.append(("ELSE",len(pcode)))  # flag for following THEN
    pcode.append(0)                     # slot to be filled in
    pcode[slot] = len(pcode)            # close JZ for IF

def cThen (pcode) :
    if not cStack : fatal("No IF for ELSE for THEN to match")
    code,slot = cStack.pop()
    if code not in ("IF","ELSE") : fatal("THEN preceded by %s (not IF or ELSE)" % code)
    pcode[slot] = len(pcode)            # close JZ for IF or JMP for ELSE
```

This should look familiar after studying the compile time code for BEGIN and UNTIL. IF will generate a rJz to the ELSE if there is one, or, if not, to the THEN. An ELSE will complete the jump for the IF and set up an unconditional jump to be completed by the THEN. THEN has to complete whichever jump is needed, from an IF or ELSE.

Here is a second factorial word definition. Instead of using iteration with BEGIN and

UNTIL, we use recursion, a topic that will be discussed later in more depth. And rather than using a flag for establishing an end-point to the computation, we'll use our new IF, THEN team.

```
# fact2.4th  Recursive factorial      # n --- n!

: fact  dup 1 > if                  # if 1 (or 0) just leave on stack
          dup 1 - fact              # next number down - get its factorial
    dump    * then                  # and mult - leavin ans on stack
  ;
```

Again, a DUMP has been stratigically placed to watch the action

```
Forth> @fact2.4th
Forth> 5 fact .
ds =  [5, 4, 3, 2, 1]
ds =  [5, 4, 3, 2]
ds =  [5, 4, 6]
ds =  [5, 24]
120
Forth>
```

Variables, Constants and the Heap

There comes a point when we need to store data outside of the data stack itself. We may want to more conveniently hold onto values independently of a changing data stack and we may want to store data in single or multi-dimensional arrays.

Forth systems set aside an area of memory for this purpose and contain a few builtins that are used to build words that, in turn, build other words. It's quite neat.

First, let's look at the builtins themselves. In our model the "heap" is simply a list of 20 integers. Obviously, it's not meant for real time.

```
heap     = [0]*20      # The data heap
heapNext =  0          # Next avail slot in heap

def rCreate (pcode,p) :
    global heapNext, lastCreate
    lastCreate = label = getWord()      # match next word (input) to next heap address
    rDict[label] = [rPush, heapNext]    # when created word is run, pushes its address

def rAllot (cod,p) :
    global heapNext
    heapNext += ds.pop()               # reserve n words for last create
```

Now, interestingly, both rCreate and rAllot are runtime words. When rCreate runs, the compiler is in immediate mode and the next word in the input (NOT in the pcode) will be the word being defined. The runtime action of the defined word will be to simply place its reserved heap address onto the data stack. ALLOT will actually reserve one or more words, advancing heapNext to be ready for a future CREATE. Let's look at an example. We'll use forth.main to both compile and execute our Forth code.

```
>>> import forth
>>> forth.main()
Forth> create v1 1 allot
Forth> create v2 3 allot
Forth> create v3 1 allot
Forth> dump
ds =  []
Forth> v1 v2 v3 dump
ds =  [0, 1, 4]
Forth>
```

Here we have created three new words V1, V2, and V3. Each has a corresponding heap address and when they are run each pushes its address. Notice the "space" between V2 and V3 due to the ALLOT used with the creation of V2.

And here is how we use these heap locations. Two builtins "@" and "!" fetch and set words in the heap. Let's play with this a bit

```
>>> import forth
>>> forth.main()
Forth> create v1 1 allot
Forth> create v2 3 allot
Forth> create v3 1 allot
Forth> ^D
>>> print forth.heapNext, forth.heap
5 [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
>>> forth.main()
Forth> 22 v2 !          # set v2[0] = 22
Forth> 23 v2 1 + !      # set v2[1] = 23
Forth> v2 @ .           # print v2[0]
22
Forth> ^D
>>> print forth.heapNext, forth.heap
5 [0, 22, 23, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
>>> forth.rDict['v3']  # See definition of v3□
[<function rPush at 0x8ba0454>, 4]
```

So, we created 2 simple variables (V1 and V3) and an array of 3 words V2. Dropping back into Python we can see the heap being modified and also a runtime definition for one of the variables.

Another useful builtin is "," which pops the TOS and pushes it onto the heap at the next free location. This can be used instead of ALLOT to initialize variables to zero.

```
>>> forth.main()
Forth> : varzero create 0 , ;
Forth> varzero xyz
Forth> xyz @ .
0
```

Finally, there is a builtin DOES> that works together with CREATE inside a definition. It takes all following words up to the closing ";" and appends them to the rDict entry for the word just created. First consider this definition and usage of a CONSTANT

```
Forth> : constant create , ;
Forth> 2009 constant thisYear
Forth> thisYear @ .
2009
```

That's fine, but there is nothing to prevent you from simply "!"ing thisYear to another value. It's the same as any variable. But if we instead do the following

```
Forth> : constant create , does> @ ;
Forth> 2009 constant thisYear
Forth> thisYear .
2009
Forth> ^D
>>> forth.rDict['thisyear']
[<function rPush at 0x9865454>, 3, <function rAt at 0x9865534>]
>>>
```

then you can see how a call to rAt has been attached to thisYear's runtime. There is no need to use an "@" to retrieve its value and "!" cannot be used to change its value.

With these builtins, it is not difficult to make arrays, including multidimensional ones. We can even make primitive structs (like C) assigning constants for field offsets.

Like C, however, there is not much memory protection. If an array reference is out of bounds, it will simply write into something else. Of course, in this model, this will stay within our heap but in a C adaptation of this code, even that would be lost.

We'll end this section with a final factorial program, this time using variables. The code looks more like a traditional language, albeit in postfix

```
# fact3.4th

: variable create 1 allot ;          #   ---    create var and init to TOS
  variable m
  variable answer

: fact                              #  n --- n!  replace TOS with factorial
     m !                            # set m to TOS
  1 answer !                           # set answer = 1
  begin
    answer @ m @ dump * answer !      # set answer = answer * m
    m @ 1 - m !                     # set m = m-1
    m @ 0 = until                    # repeat until m == 0
  answer @                           # return answer
  ;

 15 fact .                        # compute 15! and print
```

And now, let's run it

```
Forth> @fact3.4th
ds =  [1, 15]
ds =  [15, 14]
ds =  [210, 13]
ds =  [2730, 12]
ds =  [32760, 11]
ds =  [360360, 10]
ds =  [3603600, 9]
ds =  [32432400, 8]
ds =  [259459200, 7]
ds =  [1816214400, 6]
ds =  [10897286400L, 5]
ds =  [54486432000L, 4]
ds =  [217945728000L, 3]
ds =  [653837184000L, 2]
ds =  [1307674368000L, 1]
1307674368000
Forth>
```

Since our model is built in Python, we inherit its nice automatic switch to long integers.

Notice that the variables "m" and "answer" are defined outside the "fact" definition. We don't have private local variables within a definition.

Other Issues

As mentioned earlier, Forth can be very efficient both with memory and CPU time. Consider the following bits of PDP-11 assembler code. It is a recreation of a little bit of our first Forth expression "5 6 +".

```
stack:
        ...      ;     more space here for the stack to grow
        6         ; <--- r4 stack pointer (stack is upside down)
        5
ds:     0         ; base of stack

        rPush     ; <--- r3 tracks the pcode thread
        5
        rPush
        6
        rAdd
        .
        .        ; thread continues ...


rPush:   mov    (r3)+, -(r4)       ; ds.append[pcode[p]]; p += 1
        jmp   @(r3)+             ; return to thread

rAdd:    add    (r4)+, (r4)        ; tmp=ds.pop(); ds[-1] += tmp
        jmp   @(r3)+             ; return to thread

rDup:    mov    (r4),-(r4)         ; ds.append(ds[-1])
        jmp   @(r3)+             ; return to thread
```

This should look somewhat familiar from the assembler model in the previous chapter. We have the data stack on top, a bit of "threaded" code in the middle and 3 builtins. The threaded code (the name will be obvious in a minute) is essentially the same as our "pcode" array in the Python model. Machine register r3 is our "p" indexe to the next word in the pcode. The program counter, PC jumps between the builtins. The instruction "jmp @(r3)+" loads the program counter with the memory word indexed by r3 and then increments r3 to point at the next word. The program execution weaves through the threaded code out of one builtin (rPush) and into the next (rAdd). Register r4 is the ds index. On the PDP-11 the stack grew downward and lower machine addresses were actually higher on the stack. The instruction "mov (r3)+,-(r4)" pushes the next word in the thread (5 say) onto the data stack, first decrementing r4 to the next higher stack location.

Now if we were writing this in assembler we might do the following

```
        mov  #5, -(r4)       ; push 5 onto stack
        mov  #6, -(r4)       ; push 6 onto stack
        add  (r4)+,(r4)      ; add in stack
```

But if we add up the memory use and execution cycles, only the "jmp @(r3)+" instructions, the needle, if you will, that sews the code together are missing. These jumps constitute very little overhead.

In the early 1980's I developed a process control system for a newspaper mailroom that tracked bundles of newspapers on conveyor belts. Each bundle had a specific chute designation which would deliver it to a truck being loaded. We put together a small Forth system in less than 1000 lines of assembler. This system was concurrent having a special word so that one thread could yield control of the cpu to another. Like Forth itself, this application was divided into tiny pieces sewn back together. One pcode thread for example monitored keyboard input from one of the terminals. Another output messages to all the screens which included echoing input from either keyboard. Still other threads would handle a sensor on a belt or update a light display. Each thread had its own data stack but they shared variables on the heap and of course word definitions. There were no locking problems because the threads themselves yielded control only when safe to do so. Such a system was possible because throughput was only at human speed.

One final point. We used recursion in the second factorial example. This is unusual in Forth. Normally a word must be defined before being used in another definition. But in our compile function the last "except" clause allows us to build an rRun to an undefined word with the assumption that it will be defined before it is actually used. But this in turn leads to another issue. Our rRun runs a word dynamically, that is, it looks up the definition in rDict just before running it. Most Forths would not do that. It's expensive when the computation for most words is usually so small. So rather than following a pcode entry "rRun" with the name of a word, it would be reference to the words pcode and the dictionary lookup is avoided. This also has an interesting implication. If you redefine a word that has been used in the definition of other words, those other words do not change their behaviour. They are still locked to the old code. The programmer might well find this unexpected.

Index

Lisp in Python

Why Lisp?

Even though we are already programming in Python, which has many of the features of Lisp, it is instructive to look at the original Lisp evaluation mechanism. At the heart of the Lisp language is an recursive interplay between evaluating expressions, which means applying functions to arguments. But that requires further evaluation of the functions arguments and internal expressions. The result is an extremely elegant piece of code.

When I was a student of computer science at the University of Oregon in 1971, I took a course in Artificial Intelligence. We got exposed to some of the classic Lisp programs but, unfortunately, had no way of running them. So our projects were written in Fortran and were not very impressive.

So when I came across "Lisp in Lisp" taking up most of page 13 of the Lisp 1.5 Users Manual I had a stronger motivation than just esthetics to do something with it. I figured that if I could translate just that much Lisp into Fortran, then I would have the means to run other Lisp programs. It was much easier said than done. Fortran did not support recursion and this was an example of true "functional programming". There was not a single GOTO or variable assignment. Each function call and return had to done in Fortran with computed GOTO's, first pushing or popping information on stacks implemented with Fortran arrays. It was very messy. Later I did a version in PDP-11 Assembler and, still later, one in Pascal which was pretty clean.

This Lisp implemented in Python is mostly a translation of the original Lisp. If you have a chance to look at the original Lisp in Lisp, I think you'll agree with me that the Python code is much easier to get your head around. Like the original it is very short and, except for input of S expressions, completely functional. That basically means no variable assignments or explicit loops. Where we would normally use a "for" or "while" loop, you will see tail recursion.

I made a few small changes in this design. Lisp supports both lists and binary trees. In fact lists in Lisp are simply a special form of tree. Our Python Lisp supports only lists. In addition I added two small commands that will let us use the program interactively.

Introduction to Lisp

The basis for both programs and data in Lisp is the S (symbolic) expression. An S expression may be a symbol, a number, or a series of S expressions in parentheses. Here are some examples.

```
george
```

```
54
(george 54 (sue 48))
```

As you can see S expressions may be nested.

Certain forms of S expressions may be evaluated. For example "(+ 5 4)" would apply the primitive function '+' to the arguments (5 and 4) and return the number 9, which is also an S expression. All function calls list the function name first, followed by the arguments. Here are some more examples of S expression evaluation.

```
S expression        Evaluation   Comments

234                 234          numbers evaluate to themselves
(quote charlie)     charlie      quote stops further evaluation
(quote (a b c))     (a b c)      quote stops further evaluation
'charlie            charlie      'x is shorthand for (quote x)
t                   t            symbol for "true"
nil                 nil          symbol for "false" same as ()
(eq 5 5)            t            eq returns t or nil
(eq 5 6)            nil
(car '(a b c))      a            car returns 1st item in a list
(cdr '(a b c))      (b c)        cdr returns rest of a list
(cons 'a '(b c))    (a b c)      cons combines args to make a new list
```

Notice that we used (car '(a b c)) instead of (car (a b c)). The quote is necessary to keep (a b c) from another layer of evaluation. This will be clearer as we proceed.

When eval is called it is passed the S expression and also an association list. The above evaluations did not need the association list (alist for short) because we were evaluating either constants or functions whose arguments are constants.

Here is an example alist.

```
((a 2) (b 6) (c (george 45)))
```

It pairs the variables a to 2, b to 6, and c to (george 45). Here are some more sample evaluations assuming this alist.

```
S expression  Evaluation  Comments

c             (george 45) variables are looked up in the alist
(eq a 2)      t           arguments to a function are evaluated first
(eq a b)      nil
(car c)       george
```

Finally, there are a few special forms of S expressions. These are not functions even though the look like function calls. Their arguments are not evaluated before processing. One we've already seen is `quote`. Another is the "conditional" `cond` which is very much like a `case` or `switch` statement in other languages, or like a Python `if`, `elif`, ... `else`. It takes the following form.

```
(cond  A B ...)
```

where A, B, etc. are lists of two elements each. The first element of each pair is evaluated until one is true (not nil). Then the second element of that pair is evaluated and that value is returned as the value of the cond. The remaining pairs are not evaluated. Generally the last pair has `t` for its first element which makes it work like an `else`. For example with the `alist` above

```
(cond ((eq a 1) (cdr george)) (t 3))   would return 3
(cond ((eq a 2) (cdr george)) (t 3))   would return (45)
```

Another special form is used for user defining functions. It is easiest to provide an example and explain it. The following is a function definition to square a number.

```
(lambda (x) (* x x))
```

The symbol `lambda` introduces this form. It is followed by an S expression with the function parameters, and an S expression which is the body of the funciton. It may be applied to arguments just like any primitive function. Again, assuming we have the alist above ...

```
((lambda (x) (* x x)) a)     evaluates to 4.
```

In evaluation the argument a is evaluated (yields 2). Then the lambda expression is applied to 2. The parameter `x` is paired with 2 on the alist which now looks like

```
((x 2) (a 2) (b 6) (c (george 45)))
```

Finally, (* x x) is evaluated. x is replaced with 2 from the alist and the primitive function "*" is applied yielding 4.

I added one special form not in the original code. (def x y) will bind a name x to an S expression y. The alist is saved in a global variable when these definitions are made and therefore remain for later evaluations. This form is especially useful to bind names to functions. For example

```
(def square (lambda (x) (* x x)))
(sq 4)
```

Representing S expressions in Python.

Python lists are convenient to store S expressions. Nested S expressions can be handled simply with nested lists. Strings may be used for symbols and numbers can represent themselves. So our S expression `(lambda (x) (* x x))` would be `['lambda', ['x'], ['*','x','x']]`.

Tail Recursion

We should talk about tail recursion a bit. It is used in our Python code although sometimes we could have used a `for` or `while` loop instead. However if you want to create Lisp functions then you must use tail recursion because we are not providing any other means of iteration!

Lets look at an example. A call to `assoc(x,alist)` walks down the name/value pairs in the alist until it finds a pair whose 1st element matches x. Then it returns the 2nd element (the value). Here is how to write assoc using a `for` loop.

```
def assoc (x, alist) :
   for pair in alist :
      if pair[0] == x : return pair[1]
   raise "Lisp error"
```

With tail recursion the function looks like

```
def assoc (x, alist) :
   if   not alist        : raise "Lisp error"
   elif alist[0][0] == x : return alist[0][1]
   else                  : return assoc(x,alist[1:])
```

There are 3 possibilities. If the first pair on the alist is the one we want, return its 2nd element. If there is no 1st element, raise an error. Or simply search the rest of the alist recursively. Eventually either the right pair will be found or an error will be raised.

Walking through the Code.

At this point you will probably want to bring up the code in a separate window, or just print it out. It's a couple of pages. The two modules are lisp.py and lispio.py

In the module lispio.py we have a function `getSexp` to input an S expression from the user and return the equivalent Python list. We also the have a function `putSexp` to convert a Python list (representing an S expression) to a string in Lisp.

The function `getSexp` uses `getToken` to extract symbols, numbers and special characters from the input provided by the user. It also uses itself to extract nested S expressions. In turn, `getToken` uses `nextChar` to see if the next character is part of the token it is building and `getChar` to grab it. So at the bottom of the food chain is `nextChar` which actually has to deal with getting input from the user.

Lets play with getToken a bit.

```
Python 1.5.2 (#6, Aug 31 2000, 10:56:07)  [GCC 2.8.1] on sunos5
Copyright 1991-1995 Stichting Mathematisch Centrum, Amsterdam
>>> import lispio
>>> while 1 :
...     print lispio.getToken()
...
Lisp>a b
a
b
Lisp>(car '(a b))
(
car
'
(
a
b
)
)
Lisp>
```

Now let's play with getSexp to see how Lisp S expressions are converted to Python lists.

```
Copyright 1991-1995 Stichting Mathematisch Centrum, Amsterdam
>>> import lispio
>>> while 1 :
...     print lispio.getSexp()
...
Lisp>(car '(a b))
['car', ['quote', ['a', 'b']]]
```

```
Lisp>(eq a 5)
['eq', 'a', 5.0]
Lisp>
```

Notice that there are 2 versions of the function `nextChar`. One just uses `raw_input` which means the user must type everything in. The other uses the userInput module letting us put programs in files and running them thru the indirect file mechanism. For more information on the UserInput module click here.

The main functions in lisp.py are apply and eval. Each is a set of if/elif/else statements that handle the various options discussed above. The handling is done with the Python list operators. The other functions, pairlis, assoc, evcon, and evlis are just auxillary functions. The names are the same as in the original Lisp code. You will notice that they are tail recursive.

The function `pairlis` adds pairs to the alist (returning an expanded alist) and `assoc` finds values for variables in the alist passed. The function `evcon` is special for handling cond expressions as described above. `evlis` is similar to the Python `map` function. It evaluates each item in the list passed and returns a list of the values.

The lisp main function interacts with the user. As long as the user inputs an S expression to evaluate, it is passed to eval and the result printed. If lispio.nextchar is using the userInput module then the S expressions may be loaded from a file. This is generally used to define functions.

There are two extra commands handled in eval. `def` allows us to set variables or define functions by adding a pair onto the global Alist. And the special symbol `alist` is permanently set to the global Alist letting us view it easily. When the user enters a new S expression to evaluate, we start out with this preloaded alist.

Lets use the interactive mode to evaluate some expresssions. Here we use `def` to set the variable a to 6 on the alist.

```
>>> import lisp
>>> lisp.main()
Lisp>(def a 6)
a
Lisp>alist
((a 6.0))
Lisp>
```

Next we'll do an addition. The global "debug" is set to one so that each call of eval and apply will be printed.

```
Lisp>(+ a 7)
--Eval--- (+ a 7.0)  alist= ((a 6.0))
--Eval--- a  alist= ((a 6.0))
--Eval--- 7.0  alist= ((a 6.0))
--Apply-- +  Args= (6.0 7.0)  alist= ((a 6.0))
13.0
```

Next we'll define a function sq to square a number and then use it to calculate a**2.

```
Lisp>(def sq (lambda (x) (* x x)))
--Eval--- (def sq (lambda (x) (* x x)))  alist= ((a 6.0))
sq
Lisp>(sq a)
--Eval--- (sq a)  alist= ((sq (lambda (x) (* x x))) (a 6.0))
--Eval--- a  alist= ((sq (lambda (x) (* x x))) (a 6.0))
```

```
--Apply-- sq  Args= (6.0)  alist= ((sq (lambda (x) (* x x))) (a 6.0))
--Eval--- sq  alist= ((sq (lambda (x) (* x x))) (a 6.0))
--Apply-- (lambda (x) (* x x))  Args= (6.0)  alist= ((sq (lambda (x) (* x x))) (a 6.0))
--Eval--- (* x x)  alist= ((x 6.0) (sq (lambda (x) (* x x))) (a 6.0))
--Eval--- x  alist= ((x 6.0) (sq (lambda (x) (* x x))) (a 6.0))
--Eval--- x  alist= ((x 6.0) (sq (lambda (x) (* x x))) (a 6.0))
--Apply-- *  Args= (6.0 6.0)  alist= ((x 6.0) (sq (lambda (x) (* x x))) (a 6.0))
36.0
Lisp>
```

Setting debug back to 0 will enable a more natural, if less informative, interaction.

We can prepare a function definition in a file and invoke its definition. Here is a definition for the function `length` which returns the number of S expressions in a list. I used an indentation style that matches left and right parens either on the same line or vertically.

```
(def length
   (lambda (x)
      (cond
         ((not x) 0)
         (   t   (+ 1 (length (cdr x))))
      )
   )
)
```

This function is another example of tail recursion. It counts one for the first element of a list and adds that to the length of the rest of the list. An empty list returns zero.

```
Lisp>@length.lsp
length
Lisp>(length '(a b c d e f g))
7.0
Lisp>(length length)
3.0
```

Can you explain why the length of length is 3?

Dynamic Scope

An interesting property in this language emerges from using the alist to hold values. Consider the following.

```
Lisp>(def a (lambda (x) (b)))
a
Lisp>(def b (lambda () (+ x x)))
b
Lisp>(a 5)
10.0
```

The function `b` is able to see the value for `x` even though `x` is not an argument. In fact the function `b` doesn't take arguments. But since the value of `x` is determined by a simple search of the alist, its setting from the calling function `a` is found.

Some Ideas for Projects.

You can extend this program in many ways. Extra primitives like turning the debug flag on and off, or providing access to user input would be useful. The system could also be expanded with functions written in Lisp (like length) that are loaded at input. A more complete system could be a combination of Python and Lisp. For the ambitious, the

Python code could be translated back to Lisp and then you could run Lisp in Lisp in Python. Compare its speed to just Lisp in Python and then to a regular interpreter like Scheme.

Index

Prolog in Python. Introduction

As of May 2009 this web page has been updated

Someone once made the remark that there were only two kinds of programming languages, Lisp and all the others. At that time the primary languages like Fortran were much more machine centric than those of today. That is, the way you programmed was, although more efficient, not too different than how you would program in machine language. Lisp with dynamic data, automatic garbage collection, and the ability for a lisp program to easily create and run more lisp code was very much an exception.

However over time, modern languages, like Python, came to support the kind of features found in Lisp. Today, the above remark might be changed to "Prolog and all the others".

The motivation for this project has been to deepen my own understanding of Prolog and hopefully share some of what I learn with others. The program is too slow to be used for any real world programming. But the ability to play with the source, add new features, maybe sprinkle print statements or use the trace feature when something is obscure is invaluable. Lots of real prolog interpretors can be found on the internet. Or if someone is interested, porting this code to C++ would increase the speed considerably.

Project Development

I developed this code from scratch several years ago through trial and error (lots and lots of error) and did not look at any other implementation. Frankly, I wanted to solve the puzzle instead of looking up the answer. But after thrashing around for awhile I found I did need a hint. That hint led to the use of the goal stack processed from a loop (function "search"). Once I started doing that things started to fall into place.

The program is built in three versions, each expanding on its predecessor. From the start I assumed that a single complete program would be too much to make work at one go. And also too much to explain.

Version one contains the essential goal searching logic and storage of rules. Terms remain very simple.

Version two allows nested terms which make lists possible and also supports the square bracket syntax for lists.

Version three provides some arithmetic and comparison operations as well as the "cut" and "fail" operators.

Additional Resources

Lots of information and real Prolog interpreters (and even compilers) are available on the internet. A google search for "prolog" will get you started.

I also found the Clocksin's book "Clause and Effect" a delightful read. It is short, only about 150 pages, but contains a good introduction to Prolog and some surprising case studies. It is not aimed at the beginning programmer, but if you find these pages digestable then his book may be right for you as well. The book was a strong motivator for me to start this website and also tweaked my interest in Logic circuits and the small compiler. Clocksin presents Prolog versions of similar programs.

Index

Prolog in Python. Version 1

Nature of a prolog program

With "normal" programming languages a program consists of statements telling the computer how to solve a problem step by step. A Prolog program is different. Facts and rules define the boundarys for a solution (or set of solutions) to queries issued by the user and Prolog searches for solutions automatically.

This will become clearer with some examples. Suppose we have some facts about some relationships between people and some rules that let us infer more relationships. Facts might be statements like "Bill is a boy" and "Bills mother is Alice". A rule might let us infer from these two statements that "Bill is Alice's son" or that "Alice has children".

In Prolog facts are represented in a syntax that will remind you of a function calls. To indicate that Bill is a boy we would create the following "term".

```
boy(bill)
```

It would also be possible to use the term "bill(boy)" but that wouldn't be as useful because of the kind of questions we will want to ask. It is also important to remember that "boy(bill)" is NOT a function call but simply a statement about a relation. Also note that "bill" is not capitalized. This is because Prolog generally uses capitalized words as variables and lowercase words as constants. Bill is a constant just because Bill will always be Bill.

To represent the fact that Bill's mother is Alice we could use the term

```
mother(alice,bill)
```

indicating the "mother" relationship between alice and bill, that is "alice is the mother of bill". It would be fine to write the term as "mother(bill,alice)" meaning "Bill's mother is Alice" as long as we are consistant. Prolog does not deduce any deep meaning from our facts and rules. It just manipulates them to answer our questions.

Let's stick with "boy(bill)" and "mother(alice,bill)". Now we can create a rule that means "If X is the mother of Y, then Y is the child of X". Here's how such a rule might be written.

```
child(X,Y) :- mother(Y,X)
```

where the "deep meaning" of child(X,Y) is "X is a child of Y". Notice that X and Y are captialized indicating they are variables. Rules will generally have variables. Also notice that the order is reversed from our "if" statement above. The part to the left of the ":-" is called the head and the part to the right is the body whcih consists of one or more "goals" to be satisfied. Here is another rule.

```
child(X,Y) :- father(Y,X)
```

and finally

```
son(X,Y) :- child(X,Y),boy(X)
```

which says that X is the son of Y if X is the child of Y and X is a boy. Notice that the body has two terms separated by a comma. Both terms in the body must be satisfied to make the head term true.

A Prolog program consists of an ordered set of facts and rules in what is termed the "database". Let's play with the program a bit before looking at the Python code.

```
? boy(bill)
? boy(frank)
? mother(alice,bill)
```

Here we have entered three facts into the database. Even now we can query them. A query is simply a term followed by the '?' mark.

```
? boy(bob)?
? boy(bill)?
Yes
? boy(X)?
{'X': 'frank'}
{'X': 'bill'}
? mother(X,bill)?
{'X': 'alice'}
? mother(alice,X)?
{'X': 'bill'}
? mother(X,Y)?
{'X': 'alice', 'Y': 'bill'}
```

Asking if bob is a boy gave no response (meaning No). Asking if bill is a boy gives the answer "Yes" since the rule "boy(bill)" is in the database. The query "boy(X)?" means we want Prolog to find all values of X for which boy(X) is a fact in the database. The query matches both "boy(bill)" and "boy(frank)" setting the variable X and reporting it for each. Now look at all the questions we can ask about the single fact that alice is the mother of bill. We can ask "Is anyone (X) bill's mother?", "Is alice the mother of anyone?" and even "Is anyone the mother of anyone else?"

When variables are set they are displayed in a format that should be familiar. You may have already guessed that our little Prolog interpreter stores variable settings in a Python dictionary,

Next let's add the two rules above and extend our querys.

```
? child(J,K) :- mother(K,J)
? son(X,Y) :- child(X,Y),boy(X)
? son(X,alice)?
{'X': 'bill'}
```

You might be wondering why I used variables "J" and "K" instead of "X" and "Y". Either is fine. Like local variables in Python functions, the variables are local to the rule. But a little later I want to walk through a query in detail without the confusion of separate variables with the same name.

Now let's add some information about dads.

```
? child(G,H) :- father(H,G)
? father(alex,bill)
? son(bill,X)?
{'X': 'alex'}
{'X': 'alice'}
```

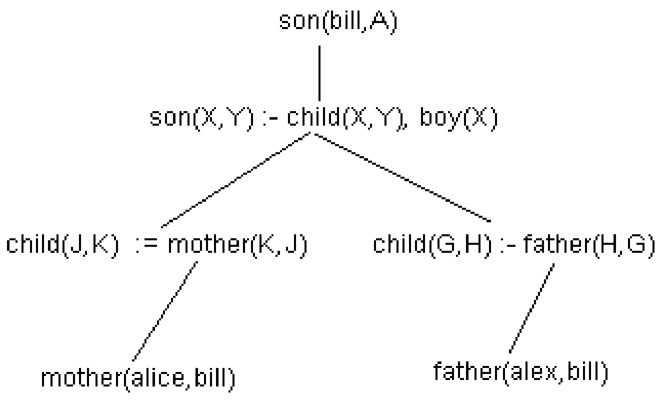
How it works

Let's look more closely at how queries are processed in order to be ready to look at the Python code. The first thing to do is to nail down some terminology.

Terms like "boy(bill)" consist of a predicate (boy) and one or more arguments (bill). Each argument can be (for now!) a constant (lower case) or a variable (capitalized). A rule consists of a head term and a set of goal terms, each of which must be satisfied in order for the rule to succeed.

A query is a term matched against facts and rules in the database. With the database as we have loaded it, the query "boy(X)?" matches both "boy(bill)" and "boy(frank)" since X is a variable which can be set (unified) to either "bill" or "frank". In general two terms will match if their predicates match and each of their arguments match. Arguments fail to match if they are different constants or variables set to different constants. Unset variables are set by this matching process. This unification process works in both directions; as queries are applied to rules and as results are returned. With the simple query "boy(bill)?" a match was found without needing to set any variables. As success is achieved, Prolog answers our query by showing how variables were set, or if no variables were set, simply with the word "yes".

Let's look a query in more detail. The following is a tree diagram showing how one query will spawn subqueries with constraints imposed by unification.


son(bill,A)
son(X,Y) :- child(X,Y), boy(X)
child(J,K) := mother(K,J)
child(G,H) :- father(H,G)
mother(alice,bill)
father(alex,bill)

The results of the query "son(bill,A)?" will be "A=alice" and "A=alex". But let's take it step by step. We have a search tree and as we both descend and return (ascend) the tree, variables will be set through unification. The goal "son(bill,A)" spawns a subgoal "son(bill,Y) :- child(bill,Y), boy(bill)" where we have just set the variable X to bill. Next "child(bill,Y)" will spawn "mother(K,bill)" and "father(H,bill)", setting variables J and G. "mother(K,bill)" finds a solution, setting k to "alice". Now we ascend the tree, setting Y to "alice" and resuming the goal "son(bill,Y) :- child(bill,alice), boy(bill)" to attack the second part "boy(bill". This succeeds (no variables set) and we hit the top of the tree setting A to "alice". Then we descend again to find "alex" as a second answer to the query.

A look the code

Having walked through a non trivial example, we're now in a position to see how the two pages of code come together. Click here to access the Python and perhaps get a printout. Since I'll refer to line numbers in the code it might be most convenient to have it in a text editor where navigation by line number is easy.

In the Python code terms and rules are represented by classes which really just hold their attributes. They each support just two methods, an initializer to compile a string representation of the term or rule the __repr__ method to display it.

The Term and Rule Classes

Let's play with the Term class.

```
>>> import prolog1 as prolog
>>> a = prolog.Term("mother(alice,bill)")
>>> a
mother(alice,bill)
>>> a.pred
'mother'
>>> a.args
['alice', 'bill']
```

Here the __init__ method (line 16) has compiled the string "mother(alice,bill)" into a Term instance with a predicate and arguments. These are represented by Python strings. The __init__ method uses the 'split' function in the string module to split fields on '(' or ',' characters. The input string has had all whitespace removed. (line 64)

Let's try a compilation of a rule

```
>>> b = prolog.Rule("son(X,Y):-mother(Y,X),boy(X)")
>>> b
son(X,Y) :- mother(Y,X),boy(X)
>>> b.head
son(X,Y)
>>> b.goals
[mother(Y,X), boy(X)]
>>>
```

A rule is made up of a head term and a list of goal terms. In the __init__ function (line 27) 'split' is again used to cut up the pieces. However re.sub first changes "son(X,Y):-mother(Y,X),boy(X)" to "son(X,Y):-mother(Y,X);boy(X)" so that terms in the body can be easily split around the ';' character.

Let's look at the 'unify' function now (line 92). 'Unify' takes four arguments; a source term and environment, and a destination term and environment. If unification is successful the destination environment may be updated. Both environments are just dictionaries with variable bindings. Unify will return 1 if successful, otherwise 0.

The code should be straight forward, although it will not stay so simple in the later versions of our program as the definition of a term becomes more general. Notice the use of the dictionary get method to try to resolve a variable in its own enviroment. If the variable is not set, "get" will return None, which can indicate permission to set the variable.

```
>>> from prolog1 import *
>>> unify(Term("boy(bill)"),{},Term("boy(frank)"),{})
0
>>> e = {}
>>> unify(Term("boy(bill)"),{},Term("boy(bill)"),e)
1
>>> e
{}
```

Here we can see that different terms do not unify but two copies of the same term do. Since there were no variables nothing was set in the destination environment 'e'.

Let's try something a little more complex.

```
>>> e = {}
>>> unify(Term("boy(bill)"),{},Term("boy(X)"),e)
1
>>> e
{'X': 'bill'}
```

This time the variable X is set in the destination environment. If we now try..

```
>>> e = {'X': 'bill'}
>>> unify(Term("boy(frank)"),{},Term("boy(X)"),e)
0
>>> e
{'X': 'bill'}
```

It fails because X was already set to 'bill'. Source arguments may also be variables. Let's look at a couple of situations.

```
>>>
>>> e = {}
>>> unify(Term("boy(G)"),{},Term("boy(X)"),e)
1
>>> e
{}
```

Here unification succeeded but the variable X was not set since G has not yet been set. However if we do set G then X will be set.

```
>>> e = {}
>>> unify(Term("boy(G)"),{'G':'frank'},Term("boy(X)"),e)
1
>>> e
{'X': 'frank'}
```

The Goal class

A Goal object is a bit of a misnomer but I can't think of any better name. A goal is a rule, some of whose subgoals may have been matched and variables set. In the tree diagram above, each branch and node represents a goal. Except for our original query, each goal has a parent goal. The attributes of a Goal object are the rule, parent, environment and an index which indicates which subgoal of the rule needs to be matched next.

When a Goal object is created (line 43) it is vital that its environment is completely independent of another goal. We don't want to be sharing dictionaries here! That's why copy.deepcopy() (line 50) is used to create a completely independent environments.

Searching for goals

Let's look at the search function next. Search will work with a stack of goals. Our query (a term) is converted into a rule that then is used to create a Goal object that is in turn used to initialize the stack. (line 117)

Until the stack is empty goals are popped off the top and matched to facts and rules in the database (line 136). When all goals in a rule succeed the parent goal is resumed, actually a copy is stacked anew where it was left off. Where a goal requires a rule in the database to be satisfied, a child goal is started. In both cases unification is applied to make (or update) the environment and the new goal is stacked for processing.

Whenever the original goal (no parent) is found, results are printed (line 123). If variables have been set, they are output. Otherwise the program simply replies "Yes".

Now, you may be wondering (and probably should be!) why we need a stack for processing. Why not instead simply make "search" a recursive function? Well, it won't work. In fact, I tried that at first. But the nature of prolog is that a "call" to "search" would need to "return" multiple times (or none), once for each solution found. Our Python functions just can't do that.

The main program

This program is set up to process statements (and queries) from any files on the command line and then to interact in the same fashion with the user. Interaction with the user may be turned off by using '.' (meaning STOP) as the last filename. Within a file lines starting with '#' are comments. The command "trace=1" will turn on the trace feature. When trace is on a line is printed by "search" each time a goal is stacked or popped from the stack. The command "dump?" will display the database. Otherwise if the line ends with a '?' it is assumed to be query; if not, it must be a rule or fact to be added to the database. Rules and facts may optionally end with a period.

An example with Trace

Here is a non-trivial example that shows the trace feature.

```
? child(X) :- boy(X).
? child(X) :- girl(X).
? girl(alice).
? boy(alex).
? trace=1.
? child(Q)?
search child(Q)
stack Goal 1 rule=got(goal) :- child(Q) inx=0 env={}
  pop Goal 1 rule=got(goal) :- child(Q) inx=0 env={}
stack Goal 2 rule=child(X) :- boy(X) inx=0 env={}
stack Goal 3 rule=child(X) :- girl(X) inx=0 env={}
  pop Goal 3 rule=child(X) :- girl(X) inx=0 env={}
stack Goal 4 rule=girl(alice) inx=0 env={}
  pop Goal 4 rule=girl(alice) inx=0 env={}
```

```
stack Goal 3 rule=child(X) :- girl(X) inx=1 env={'X': 'alice'}
  pop Goal 3 rule=child(X) :- girl(X) inx=1 env={'X': 'alice'}
stack Goal 1 rule=got(goal) :- child(Q) inx=1 env={'Q': 'alice'}
  pop Goal 1 rule=got(goal) :- child(Q) inx=1 env={'Q': 'alice'}
{'Q': 'alice'}
  pop Goal 2 rule=child(X) :- boy(X) inx=0 env={}
stack Goal 5 rule=boy(alex) inx=0 env={}
  pop Goal 5 rule=boy(alex) inx=0 env={}
stack Goal 2 rule=child(X) :- boy(X) inx=1 env={'X': 'alex'}
  pop Goal 2 rule=child(X) :- boy(X) inx=1 env={'X': 'alex'}
stack Goal 1 rule=got(goal) :- child(Q) inx=1 env={'Q': 'alex'}
  pop Goal 1 rule=got(goal) :- child(Q) inx=1 env={'Q': 'alex'}
{'Q': 'alex'}
```

Index

Prolog in Python. Version 2

This second version of our Prolog interpretor implements only a single new feature. But that feature will enable the use of trees and lists and is necessary for more interesting programs.

The feature is simply this. In addition to constants and variables, arguments in a term may be other terms. This complicates parsing and unification, because both will now need to be recursive, but leaves other parts of the program pretty much intact.

In this chapter, we'll look at compound terms (terms that contain other terms) only in regard to building Prolog lists.

Building lists with compound terms

Prolog does not support lists directly but builds them as a set of nested terms The predicate "." is used to bind the elements of the list together. The term ".(a,b)" may be represented as a mini tree

```
.__ b
|
a
```

and .(a, .(b, .(c, .()))) becomes

```
.__.__.__nil
|  |  |
a  b  c
```

The first tree may also be represented in Prolog with the syntax "[a|b]" and the second, which is a proper list may be represented by the familiar "[a,b,c]". The term ".()" is the empty list represented by "[]". With the square bracket syntax, Prolog lists look just like Python lists. Prolog lists may contain constants, variables, and of course other lists.

Compound terms, and therefore lists, unify in the same way that we are already familiar. That is, the predicates must match, they must have the same number of arguments, and each argument must unify with the corresponding one in the other term. But since arguments themselves may be terms the process must be done recursively. Here are some examples. assume all variables in the second term are unbound.

```
Unification ...
[a,b,c]  with X        binds X to the [a,b,c]
[a,b,c]  with [X,b,c]  binds X to a
[a,b,c]  with [X|Y]    binds X to a, Y to [b,c]
```

It is important to remember that internally in the program lists are nested terms with the "." predicate, but that they are translated to and from the square bracket notation for input and display. The pipe character "|" effectively splits a list into its first element and the remaining elements.

A better parser for terms

Click here to access the Python code for prolog2.py. You may find it convenient to save it into a file and access it with a text editor.

In the first version of our Prolog interpreter we used the split function in the string module to break a term like "mother(mary,bill)" first into a predicate "mother" and arguments "mary,bill" (by splitting on the "(" character), and later splitting the arguments apart (on the commas). Terms in a rule are also separated by commas but we got around that by using re.sub to turn just those commas into semicolons, and then splitting the terms apart.

Alas, this approach just won't work anymore. If we have nested list like "[a,[b,c,d],e]", we want to extract three elements, "a", "[b,c,d]", and "e". Not "a","[b","c","d]","e".

To accomodate this prolog2.py contains its own split function that splits only when a separater is at a zero nesting level. As a string is scanned, the nesting level is increased by one whenever a "(", or "[" is encountered, and decreased by one whenever a ")" or "]" is encountered. In addition an optional "All" parameter, when 0, lets us split off just the first predicate in the term.

As in Prolog1.py terms are objects and compiled from text. There is also a mode where a predicate and argument list may be compiled into a term. (line 35). This is used by the eval function. With text however, either a list (line 39) or a standard term may be compiled. But now that arguments are also terms, Term initialization is recursive. The Python map function builds all the argument terms with a single call. (lines 42 and 52). Finally a simple constant or variable is a term with no arguments, like it was in prolog1.py (line 55).

The Term __repr__ method will determine how terms are displayed when printed or passed to the builtin str function. Since lists are really nested terms using the "." predicate, they are now translated back to the square bracket notation. (line 59). Other terms are output the same way they were in prolog1.py.

Let's play with the Term class a bit

```
>>> from prolog2 import *
>>> a = Term("bill")
>>> print a
bill
>>> a.pred
'bill'
>>> a.args
[]
>>>
```

Here we have compiled the simplest possible term, a constant. It prints its name and in order to see the insides (pred and args) we need to look at the attributes directly.

```
>>> b = Term("tasty(food(meat))")
>>> b
tasty(food(meat))
>>> b.args
[food(meat)]
>>>
```

This is a compound term showing the argument of "tasty" as a nested term.

Next we'll construct a list with the "." operator directly.

```
>>> c = Term(".(a,.(b,.()))")
>>> c
[a,b]
```

But of course, it is more natural to create lists from the list syntax itself.

```
>>> d = Term("[x,y,[a,b,c],z]")
>>> d
[x,y,[a,b,c],z]
```

We can also use the "|" operator to prepend 'x' onto the list "[y,z]".

```
>>> e = Term("[x|[y,z]]")
>>> e
[x,y,z]
>>>
```

An improved unify function.

The Rule and Goal classes are the same as in prolog1.py but unification is more complex since looking up variables needs to be done recursively. The function `unify` returns one if the source term can be unified to the destination term. Variables in each are looked up in their own enviroments and destination variables may be bound to constants in the source.

The new function `eval` is used to "look up" variables. (line 206). `eval` is recursive so that if a term is evaluated, each argument is evaluated internally. `eval` then returns a term with all variable references converted to constants or the value `None` if this is impossible.

The function `unify` returns true (1) if the source (or parts of it) are still variables. Otherwise it returns true if destination can be matched to it piece by piece in basically the same manner as in prolog1.py.

Notice that all returns from the function `unify` are channeled through the function `sts`. This is for tracing purposes and results in nested unifications being indented in a trace. Let's play with this.

First we'll set up an empty destination enviroment "e" and unify a constant list with a simliar list with variables.

```
>>> import prolog2
>>> e = {}
>>> prolog2.unify(Term("[a,b,c]"),{},Term("[A,B,C]"),e)
1
>>> e
{'B': b, 'C': c, 'A': a}
>>>
```

Now we'll unify again (just matching values) since e is already set. But this time we'll turn on the trace in the module prolog2.

```
>>> prolog2.trace = 1
>>> prolog2.unify(Term("[a,b,c]"),{},Term("[A,B,C]"),e)
>>> prolog2.unify(Term("[a,b,c]"),{},Term("[A,B,C]"),e)
```

```
  Unify [a,b,c] {} to [A,B,C] {'A': a, 'C': c, 'B': b}
    Unify a {} to A {'A': a, 'C': c, 'B': b}
      Unify a {} to a {'A': a, 'C': c, 'B': b}
      Yes All args unify
    Yes Unify to Dest value
    Unify [b,c] {} to [B,C] {'A': a, 'C': c, 'B': b}
      Unify b {} to B {'A': a, 'C': c, 'B': b}
        Unify b {} to b {'A': a, 'C': c, 'B': b}
        Yes All args unify
      Yes Unify to Dest value
      Unify [c] {} to [C] {'A': a, 'C': c, 'B': b}
        Unify c {} to C {'A': a, 'C': c, 'B': b}
          Unify c {} to c {'A': a, 'C': c, 'B': b}
          Yes All args unify
        Yes Unify to Dest value
        Unify [] {} to [] {'A': a, 'C': c, 'B': b}
        Yes All args unify
      Yes All args unify
    Yes All args unify
  Yes All args unify
1
>>>
```

Two examples with lists

Finally we'll examine two classic operations with lists, element membership and the appending of lists.

```
? member(X,[X|T])
? member(X,[H|T]) :- member(X,T)
```

Two rules determine membership. If X is the head of a list or if X is a member of the tail of the list. In the first case it doesn't matter what the tail "T" of the list is. In the second case it doesn't matter what the head "H" of the list is.

With these two definitions we can test for membership

```
? member(a,[a,b,c])?
Yes
? member(b,[a,b,c])?
Yes
```

And finally Prolog can compute "backwards" so to speak to find all the members of the list. In this mode we have something similar to the Python "for" statment.

```
? member(X,[a,b,c])?
{'X': a}
{'X': b}
{'X': c}
?
```

The rules for append are a bit trickier. It's always wise to remember that the rules constrain the solution but don't specify the computation. Here are the two rules for append. The first two arguments are the input lists and the third argument is the result.

```
? append([],L,L)
? append([X|A],B,[X|C]) :- append(A,B,C)
```

The first rule says that any list "L" appended to the empty list results in the same list "L".

The second rules says that if C is A appended to B, then it's also true if X is prepended to both A and C.

Let's run some tests computing in both directions.

```
? append([a,b],[c,d],X)?
{'X': [a,b,c,d]}

? append([a,b],Y,[a,b,c,d])?
{'Y': [c,d]}

? append(X,Y,[a,b,c])?
{'X': [], 'Y': [a,b,c]}
{'X': [a], 'Y': [b,c]}
{'X': [a,b], 'Y': [c]}
{'X': [a,b,c], 'Y': []}
?
```

In the last example, Prolog gives us all possible ways the list "[a,b,c]" can be constructed. I think this is kind of wild.

Index

Prolog in Python. Version 3

Arithmetic operations

The final version of our Prolog interpreter implements a few of the arithmetic operators as well as the "cut" and "fail" terms. Up to now we've been unable to do any numerical computation. Actually, this is not Prolog's strong point anyway. But some numerical computation is always necessary. For example here is a pair of rules to find the length of a list.

```
length([],0)
length([H|T],N) :- length(T,Nt), N is Nt+1
```

This says that the length of an empty list is zero and the length of any other list is one greater than the length of its tail (the list less its first term).

The interesting term is "N is Nt+1". For one thing it's in infix. There is an equivalent form "is(N,+(Nt,1))" which looks more Prolog-like (and Lisp-like) but is harder to read. It turns out that it requires only a few extra lines of Python to implement the infix form, although the operator "is" is renamed (internally only) to "*is*" since "is" should remain a valid name.

It is necessary to discuss these operations in some detail. Terms in Prolog are used as goals within rules and either succeed or fail. During unification variables are sometimes set as goals succeed. With "N is Nt+1", or better, "is(N,+(Nt,1))", first the inner term must succeed before the outer term is tried.

The '+' operator succeeds only if both arguments evaluate to numbers. The term then evaluates to the sum of the numbers. Operators like '-', '*'. '/' work in exactly the same way. The boolean operators "<", "==", etc. also expect numeric arguments but then simply succeed or fail. They are only used as the top term in a goal.

The 'is' operator is a combination of both the Python "=" and the '==' operators. A variable on the left that is unset is set to the computation on the right and the term succeeds. If the left side is already set then the term succeeds only if the two sides are equal. Although it's not obvious at this point, this lets us do both of the following.

```
? length([a,b,c],X)?
{'X': 3}
? length([a,b,c],3)?
Yes
?
```

Code changes for aritmetic

Click here to access the Python code for prolog3.py. You may find it convenient to save it into a file and access it with a text editor.

In prolog1.py we used the split function in the string module to split terms in a rule and arguments in a term. In prolog2.py we had to write our own "split" function in order to correctly handle nested terms. We still were only separating on commas or the left parenthesis (to pull a predicate from its arguments).

Some of our infix operators are now more than a single character, such as "<=" or "*is*". A small adaptation using the variable "lsep" (line 18) which stands for "length of separator" handles this.

Some infix operators are only allowed at the top level of a term. A new function "splitInfix" (line 35) and called from "Term" init (line 42) looks for infix operators (the list "infixOps" is far from complete) and essentially makes the string "a<=b" equivalent to "<=(a,b)".

Finally we come to the execution of our new operators. Up to now the search function took a term from a rule and then searched for matches in the database of other rules. These new operators do not initiate a search. Instead they are simply evaluated (with possible side effects) and if they succeed the rule is continued with the next term. The code for this (lines 201 to 216) check for *is*", "cut", "fail", and generic functions like "<" all of which are found only at the top level of a term.

Other new operators like "+" exist in nested terms and are processed by the eval function (line 244). Each of these operators is handled by its own function which builds a new term from its arguments.

Cut and Fail

Consider the following piece of Prolog.

```
childOf(X,Y) :- parent(Y,X)
parent(chris,jon)
parent(maryann,jon)
childOf(A,B)?
{'B': chris, 'A': jon}
{'B': maryann, 'A': jon}
?
```

Jon is the child of both parents so Prolog returns two answers. But if we want to only find a first answer we can do the following instead.

```
childOf(X,Y) :- parent(Y,X),cut
parent(chris,jon)
parent(maryann,jon)
childOf(A,B)?
{'B': chris, 'A': jon}
?
```

Cut stops alternatives in the search and then succeeds. In prolog3.py this is accomplished by simply truncating the queue of alternatives (line 220).

Fail is almost the exact opposite. It stops the current rule, leaving any alternatives alone. "Cut" and "fail" are sometimes used together to declare complete failure of the search.

Searching with a Queue, instead of a Stack

You may or may not have noticed in prolog2.search function that the stack of goals became a queue. Other than the change of the variable name, the only difference is the "queue.insert(0,c)" instead of "stack.append(c)" (line 224 in prolog3.py).

The effect of this change is subtle, but interesting. It changes the tree search from depth-first to breadth-first. That, in turn, means that multiple goals are processed in parallel rather than one goal being completed before another is started. This was also discussed in "Queues, Trees and Water Buckets". It opens the door to parallel processing but also creates problems, especially with the "cut" operator. Not only does the queue need to be emptied but processes running in parallel need to stop so as not to add any new goals to the queue afterwards. It's basically a synchronization problem. I have read, however, that most modern prologs do use a breadth-first search.

Where from here?

This is as far as I intend taking the Prolog project, but it can certainly be extended further. I'm quite surprised so much could be done in about 260 lines of Python, including whitespace.

If you do extend the program I would enjoy hearing from you.

Index

The Basics of Data Compression

In earlier days, computers were small and conserving space in memory or on a disk drive was always a premium. The computer I am using now has over 1 million times the memory of my first PC that I bought in 1983, and the new 80 gigabyte disk holds 4000 times the amount of data. The first computer also cost a lot more.

Ironically, space is still at a premium primarily because as the machines have gotten bigger and faster, so have the problems we want to solve.

In this study we'll look at an early algorithm developed by David Huffman in 1952 when he was a graduate student at MIT. The algorithm squeezes the "fluff" out of data but in a way that the original can be reproduced exactly. You see this done in programs like Winzip, stuffit, or for Unix folk, gzip. This is called "lossless" compression and is different from "lossy" compression used in audio and video files where the exact original cannot be reproduced.

For demonstration purposes we'll stick with character data and mostly just use a small subset of the alphabet. As usual, we'll use small amounts of Python code to make it all work.

Computers store text in several ways. The most common encoding is Ascii where each character is stored in an 8 bit byte. So, for example, the character 'e' is the bit sequence '001100101'. As any byte can take one of 256 distinct values (0 to 255) there is plenty of room for the Latin alphabet, the numbers zero to nine and lots of punctuation characters such as periods, commas, newline indicators and so on. Actually, if we stick to English text, we can get by the first 127 values. Other European languages require some more letters and use the upper range 128-255 as well. Still other languages, like Russian or Chinese, require more elaborate coding systems such as Unicode where multiple bytes are used per character.

The secret of compressing data lies in the fact that not all characters are equally common. For instance, in typical English text the letter 'e' is much more common than the letter 'z'. In fact there are roughly 70 'e's for every 'z'. If we could encode the letter 'e' with less than the usual 8 bits and in exchange let the letter 'z' be take more, we'd be money ahead.

This sort of thing is pretty common. Human languages have mostly evolved so that the most common words such as "I", "am", "Yo", "soy", "ich" and "bin" are nearly as short as possible. Uncommon words like "ominous", "peligroso", or "Einkommensteurerklärung" come rarely enough to not make too much difference.

Huffman Data Compression.

We will look at several functions that bring together an example of Huffman data compression for text files. These functions do the following.

1. Examine text to be compressed to determine the relative frequencies of individual letters.

2. Assign a binary code to each letter using shorter codes for the more frequent letters. This is the heart of the Huffman algorithm.

3. Encode normal text into its compressed form. We'll see this just as a string of '0's and '1's. This will turn out to be quite easy.

4. Recover the original text from the compressed. This will demonstrate a nice use of recursive traversal of a binary tree, but will still remain fairly simple.

To view the complete computer program's source code Click Here.

Determining Relative Frequencies.

Our function "frequency" takes a text string as input and returns a dictionary of the letters encountered with a count of how often they appear.

```
def frequency (str) :
    freqs = {}
    for ch in str :
        freqs[ch] = freqs.get(ch,0) + 1
    return freqs
```

Here is an example run. To keep things simple we are using text consisting of only 7 characters (a-g), but we will vary their frequency.

```
% python >>>
import huffman
>>> freqs = huffman.frequency("aaabccdeeeeeffg")
>>> print freqs
{'a': 3, 'c': 2, 'b': 1, 'e': 5, 'd': 1, 'g': 1, 'f': 2}
>>>
```

The dictionary returned lets us look up the frequency for each letter. Notice the use of the .get function. If the letter is not already in the dictionary, the default value zero is returned, otherwise its existing count. In either case the count is incremented and the key/value pair is updated or newly added to the dictionary.

Assigning codes to the Characters

Our next function builds and sorts a list of tuples. This list will be the input for the main Huffman algorithm.

```
def sortFreq (freqs) :
    letters = freqs.keys()
    tuples = []
    for let in letters :
        tuples.append((freqs[let],let))
    tuples.sort()
    return tuples
```

The dictionary "freqs" that we built above is the input to the function "sortFreq"

```
>>> tuples = huffman.sortFreq(freqs)
>>> print tuples
[(1, 'b'), (1, 'd'), (1, 'g'), (2, 'c'), (2, 'f'), (3, 'a'), (5, 'e')] >>>
```

The sort method for a list compares individual elements. In this case the elements are tuples of two values, a number and a character. The sort method will first compare the numbers and only if they are equal will it compare the character values. Notice that sorting the list has the effect of moving the least common letters to the front of the list. This will be very convenient.

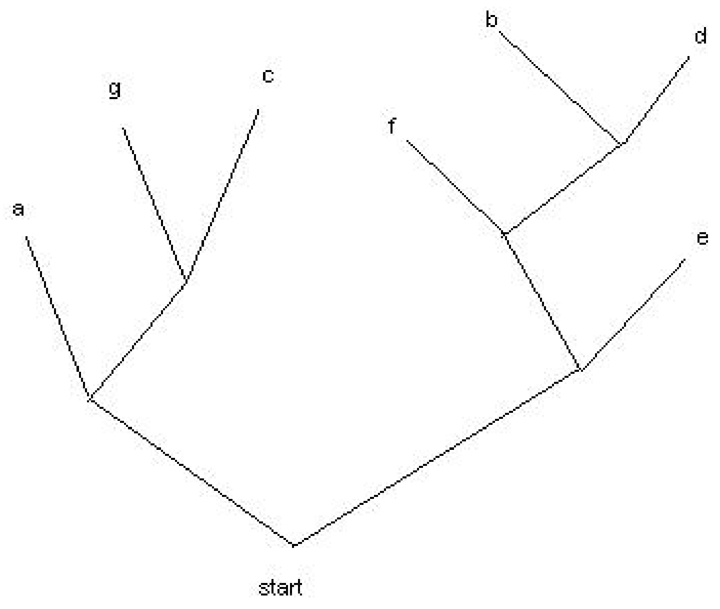
Let's jump ahead just a bit. Here are the codes that will be assigned to each of the seven characters.

```
    a   '00'
    b   '1010'
    c   '011'
    d   '1011'
    e   '11'
    f   '100'
    g   '010'
```

Each character is assigned to a string of 0's and 1's. We're cheating a bit. It's easier to use strings in our program than the actual bits we would use in real life. But see the exercises at the end.

Notice that the two most common characters, "a" and "e" are represented by just two bits, whereas the two least common characters "b" and "d" each use four bits.

Perhaps not so easy to see is that there is no ambiguity in this encoding. Any string of zero's and one's will map to a unique string of the characters 'a' to 'g'. This is much more intuitive if we draw a tree for the code table.



If we start at the bottom and traverse up the tree through the branches we can arrive at any character in a series of left and right turns. If we add a "zero" for each left turn and a "one" for each right turn, the coding for the character at the end is produced. For example, a left-right-left takes us to the letter "g" which has "010" for its code.

Since a unique path leads to each letter we don't need anything to delimit one character from another. We will be able to just run their bits together.

So now let's back up and see how this tree is built from the frequency distribution. It is both clever and actually not too hard.

Remember we had the following list of tuples. The tuples are in order of the frequency of the letter each represents. We will turn this list into a tree.

```
>>> print tuples
[(1, 'b'), (1, 'd'), (1, 'g'), (2, 'c'), (2, 'f'), (3, 'a'), (5, 'e')]
```

Unlike real trees, our tree will grow from the top to the bottom, from the leafs back to the root. (Whimsical fantasy should be in every programmers toolbox). We'll pull two elements from the front of the list and combine them into a new element. Remember these 2 elements will have the least frequencies. The new element will be the branch point to the 2 elements. The frequency of the new branch point is simply the sum of the frequencies of its two parts. Think about this for a just a moment and it should make sense. The first time we do this the new element will look like

```
(2, ((1, 'b'), (1, 'd')))
```

which is the branch point in the upper right of the tree leading to the characters 'b' and 'd'. Since both 'b' and 'd' have a frequency of 1, their branch point has a frequency of 2. Next we add this new element onto the list and resort it putting the new element (branch point) into its proper position in the list. The process is repeated. Each time two elements are replaced by one until the list has a single element which represents the complete tree.

Here is the Python function that does this. It's even less wordy than the above explanation.

```
def buildTree(tuples) :
    while len(tuples) > 1 :
        leastTwo = tuple(tuples[0:2])                  # get the 2 to combine
        theRest  = tuples[2:]                          # all the others
        combFreq = leastTwo[0][0] + leastTwo[1][0]     # the branch points freq
        tuples   = theRest + [(combFreq,leastTwo)]     # add branch point to the end
        tuples.sort()                                  # sort it into place
    return tuples[0]            # Return the single tree inside the list
```

Once we have the tree, we can trim of the frequencies leaving us with a nice nested representation.

```
def trimTree (tree) :
     # Trim the freq counters off, leaving just the letters
    p = tree[1]                                    # ignore freq count in [0]
    if type(p) == type("") : return p              # if just a leaf, return it
    else : return (trimTree(p[0]), trimTree(p[1])) # trim left then right and recombine
```

```
>>> tree = huffman.buildTree(tuples)
>>> trim = huffman.trimTree(tree)
>>> print trim
(('a', ('g', 'c')), (('f', ('b', 'd')), 'e'))
```

So "trim" has the tree above as coded into a set of nested tuples.

Codes may be assigned by recursively traversing the tree, keeping track of the left and right turns in the variable "pat". When we reach a leaf of the tree, i.e. a string rather than a nested tuple, we can assign the code calculated enroute. We have a global dictionary "codes" which will be filled in along the way.

```
def assignCodes (node, pat='') :
    global codes
    if type(node) == type("") :
        codes[node] = pat              # A leaf. set its code
    else  :                         #
        assignCodes(node[0], pat+"0")   # Branch point. Do the left branch
        assignCodes(node[1], pat+"1")   # then do the right branch.
```

Here we will run the function assignCodes on the trimmed tree to build our lookup dictionary for encoding.

```
>>> print trim
(('a', ('g', 'c')), (('f', ('b', 'd')), 'e'))
>>> huffman.assignCodes(trim)
>>> print  huffman.codes
{'a': '00', 'c': '011', 'b': '1010', 'e': '11', 'd': '1011', 'g': '010', 'f': '100'}
```

Encoding and Decoding a Text Stream.

Once we have our codes dictionary populated encoding a text string is simple. We have only to look up the bit string for each character and paste it on.

```
def encode (str) :
    global codes
    output = ""
    for ch in str : output += codes[ch]
    return output
```

This turns on our original string "aaabccdeeeeeffg" into 44 bits that we represent with "00000010010100101010111111111101101110111000". The original text at one byte per character requires 120 bits.

Decoding the bit stream back into text is a little harder but still not bad. Each "zero" in the bit stream means a left turn up the tree, each "one" a right turn. When a leaf is reached, its character is sent to the output and we restart at the base of the tree for the next character.

```
def decode (tree, str) :
    output = ""
    p = tree
    for bit in str :
        if bit == '0' : p = p[0]     # Head up the left branch
        else          : p = p[1]     # or up the right branch
        if type(p) == type("") :
            output += p              # found a character. Add to output
            p = tree                 # and restart for next character
    return output
```

A Real Example.

Feeding this document through our program resulted in the following set of codes. I have omitted most of the middle. You can see that the space character is by far the most common followed by 'e' and 't'.

```
[(1, '%'), (1, '/'), (1, 'Y'), (2, '7'), (2, '8') ... (608, 't'), (803, 'e'), (1292, ' ')]
```

As you can see the algorithm compressed 22135 bytes to 13568 and then successfully restored the original. The function "main" reads text from standard input and puts it through the paces. Your results may vary as the file is undergoing change.

```
% python ./huffman.py < huffman.html
Original text length 22135
Requires 108543 bits. (13568 bytes)
Restored matches original True
Code for space is  100
Code for letter e  1110
Code for letter y  101010
Code for letter z  1111101100
```

Ideas for Further Exploration

In order to compress a text file and then later restore it, the tree must be included along with the bit stream. Modify the code to produce compressed files along with their decoding tree.

Suppose we made a reverse dictionary from "codes" where the keys are the compressed bits and the values are the original Ascii characters. Could you use this instead of the tree to decode compressed text? Modify the function "decode" to work this way. Compare the two functions for simplicity and speed.

Create binary ones and zeros so that real compression actually takes place. You will need to investigate the bit operators in Python.

Can the program as it stands compress binary data as well? Test it.

Try using text from other types of sources like reports and even other languages.

German and English keyboards have the "Y" and "Z" keys switched. Why? Run some German text through the program to check relative frequencies of "Y" and "Z" in the two languages. If you need to find some German text just google "Einkommensteurerklärung".

Natural Language Processing

Introduction

This is the first part of a multi-step project to parse and process small subsets of natural language. Here i part 1 we will explore the use of RTN's for representing grammars and Python code to both generate and parse sentences in a grammar. In part 2 we will match parsed sentences against a "Knowledge" base to perform sematic processing, that is, extract meaning from the sentence. Eventually, I hope to replicate Terry Winograd's "Blocks World" which will require sentence parsing, extraction of meaning, goal planning, and execution of instructions.

Keeping the code as simple as possible

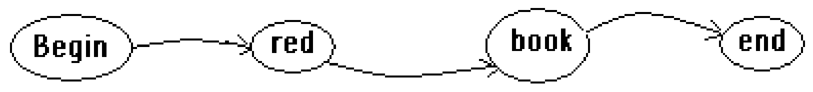
Recursive transition networks

Recursive transition networks (RTN) are useful devices for representing grammars in both human and computer languages. The kind of grammars that can be represented with RTN's are called context-free. They consist of rules for how words may be put together into structures and, in turn, how those structures may be built into larger structures. For a particular grammar a valid "sentence" is a list of words that follow the rules of the grammar. We will use the word "sentence" with this more restricted meaning a lot.

The computer language Pascal (among others) is formally defined using RTN's.

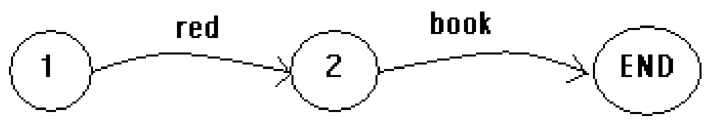
Here is a very simple example. The grammar in figure 1 consists of a single "sentence", the word "red" followed by the word "book".

Fig 1



For our purposes an alternative, but equivalent, format is more useful because it is easily translated into a Python dictionary. The network that defines this "sentence" consists of nodes (1, 2, END) connected by named arcs. The name of an arc indicates what word(s) will carry us from one node to another, and in which direction. You traverse the network by starting at node 1 and finding a path to node END.

Fig 2



And here is one way to represent the network in Python

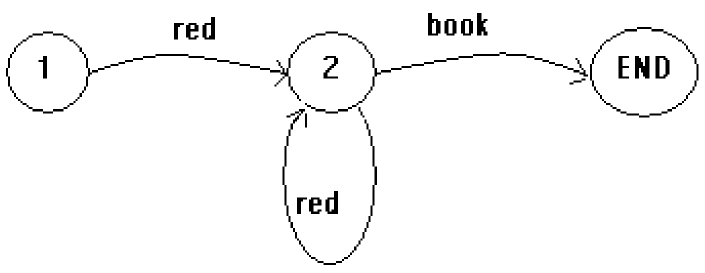
```
net1 = {
 'sentence': {
        1: (("red",  2),    ),
        2: (("book", END), )
        }
}
```

The network "sentence" refers to a dictionary whose entries are the states. Each state (1,2..) consists of a tuple of transitions where. The word "red" in state 1, for example, takes us to state 2. The "sentence" network is itself inside a dictionary of networks (net1). Now this is, momentarily, a bit more complex than it needs to be. But we're preparing the groundwork to have a family of networks working together.

So, the nodes in the network are key/value pairs in a dictionary. The key is the name of the node (1,2) and the value is a list of arcs. Each arc is a two valued tuple describing the word and destination. The END node does not need to be defined since no arcs emanate from it. END is simply a defined constant.

Let's move on to a somewhat more interesting grammar. Figure 3 describes a grammar that consists of the word `book` preceded by one or more words `red`. So `red book` and `red red red book` are both valid sentences in the grammar.

Fig 3



And here is the Python version.

```
net2 = {
 'sentence': {
         1: (("red", 2), ),
         2: (("red", 2), ("book", END))
         }
}
```

Generating sentences from a grammar

It's easy to write a python function that will generate a random sentence in a grammar. For our simple non-recursive grammars above the following function works well.

```
import random
END   = None

def gen1(net) :
    "Generate random sentence from simple net. Non recursive"
 # p points to state 1 initially. s is the sentence being build
    p = net[1]; s = ""
    while 1 :
        choose = random.randrange(len(p)) # A random transition in this state
        s = s + " " + p[choose][0]        # The word on the arc
        next = p[choose][1]               # To the next state
        if next == END : return s
        p = net[next]
```

The variable `p` keeps track of the current node and `s` is the sentence as it is being built. From each node we choose an arc at random (line 8), add its word to the sentence (line 9) and advance to the next node (line 12). The sentence is returned when the END node is reached (line 11).

Let's play with this a bit. You can access code and grammar at rtn.py and simple.py

```
>>> import rtn
>>> import simple
>>> rtn.gen1(simple.net1['sentence'])
' red book'
>>> rtn.gen1(simple.net1['sentence'])
' red book'
>>> rtn.gen1(simple.net2['sentence'])
' red red red book'
>>> rtn.gen1(simple.net2['sentence'])
' red red book'
```

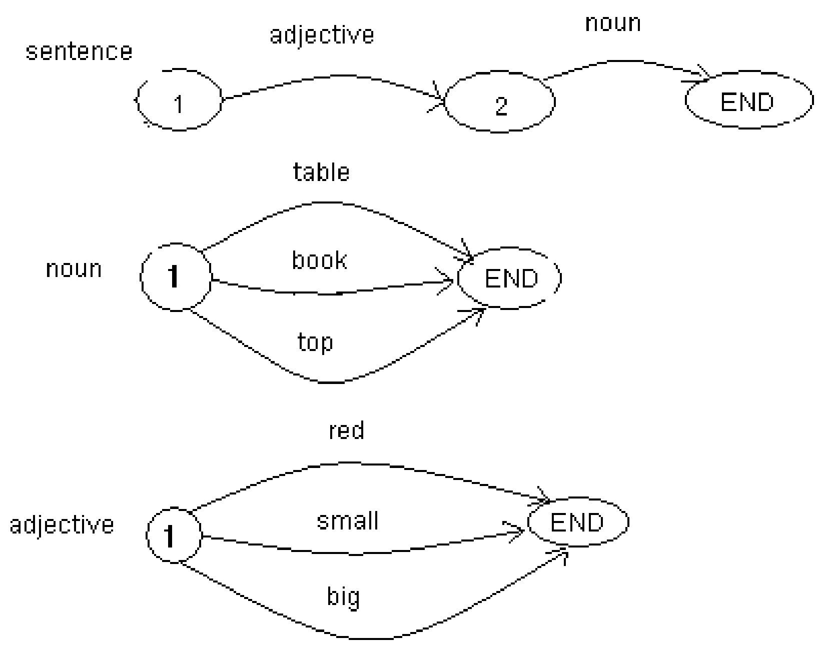
The network `net1` always produces the same sentence, but `net2` generates a random number of `red` words preceding `book`.

Nested Networks

The following assumes a rudimentary knowledge of english grammar. You need to at least know the difference between nouns, prepositions, verbs and adjectives. There are lots places to get this information.

Extending our simple networks to recursive networks requires just one change. Allow the label on an arc to reference either a word or another network. Here is an example of 3 simple networks `article`, `noun1` and `adjective` along with a recursive network `noun2`. These "2nd order nouns" are optionally preceded by the word `the` or `a` followed by zero or more adjectives.

Fig 4. Recursive Networks



The recursive network `net3` is defined in Python as you probably already suspect. The subnetworks are defined first.

```
net3 = {
 'sentence': {
         1: (('adjective', 2), ),
         2: (('noun'  , END),  )
         },

 'noun'  : {
         1: (("book",END),("table",END),("top",END),("cover",END))
         },

 'adjective': {
         1: (("big",END),("small",END),("red",END))
         }
}
```

A small modification in the generator function is needed to make it work recursively.

```
def gen2(network, name) :
    "Generate random sentence, allow recursive network"
    net = network.get(name)
    if not net : return name     # terminal node
    p = net[1]; s = ""
    while 1 :
        choose = random.randrange(len(p))
        s = s + " " + gen2(network, p[choose][0])
        next = p[choose][1]
        if next == END : return s
        p = net[next]
```

Now the `name` passed to the function may be either a word or a subnetwork. The fourth line checks if it refers to a network and, if not, simply returns it. Otherwise line 8 calls gen2 recursively to expand the subnetwork. Let's generate some nouns, adjectives and sentences in our tiny grammar.

```
>>> import rtn, simple
>>> rtn.gen2(simple.net3, "noun")
' table'
>>> rtn.gen2(simple.net3, "noun")
' top'
>>> rtn.gen2(simple.net3, "adjective")
' big'
>>> rtn.gen2(simple.net3, "adjective")
' small'
>>> rtn.gen2(simple.net3, "sentence")
'  big   table'
>>> rtn.gen2(simple.net3, "sentence")
'  red   book'
>>>
```

Now suppose that instead of generating a random sentence, we want to generate all possible sentences in a grammar. We can do this because our little grammar still finite.

But is there an easy way to generate all sentences in a grammar? This brings us to a new topic

Python generators

Generators are new to Python 2.2. You may think of them as functions that return multiple values one at a time to the caller, but maintaining their internal state (where they are and the value of local variables) between calls. They are very convenient when used with a "for" loop.

Here's a very simple example

```
>>> def gg() :
...     yield 2
...     yield 4
...     yield 8
...
>>> for i in gg() : print i
...
2
4
8
>>>
```

What turns a function definition into a generator is the use of the keyword `yield` which returns a value but keeps a bookmark of where it left off and retains values for local variables. The `for` loop will keep getting values from the generator until an explicit `return` in the generator is encountered or the generator falls off the end (an implicit return).

Generators are actually objects with a `next` method that is used by the `for` loop. A generator raises a specific exception when a `return` is excecuted, explicitly or implicitly.

Here is a generator that will generate all the sentences in the little recursive grammar above.

```
def gen3(network, label, state) :
    "Use generator to yield all possible sentences"
    if not network.get(label) : yield label  # if not a subnet stands for itself
    elif state == END         : yield ""     # no state. end of road
    else :
        net = network[label]
        p = net[state]; s = ""
        for labl,nextState in p :
            for word in gen3(network,labl,1) :
                for rest in gen3(network,label,nextState) :
                    yield word + " " + rest
```

This generator is doubly recursive and deserves a close study. When it is entered `label` may be subnetwork or a terminal word (like "red"). Basically, from a given state, each arc is tried. For each "word" (terminal or subnetwork) the arc label produces, we check from the target node to see if the END can be reached and yield the sentence accordingly. It is tricky.

Let's try this out.

```
>>> import simple
>>> import rtn
>>> for x in rtn.gen3(simple.net3,"noun",1) : print x
...
book
table
top
cover
>>> for x in rtn.gen3(simple.net3,"adjective",1) : print x
...
big
small
red
>>> for x in rtn.gen3(simple.net3,"sentence",1) : print x
...
big  book
big  table
big  top
big  cover
small  book
small  table
small  top
small  cover
red  book
red  table
red  top
red  cover
>>>
```

As an exercise, figure out why the words are separated by 2 spaces instead of 1.

Matching input sentences

Usually, we want to see if a given sentence is valid in the grammar and, if so, what are the constituent parts and how are they related to one another. Sometimes there are multiple possibilities. For example, with the sentence, "The book on the table that is red", you can't tell whether it is the table or the book that is red. Of course we humans will rely on context for resolving ambiguities. In the sentence, "The man wearing the hat that is green", it is almost certain that it is the hat that is green since you almost never see martians wearing hats. But computer programs cannot be expected to know such things. At least not yet! So if our grammar allows ambiguity, the matching algorithm should find multiple interpretations.

Our approach will be to generate all sentences that match an input sentence. If we are unable to find at least one match the sentence is not valid for our grammar. Later, as the match is generated we will track the structures built in subnetworks and combine them into nested list structures. The nesting will show the relationship of the parts in the same way that a sentence diagram does.

We are going to build a noun phrase grammar that includes prepositional phrases. Much like adjectives, these qualify (or zero in on) a noun. "The book on the table with a red cover" has two prepositional phrases both of which distinguish this book from, perhaps, others nearby. However, notice the ambiguity. Is the book or the table covered?

Let's look at a grammar that brings this together. You can access the code in english.py. Here it is.

```
#
# english.py
#
END = None

net = {
   ":noun1": {
      1: (("book",END),("table",END),("top",END),("cover",END))
      },

   ":article": {
      1: (("the",END),("a",END),("an",END))
      },

   ":adjective": {
      1: (("big",END),("small",END),("red",END))
      },

   ":preposition": {
      1: (("on",END),("of",END),("in",END),("with",END))
      },

   ":noun2": {
      1: ((":article", 1), ("",2)),
      2: ((":adjective", 2), (":noun1", END))
      },

   ":prepPhrase": {
      1: ((":preposition", 2), ),
      2: ((":noun3", END), )
      },

   ":noun3": {
      1: ((":noun2", END), (":noun2", 2)),
      2: ((":prepPhrase", END), (":prepPhrase", 2))
      },
}
```

Let's note a few things. The names of the networks now start with a ":". There are two reasons for this. One, we can immediately see that a word represents a subnetwork rather than itself. Two, it allows the name of a subnetwork to stand for itself. Using the colon is simply a convention. The code does not care.

So, here we have basic nouns (:noun1) like "book" and "table", modified nouns (:noun2) that include a possible article (the, a) and perhaps adjectives (red, big) and now noun phrases (noun3) that include prepositional phrases

Two special things need to be brought to attention. First, notice the `("",2)` in state 1 in the :noun2 network. If an :article is not found, we basically can proceed to state2 for free, without consuming any input

Next, notice that in state 2 of :prepPhrase the noun inside the prepositional phrase is also a :noun3. That means, of course, that its noun can also be modified by its own prepositional phrase. We'll see the implications of that shortly.

The match generator

Our first match generator is very simliar to `gen3` above.

```
def match1(network, label, state, input) :
    if   state == END : yield input[:]
    elif not input     : return
    elif not network.get(label) :
        if label == ""        : yield input[0:]  # free pass
        if label == input[0] : yield input[1:]
    else :
        net = network[label]
        p = net[state]; s = ""
        for labl,nextState in p :
            for input1 in match1(network,labl, 1, input) :
                for rest in match1 (network,label,nextState, input1) :
                    yield rest
```

Breathe deeply and relax. It's mostly the same code with the addition of "input" to be matched. The input consists of words in a list like "['the','big','red','book']". As possibilities are generated they must match the input and as that happens the REST of the input is yielded as a result. At the single word level this is "yield input[1:]" and at the phrase level this is "yield rest".

In line 5 (free pass) we are allowing an arc to jump from one state to another without consuming input. Basically an unlabeled arc. This will allow an article like "the" to be optional.

A subtle point should be noted. Yielding `input[0:]` and `input[1:]` create new lists in Python which will be independent of whatever else might happen to `input`. Lists are mutable objects and the program may, at any time, may be generating multiple matches. If these matches were all modifying a common list, they could easily get into the situation of stepping all over their own toes.

Let's play with this a bit. Notice that the words of our input are already split into a list and that a flag word "xx" is being appended.

```
>>> import english, rtn
>>> for s in rtn.match1(english.net, ':noun1', 1, ['book','xx']) : print s
['xx']
```

Here we have a simple match on the word book. It is consumed leaving 'xx' unmatched.

```
>>> for s in rtn.match1(english.net, ':noun1', 1, ['the','book','xx']) : print s
>>> for s in rtn.match1(english.net, ':noun2', 1, ['the','red','book','xx']) : print s
['xx']
```

Trying to match "the book" to a :noun1 fails, but a :noun2 can handle that along with an adjective "red".

Retrieving the sentence structure

Our match generator generates values containing the input that remains to be matched. We would also like to generate a data structure showing the sentence structure of the input that has been matched.

We'll generate nested tuples, one tuple for each network. An example should make this fairly clear. The `noun2` sentence "the red book" is represented by the set of nested tuples

```
>>> for s in rtn.match2(english.net, ':noun2', 1, ['the','red','book','xx']) : print s
(['xx'], (':noun2', (':article', 'the'), (':adjective', 'red'), (':noun1', 'book')))
```

The remaining input "xx" is returned along with a parse tree, each node a 2 element tuple. The first element of each tuple contains the name of the network and the second is the subtree or node that satisfies it. This format makes it fairly easy to do semantic analysis.

And this is the python code:

```
def match2(network, label, state, input,ind=0) :
    if   state == END : yield input[:],None
    elif not input    : return
    elif not network.get(label) :
        if label == ""        : yield input[0:],label  # free pass
        if label == input[0] : yield input[1:],label
    else :
        net = network[label]
        p = net[state]; s = ""
        trace = [label]
        for labl,nextState in p :
            sav1 = trace[:]
            for input1,trace1 in match2(network,labl,1,input) :
                if trace1 : trace.append(trace1)
                sav2 = trace[:]
                for rest,trace2 in match2(network,label,nextState,input1) :
                    if trace2 : trace = trace + list(trace2[1:])
                    yield rest, tuple(trace[:])
                    trace = sav2[:]
                trace = sav1[:]
```

This is essentially match1 extended to generate "trace" tuples. At each level a different trace is started at line 10, grown at line 14 when the first arc is traversed and the rest of it gathered at line 17. The appends and additions are then backed off in lines 19 and 20 in order not to corrupt further possibilities. Incidentally, Don't feel bad if you have to study this closely. I spent hours getting this to work correctly.

Finally, our third matching function is very simple. It generates solutions only where the complete input is matched. It simply yields the trace.

```
def match3 (network, label, state, input) :
    "Use generator to yield all possible matching sentences that consume input"
    import string
    for input,trace in match2(network, label, state, input) :
        if not input : yield trace
```

Dealing with Ambiguity

Let's now play with a sentences that lead into ambiguity.

```
>>> import rtn, english
>>> sentence = ['a','book','on','the','table']
>>> for t in rtn.match3(english.net, ':noun3', 1, sentence) : print "Got one!!", t
Got one!! (':noun3', (':noun2', (':article', 'a'),
              (':noun1', 'book')), (':prepPhrase',
                 (':preposition', 'on'), (':noun3',
                   (':noun2', (':article', 'the'), (':noun1', 'table')))))
>>>
```

That's straightforward. I fussed with the indentation on the trace tree to make it more readable. But now notice this variation.

```
>>> sentence = ['a','book','on','the','table','with','a','cover']
>>> for t in rtn.match3(english.net, ':noun3', 1, sentence) : print "Got one!!", t
...
Got one!! (':noun3', (':noun2', (':article', 'a'),
             (':noun1', 'book')), (':prepPhrase',
                (':preposition', 'on'), (':noun3',
                    (':noun2', (':article', 'the'), (':noun1', 'table')),
                      (':prepPhrase', (':preposition', 'with'), (':noun3',
                         (':noun2', (':article', 'a'), (':noun1', 'cover')))))))

Got one!! (':noun3', (':noun2', (':article', 'a'),
             (':noun1', 'book')), (':prepPhrase',
                (':preposition', 'on'), (':noun3',
                    (':noun2', (':article', 'the'), (':noun1', 'table')))),
            (':prepPhrase', (':preposition', 'with'), (':noun3',
               (':noun2', (':article', 'a'), (':noun1', 'cover')))))
>>>
```

Do you see the difference? In the first interpretation, "with a cover" applies to to the table. In the second, it is the book that has a cover. It's all in counting the parans.

Parsing another language

Finally, let's contrast an english phrase with a japanese one and create a little RTN grammar for japanese.

The phrase we will parse is in english "the book on top of the table".

Now in japanese there are no articles (the, a, an). Prepositional phrases precede the noun and the preposition itself comes at the end. So the sentence looks like the following

```
taberu   no   ue   ni  hon
table    of   top  on  book
```

See japanese.py for the networks.

```
>>> import japanese
>>> sentence = ["taberu","no","ue","ni","hon"]
>>> for t in rtn.match3(japanese.net, ':noun3', 1, sentence) : print "Got one!!", t
Got one!! (':noun3', (':prepPhrase', (':noun2', (':noun1', 'taberu')),
             (':preposition', 'no')), (':prepPhrase', (':noun2', (':noun1', 'ue')),
             (':preposition', 'ni')), (':noun2', (':noun1', 'hon')))
>>>
```

Practical considerations

We just used a network for lexicons of nouns, adjectives, etc. but this would be impractical for anything other than the toy grammars shown here. Instead, having ":noun1" be a network of individual words it would be more practical to have it call a function that accesses a lexicon with a structure indicating a number of things about the word, including that it is a noun and its japanese translation is "taberu". Of course, "table" can also be a verb in which case it would have a different a second lexicon entry

In the next section (late 2009) we'll add the ability to have function calls inside the network. This will make it possible to extend program in interesting ways. We'll also start to play with the semantics of our input sentences, extracting meaning. Keep in touch.

Index

SIR - Semantic Information Retrieval

Introduction.

I was exposed to the original program, written in Lisp by Bertram Rapheal as part of his PHD, in the early 70's in my Artificial Intelligence class. The program inputs information from the user and builds a semantic network, that is, a network of objects connected by their relationships to each other. It can then, in turn, answer questions from the user making deductions from those same relationships.

Objects and Relationships

Consider the following dialogue

```
? the cat owns a collar
 I understand
? fluff is a cat□
 I understand
? does fluff own a collar□
 Not sure
? every cat owns a collar
 I understand
? does fluff own a collar□
 Yes
?
```

The program parses each sentence, which must be in a fairly rigid format, and outputs a response. The way this is done will become more clear as we proceed.

There are basically these kinds of objects

1. Specific objects such as "fluff" or "the cat"

2. Generic objects such as "collar" and "cat" and "animal"

Relationships between objects are the following

1. A specific object may be a member of set of generic objects

2. A specific object may have multiple names (equivalence). "Dave" and "David" may refer to the same person.

3. A generic object may be a subset of another generic

4. A specific object may own (possess) another specific or generic object

5. Each member of a generic set may own another generic object

The original Lisp program as published in Shapiro's "Techniques of Artificial Intelligence" is about 13 pages of code and difficult. Several years ago I tried to basically translate it into Python but with unsatisfactory results. Recently, I took another look at the program with the idea of simplifying it somewhat and of using regular expressions to both parse the input and to express validations for relationship paths when answering questions. The result is a fairly simple program that is only a page and half of Python and should be fairly easy to understand.

Just enough regular expression syntax

I don't want to spend much time on regular expression syntax. There are lots of resources including http://www.amk.ca/python/howto/regex. We can see what this program uses with a single example.

```
>>> import re
>>> m = re.match ("(everylanylanla) (.*) is (alan) (.*)", "a man is a person")
>>> m
<_sre.SRE_Match object at 0xb7ef4cd8>
>>> m.groups()
('a', 'man', 'a', 'person')
>>>
```

The "re" module contains a "match" function which takes 2 parameters; a pattern and a string to match against the pattern. If the match is successful a "match" object is returned. The "groups" method returns a tuple with the parts matching subpatterns enclosed in parentheses. The "|" character makes alternative matches possible. The "." character matches any single character and finally the "*" character indicates that the preceding character (or group) may be present zero or more times.

That's basically all we need to know

A Quick Overview of the Code

At this point, bring the code for sir.py into another window or make a printout.

The first thing you'll notice is the "rules" table that matches sentence patterns with actions. Statements that are input result in calls to the function "addFact". Questions result in calls to the function "getPath". There are also simple commands to dump the facts table, toggle the debug flag, and exit the program gracefully.

As always, the program starts in the function "main" which reads input from either a file (or files) followed by input from the user. A file name of "." bypasses input from the keyboard which makes it convenient to do regression testing as the program is developed

The interesting bits will now be looked at in more detail

Relationship Paths

It's pretty easy to see how regular expressions can match our input statements and questions. They can also be used to describe valid chains of relationships used to answer questions.

Inputting a sentence like "every man is a person" adds a relation to our "database" of relations. This is actually just a list of relations ("facts"). Each relation is a tuple. In this case the tuple will be ('man','s','person') where 's' means 'subset'.

The function "matchSent" matches your input sentence against the patterns in "rules". If a match is found, the function "addFact" is called with 2 arguments; the groups from the match and a command string like "1s3|3S1". Commands in the string are seperated by a "|". This string means add the relation "group(1) is a subset of group(3)" and add the relation "group(3) is a superset of group(1). Here are some more examples.

```
rules = (
 ("(everylanylanla) (.*) is (alan) (.*)",lambda g: addFact(g,"1s3l3S1")),
 ("(.*) is (alan) (.*)",               lambda g: addFact(g,"0m2l2M0")),
 ("(.*) is (.*)",                    lambda g: addFact(g,"0e1l1e0")),
 ...
```

Single characters to denote relationships. Lower case letters go one way and upper case the opposite. The meanings are as follows

"s" is subset, "S" is superset

"m" is member, "M" is contains

"e" is equivalent

"p" is "possess", "P" is "possessed by"

If we run the program in debug mode, the function addFact give us more information than just the mysterious "I understand".

```
: python sir.py
? debug
? every man is a person
  adding fact ('man', 's', 'person')
  adding fact ('person', 'S', 'man')
  I understand
?
```

Things get interesting when we start asking questions. Leaving debug on, watch the following

```
? every man is a person
  adding fact ('man', 's', 'person')
  adding fact ('person', 'S', 'man')
  I understand
? fred is a man
  adding fact ('fred', 'm', 'man')
  adding fact ('man', 'M', 'fred')
  I understand
? is fred a person
  path -  fred  to  person
    path -  man  to  person
  Yes e*ms* ['ms']
?
```

Now "fred" is a member of the set "man" and "man" is a subset of "person". Finding a path from "fred" to "person" becomes a two step process that we can denote with "ms" (member-subset)

The rule matching "is fred a person" is

```
( "is (.*) (alan) (.*)",   lambda g: getPath(g,"0e*ms*2"))
```

which results in a call to the function "getPath" finding a path from "fred" (g[0]) to "person" (g[2]) using the pattern "e*ms**". A path "ms" (member-subset) was found to link "fred" to "person". This is the simplest case for this pattern; there were no "equivalence" or additional "subset" steps necessary.

Moving along, let's add an equivalence at the front end and another subset on the back end to see the full pattern put to use.

```
?  every person is a creature of god
   adding fact ('person', 's', 'creature of god')
   adding fact ('creature of god', 'S', 'person')
   I understand
? the big guy is fred
   adding fact ('the big guy', 'e', 'fred')
   adding fact ('fred', 'e', 'the big guy')
   I understand
? is the big guy a creature of god
   path -  the big guy  to  creature of god
     path -  fred  to  creature of god
       path -  man  to  creature of god
         path -  person  to  creature of god
       path -  the big guy  to  creature of god
   Yes e*ms* ['emss']
?
```

Forward and backward paths

Consider the following dialogue from the file sir.1.txt

```
$ python sir.py sir.1.txt .
bill is a man
  I understand
is bill a person
  Not sure
bill owns a porsche
  I understand
does any person own a car
  Not sure
a man is a person
  I understand
does any person own a car
  Not sure
a porsche is a car
  I understand
debug
does any person own a car
```

```
  path -  person  to  car
    path -  man  to  car
      path -  bill  to  car
        path -  porsche  to  car
  Yes S*Me*ps* ['SMps']
```

Now we can see the reason for adding bidirectional rules. The path is 'SMps'. "person" is a superset of "man". "man" contains "bill" as a member (M relation), "bill" owns a "porsche", and finally, "porsche" is a subset of "car".

Efficiency in path searching

The function "path" is recursive and left to its own devices can conduct a very large search. It needs to be controlled a bit. A couple of techniques are used.

The "used" dictionary at each recursion level keeps the search from getting into a loop. It requires that each relationship added to a path is "fresh", not used somewhere earlier in the path. The problem is easy to see if we have "Joe is Joseph" and an "e*" in the validation pattern. Then we quickly get "Joe->Joseph->Joe->Joseph..." until we have a stack overflow. The following clips this possibility.

```
if used.get(fact) : continue
```

At each level of recursion "used" carries all the earlier relationships and adds its own. A new dictionary is created and is updated with the preceding relations already used. Backing out again our new entries are forgotten so that other paths may still be explored that use the same relationships in other ways.

The function "okSoFar" checks that as each relationship is added to the path the pattern is still partially matched. If that is not the case then we have started down a blind alley.

A final check involves the variable "indent" in the function "path" which grows with each level of recursion. Simply checking that it is not growing too big will eventually abandon any path caught up in an infinite loop.

With a little unix (linux) magic we can get a feel for the importance of these checks. Let's run the above dialogue but pipe the output to "grep" to filter out all lines not containing the word "path".

```
$ python sir.py sir.inp . l grep path
  path -  person  to  car
    path -  man  to  car
      path -  bill  to  car
        path -  porsche  to  car
```

Now let's pipe this output to the program "wc" (word count) to see the number of lines

```
 python sir.py sir.inp . l grep path l wc
     4     20     112
```

So, 4 lines, 20 words, 112 characters. If you now comment out the lines

```
sts = okSoFar(pat, sofar+rel)
if not sts : continue
```

you will see the 4 lines jump to an 8. If you additionally comment out the line

```
if used.get(fact) : continue
```

you see the 8 jump to 232! The contraints are quite important.

Index

Using Unicode with Python

Computers really work only with numbers; binary numbers. But to communicate with people computers had to have some ability to input and output words almost from day one.

The trick has always been to provide a mapping of numbers to characters. There were several standards, including EBCDIC by IBM, but the Ascii character set emerged as a clear winner in the race for a standard by the late 1970's. It provides for 127 characters, of which about 96 are printable. These include most punctuation characters, decimal digits and both the upper and lower case alphabet. In addition, low numbers from 0 to 32 are used for control characters such as space, tab, and newline.

Most computers use an 8 bit byte to store a character (or more accurately the number representing the character) but there have been exceptions. The DEC PDP-10, a timesharing computer we used at the University of Oregon in the 70's, had a word size of 36 bits and would store 5 characters per word with a bit left over.

If 8 bit bytes are used to store a character that leaves 1 bit unused, effectively doubling the number of possible characters. And there are lots of candidates. Most European languages need several letters beyond those provided in the basic Ascii set, and some like Greek and Russian need entire alphabets. Individually, many of these needs can be accomodated but not all at the same time. This led to a variety of extensions to Ascii, mutually incompatible. We will write a bit of Python code that generates a web page to illustrate this in a bit.

Octal and Hexidecimal numbers

Computers actually do everything in binary numbers but reading binary numbers is hard on people. For example decimal 133 is "01000101" in binary (the 3 '1' bits adding up to 128+4+1=133) but one would hardly know to look at it. Humans process information better in fewer but bigger chunks. Octal and hexidecimal numbers are a convenient compromise. We get smaller chunkier numbers that still let us see the binary pattern in the number.

Octal (base 8) uses the digits 0-7 and is directly transformed from binary 3 bits at a time. Octal became popular since several early computers with 16 bit word size, used 3 bit fields for fields within a machine instruction. In the PDP-11, for example, 0001000101000011 moves the contents of register 5 to register 3. If this number is first chunked into 0 001 000 101 000 011 it becomes 010503 in octal. The fields are 01=move; 05 and 03 are the registers. In hexidecimal the same number chunks into 0001 0001 0100 0011 and is represented by 1143. That's not as useful.

Other computers use 4 bit chunks for fields. The DEC VAX-11 which basically replaced the PDP-11 over time has a 32 bit word size. It has 16 registers instead of the 8 on the

PDP-11. So hexidecimal is a natural choise to represent Vax instructions. Hexidecimal needs 16 digits and uses 0-9 plus A-F (either case).

Both octal and hexidecimal may be used to represent 1 byte characters. In Python any character, especially a non-printing character may be input to a string by using its 3 digit octal value following a '\'. It may also be input with its 2 digit hexidecimal value following a '\x'. Python displays non-printing characters in a string in either format with newer versions of Python preferring hexidecimal and older versions octal.

```
>>> a = '\x41bc'
>>> a
'Abc'
>>> '\x10bc'
'\020bc'
>>>
```

For character data I find hexidecimal better. Exactly 2 digits are needed for a byte and if a number occupies multiple bytes the digits break evenly on the byte boundaries. We will stick with hexidecimal in this tutorial.

A look at some encodings

Different encodings are just different mappings of numbers to characters. Many go by "ISO-8859-x" where "x" is 1 to 15 or so. These encodings generally also have a more common name such as "Ascii", "Latin-1", "Nordic", etc.

The following program table.py creates a small web page with a table. The table is 16x16 cells, each containing a character and its corresponding hexidecimal value. Load table.html into your browser, either Netscape, Mozilla or Internet Explorer. (Warning: Opera has problems with foreign character sets) Next go to the view menu and choose "character encoding" if you are using Netscape, "encoding" if Internet Explorer or "character coding" for Mozilla. Select various 8 bit sets such as ISO-8859-1, ISO-8859-15, Cyrllic (windows), Greek etc. Don't worry about UTF-8 or UTF-16 if you see them on the menu. Just ignore them for now.

As you select alternate encodings, notice how the first half of the table stays the same but how the second half (hex 80-ff) changes quite a bit from one encoding to another. For example, You may find that hexidecimal "DF" is the Russian letter **Я** in ISO-8859-5 or "Cyrillic" but something else in another encoding.

With these 8 bit encodings it is possible to mix English with one or more other european languages, but not two foreign languages together. It is also difficult for the software to keep track of which encoding is in use in a particular document.

An entirely different problem comes up with languages like Chinese and Japanese which use thousands of characters within a single language.

Enter Unicode

There is a simple and obvious way to get around all of this. Stop trying to cram all of the characters in all of the worlds languages into 8 bits. Basically, Unicode uses 16 bit numbers to assign a unique number to every character in every alpabet. 16 bits yields 65536 possible values. With Unicode you can have Russian, Greek and any other language all in the same document with no confusion.

For convenience Unicode uses the values 00-7f for the same charaters as Ascii. In addition, the character values 80-ff match those in the iso-8859-1 encoding.

You can find out everything about Unicode here. Click on "Charts" to see whole sets of encodings.

Unicode and Python

Python supports Unicode strings whose individual characters are 16 bits. A Unicode string literal is preceded with a 'u'. For example

```
>>> a = u'abcd'
>>> print a
abcd
```

Characters in the Unicode string may be simple Ascii characters which map to Unicode characters with the same value or they may be extended characters. Any extended character (whose value is greater than 0x7F) may be input with the string '\uxxxx' where xxxx is a 4 digit hexidecimal number. Python will display it in the same format. For example

```
>>> a = u'abc\uabcd'
>>> a
u'abc\uABCD'
>>> len(a)
4
>>> a[2]
u'c'
>>> a[3]
u'\uABCD'
>>>
```

The hexidecimal number 'ABCD' above is a single character. Unfortunately we can't see directly what character in what language it represents from Python's interactive mode. However we'll see how to view it shortly in a web browser.

There are some Python builtin functions for Unicode. The function `unicode` will convert 8 bit encodings into Unicode. It takes 2 arguments, a normal string with 8 bit characters and a string describing the encoding. For example

```
>>> a = 'abc\x81'
>>> unicode(a, 'iso-8859-1')
u'abc\x81'
>>>
```

The function converts a string of 8-bit character to one of 16-bit characters. The result is the corresponding Unicode string. If the encoding is 'ascii' then the function will complain if any character is above 7F hex.

```
>>> unicode(a, 'ascii')
Traceback (most recent call last):
File "<stdin>", line 1, in ?
UnicodeError: ASCII decoding error: ordinal not in range(128)
```

Now consider the following

```
>>> unicode('abc\xe4', 'iso-8859-1')
```

```
u'abc\xe4'
>>> unicode('abc\xe4', 'iso-8859-5')
u'abc\u0444'
```

In the iso-8859-1 encoding (Central European) the 8 bit character is mapped to the same value in Unicode (hex e4) which represents the character "**ä**" but in iso-8859-5 \xe4 is mapped to Unicode \x0444 which represents the russian character "**я**". The second argument in the unicode function call may be omitted if it is 'Ascii'.

With ordinary strings two functions convert a single character string to and from its numeric value. The function "ord" works with both ordinary and Unicode strings returning the numeric value of a character.

```
>>> print ord(u'A')
65
>>> print ord(u'\u0444')
1092
```

The inverse function for ordinary strings is "chr" which returns a string of length one for a numeric value in the range 0-255. A new function "unichr" returns a Unicode string of length one for a number value 0-65535.

```
>>> chr(0xef)
'\357'
>>> unichr(1092)
u'\u0444'
```

There is much more support built into Python for Unicode strings. The "string" module will split, join, strip, etc Unicode strings just like ordinary strings. In fact you can mix them much like you might mix intergers and longs in numeric expressions. Python will first convert the ordinary string into a (more general) Unicode string and then perform the operation requested. Here is an example of splitting and joining strings.

```
>>> a = u'abcd:efg'
>>> string.split(a, u':')
[u'abcd', u'efg']
>>> string.join(['abcd',u'efg'],':')
u'abcd:efg'
>>> a == 'abcd:efg'
1
```

This also works with other modules like "re" that deal with strings.

Encoding Unicode strings

The inverse of the `unicode` function is to invoke the `encode` method on a Unicode string. The result is an ordinary string in the 8 bit encoding. Unicode strings are objects. Here is an example.

```
>>> a = u'abcd\u0444'
>>> a.encode('iso-8859-5')
'abcd\344'
>>>
```

Remember that in Unicode 0x444 is the Russian character "**я**" and corresponds to 0xe4 in 8 bit iso-8859-5.

New encodings for Unicode

Files on the disk consist of byte streams, 8 bits each, and in order to handle Unicode in a file we need to use more than one byte per character. The two most common encodings are "utf-16" and "utf-8", each with some advantages.

Utf-16 simply represents Unicode values in two bytes each. So our Unicode character "**я**"=u"\u0444" becomes 0x04 0x44 in utf-16. Or possibly 0x44 0x04. The first is in big endian and the second in little endian format. Which one is being used is determined by the first two bytes of a Utf-16 string (or file). They are either 0xFF 0xFE or 0xFE 0xFF.

Utf-16 is a good encoding to use for Japanese or Chinese with their huge character sets since 2 bytes per character is good fit. But it's not so good for English or most other european languages. With English alternate bytes are basically Ascii and zeros in between. So half the space is wasted.

Utf-8 encoding is very effective in dealing with this. Utf-8 encodes a Unicode character in one or more bytes, but only as many as are needed to represent characters number. For normal Ascii characters that is a single byte. So Utf-8 strings and files containing only characters 0-7f are exactly the same. Above that range Utf-8 uses its own values which are carefully chosen so that it can be determined exactly how many bytes to grab for the next character. If the first byte is 0x80 or above at least one more byte will follow and depending on the value in the 2nd byte a third byte might follow and so on. For example the Korean character represented in Unicode as u'\uC5D0' is the 3 byte sequence 0xec 0x97 0x90 in Utf-8.

So you can see that Utf-8 is more compact for european languages where most characters require a single byte except where two bytes are required for accents, umlauts, etc. But if most or all of the text is in Chinese then Utf-16 with 2 bytes per character will beat Utf-8 for compactness.

Examples of UTF-8 and UTF-16

Sometimes the best way to understand something is to simply see some examples. Lets start with a little piece of UTF-16 which contains both Ascii characters and some japanese characters. With the aid of a little program decode.py we can look at the file byte by byte. The program replaces bytes with value 0 (about half) with a '.' and bytes above 0x80 with their hexidecimal value bracketed by '-'s. The result may be viewed without the html being interpreted by the browser. The english part is fairly readable.

Notice that the first two bytes are 0xFF an 0xFE which specify the byte order of all the characters to come. The Ascii compatible characters have their Ascii value in the first byte and zero in the following byte.

Notice the meta tag contains 'content="text/html; charset="utf-16"'. This is picked up by the browser and controls how the page is interpreted. Without it the page does not display.

Notice too, that when we get into the japanese characters themselves there are no "zero" bytes. The first character is "S0" (the character zero, not the value), which together make the hexidecimal value 0x3053 which is the Japanese hiragana こ.

Utf-8 is really the more interesting of the two common Unicode encodings. Our little decode.py works fine for utf-8 as well. Here a sample in html and the same text decoded. Thanks to "Joel on Software" from whose pages I lifted little snippets. Check out his

Unicode writeup here.

A little application

I enjoy foreign languages but my keyboard is standard American. In order to type Russian or even German I would normally have to jump through some hoops. But here's some code to make it a bit easier.

In German the special characters **ä, ö, ü and ß** may be represented by the pairs **ae, oe, ue and ss**. This is an old trick that dates back to Morse code, I believe, and is how I generally type German. With a small table german.py and a Python program utf8.py any string with such pairs will be transformed into a utf-8 string with these pairs properly encoded.

The program transforms stdin to stdout looking for text between pairs of xml tags (<german>...</german>, <russian>...</russian> and <unicode>.</unicode>). In the case of german and russian any amount of text can be transformed. With the "unicode" tag only a single 4 digit hex unicode value is encoded to utf-8. In addition, the meta tag with 'charset="utf-16"' is inserted into the header so your browser interpretes the codes correctly.

The function `encode`imports `table` from a module passed or, if no module is passed, translates a single Unicode character into utf-8. Besides german, the table russian.py lets me type russian in semi-phonetic latin characters like **Ya nye znal yego ochyen khorosho** (I didn't know him very well) and have it come out **Я не знал его очен кhорошо**. (Actually, I'm missing a soft sound at the end of **очен**.) Incidentally, If you do a "view source" on this page you will see that it is utf-8 and in fact I used this program to convert this writeup from its original html (generated from lore) into what you are now viewing.

Index

Logic Circuits - Part One

There are several motivations for simulating logic circuits in Python. First it is a nice simulation exercise. Second it shows off object oriented programming well, especially the power of inheritance. Finally real logic circuits built with electronic components are the basis for computer hardware. The simplest are gates (AND, OR, NOT) made with only a few transistors and resisters. Since the mid 1960's these circuits have been fabricated in integrated circuits (chips). Before that they were built with seperate transistors and resisters wired together on circuit boards. Today you can purchase chips with, say, six NOT gates or four 2 input NAND gates (an AND gate followed by a NOT) for as little as a quarter at your local electronics retailer.

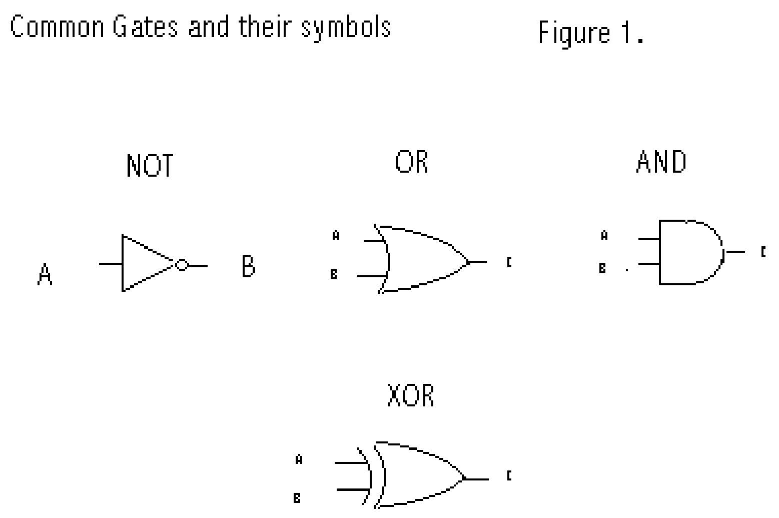
We will use a couple of Python classes to simulate the behavior of logic circuits. We will have a base class LC (logic circuit) which will be subclassed to actual gates and other circuits. A second class, Connector, will be used to provide inputs and outputs to the logic circuits. An output connectors *connect* method will simulate the wiring of LC's to one another.

Click Here for Full Python Source

We will start with 3 simple LC's and proceed to build everything else from them. When we are done with this part we will have a LC that adds two binary numbers together. In the next part we will see how the same simple LC's can be used to provide computer memory and even multiplication of binary numbers.

Our 3 basic LC's are the two input AND and OR gates and the NOT gate (sometimes called an inverter).

The AND gate has two inputs (generally labeled A and B) and and output C. The inputs and output may have the value 0 or 1. The output C is 1 if and only if both A and B are 1. Otherwise it is 0. (see fig. 1)

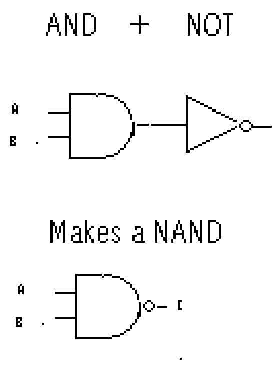


The OR gate is very similar with the same inputs A and B, and output C. The output C is 1 if either A or B is 1, Otherwise it is 0.

The NOT gate has a single input A and output B. The output is the opposite of the input. If the input is 0, the output is 1. If the input is 1, the output is 0.

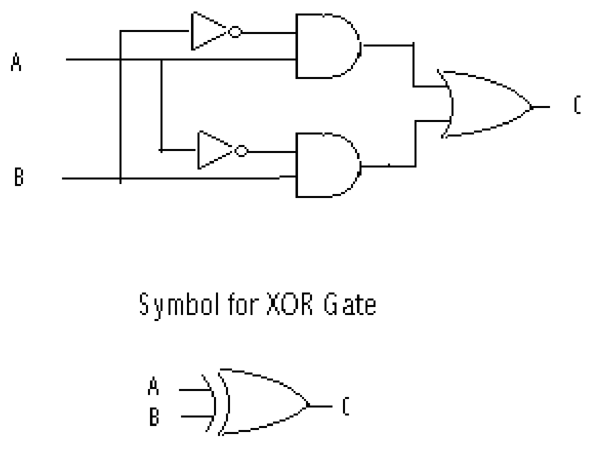
From these basic 3 LC's (or gates), everything else is built by using existing LC's and connecting outputs to inputs in certain ways. For example to make a NAND gate where the output C is 0 if and only if both inputs are 1, we can use and AND gate followed by a NOT gate. The inputs of the AND gate are the inputs to the NAND. The AND output is connected to the NOT input. The NOT output is the output for the NAND. See figure 2.

Figure 2. Simple Composite Gate



A more interesting LC (see figure 3) is the XOR gate. It is built with 2 NOT's, 2 AND's and an OR gate. Its output C is 1 if the inputs A and B are not the same. We can see how this works looking at the figure. The LC output is the OR gate output and will be 1 if either of the AND gate outputs are 1. The top AND gate output is 1 if input A is 1 and B is 0. The lower AND gate output is 1 if A is 0 and B is 1. If A and B are the same, both AND gate outputs are 0. You should convince yourself of this in more detail.

Figure 3. Composite Circuit for XOR Gate



Lets look at our Python classes now. The class Connector will provide inputs and outputs for instances of the LC class (and subclasses). Each Connector will belong to an owner and have a name. By convention the name is a string that reflects the variable name we choose for the connector. This will let us monitor outputs during execution. Every connector has a current value, either 0 or 1. This will change as the program executes.

An output connector will normally be connected to one or more input connectors unless it is a final output, in which case it will probably be monitored. Input connectors will have their "activates" attribute true, so that when their value is changed they wake up their owner (an LC) to reevaluate its output(s).

So this is the code to initialize a Connector instance

```
class Connector :
  def __init__ (self, owner, name, activates=0, monitor=0) :
    self.value = None
    self.owner = owner
    self.name  = name
    self.monitor  = monitor
    self.connects = []
    self.activates= activates
```

For output connectors we will call the connect method to provide the output with a list of its input connectors. For convenience, a single input does not have to be placed in a list.

```
def connect (self, inputs) :
  if type(inputs) != type([]) : inputs = [inputs]
  for input in inputs : self.connects.append(input)
```

The set method will be called from different places depending on whether we have an input or output connector. Outputs will be set by their owners evaluation functions. Outputs will in turn set their inputs. As an input is set, it triggers its owners evaluation, cascading the logic activity. If a connector is being monitored (self.monitor==1) it prints the owners name, its own name and the value it is being set to.

```
def set (self, value) :
  if self.value == value : return      # Ignore if no change
  self.value = value
  if self.activates : self.owner.evaluate()
  if self.monitor :
    print "Connector %s-%s set to %d" % (self.owner.name,self.name,self.value)
    for con in self.connects : con.set(value)
```

Logic circuits will be simulated in the LC class and its subclasses. There are two types of logic circuits, simple and composite. Simple logic circuits have inputs, outputs, and an evaluate method. Composite circuits have inputs, outputs, and a collection of internal LC's and connector objects "wired" together. Each input of a composite LC will be connected to an input of an internal LC. Likewise each output of the composited LC will come from an output of an internal LC. For Composite LC's the evaluation function is unused.

Our base class LC is a virtual class. That is, we won't build instances (objects) from it directly. It will be subclassed to more useful purposes. It provides the logic to give an LC a name and a default evaluation function.

```
class LC :
  def __init__ (self, name) :
  self.name = name

  def evaluate (self) : return
```

Our first real LC is a NOT gate or inverter. It calls its superclass, sets up an input connector A, an output connector B, and an evaluation method. The evaluation method sets B to 1 if A is zero, or B to 0 if A is 1. A Not LC is a example of a simple LC.

```
class Not (LC) :         # Inverter. Input A. Output B.
  def __init__ (self, name) :
    LC.__init__ (self, name)
    self.A = Connector(self,'A',activates=1)
    self.B = Connector(self,'B')

  def evaluate (self) : self.B.set(not self.A.value)
```

Next we will define classes for our other two simple LC's, the AND and OR gates. First lets define a convenience class for two input gates in general that have inputs named A and B and an output C.

```
class Gate2 (LC) :  # two input gates. Inputs A, B. Output C.
  def __init__ (self, name) :
    LC.__init__ (self, name)
    self.A = Connector(self,'A',activates=1)
    self.B = Connector(self,'B',activates=1)
    self.C = Connector(self,'C')
```

With that out of the way we just have to add evaluation functions to make the AND or OR classes.

```
class And (Gate2) :       # two input AND Gate
  def __init__ (self, name) :
    Gate2.__init__ (self, name)

  def evaluate (self) : self.C.set(self.A.value and self.B.value)

class Or (Gate2) :        # two input OR gate.
  def __init__ (self, name) :
    Gate2.__init__ (self, name)

  def evaluate (self) : self.C.set(self.A.value or self.B.value)
```

Let's play with this a bit. We'll instantiate an AND gate, set its output for monitoring and then set its input connectors both to 1.

```
>>> from logic import *
>>> a = And('A1')
>>> a.C.monitor=1
>>> a.A.set(1)
>>> a.B.set(1)
Connector A1-C set to 1
```

Next we will instantiate an inverter (NOT gate) and connect it to the output of the AND gate. See figure 2. Then we'll change one the inputs.

```
>>> n = Not('N1')
>>> a.C.connect(n.A)
>>> n.B.monitor=1
>>> a.B.set(0)
Connector A1-C set to 0
Connector N1-B set to 1
>>>
```

Now we will build a class for a composite LC, the XOR gate. See figure 3. When an instance of an Xor is built, we grab 2 AND gates, 2 Inverters, and an OR gate and wire them up as shown in the figure. Notice in particular how we connect the external input (self.A) to an input of an internal gate (self.A1.A)

```
class Xor (Gate2) :
  def __init__ (self, name) :
    Gate2.__init__ (self, name)
    self.A1 = And("A1") # See circuit drawing to follow connections
    self.A2 = And("A2")
    self.I1 = Not("I1")
    self.I2 = Not("I2")
    self.O1 = Or ("O1")
    self.A.connect    ([ self.A1.A, self.I2.A])
    self.B.connect    ([ self.I1.A, self.A2.A])
```

```
        self.I1.B.connect ([ self.A1.B ])
        self.I2.B.connect ([ self.A2.B ])
        self.A1.C.connect ([ self.O1.A ])
        self.A2.C.connect ([ self.O1.B ])
        self.O1.C.connect ([ self.C ])
```

So let's play with an XOR gate for a minute. Remember the output should be 1 if the inputs are different.

```
>>> o1 = Xor('o1')
>>> o1.C.monitor=1
>>> o1.A.set(0)
>>> o1.B.set(0)
Connector o1-C set to 0
>>> o1.B.set(1)
Connector o1-C set to 1
>>> o1.A.set(1)
Connector o1-C set to 0
>>>
```

Now, finally, we are ready to talk about addition. So let's start with something familiar. Suppose we want to add 168 and 234. Hmm. 8 plus 4 is 2 and carry 1. 1 plus 6 plus 3 is 0 and carry a 1. Finally 1 plus 1 plus 2 is 4. Answer 402.

So when we were all 7 years old we memorized 100 (10 times 10) rules of addition like 8+4=12 (or 2 and carry 1). Of course if you get tricky and remember that 8+4 is the same as 4+8 and that zero plus anything is just that anything, you can reduce the amount of memorization to 45 items.

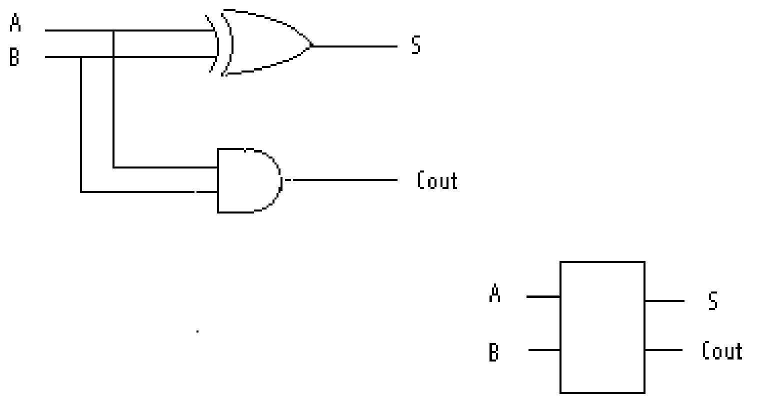
In binary it is so much simpler. 4 rules (2 times 2) instead of 100. If you are tricky, just 1. Basically that 1+1=2 (well 10, that is 0 and carry 1). Now to represent a number in binary requires about 3.5 times as many digits as decimal but you can start to see that this may be a good tradeoff.

Actually so far we're just talking about the first (rightmost) column. There is no carry in yet. But bear with me for a moment. Lets summarize the 4 rules of binary addition

```
0 + 0 = 0
0 + 1 = 1
1 + 0 = 1
1 + 1 = 0  and carry a 1
```

We will build a half adder circuit with 2 inputs and 2 outputs. See figure 4. The Sum output is the same as the XOR gate above. That is the sum bit is one if one and only one of the input bits is 1. The carry out is 1 only if both inputs are 1. For that we need an AND gate.

Figure 4. Half Adder Circuit and Symbol



```
class HalfAdder (LC) :        # One bit adder, A,B in. Sum and Carry out
  def __init__ (self, name) :
    LC.__init__ (self, name)
    self.A = Connector(self,'A',1)
    self.B = Connector(self,'B',1)
    self.S = Connector(self,'S')
    self.C = Connector(self,'C')
    self.X1= Xor("X1")
    self.A1= And("A1")
    self.A.connect    ([ self.X1.A, self.A1.A])
    self.B.connect    ([ self.X1.B, self.A1.B])
    self.X1.C.connect ([ self.S])
    self.A1.C.connect ([ self.C])
```

Let's build a half adder, monitor its outputs as we play with the inputs

```
>>> h1 = HalfAdder("H1")
>>> h1.S.monitor=1
>>> h1.C.monitor=1
>>> h1.A.set(0)
Connector H1-C set to 0
>>> h1.B.set(0)
Connector H1-S set to 0
>>> h1.B.set(1)
Connector H1-S set to 1
>>> h1.A.set(1)
Connector H1-S set to 0
Connector H1-C set to 1
>>>
```

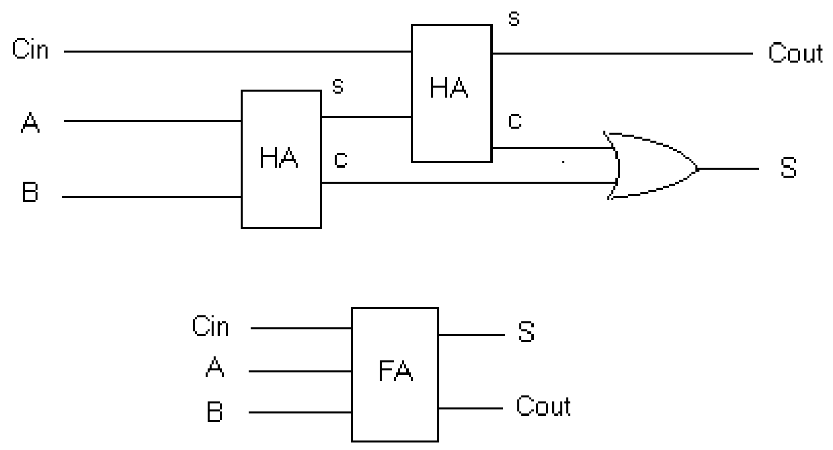
Now we're ready for the coup de grace. Let's go back to decimal for just a moment. Suppose we are adding the following.

```
  189
  243
```

and we are working on the middle column. We'll have a carry in of 1 (9+3=2, carry the 1) plus 8 plus 4. We'll add the 8 and 4 to get 12 (actually 2 and carry a 1) and then add the carry in 1 to the 2 to get 3 for sum digit. In the 2 additions, at most only one will generate a carry. You might want to play with some examples to convince yourself of this.

Look at figure 5. We add A and B and take the sum bit and add it to the carry in bit. Only one of these additions might produce a carry out which is propogated to the next column.

Figure 5. Full Adder Circuit

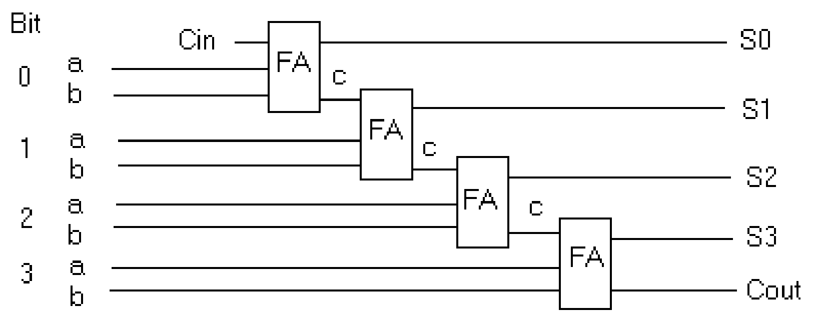


So here is our class for a full adder. 3 inputs, 2 outputs, and wired like shown in the figure.

```
class FullAdder (LC) :         # One bit adder, A,B,Cin in. Sum and Cout out
  def __init__ (self, name) :
    LC.__init__ (self, name)
    self.A    = Connector(self,'A',1,monitor=1)
    self.B    = Connector(self,'B',1,monitor=1)
    self.Cin  = Connector(self,'Cin',1,monitor=1)
    self.S    = Connector(self,'S',monitor=1)
    self.Cout = Connector(self,'Cout',monitor=1)
    self.H1= HalfAdder("H1")
    self.H2= HalfAdder("H2")
    self.O1= Or("O1")
    self.A.connect    ([ self.H1.A ])
    self.B.connect    ([ self.H1.B ])
    self.Cin.connect  ([ self.H2.A ])
    self.H1.S.connect ([ self.H2.B ])
    self.H1.C.connect ([ self.O1.B])
    self.H2.C.connect ([ self.O1.A])
    self.H2.S.connect ([ self.S])
    self.O1.C.connect ([ self.Cout])
```

Instead of playing with a single full adder (you are welcome to, of course), we will cut to the chase and write a function that wires up four full adders and let's us do addition in binary. Figure 6 is a schematic of our circuit.

Figure 6. Four Bit Binary Adder Circuit



We will represent a binary number in a string of zeros and ones. For example '0101'. We'll use a helper function to extract a bit from a given position as an actual 0 or 1 instead of a character.

```
def bit (x, bit) : return x[bit]=='1'
```

Our test function will take in two binary numbers (as strings), build an 4 bit adder, set the inputs and then print out the result, including the final carry out. Just as in decimal, where 2 four digit numbers may sum to a five digit result, we need to provide the last carry out as a digit too.

```
def test4Bit (a, b) :    # a, b four char strings like '0110'
   F0 = FullAdder ("F0")
   F1 = FullAdder ("F1"); F0.Cout.connect(F1.Cin)
   F2 = FullAdder ("F2"); F1.Cout.connect(F2.Cin)
   F3 = FullAdder ("F3"); F2.Cout.connect(F3.Cin)

   F0.Cin.set(0)
   F0.A.set(bit(a,3)); F0.B.set(bit(b,3))  # bits in lists are reversed from natural order
   F1.A.set(bit(a,2)); F1.B.set(bit(b,2))
   F2.A.set(bit(a,1)); F2.B.set(bit(b,1))
   F3.A.set(bit(a,0)); F3.B.set(bit(b,0))

   print F3.Cout.value,
   print F3.S.value,
   print F2.S.value,
   print F1.S.value,
   print F0.S.value,
```

Let's try it out. Notice that in the FullAdder class definition we told it to monitor all inputs and outputs. This will let us watch the logic cascade.

```
>>> test4Bit ('0100','0010')
Connector F0-Cin set to 0
Connector F0-A set to 0
Connector F0-Cout set to 0
Connector F1-Cin set to 0
Connector F0-B set to 0
Connector F0-S set to 0
Connector F1-A set to 0
Connector F1-Cout set to 0
Connector F2-Cin set to 0
Connector F1-B set to 1
Connector F1-S set to 1
Connector F2-A set to 1
Connector F2-B set to 0
Connector F2-S set to 1
Connector F2-Cout set to 0
Connector F3-Cin set to 0
Connector F3-A set to 0
Connector F3-Cout set to 0
Connector F3-B set to 0
Connector F3-S set to 0
0 0 1 1 0
>>>
```

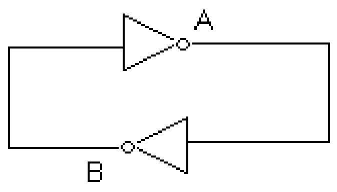
Or, in other words, 2+4=6. In case you think there is any cheating going on, search for the '+' character in the file logic.py. You won't find one.

Logic Circuits - Part Two

In this study we will look at how logic gates may be used for computer memory, counters and shift registers. We'll cap it off with a discussion about a possible circuit for binary multiplication. I'm assuming you have already studied "Logic Circuits - Part One" fairly throughly. As before we will simulate these circuits in Python by extending the LC class in order to watch them in action.

So far all of the circuits we have seen process inputs to yield outputs. The information flow has been "left to right" in the diagrams. Occasionly this flow can be fairly involved; for example, in the 4 bit adder of the previous chapter carry signals must be transmitted from bit to bit. However in order for logic gates to be used as memory devices we need something more. With "positive feedback" an output is used to reinforce its own input in order to hold a value after the original input has gone. Such circuits go by the generic name of "flip flops".

Let's look at the simplest possible example of this. Here we have 2 inverter (NOT) gates connected in a circular fashion to provide feedback. Recall that a NOT gate's output is always the opposite of its input.



Simplest Memory Circuit

So if the output A is 1 this would make output B 0. And if B is 0 then it would in turn hold A at 1. This is the positive feedback that provides the ability of the circuit to "remember" it value. The situation could just as easily be reversed with B at 1 and A at 0. In either case the arrangement is stable.

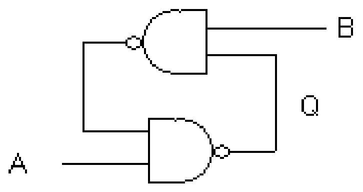
Now if for some reason both outputs were ever the same (such as when they are being powered up), both would try to change. Whichever gate was tiniest bit faster would flip its output just ahead of the other and the circuit would fall into one of the two stable states.

Of course this circuit is rather useless since we have no way of changing the state. That's about to be fixed with our next flip flop.

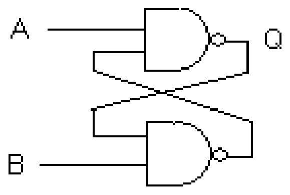
The Latch Flip Flop

The next flip flop, the LATCH, is similar but has inputs as well as outputs. It uses 2 NAND gates each of which, as you recall, consists of an AND gate followed by an inverter. The output of a NAND gate is 0 only when both inputs are 1.

Look at the circuit on the left. If the inputs A and B are both 1 (the normal mode) this circuit would act the same as the one above. But if input A is taken to zero then Q will be forced to 1. Q being 1 and input B being one makes the output of the upper NAND gate 0 which holds Q at 1 even after A is taken back to 1. The opposite happens when B is temporarily taken to 0. So a LATCH circuit remembers which input was last "dropped" to zero.



Simple Latch Flip Flop



How it's normally represented

A LATCH flip flop can be simulated with the following extension of the LC class. The first thing we'll do is define a two input NAND gate. We've made it a basic gate with an evaluation function but it could have as easily been programmed as a composite gate consisting of an AND gate and a NOT. Compare this code with the class definition for an AND gate in the prior study. You should find only one tiny difference.

```
class Nand (Gate2) :      # two input NAND Gate
  def __init__ (self, name) :
    Gate2.__init__ (self, name)

  def evaluate (self) :
    self.C.set(not(self.A.value and self.B.value))
```

And here is the code for the LATCH flip flop as a composite gate where 3 connectors connect the 2 NAND gates to form the circuit and provide the A and B inputs along with the output Q. This is followed by a test function which will let us drop either the A or B input temporarily to zero and then immediately raise it back to one.

```
class Latch (LC) :
  def __init__ (self, name) :
    LC.__init__ (self, name)
    self.A = Connector(self,'A',1)
    self.B = Connector(self,'B',1)
    self.Q = Connector(self,'Q',monitor=1)
    self.N1 = Nand ("N1")
    self.N2 = Nand ("N2")
    self.A.connect ([self.N1.A])
    self.B.connect ([self.N2.B])
    self.N1.C.connect ([self.N2.A, self.Q])
    self.N2.C.connect ([self.N1.B])

  def testLatch () :
    x = Latch("ff1")
    x.A.set(1); x.B.set(1)
    while 1 :
      ans = raw_input("Input A or B to drop:")
      if ans == "" : break
      if ans == 'A' : x.A.set(0); x.A.set(1)
      if ans == 'B' : x.B.set(0); x.B.set(1)
```

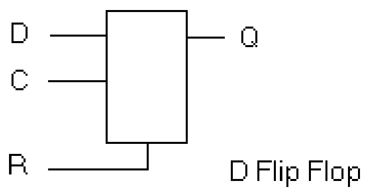
Let's run the test function.

```
>>> from logic2 import *
>>> testFlipFlop()
Connector ff1-Q set to 0
Connector ff1-Q set to 1
Input A or B to drop:A
Input A or B to drop:A
Input A or B to drop:B
Connector ff1-Q set to 0
Input A or B to drop:A
Connector ff1-Q set to 1
Input A or B to drop:
>>>
```

As we can see, Q starts out at zero and on its own shifts to one. This is the racing condition we talked earlier and results from the way we connected the NAND gates in the circuit. As A is dropped Q remains one. Not until B is dropped does Q drop as well. It stays zero until A is dropped again.

The "D" Flip Flop

The next flip flop (D) is somewhat more complex in its behaviour. It can also be made from simple gates (about 8 of them) but we're going to program it as a basic LC circuit with an evaluation method. This "D" flip flop has two inputs (C and D) and an output Q. The "D" input is "Data In" and the "C" input is the clock. As the clock falls from 1 to 0 "Q" is set to whatever value "D" has at that instant. And "Q" can't change this setting until "C" again falls from 1 to 0. For this reason this is called an edge triggered flip flop. This is a bit like a camera where D is the lens bringing in the image; Q is the film; and C is the shutter button. As you click the shutter the film is exposed but even if you continue to hold the shutter button down the shutter itself closes independently. To take the next picture you must release the shutter button in order to click it again. Here is the symbol for a "D" flip flop.



The extra input, "R", is a reset. If it is zero then Q is forced to zero. If "R" is one then the "D" flip flop works as stated above. We will mostly ignore the reset input in the circuit diagrams below to keep things less confusing.

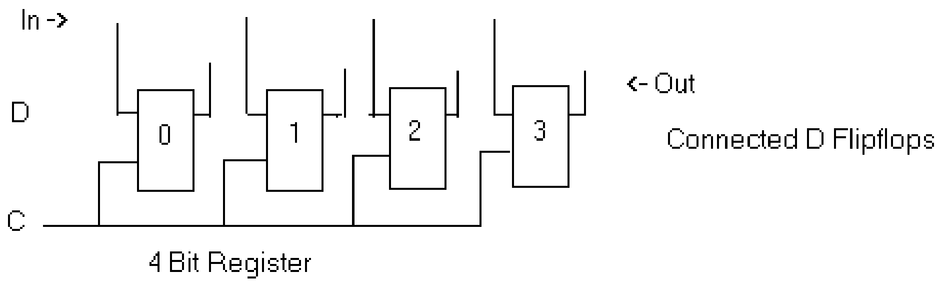
Here is the class to represent it as a basic LC gate (without the reset).

```
class DFlipFlop (LC) :
  def __init__ (self, name) :
    LC.__init__ (self, name)
    self.D = Connector(self,'D',1)
    self.C = Connector(self,'C',1)
    self.Q = Connector(self,'Q')
    self.Q.value=0
    self.prev=None

  def evaluate (self) :
    if self.C.value==0 and self.prev==1 :  # Clock drop
      self.Q.set(self.D.value)
      self.prev = self.C.value
```

The attribute 'self.prev' saves the previous value of the C input so that the evaluate function will set Q only on the transition from one to zero.

The Simple Data Register

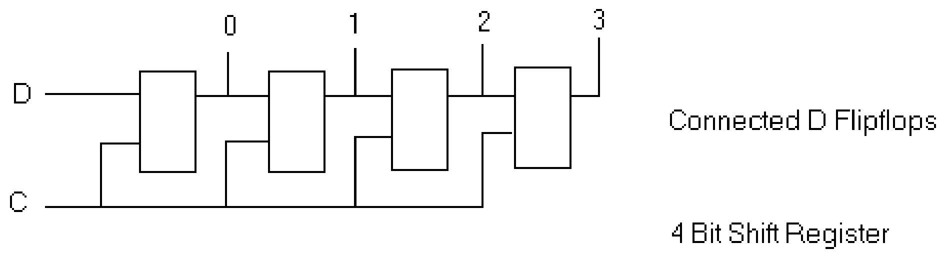
We can use D flip flops to build registers that hold binary numbers. Each flip flop remembers a single "bit" of the number and by clocking them together the binary number on the input lines is stored in the register whenever the clock line is dropped. Here is a diagram of a 4 bit data register.



In practice the reset inputs would also be connected together. This would allow the register to be set to zero independently of what is on the input lines. We'll use that feature later.

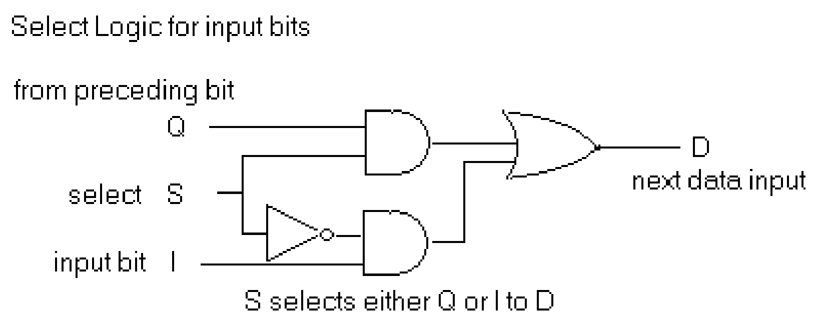
The Shift Register

If we change the connections so that each output becomes the data input for the the next bit, we'll have a shift register. Each drop of the clock line sets each bit to whatever was in the neighbor on the left. Here is where it becomes critical that the individual flip flops only respond to the edge of the clock drop. If they continued to load data while the clock was 0, the data would simply flush through and all of the bits would be set to the leftmost input.



We'll be using two shift registers below for multiplication. One shifts bits from low to high (multiply by 2), and the other will shift bits from high to low (divide by 2). Either is possible, of course. It's just however we connect the flipflop outputs to the next input; low to high or high to low.

It is also possible (and desirable) to be able to load a number from input lines into a shift register. This can be done with some selection logic between each bit that channels either the input bit I or the neighboring Q output into each D input.

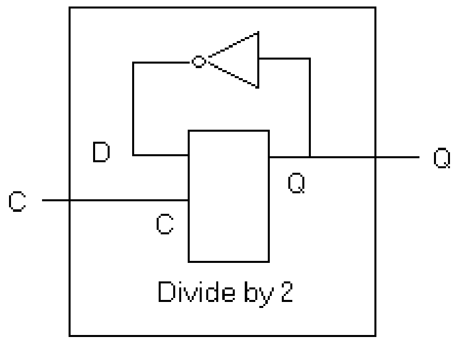


If S is 1 then the lower AND gate will block signal I. D will be the same value as Q. If S is 0 then the opposite happens and D will take whatever value I has.

This gating would be used between each pair of bits. The S lines are connected together to control whether a clock drop loads data from the input lines "I", or from the preceding bits output "Q".

The Divide by 2 Circuit

The following circuit feeds the Q output of its D flip flop back to its D input inverted. This creates a "Divide by 2" circuit. The D input is always opposite of Q so that each time the clock drops, Q changes, going either 0 to 1 or 1 to 0. Therefore Q itself drops at exactly half the rate of C.



Here is a class to simulate a divide by 2 circuit followed by a test function. It's built as a composite gate containing both a D flip flop and a NOT gate.

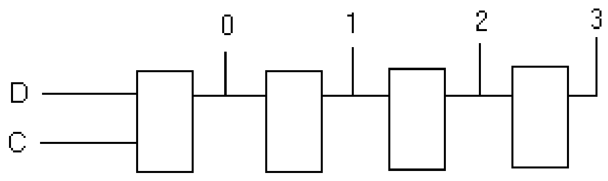
```
class Div2 (LC) :
  def __init__ (self, name) :
    LC.__init__ (self, name)
    self.C = Connector(self,'C',activates=1)
    self.D = Connector(self,'D')
    self.Q = Connector(self,'Q',monitor=1)
    self.Q.value=0
    self.DFF = DFlipFlop('DFF')
    self.NOT = Not('NOT')
    self.C.connect ([self.DFF.C])
    self.D.connect ([self.DFF.D])
    self.DFF.Q.connect ([self.NOT.A,self.Q])
    self.NOT.B.connect ([self.DFF.D])
    self.DFF.Q.activates = 1
    self.DFF.D.value = 1 - self.DFF.Q.value

  def testDivBy2 () :
    x = Div2("X")
    c = 0; x.C.set(c)
    while 1 :
      raw_input("Clock is %d. Hit return to toggle clock" % c)
      c = not c
      x.C.set(c)
```

Let's run the test function. Notice that Q drops to 0 with every second clock drop.

```
>>> testDivBy2()
Clock is 0. Hit return to toggle clock
Clock is 1. Hit return to toggle clock
Connector X-Q set to 1
Clock is 0. Hit return to toggle clock
Clock is 1. Hit return to toggle clock
Connector X-Q set to 0
Clock is 0. Hit return to toggle clock
Clock is 1. Hit return to toggle clock
Connector X-Q set to 1
```

The Counting Register

Multiple Divide-by-2's can be used to build binary counting registers. To get an idea of how this works, consider for a moment the tachometer on your car. With each mile (or kilometer) driven the units digit advances by one. Each time a digit drops from "9" to "0" the digit to its left advances by one. Of course binary numbers are simpler. If we had binary tachometers then each mile driven would flip the units bit and each drop from "1" to "0" would advance the bit to its left. We can achieve this by simply connecting Divide-by-2's together, output to input and we have our binary counter.



Connected Divide by 2's

4 Bit Counter

In practice we would want to add 2 additional features. We would use the same selection logic as we did in the shift register to enable a number to be loaded to the counter from input lines. And we would connect the reset lines together so that the counter could be easily reset to zero.

Here is a Python simulation of the above circuit without these extra features.

```
class Counter (LC) :
  def __init__ (self, name) :
    LC.__init__ (self, name)
    self.B0 = Div2('B0')
    self.B1 = Div2('B1')
    self.B2 = Div2('B2')
    self.B3 = Div2('B3')
    self.B0.Q.connect( self.B1.C )
    self.B1.Q.connect( self.B2.C )
    self.B2.Q.connect( self.B3.C )
```

And here is a test function. Each time through the loop the 4 output bits of the counter are printed and then the clock is toggled 1 to 0 and then back to 1.

```
def testCounter () :
  x = Counter("x")    # x is a four bit counter
  x.B0.C.set(1)      # set the clock line 1
  while 1 :
    print "Count is ", x.B3.Q.value, x.B2.Q.value,
    print            x.B1.Q.value, x.B0.Q.value,
    ans = raw_input("\nHit return to pulse the clock")
    x.B0.C.set(0)   # toggle the clock
    x.B0.C.set(1)
```

```
>>> testCounter()
Count is  0 0 0 0
Hit return to pulse the clock
Connector B0-Q set to 1
Count is  0 0 0 1
Hit return to pulse the clock
Connector B0-Q set to 0
Connector B1-Q set to 1
Count is  0 0 1 0
Hit return to pulse the clock
Connector B0-Q set to 1
Count is  0 0 1 1
Hit return to pulse the clock
Connector B0-Q set to 0
Connector B1-Q set to 0
Connector B2-Q set to 1
Count is  0 1 0 0
Hit return to pulse the clock
Connector B0-Q set to 1
Count is  0 1 0 1
```

Multiplication

We now have all the pieces to build a binary multiplier circuit. We won't actually do that, but we will look at how the major parts are connected and the logic required to make those parts work together.

First consider the following multiplication example.

```
233   (multiplicand)
*  123   (multiplier)
------
699   (3 times 233)
4660   (2 times 233)
23300   (1 times 233)
------
28659   (product)
```

This should hold no surprises. We all learned to do this in school. Each digit in the multiplier (123) is multiplied by the multiplicand (233), shifted appropriately and then the results are added together.

Now let's look at something simliar in binary. Here we will multiply 2 6-bit numbers whose decimal values are 38 and 36. Each one bit in a binary will represent the appropriate power of two.

```
100110     (multiplicand - decimal 38: 32+4+2)
* 100100     (multiplier   - decimal 36: 32+4)
--------
000000
000000
100110
000000
000000
100110
-------------
10101011000    (product - decimal 1368: 1024+256+64+16+8)
```

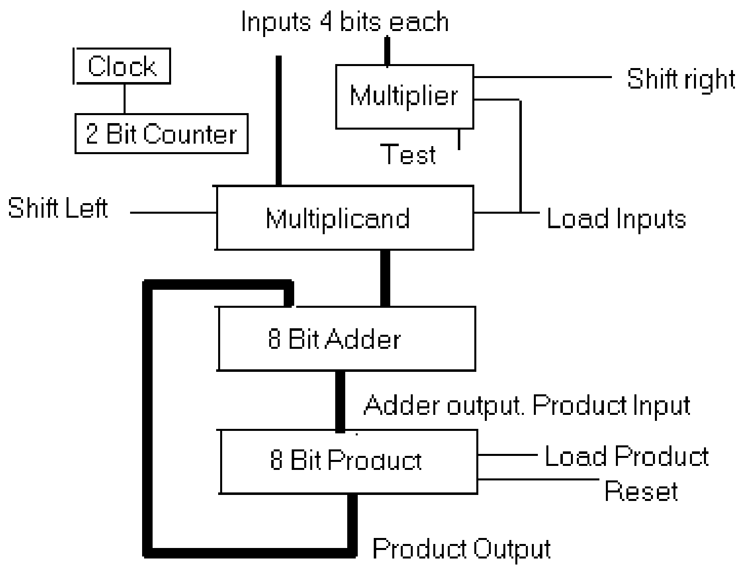
Now a number in binary (say 1001) has a decimal value of 1*1 + 0*2 + 0*4 + 1*8. Or we could say simply 8+1. In the multiplication above we have in decimal 38*36 which is 1368. Notice how simple the multiplication is however. For each bit in the multiplier the multiplicand is simply added to the product if that bit is one. Then the multiplicand is shifted to the left (multiplied by 2).

This can be duplicated with the circuits we have seen so far.

Now we would need 6 decimal digits to hold the product of 2 3-digit numbers. When multiplying two 4 bit numbers we need to have room for an 8 bit result.

Look at the following circuit diagram. We have two shift registers for the multiplier and multiplicand. In the 4 step procedure the multiplicand will be shifted to the left so that it is properly placed to be added to the product in the 8 bit adder. The multiplier is shifted to the right in order that the bit that we are testing is conveniently in the lowest bit position. The thick lines represent either 4 or 8 individual lines.

To start the multiplication the multiplier register and multiplicand are first loaded with their 4 bit values. The 8 bit product register and the 2 bit counter are reset to 0. Next a clock signal is generated and on each transistion several things take place with the help of some gating logic (which is not shown).



The clock signal will alternate between a 0 and 1 value on the order of one nanosecond. It needs to be slow enough so that signals can perculate through the adder completely. Light travels about a foot each nanosecond and when computers were built from discrete components this would have been close to the speed limit.

Here is what needs to happen in each of the four steps.

```
As the clock goes from 0 to 1 ...
if bit 0 of the multiplier is one the product register is
loaded with the sum of itself and the multiplicand

As the clock goes from 1 to 0 ...
the counter register is incremented.
the multiplier is shifted to the right
the multiplicand is shifted to the left
```

When the counter resets to zero stop and read product from the product register.

Now there are lot of details that we did not build circuitry for but you should be able to get enough of an idea that you could do this if you had to.

Negative Numbers

There is a scheme to represent negative numbers in binary that works as follows. Suppose you get into a brand new car and drive it in reverse for one mile. Assuming the odometer started at zero it should now read 999999. Now if you drive forward 6 miles the odometer will read 000005. So 999999 was a convenient representation for -1.

The same is done with binary numbers and the scheme is called "two's complement". 4 bit number which could take values from 0 to 15. (0000 to 1111) would instead take values -8 to -1 (1000 to 1111) and 0 to 7 (0000 to 0111).

This works automatically in the adder, since it will play the same trick that we saw with the odometer above. However the multiplier requires just a little help. Consider 2*-1 which is 0010 * 1111 in 4 bit two's complement. The 8 bit result should be 11111110, which is -2. What needs to be changed in the steps above to make this possible?

Taking it Further

Here are a few ideas for exercises.

Write a test function for the shift register.

Create a D flip flop from simple gates. Here a couple of hints. It is easy to make one using a LATCH circuit and some extra gating if it is NOT edge triggered. For edge triggering consider two in a row.

Work out, to some extent, actual control circuitry for the multiplier.

I would be happy share ideas. Email me at cmeyers@guardnet.com

Click Here for Full Python Source

Index

Simulator for a Mythical Machine

Click here for the Python code for the MM Simulator

What is Machine Language

Machine language is the basic language understood by the electronics of the computer's CPU (central processing unit). It is strictly numerical rather than having any sort of syntax you are used to with Python or perhaps other languages. Each type of CPU has its own unique machine language. The machine language for a Macintosh is very different from an Intel based PC.

Machine language is much simpler, in general, than high level languages that came later. Computers were first designed around just a few basic ideas. There needed to be a store of memory containing fixed sized numbers. There would be instructions to move numbers between memory and one or more special registers which could also hold numbers of the same size as the memory store. Other instructions would apply arithmetic operations to the numbers in the registers. Finally, special instructions could alter the flow of the program allowing repetition and conditional execution of instructions. What you do in Python when you use `while`, `for` and `if` statements.

Originally the numeric instructions were wired in with patch cords. Later someone had the bright (and simple) idea of using the memory store to hold both data and instructions. To make this work the numeric instructions had to be the same size as the numbers being stored in memory. Because almost all computers use the binary number system, this "word size" is expressed as a certain number of bits for each number. Practial computers have had word sizes from 12 to 64 bits.

In this study we will develop the Mythical Machine Language (MML) for, of course, the Mythical Machine (MM). Since we can't build a real machine we will create a small Python program to simulate its operation. MM is much simpler than any real computer but is still be capable of doing real computations. To make our task simpler MM is a decimal based computer. Each of its memory words can hold a 6 digit decimal number. The machine language instructions use discrete digits for the parts of the instructions. This will become clearer with some examples.

Design of MM

Our machine will have exactly 1000 words of memory, each with a address of 000 to 999. So a memory address requires exactly 3 decimal digits. Each word of memory holds a 6 digit decimal number and may be used for either data or program.

In addition our machine will have 10 general registers that also hold a 6 digit number. The registers generally hold temporary values being computed. Also there are 2 special registers. The "pReg" is the program counter. It contains the memory address of the next instruction to be executed. The "iReg" contains the instruction currently being executed.

Our simulator will let us load a program into memory and then run (execute) the program step by step. First the word of memory addressed by the pReg is copied (loaded) to the iReg. Next the instruction is carried out (executed). This process repeats until a "Halt" instruction is executed which will cause the simulator to exit. While the program is running the contents of the pReg, iReg and 10 general registers are updated and displayed.

Machine Language for MM

The following describes the instructions for MM. Each instruction will use 1 word of memory. Of the 6 digits in the word, 2 are reserved for the operation code (what the instruction will do), 1 digit will specify a general register, and the remaining 3 will specify either a memory address 000 to 999, an actual number or, depending on the instruction, a second general register.

Two digits for the operation code allows us to have 100 different instructions. We'll only need a dozen or so. These will let us move numbers to and from memory and registers, do arithmetic (add, subtract, multiply, and divide) on numbers in registers, and alter the flow of the program.

```
-------------- Instruction Set for MM -------------
000000     Halt
01rmmm     Load register r with contents of address mmm.
02rmmm     Store the contents of register r at address mmm.
03rnnn     Load register r with the number nnn.
04r00s     Load register r with the memory word addressed by register s.
05r00s     Add contents of register s to register r
06r00s     Sub contents of register s from register r
07r00s     Mul contents of register r by register s
08r00s     Div contents of register r by register s
100mmm     Jump to location mmm
11rmmm     Jump to location mmm if register r is zero
```

In the above diagram the letters "r" and "s" represent general registers, "mmm" represents arbitrary 3 digit addresses and "nnn" an arbitrary 3 digit number.

The first instruction with an opcode of 00 means the program is to halt. It doesn't matter what is in the remaining 4 digits but they are usually zero also. Executing a halt in the simulator causes it to exit but leaves the display on the screen so you can read the result of your computation in one of the general registers.

The next four instructions copy numbers between memory and the 10 general registers. The third digit specifies the register and the last 3 digits the source in memory. So 016234 means load register 6 with the number at memory address 234. The memory word itself is not changed. 023234 copies the number in register 3 to memory address 234. This also leaves the register 3 unchanged. Opcode 03 is a little different and has the name "load number". 035123 puts the number 123 into register 5. Opcode 04 uses another register as an index to memory. If register 2 contains the number 546 then the instruction 043002 loads whatever is in memory location 546 into register 3. This instruction will allow us to operate on a list (or array) of numbers.

The next four instructions operate just between the special registers. Instead of the low 3 digits specifying a memory location, they specify a second register. So 057008 adds the number in register 8 to register 7. Register 7 gets the sum and register 8 is unchanged. The same pattern is used for subract (06), multiply (07), and divide (08).

Finally, the last two instructions make it possible for our programs to do repetitive and conditional logic. The instruction 100452 puts the number 452 into the program counter (pReg). So whatever instruction is at 452 will be the next one fetched and executed. The instruction 113764 will set the number 764 into the pReg if and only if register 3 contains the number zero. These instructions are called jumps. The first is an unconditional "jump" and the second is called "jump if zero".

Our First Program

In this section we will write a little program that simply adds two numbers together.

To prepare data for the simulator we need to prepare a file that contains the address and its content for each memory location that we will use. Once the program is loaded the simulator sets the pReg to 100 and execution begins. There is nothing magic about 100. But generally programs do not start at location zero.

During each execution cycle an instruction is fetched, the pReg is advanced and the instruction is executed. If the instruction is a jump instruction the pReg may be changed. Otherwise it contains the address of the next word in memory. This cycle repeats until a Halt instruction is executed and the simulator stops.

Our program file may also contain any arbitary comments after the address and its data. The simulator will only look at the first 2 fields of each line. Here is the program to add the number 12 and 13.

```
100   031012   Load register one with the number 12
101   032013   Load register two with the number 13
102   051002   Add register two to register one.
103   000000   Halt. The answer is in register one.
```

Put the above 4 lines into a file called "prog1.mml" and run the simulator program with "python simulator.py prog1.mml". You will be prompted to hit the return key before each instruction fetch and before each instruction execution. When the program stops the screen should show

```
The Mythical Machine

P reg  000104       I reg 000000

reg 0  000000       reg 5 000000
reg 1  000025       reg 6 000000
reg 2  000013       reg 7 000000
reg 3  000000       reg 8 000000
reg 4  000000       reg 9 000000
```

showing that the last thing that happened was that the halt instruction was retrieved from address 103 and the registers 1 and 2 modified as per instructions. Notice that the pReg was advanced to 104. It gets advanced after each instruction.

The Python Simulator for MM

Let's spend a little time with the code in simulator.py. First of all this program updates the screen in place. To do that it uses some "escape sequences" to position the cursor on the screen and to erase the screen. These escape sequences go back quite a ways and early computer terminals connected to mainframe computers would respond just as your Linux or Windows CRT screen does. Incidentally for this to work in Windows you need to have the program ANSI.SYS running and be running Python from the DOS window. For Windows 95 or 98 this can be done with a command in CONFIG.SYS in the root directory. The line should read "Device=c:\windows\command\ansi.sys".

Escape sequences always begin with the "escape" character, which in Python is represented "\33". The string "\33[1;1H" moves the cursor to the top left corner of the screen. The string "\33[5;6H" moves the cursor to row 5, column 6. The other escape sequence we use is "\33[0J" which erases the screen from the cursor location. You can also use the "curses" module to do this kind of thing, but our requirements are so simple that it makes sense to do it directly. Anyway, this gives a hint on how the curses module itself does its magic.

For the most part the simulator program is straight forward. Function "main" calls "loadProgram" with the filename as passed on the command line. It then erases the entire screen and prints "panel", a "multiline string. With the program loaded the program counter "pReg" is set to 100 and the panel is updated. Then the function "cycle" is called for each instruction. "Cycle" pauses before each retrieve and execution of an instruction. If you hit an "a" (for "all") before the return the simulator will continue without pausing to the halt instruction. Function "cycle" extracts the opcode, register and address fields from the instruction and then in a if/elif block takes the appropriate action for the opcode. The screen is updated with the updatePanel function. UpdatePanel uses a convenient "loc" dictionary which stores the screen coordinates of each register in a tuple accessed by the register number.

A more complex program

Let's now look at a program that computes the sum of several numbers. We'll put the numbers (the data) in memory starting at location 200. We'll put the program instructions at location 100. The program will add numbers to the running sum until it encounters the number zero which is a sign to stop.

In this program we need to use the "load indirect" (04) instruction and use register 1 to point to the data as we add them to the sum in register 0. Here is the program with comments

```
100 030000   reg 0 holds the sum, set to zero
101 031200   put address (200) of 1st number in reg 1
102 032001   reg 2 holds the number one.
103 043001   next number (via reg 1) to reg 3
104 113108   if zero we're done so jump to halt inst
105 050003   otherwise add reg 3 to the sum in reg 0
106 051002   add one (in reg 2) to reg 1 so it points to next number
107 100103   jump back to 103 to get the next number
108 000000   all done so halt. the sum is in reg 0

200 000123   the numbers to add
201 000234
202 000345
203 000000   the end of the list
```

What's missing?

MM is really too simple to be of much use although we will use it to calculate factorials both in assember and a little high level language that we'll design for it. Several additional features would be found in any real comupter. Let's look at a few of these.

We can only store integers up to 999999 in the memory or registers. Real computers use floating point numbers, character data, bitmaps and so on. But its all (binary) numbers.

Each computer stores floating point numbers in its own way and has special machine instructions for doing arithmetic with such numbers, although early computers emulated this in software. You may already be familiar with how numbers represent characters in ascii character set and there are others as well. Character strings and lists, or arrays, are stored using sequential addresses in the memory very much like we did in the second program.

With the Jump instructions we can do loops and "if" kinds of logic. Along with the conditional "jump if zero" would be other kinds of tests as well such as "jump if negative". Also, to call a subroutine (function) requires both a jump and a way to return to the instruction after the "jump to subroutine". This was often awkward until machine code used stacks to keep track of return addresses. Stacks are simply sections of memory where a general register addresses the top element. Stacks make recursive functions possible even in machine language.

Finally, every computer needs some way to communicate with the outside world. We gave MM a panel showing its registers but nothing else. Real computers, even the earliest ones had keyboards, screens or paper prinouts, and so on. Typically the electronics are designed so that special addresses reference registers in these devices instead of main memory.

We'll continue to use MM in the next two sections on assembler language and the small compiler.

Index

Assembers and Assembly Language

Click here for the Python code for the MM Assembler In the last section we looked at a simulator program for the Mythical Machine. In this section and the next we'll see how software evolved to make programming computers much easier.

Problems with Machine Language

Programming in machine language is tedious for some obvious reasons. One is that you have to keep track of the numerical operation codes and not choose the wrong one. In coding the little programs of the last section we cheated a bit in that our format allowed us to put a comment after the address and assembled instruction which the simulator simply ignored. Without those comments the code would be extruciating to follow.

Another problem occurs if we are assembling a jump instruction. We only know the target address if the jump is back to a previous instruction. If the jump is forward to an instruction yet to come then we probably don't know what the address will be. So we have to come back later to complete the jump instruction.

But here is the worst problem. Suppose our initial code contains errors that require more (or fewer) instructions to fix. Then some jump instructions may have the wrong target addresses when the code is fixed and other instructions get shifted in memory. A major headache. One quick and dirty approach to this problem was apply what was called a binary patch. Where the code change occurred, we replaced whatever instruction was presently there with a jump to some available memory. At that address we put the replaced instruction, any new instructions and finally a jump back to the original spot. That saved other jumps around the patch from needing modification but code like this very quickly becomes messy and unmanagable.

Assembler Programs

The answer to these problems was to create "assembler" programs that let us represent the operation codes, registers, and addresses symbolically and let this program assemble the numeric instructions for us. When changes are made in the program, the assembler code is modified and the entire program re-assembled to machine code.

In assembler pieces of each instruction are represented in a way that is much more readable to humans. For example "add r1,r2" could mean add register 2 to register 1. The assembler would assemble the opcode (05) with the 2 register arguments to create the instruction 051002. Jump destinations and data addresses are determined by applying a label to an instruction or data point. This will be clearer with an example. Here is the assembly language version of our previous program to add a list of numbers together.

```
go    ld#  r0,0       register 0 will hold the sum, init it
      ld#  r1,nums    register 1 points to the numbers to add
      ld#  r2,1       register 2 holds the number one.
loop  ldi  r3,r1      get next number into register 3
      jz   r3,done    if its zero we're finished□
```

```
       add  r0,r3     otherwise add it to the sum
       add  r1,r2     add one to register one (next number to load)
       jmp  loop      go for the next one
done  hlt  00          all done. sum is in register 0
nums  123              the numbers to add
       234
       345
         0            end of the list
```

You may already see what is going on here. The symbols r1,r2 represent the general registers. The symbols "ld#", "ldi", "ldr", "jz", "jmp" and "hlt" are operation codes we are using. Finally the symbols go", "loop", "done" and "nums" are labels arbitrarily chosen to represent memory addresses. We don't know what those memory addresses will be and we don't really care. The assembler program will figure that out for us.

Each line has the information for the assembler to build a single machine instruction. One, two, or three fields may be followed by an optional comment. In the first line the label "go" will make the symbol "go" the same as the address of this instruction. A label starts at the beginning of the line with no white space preceding it. The second field is the operation code "ld#" (load number) and the third field "r0,0" provides the information to complete the instruction. Some instructions require both a register and address argument, "jmp" requires only an address, and "hlt" needs no argument at all. Finally, the operation code may be replaced by a simple number which lets us put data into the program.

An Assembler for MM in Python

If you haven't already get a printout of the program code for the MM Assembler or put it into another window.

The assembler works in two passes. The first pass determines what address values need to be assigned for each of our labels. These are stored in the dictionary "lookup" along with the register definitions.

The second pass uses the opcode field and argument field to build instructions, substituting lables in the argument field with their numeric addresses.

Assemblers usually take an input file with contents like the above and produce two output files. One is the machine language in a form that the computer can load for execution. Generally it is combined with others that are "linked" to form a complete executable program. The other output is a listing file that would look like the following.

```
100 030000   go    ld#  r0,0      reg 0 will hold the sum
101 031107         ld#  r1,nums   reg 1 points to next num
102 042001   loop  ldi  r2,r1     get next number into reg 2
103 112106         jz   r2,done   if its zero we're done
104 050002         add  r0,r2     else add it to the sum
105 100102         jmp  loop      go for the next one
106 000000   done  hlt            all done. sum is in reg 0
107 000123   nums  123              the numbers to add
108 000234         234
109 000345         345
110 000000           0           zero marks the end
```

Function "main" reads the entire program from standard input to a list of lines and then passes the list to functions "pass1" and "pass2".

The function "pass1" looks at each line of the program. Any label in the instruction is equated to the current value of the program counter in the dictionary "lookup". Then if a valid opcode is found the program counter is advanced since there will be an instruction generated from this line in pass 2.

The function "pass2" looks at each line again and this time assembles the instructions since all labels have had their values set in the "lookup" dictionary. The operation code for each instruction is either an opcode in "codes" or a simple number. Anything else raises an error and will show as "***" on the output.

It is ok to have a label alone on a line. This makes it possible to have several symbols equated to the same address. We will use this feature in the next section when we develop a tiny compiler for MM.

Other Considerations

There are a couple of things that should be pointed out. Our assembler creates code that is then loaded directly into our computer (or simulator) in an address space determined by the assembler. In real life final addressing is actually determined later by another program called the "linker". The machine code is also designed to facilitate this. Instead of actual addresses in the instructions, it's more likely that an offset from the current instruction would be used. So that "jmp loop" would not use the address 102, but rather -3. This complicates other things because now we would need two different instructions for "ld# r0,0" and "ld# r0,nums" since in the first instance we really want the number zero but in the second case we want whatever the final address of "nums" will be.

However there is a tremendous advantage to code like this. It may be loaded anywhere in memory, and the linker program may link together many separate modules to create a single executable. On modern systems it is even possible to load modules at runtime and link them together. In fact, this is exactly what you do whenver you import a module into your Python program that was written in the C language and compiled to machine code.

Index

Simple Compiler for MM

Click here for the Python code for the MM Compiler

In this section we will build a small compiler that will translate expressions and statements in a python like mini-language into assembly language for our MM computer.

Motivation for Compilers

The assembler certainly makes it easier to write machine language programs but it would be still nicer to be able to program with expressions like "a=(a+b)*c" instead of, say

```
ld  r0,a
ld  r1,b
add r0,r1
ld  r1,c
mul r0,r1
sto r0,a
```

So just as assemblers can assemble machine instructions from their symbolic parts, compilers are programs to compile sets of assembler instructions for more complex expressions. Some of the earliest compilers were for the language Fortran which stands for Formula Translation. With Fortran a programmer could write code much more concisely and easily than with assembler. The compiled machine code was generally not quite as good as what could be produced by hand in assembler but it was often good enough, even when machines were absolutely miniscule by today's standards. Furthermore, knowing Fortran (or other languages like Cobol or Algol) meant that a programmer could write code for any computer that had the appropriate compiler, not just for the computer for which one knew the assembler code. Today the most common compiled language that produces machine code for personal computers is probably C.

Interpreters

From the start there was also a parallel effort to design and build interpreted languages. In general these languages do not try to produce machine code directly but rather interpret instructions that can do much more advanced operations than what the basic hardware supports. Early interpreted languages include Basic, APL (A Programming Language), and Lisp. Interpreted programs run much slower than compiled programs but in general are often easier to write and maintain. Python is an interpreted language and for programs that easily translate from Python to C, the C version will often run 30 to 50 times faster. But if we are doing the kinds of things Python does well (our MM compiler is a good example) the speed difference is less pronounced. We often are quite willing to give up a little speed to gain some very powerful operations that in turn result in consise code that is easy to understand.

The MH Compiler

Our language, which we'll call MH, is designed to be as small as possible but able to have programs like the factorial program from the previous section. We'll have only interger variables and, like Python, variables do not have to be declared. Variables and numbers may be combined in expressions with operators (+,-,*, and /) and anywhere a variable or number may appear a subexpression in parentheses may be used instead.

In order to keep the compiler very small we'll support just two kinds of basic statements; assignment and the "while" statement. In addition there is a compound statement, a series of other statements inside a pair of curly braces "{\}".

So lots of stuff (most) is left out. But there is enough to write the simple factorial program which will use most of the features just mentioned. Near the end of this section will be some suggstions for extending the language.

Let's look at an example program in MH. Here is code for computing the factorial of 5.

```
term = 5
ans  = 1
while term { ans=ans*term  term=term-1 }
ans
```

As you can see, the program has some assignment statements and a while statement. There is also a expression at the end whose purpose will be explained later. The 2 statements enclosed in curly braces form a single compound statement that forms the body of the while statement. The test expression in the while statement "term" will cause the statements in the body of the while to be repeated until the value of term is zero. This, of course, will cause the variable "ans" to be multiplied by 5, 4, 3, 2, and finally 1.

Notice that unlike Python, statements do not need to be on separate lines or indented. This syntax is closer to the C language, with the exception that in C statements are required to be terminated with a semicolon.

The Compilation Process

The compilation process consists of parsing the code in the input language (MH) and then generating code in the target language (MM assembly). The parsing phase itself consists of two parts; identifying the tokens of the language such as variables, numbers, keywords and special symbols, and then determining how these tokens relate to one another to form expressions and statements. Sometimes compilers will build tree structure is to store these relationships but our little compiler does not go to such lengths. As the various pieces of input code come together output code is generated somewhat on the fly. The reason we can do this is that the grammer of our language is simple enough that the program can be scanned and the structure determined by simply knowing what has come so far and what the next token is to the right.

A Simple Compiler in Python

This is probably a good time to look over the compiler.py and either keep it in a separate window or make a printout to follow the remainder of this discussion.

The function getToken extracts the next token from the program and returns it along with the rest of the program. It extracts keyword or variable names, numbers and special symbols. Lets try it out on some input.

```
>>> import compiler
>>> compiler.getToken("while a b=a")
['while', ' a b=a']
>>>
```

So you can see that a call to getToken removed the keyword "while" and returned it and the rest of the program in a list. Let's look at the code in getToken briefly. The string.strip function first removes whitespace including spaces, tabs and newlines. Then the next character is examined. If it is a letter then a word is gathered as in the example above. If it is a digit, a string representing an integer is returned. Finally a single character, perhaps an "+", "=", or "{" is returned. In any case what is returned is the next item of information that forms the structure of the program.

If MH were extended we would want getToken smart enough to return pairs of special characters like "**" or "<=" as single tokens. You might consider how you would do this. We would also need a way to ignore comments in the program.

Let's look at the nature of possible statments in MH. The simplest is an expression all by itself. An expression in MH is a series of "terms" separated by an operator ('+','*',etc). "Terms" are numbers, variables, or subexpressions enclosed in parentheses. Expressions and terms are recursive structures and our functions "getExpr" and "getTerm" call each other recursively as well.

This expression statement, which is also possible in Python, is a little boring because it computes a value and then promptly throws it away. An assignment statement however sets a variable to the value computed. In MH only a simple variable is allowed on the lefthand side of the assignment operator "=".

A compound statment is a series of statements bracketed by "{" and "}". The inner statements may be assignments, while statements and other compound statements.

The while statment takes the form "while <expression> <statement>". Here "<statement>" is apt to be a compound statment and is repeated as long as the "<expression>" evaluates to a nonzero value.

Some Sample Compilations

It is useful to play with the compiler before examining the code in detail. This will allow us to see what sort of assembly code is produced. The compiler accepts a mini-program on the command line (or standard input) and outputs the resultant assembly code to standard output.

Let's look at what must be the simplest possible program. A single statement which in turn is a single number, "3".

```
$ python compiler.py "3"
  ld# r0,3
  hlt
```

The generated code simply loads the number 3 into register 0. In general the compiler will generate code to get the result of any expression into a chosen register, starting with 0 and working up as it needs more registers. This, by the way, is why our MM architecture includes 10 general registers. Let's look at a slightly more complex expression, "a*3". Note that it is necessary to quote the expression to keep the shell from trying to evaluate special characters like "*".

```
$ python compiler.py "a*3"
  ld  r0,a
  ld# r1,3
  mul r0,r1
  hlt
```

Here the subexpression "a" is "calculated" in register 0 and the subexpression "3" in register 1 (the next register). Then the multiply instruction leaves the result in register 0. Not shown here is a memory word allocated to the variable "a". Next, let's try a nested expression

```
$ python compiler.py "(b+3)*a"
  ld  r0,b
  ld# r1,3
  add r0,r1
  ld  r1,a
  mul r0,r1
```

This expression requires r0 and r1 to compute "b+3" and then reuses register 1 to hold "a" before the multiply.

Let's extend this just a bit by compiling an assignment statement. Now the output looks like this.

```
$ python compiler.py "c=(b+3)*a"
  ld  r0,b
  ld# r1,3
  add r0,r1
  ld  r1,a
  mul r0,r1
  sto r0,c
  hlt
c 0
```

The calculation is followed by "sto r0,c" which stores the result of the computation into the variable "c". Also a line was added to allocate a word of memory for "c". As we'll see, these allocations all take place at the end of our program so they are out of the way of the code. The compiler will remember to allocate a word of memory for any variable that gets something assigned to it. That should include all variables that we use.

Finally let's look at the output of a very simple while loop. It's not meant to be a real one since the conditional term does not change. But it will illustrate the code that is generated.

```
python compiler.py "while a {b=b*c a=a-1}"
z1
  ld  r0,a
  jz  r0,z2

  ld  r1,b
  ld  r2,c
  mul r1,r2
  sto r1,b

  ld  r1,a
  ld# r2,1
  sub r1,r2
  sto r1,a
  jmp  z1
z2
  hlt
a 0
b 0
```

The while loop uses the "jump on zero" machine instruction to jump out of the loop and a simple jump to repeat the loop. It also generates labels "z1" and "z2" for the destinations of the jump instructions. Later while statements would generate labels like "z3", "z4", etc.

I've put in some blank lines to separate the code generated by the different expressions. You should be able to recognize which parts belong with what.

A Closer Look at the Code

We've already looked at "getToken". Let's look at the other functions from the top down.

Function "main" reads an entire program from either standard input or (as we've seen) from the command line. It repeatably calls "getStat" to peel one statement at a time from the program and print the resulting assembly code. As we'll see the dictionary vars contains an entry for each variable found and "main" reserves a word of memory for each.

Like "getToken" the 3 other functions "getStat", "getTerm" and "getExpr" return two values; the assembly code generated and the remainder of the program still to be compiled.

Function "getStat" looks at the first token. If it is "while" it gets an expression for the condition test, putting the assembly code into string "code1", and another statement (calling itself recursively) for the body (in "code2"). It then makes two unique lables in "l1" and "l2" and combines all to produce the code which is returned along with the rest of the program. Other sections of "getStat" handle the assignment statement, compound statments and a single expression. Part of compiling an assignment statement is to add an entry to the "vars" dictionary for each variable on the left of the "=".

By this point the remaining two functions, "getExpr" and "getTerm" should be fairly straightforward. The only tricky bit is the mutual recursion between them. Expressions consist of terms which may be variables, numbers or subexpressions in parentheses separated by operators. If you are confused run the test above with some "print" statements inserted.

Compiling and Assembling the Factorial Program

Now let's compile the factorial program at the top of this section and produce MM code that can be run in the simulator. We'll start by placing the following MH code into a file called "fact.mh"

```
term = 5
ans  = 1
while term { ans=ans*term  term=term-1 }
ans
```

On Unix it is possible to run the compiler and assembler together since both are using standard input and standard output. In fact many compilers take advantage of this. Here is our factorial program ready to load into the simulator.

```
$ python compiler.py < fact.mh | python assembler.py
```

```
100 030005     ld# r0,5
101 020117     sto r0,term
102 030001     ld# r0,1
103 020118     sto r0,ans
           z1
104 010117     ld  r0,term
105 110115     jz  r0,z2
106 011118     ld  r1,ans
107 012117     ld  r2,term
108 071002     mul r1,r2
109 021118     sto r1,ans
110 011117     ld  r1,term
111 032001     ld# r2,1
112 061002     sub r1,r2
113 021117     sto r1,term
114 100104     jmp  z1
           z2
115 010118     ld  r0,ans
116 000000     hlt
117 000000   term 0
118 000000   ans 0
```

The last statement of the program is simply the expression "ans". As you can see this loads it into general register zero where it can be observed in the simulator. The following creates a machine language file and runs it in the simulator.

```
python compiler.py < fact.mh | python assembler.py >fact.mm
python simulator.py fact.mm
```

Further considerations

It is interesting to compare the assembly language output from the compiler with the assembly language program for computing factorials that we built in the previous section. The hand built version is about half the size of the compiler output. For a long time people continued to program in assembler just for this advantage. The space advantage is also a runtime advantage. Smaller programs run faster with fewer instructions to execute. Later, compilers were made to optimize their output and that closed the gap somewhat.

Our compiler treats all operators that it supports with the same precedence. You can see this by compiling "a*b+c" and "a*(b+c)". They produce the same code. To fix this "getExpr" must be broken into separate functions for each precedence level. Sums can be products separated by "+" and "-" operators. Products are terms separated by "*" and "/". There are several other levels, exponentiation and logical operations, even array access.

Defining functions and calling them with arguments demands operations that work with stacks. Stacks may also be used in lieu of multiple registers for computing nested expressions and then combining them later. Compilers are sometimes defined as being either stack oriented or register oriented. In fact the way our compiler uses successive registers is very similar to using a stack.

Our compiler does not support function calls or definitions. Typically when a function is called its arguments are evaluated and pushed onto a stack. The function is then invoked saving the return address as well, either on the same stack or on another. When the function returns the arguments are popped from the stack and the function return value is pushed.

Python, Java and some other languages perform what they call compilation, but instead of compiling to machine code they compile to an intermediate code that is then interpreted very much like our simulator program works. However the basic operations in this code are much more powerful than our machine instructions. Here strings can be added, dictionaries accessed and so on. In python compilation happens automatically. When you import a module Python will try to save the compiled code in a file with the "pyc" extension. The Python interpreter then takes over to run the code. With Java the interpreters are available in web browsers which load compiled Java classes and run them. These interpreters go by the name of "Java Virtual Machine". There is also a Python compiler called Jython written in Java that translates Python source code into Java classes.

Index

SQL Introduction for Python Programmers

This tutorial on SQL is meant to demonstrate the small amount of know-how you need to write effective database programs. By looking at any SQL reference book, it is obvious that there is vastly more to the subject than what is presented here. But in several years of programming with SQL and Python I've found that about 98% of what I do is in the following pages.

We will be using PostgreSQL and Python together. At work I use Sybase for the database server but for most readers open source PostgreSQL is much more affordable and still very powerful. Another open source option that you can explore is MySQL.

Servers and Clients

Relational database systems are client/server systems. In PostgreSQL the server is called the postmaster and does all the actual interaction with the data itself. Clients are programs that make requests to the server to either read or modify the database. This approach allows several clients to simultaneously access the data with the server doing all the necessary synchronization.

There are two types of client we'll look at. One is the program psql that comes with the PostgreSQL system and lets us examine and modify data easily interactively, similar to the Python interactive mode. The other type of client is a Python program you write that imports the `pg` module to do the same kind of interactions that psql does but retrieve the result in Python data structures.

Nature of the data

In thinking about databases it is useful to consider some analogies, although they should not be taken too literally.

The server may access many databases but a client will operate on only one at a time. Think of a database as a directory in your file system. Tables in the database are something like files in the directory. Tables consist of rows and columns and you may think of them as spreadsheets. Each column has a name and can hold a particular type of data such as an integer, a floating point number, or a string. Each relational database system has its own set of datatypes. PostgreSQL has about 40. But the datatypes you'll use over and over again are numbers (ints and floats), strings, and dates.

It is important to realize at the start what columns cannot hold. You can't put a Python list or dictionary into a column, as lovely as that would be. Later we will see how we can join tables together to achieve this kind of versatility with our data.

Tables can also have indexes which maintain the data in sorted order and serve 3 main purposes. One is to make accessing data much faster. If the client requests a certain row from a table by a data value in one or more or its columns, the server will not have to read (perhaps) the entire table to find the row in question if the table is indexed by that set of columns. Instead the row is found by bouncing around a much smaller (and sorted) index. Secondly, indexes let us processes a set of rows in the sorted order. Finally, indexes are necessary if we want to guarentee that each row has a unique value in a column (or set of columns). We'll go into indexes in more detail later.

Using psql with postmaster

First make sure the server (postmaster) is running. If you are on a multiuser system a system administrator may be responsible for this and has also created a database for your use, or perhaps to share with others.

I'm using Linux on a standalone system so I will start the postmaster (from the postgres login) in local mode (just my machine) and create a test database. We'll use a dedicated window to run postmaster. The "$" is the shell prompt

```
$ postmaster -i localhost
```

Messages from postmaster will appear the in this window.

Next, logged in another window as user `postgres`, I'll create a fresh database called `school` to play with. Then I'll activate the psql client attaching it to my database `school`. We're now ready to add and manipulate tables in `school`. The program `psql` issues the prompt `school=#`, showing which database it has open.

```
$ createdb school
$ psql school
school=#
```

Operations on Tables

Tables are created by giving them a name and a set of columns. Each column in turn needs a name and a datatype. Here is a simple example.

```
school=# create table course (number int, name varchar(24), credits int);
```

Here we have created a simple table with information about courses. We are using a column for the course number, another for its name, and one for the number of credits. The name is defined as `varchar` (variable length character string) which is like a Python string except that this one has a maximum length of 24. The course number and credits are stored as integers.

The opposite operation of `create table` is `drop table`. You may think of `drop` as `delete` or, if you are unix fan, `rm`.

```
school=# drop table course;
```

Once a table is created, clients do mostly four things with them. Insert a new row, delete a row, update column values in a new row and finally select rows (read).

Inserting rows into a table

Inserting rows can be done in 2 ways. The simplest is to list the column values in the same order as the columns were defined when created. For example

```
school=# insert into course values (3, 'Computer Programming', 1);
```

Another way is to first list the column names followed by values.

```
school=# insert into student (id,name) values (411, 'Bayartsogt, Gombo');
```

This method is handy if some columns have default values or can take a null value (think Python None). These columns and their values may be omitted from the insert statment. You can check your reference book on how to expand the `create table` statement to specify defaults with columns.

Deleting rows from a table

Deleting rows from a table is as simple as identifying them. For example.

```
school=# delete from course where number=120
```

The clause `where number=120` specifies a single row, at least with what we have in the table so far. But if there were two rows in the table with this number, both would be deleted. In fact, the `where` clause may be omitted in which case all rows in the table are deleted. So `delete from course` empties the course table.

Actually, `where` clauses are a lot like boolean expressions in Python, i.e. what you find after keywords like `if` and `while`. But there are some differences you'll want to remember. In SQL you use a single `=` to test for equality instead of `==`. And where you would find variable names in Python SQL requires column names. But and's and or's are allowed and you're free to use parantheses as you would expect. One thing to watch out for in postgreSQL is that you must use single quotes for strings. Double quotes are used for another purpose.

One common practice that is quite different is pattern matching with string data. Exact matching is simply `name='Algebra II'` but the phrase `name like 'Algebra%'` would match any string starting with "Algebra". The `%` character is the wildcard character for SQL's `like` operator. With Python these kind of matches generally require the `re` (regular expression) module.

`Where` clauses are probably the most complex and powerful part of SQL. They are used with `update` and `select` statements just as they are with `delete`. We'll see several examples as we go along.

Updating rows in a table

Updating columns in a table is straighforward. A `where` clause identifies the row (or rows) to update. The `set` clause identifies the columns and their new values.

```
school=# update student set name='George' where id=411;
```

You can modify a single column, several columns or even all columns depending on your `where` clause (or lack of one). You may also update several columns in a single update statement. Just separate `name=value`(s) with commas.

Selecting rows from a table

The `select` statement is the most interesting and the one you will use most often, probably 90% of the time. Here is a simple example which select all columns (*) from all rows (no where clause) from the table course.

```
school=# select * from course;
 number |         name         | credits
--------+----------------------+---------
    101 | Algebra I            |       5
    201 | Algebra II           |       5
    150 | World History        |       2
    301 | Calculus             |       5
    314 | Computer Programming |       4
    204 | Spanish II           |       3
(6 rows)
```

Adding a `where` clause lets us control how much we get back

```
school=# select * from course where name like 'Algebra%';
 number |    name    | credits
--------+------------+---------
    101 | Algebra I  |       5
    201 | Algebra II |       5
(2 rows)
```

The `*` above indicates all columns. If only some are wanted you specify them by name.

```
school=# select number,credits from course where credits > 4;
 number | credits
--------+---------
    101 |       5
    201 |       5
    301 |       5
(3 rows)
```

Beyond the basics

Now that you've seen the four basic operations on tables it's time to extract from multiple tables. Relational databases are so named because tables are related to each other by common values. To illustrate this we need to create a couple more tables; one for students and one for teachers. To make this easier we put the necessary sql into two files, teachers.sql and students.sql. Take a look at these. They're pretty bare-boned with each student and teacher having simply a name and an id number. Id numbers are necessary to distinguish duplicate names. In practice the id numbers for people are often social security numbers. Id numbers are also great for joining tables together as we'll see.

We can load these files with psql redirecting stdin to the files. The contents of the files simply replace what you would type.

```
$ psql school < teachers.sql
$ psql school < students.sql
$ psql school
school=# select * from teacher;
  id  |     name
------+---------------
 1001 | Elkner, Jeff
 1002 | Meyers, Chris
 1003 | Downey, Allen
(3 rows)

school=# select * from student;
 id  |   name
-----+-----------
 411 | Bayartsogt, Gombo
 412 | McMahon, John
 413 | Kern, Owen
 414 | Cohen, Jonah
(4 rows)
```

Let's talk about relationships between rows in different tables. They fall into 3 catagories; one to one, one to many, and many to many.

An example of a one-to-one relationship might be a parking space assigned to a teacher. One parking space; one teacher. The easiest way to make this relationship in our database is to make the parking space number a column in the teacher table. If, perhaps, more information about parking spaces were required, we could have a separate table for parking spaces. The parking space number in both tables would tie the teacher to the extra information in the parking table.

An example of a one-to-many relationship might be teachers to courses. We'll assume each course is taught by a single teacher but that each teacher may teach several courses. In a Python program we might have a object class of teacher with an attribute of courses, a list containing the course numbers. But remember that we can't store lists in a database. What we do instead is have a column in the course table that contains a single teacher id. We'll see in a bit how this solves the problem.

Finally an example of a many-to-many relationship might be students and courses. Each course has many students and students take many courses. Representing this kind of relationship requires an extra table that ties students and courses together. We'll call this table "enrolled" and it will have a row for each combination of student and course.

```
school=# create table enrolled (studentId int, courseId int);
```

If we have 3 students each taking 4 courses, our table would have 12 rows (3*4).

Working with multiple tables

Let's drop the tables and reload them from these files; teachers.sql, courses.sql, students.sql and enrolled.sql.

```
$ psql school < teachers.sql
$ psql school < students.sql
$ psql school < courses.sql
$ psql school < enrolled.sql
```

Now things get a little more interesting. We can use the relations between the tables to make more complex queries.

Let's list all courses taught by teacher 1001

```
$ psql school
school=# select name,teacherId from course where teacherId=1001;
        name         | teacherid
---------------------+-----------
 World History       |      1001
 Computer Programming |      1001
(2 rows)
```

But suppose we want to list the teacher's name instead of their id. To do this we need to `join` the course and teacher tables. Our first attempt will be

```
school=# select course.name, teacher.name from course,teacher;
```

Because both tables have a column called `name` the columns need to be qualified by their table names; thus, course.name and teacher.name. There is still a problem, however. If you try the above query you'll get 18 rows; 3 teachers times 6 courses. That's what happens when tables are joined. To trim our answers to what we want we need to add a simple `where` clause.

```
school=# select course.name, teacher.name from course,teacher
school-#   where teacher.id=teacherId;
        name         |     name
---------------------+---------------
 World History       | Elkner, Jeff
 Computer Programming | Elkner, Jeff
 Algebra II          | Meyers, Chris
 Calculus            | Meyers, Chris
 Algebra I           | Downey, Allen
 Spanish II          | Downey, Allen
```

There was no need to qualify the column name `teacherId` since it is unambiguous

We can also sort the output with an `order` clause. `Order` clauses always come after `where` clauses. For example

```
school=# select course.name, teacher.name from course,teacher
school-#   where teacher.id=teacherId
school-#   order by course.name;
        name         |     name
---------------------+---------------
 Algebra I           | Downey, Allen
 Algebra II          | Meyers, Chris
 Calculus            | Meyers, Chris
 Computer Programming | Elkner, Jeff
 Spanish II          | Downey, Allen
 World History       | Elkner, Jeff
(6 rows)
```

Finally, we'll join 3 tables to show students enrolled in each class. We'll order first by course name and then by student name.

```
school-#select  course.name,student.name from enrolled,student
school-#  where course.number=enrolled.courseNumber
school-#  and   student.id   =enrolled.studentId
school-#  order by course.name, student.name;

        name         |   name
---------------------+-----------
 Algebra I           | McMahon, John
 Algebra II          | Bayartsogt, Gombo
 Calculus            | McMahon, John
 Computer Programming | Bayartsogt, Gombo
 Computer Programming | McMahon, John
 Computer Programming | Kern, Owen
 Computer Programming | Cohen, Jonah
 Spanish II          | Kern, Owen
 Spanish II          | Cohen, Jonah
 World History       | Bayartsogt, Gombo
(10 rows)
```

The Python connection

Importing the module `pg` into a Python program gives us the same access to the postmaster server that we have with psql.

Let's play with the pg module in the interactive mode. The first thing we must do in our program is to establish a connection to a database. Here is an example. (`>>>` is Python's prompt)

```
>>> import pg
>>> conn = pg.connect(dbname="school", host="localhost", user="postgres")
```

Remember that the postmaster server is running on our local machine with the `-i localhost` switch. If your situation is different then you would specify the actual computer in the `host` parameter.

With a database connection we can perform querys just like with psql. Let's do one.

```
>>> result = conn.query("select * from course")
>>> print result
number|name                |credits|teacherid
------+--------------------+-------+---------
   101|Algebra I           |      5|     1003
   201|Algebra II          |      5|     1002
   150|World History       |      2|     1001
   301|Calculus            |      5|     1002
   314|Computer Programming|      4|     1001
   204|Spanish II          |      3|     1003
(6 rows)
```

Now, this is nice but we will want to manipulate the results of the query, not just print them in this somewhat hokey table format. Happily, `result` is an object with attributes and methods that let us access the data as Python values. In fact its `__str__` method produces the output above. One of the most useful methods is `dictresult` which returns a list of dictionaries. Each dictionary represents a row.

```
>>> print result.dictresult()
[{'number': 101, 'name': 'Algebra I', 'credits': 5, 'teacherid': 1003},
{'number': 201, 'name': 'Algebra II', 'credits': 5, 'teacherid': 1002},
{'number': 150, 'name': 'World History', 'credits': 2, 'teacherid': 1001},
{'number': 301, 'name': 'Calculus', 'credits': 5, 'teacherid': 1002},
{'number': 314, 'name': 'Computer Programming','credits': 4, 'teacherid': 1001},
{'number': 204, 'name': 'Spanish II', 'credits': 3, 'teacherid': 1003}]
>>>
```

I massaged the line endings a bit to make it more clear. Each key/value pair in each dictionary represents a column name and value. In the small programs to come we'll use this method for making a report.

Now lets consider joining two tables.

```
>>> cmd = """select course.name,teacher.name from course,teacher
...          where teacher.id=course.teacherId order by course.name"""
>>> result = conn.query(cmd)
>>> print result
name               |name
-------------------+-------------
Algebra I          |Downey, Allen
Algebra II         |Meyers, Chris
Calculus           |Meyers, Chris
Computer Programming|Elkner, Jeff
Spanish II         |Downey, Allen
World History      |Elkner, Jeff
(6 rows)

>>> print result.dictresult()
[{'name': 'Downey, Allen'}, {'name': 'Meyers, Chris'},
{'name': 'Meyers, Chris'}, {'name': 'Elkner, Jeff'},
{'name': 'Downey, Allen'}, {'name': 'Elkner, Jeff'}]
>>>
```

Oops! Notice that our dictionaries only have the teacher names and not the course names. This is because both tables use the same column name `name`. To get around this we need to apply an alias to one of the column names. In this case we choose to give teacher.name the alias `tname`.

```
>>> cmd = """select course.name,teacher.name as tname from course,teacher
... where teacher.id=course.teacherId order by course.name"""
>>> result = conn.query(cmd)
>>> print result.dictresult()
[{'name': 'Algebra I', 'tname': 'Downey, Allen'},
{'name': 'Algebra II', 'tname': 'Meyers, Chris'},
{'name': 'Calculus', 'tname': 'Meyers, Chris'},
{'name': 'Computer Programming', 'tname': 'Elkner, Jeff'},
{'name': 'Spanish II', 'tname': 'Downey, Allen'},
{'name': 'World History', 'tname': 'Elkner, Jeff'}]
>>>
```

There is another way besides dictresult() to get at the data. Two methods are used; one returns a tuple of the field names and the other a list of tuples with the column values.

```
>>> print result.listfields()
('name', 'tname')
>>> print result.getresult()
[('Algebra I', 'Downey, Allen'),
('Algebra II', 'Meyers, Chris'),
('Calculus', 'Meyers, Chris'),
('Computer Programming', 'Elkner, Jeff'),
('Spanish II', 'Downey, Allen'),
('World History', 'Elkner, Jeff')]
>>>
```

This method probably runs somewhat faster, but programs using dictionaries are more readable (IMHO) since the column names appear in the expressions.

A little report program

Now we are going to write two versions of a tiny report program that lists teachers alphabetically and under each one another alphabetical list of the courses they teach. These programs will demonstrate two different ways to accomplish the same goal. A bit later we'll discuss when one method might be more efficient than the other.

```
#!/usr/bin/env python
#
#      Simple report of classes taught by each teacher
#
import pg

db = pg.connect(dbname="school", host="localhost", user="postgres")

def main () :
    result = db.query("select * from teacher order by name")
    trows = result.dictresult()
    for trow in trows :
        tId = trow['id']
        print "%s" % trow['name']
        query ="select * from course where teacherId=%d order by name"
        result = db.query(query % tId)
        crows = result.dictresult()
        for crow in crows :
            print "  %s" % crow['name']  # Name of course

    if __name__ == "__main__" : main()
```

In the program above we first select rows from the teacher table. Then for each teacher we select the course taught. The output looks like this

```
Downey, Allen
Algebra I
Spanish II
Elkner, Jeff
Computer Programming
World History
Meyers, Chris
Algebra II
Calculus
```

The second program below takes a different approach. A single select joins the teacher and course table together. A single `for` loop processes the results. A little extra logic is needed when the teacher changes. But the result is the same.

```
#!/usr/bin/env python
#
#    Simple report of classes taught by each teacher
#
import pg

cmd = """
select teacher.name as tname, course.name as cname
from teacher, course
where teacher.id = course.teacherId
order by teacher.name, course.name"""

db = pg.connect(dbname="school", host="localhost", user="postgres")

def main () :
    curTeacher = None
    result = db.query(cmd)
    rows = result.dictresult()
    for row in rows :
        if row['tname'] != curTeacher :
            curTeacher = row['tname']
            print "%s" % curTeacher
        print "  %s" % row['cname']  # Name of course

if __name__ == "__main__" : main()
```

So when is one method better than the other? In an example this small there is virtually no difference. However if we had 500 teachers the first method would require 501 querys; the second still only one. The second is apt to perform much faster. However, if queries get too complex, generally by joining too many large tables together, the time required for the server to process them can unexpectably explode. The reasons for this are not terribly clear. But the solution is to use the first method or a combination of the two where you balance minimizing the number of queries and their complexity. It sometimes takes some experimentation to find the best answer in each situation.

The query method will actually take any SQL statement. So you can use it to update and delete rows as well as insert rows. When used for these purposes no result object is returned.

```
>>> db.query("update teacher set name='Gardner, Martin' where id=401")
```

Using tables with indexes

Indexes make searching for rows in a table very fast. You, the user, simply tell the server to create an index for a given column (or combination of columns) and the server takes care of updating the index as new rows are inserted, deleted, or updated. You never have to tell the server to use an index. It figures that out from your query.

To create an index you can issue a simple command from psql. It is uncommon to create indexes from Python since it is generally a one-time activity.

```
school=# create index inxTeacherName on teacher (name);
```

Indexes are given a name (here inxTeacherName), a table to index and one or more columns to build the index from. To remove an index you drop it.

```
school=# drop index inxTeacherName;
```

By using the keyword `unique` the server will prohibit the insertion of a row where the index key is already present. The following pair of indexes on the enrolled table make searching both ways (students by course, or courses by student) very fast and also guarentee that no student is enrolled in the same course twice.

```
school=# create unique index inxEnrolled1 on enrolled (studentId,courseNum);
school=# create unique index inxEnrolled2 on enrolled (courseNum,studentId);
```

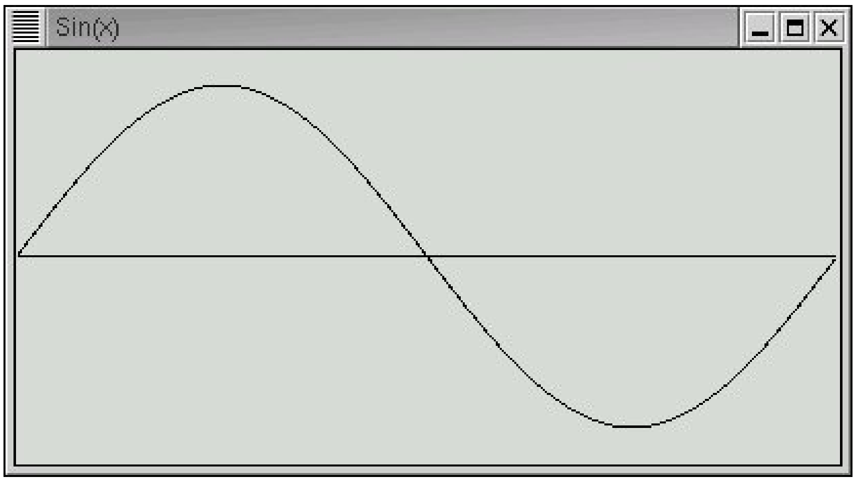
One caveat with using indexes. They can get fragmented with lots of insertions, in a manner similiar to disk fragmentation when you add lots of new files. The solution for this is to periodically drop and re-create the index. You'll know when it's time when searches start getting slower and slower.

Waves and Harmonics

The book "Who is Fourier, a Mathematical Adventure" from the Transnational College of LEX is an excellent and gentle introduction to a wide range of subjects including differential and integral calculus, Fourier Series, and even an analysis of the five vowel sounds used in the Japanese language. This study augments the book by preparing an object class for representing waveforms graphically and also manipulating them. If you can't get a copy of the book but have some background in trig and maybe just a bit of calculus, the material should be easy.

Sound waves

When a tuning fork is struck a sound of a single fixed frequency is emitted as the fork vibrates. If we use a microphone to convert the sound wave to an electrical signal and then display the signal on an oscilloscope the waveform will look the following sine wave.



The horizontal axis represents time and the vertical axis amplitude. An amplitude may represent several things such as the exact position of the tuning fork arm as it vibrates or the compression and decompression of air as the sound travels through it or perhaps the instantaneous voltage of the electical signal generated by the microphone. Each seperate point is also a point in time. The sine wave describes how any of these properties change during a single cycle. If the frequency is 440 cycles per second (440 Hertz) then the time shown above is 1/440 of a second. Only two attributes are needed to describe a sine wave; its frequency and its amplitude.

If you now strike the A above middle C on a (well tuned) piano, you will also get a wave of 440 cycles per second. However it will not sound quite the same as a tuning fork. And it will look quite different on the oscilloscope. And if you play the same note on a violin or trumpet they will sound different yet again.

The reason for the difference in sound is that these more complex instruments produce a sine wave not only at 440 hertz but also at higher frequencies (harmonics) where each frequency is an integer multiple of 440 hertz. To picture this think of a violin string vibrating so that the end points are fixed and the middle of the string vibrates back and forth. That would be the fundamental frequency of, say, 440 hertz. Next think of the center of the string also staying stationary and the points at 1/4 and 3/4 of the string length vibrating back and forth. That vibration would be at 880 hertz because the length of a vibrating string (or part of a string) determines the frequency and the relationship is a linear one. That means is you half the length of the string you will double the frequency. You can divide the string into any number of sections and get a vibration at the corresponding frequency. These vibrations are called standing waves.

Each instrument produces its unique pattern of harmonics and their relative strengths determine the characteristic sound of the instrument.

Now suppose we would like to play music from our computer. One way to do this is to capture a sound wave electrically with a microphone, convert instantaneous voltage values at some fixed rate and store the values for later retrieval. This is exactly how a music CD works. But it takes a lot of storage, as you already may know. About 10 megabytes for each minute of sound.

But a much more compact method exists if the sounds are relatively simple such as notes from trumpet or violin (or even japanese vowels). All we need to know is the base frequency and amplitude of a note, its duration, and the relative amplitudes of each harmonic. Of course, there are an infinite number of harmonics but ones that are beyond the range of human hearing (about 20,000 Hertz) may be ignored since you wouldn't hear them anyway.

The main point of this study is to demonstrate Fourier's method for finding the harmonic amplitudes from a sampled wave.

A Python Class for Waves

Let's start by creating a Python class to store a single cycle of a complex waveform (like a violin note) in the same way a CD stores the information. The data might be gathered by sampling a microphone. One attribute of our class will be an array, or list, of sampled points. With our wave class we will be able to add, multiply, divide, and subtract waves in order to add in (and subtract out) harmonic frequencies. The class will also have a method to plot the wave on the screen.

We'll use just a bit of the Tkinter package to plot the waves. Normally the learning curve for Tkinter is fairly steep but we will be able to accomplish our needs with just a few commands.

Let's look at the definition of the wave class. To see the code in full Click here.

```
class wave :
  def __init__ (self, points=400, formula=None) :
    self.data = [0.0]*points
    self.points= points
    if formula :
      for p in range(points) :
        x = p*pi*2/points
        self.data[p] = eval(formula)
```

When a wave object is instantiated we pass the number of data points that are to be stored for the range 0 to 2*pi and also an optional formula for calculating the data points. The formula is a string (like "sin(x)") that is passed to the Python eval function which calculates the value of each data point. The formula should contain a single variable "x" which will be iterated over the range. Let's look at an example. We'll create a sine wave with 400 data points.

```
>>> import wave
>>> w = wave.wave(formula="sin(x)",points=400)
>>> print w.data[0],w.data[100],w.data[200],w.data[300]
0.0 1.0 1.22460635382e-16 -1.0
>>>
```

Since there are 400 data points, 100 corresponds to pi/2, 200 to pi, and 300 to 3*pi/2. 1.22460635382e-16 is *very* close to zero, in fact we could also write it as .000000000000000122460635382. The reason it's not exactly zero is due to rounding errors in the floating point calculations.

Next we define the methods to add and multiply waves. By using Pythons magic method __add__ we're able to use the '+' operator directly on two wave objects creating a third objects whose data points are the sum of the corresponding data points in the input waves.

```
def __add__ (self, other) :
  target = wave(points=self.points)
  for i in range(self.points) :
    target.data[i] = self.data[i] + other.data[i]
  return target
```

As you can see, "target" is a new wave object created to hold the sum. It is made with the same number of data points as the source.

The subtract method __sub__ is identical to __add__ with the exception that the '-' operator replaces the '+' operator.

Mutliplication is slightly more complicated because there are two ways it can be done. If a wave is multiplied by a number then each data point is multipled by that number resulting in a wave with the same shape but with a different amplitude. If two waves are multiplied together then the corresponding data points are multiplied together resulting in a quite different wave. We'll use both forms of multiplication.

```
def __mul__ (self, other) :
  target = wave(points=self.points)
  if type(other) == type(5) or type(other) == type(5.0) :
    for i in range(self.points) :
      target.data[i] = self.data[i] * other
  else :
    for i in range(self.points) :
      target.data[i] = self.data[i] * other.data[i]
  return target
```

The final operation that we need for waves is to integrate them. For those of you without a calculus background, this means finding the area between a curve and the zero line. Where the function values are positive the area is positive and where they are negative, the area is also negative. Integral calculus has rules to compute such areas, but we will do something much simpler. We'll let the computer simply compute the area by finding the average function value in the range 0 to 2*pi and simply multiplying that value by 2*pi.

That is just what the following method does.

```
def integral(self) :
  ans = 0.0
  for pt in self.data : ans = ans+pt
  return ans*2*pi/self.points
```

This will introduce some small errors since our waves are sampled at a finite number of data points. The curves are not completely smooth, but sort of "staircasey".

Just Enough Tkinter

Finally, we need to be able to plot our wave to the screen. There are actually several ways to go about this but we'll use the Tkinter package which is fairly straightforward for operations this simple. To use Tkinter, it must be installed on your computer and if you are using Linux the you must be using the X window system.

We will use just a few simple commands to plot our waves with Tkinter. If you can, follow along from the Python interactive prompt.

```
>>> from Tkinter import *
>>> win = Tk()
```

If all is well, a small window should have popped up on your screen that is framed and 200 pixels in both width and height. The variable "win" is a reference to this window. (Windows users, don't panic. The window will appear a little later.)

The next two commands create a "canvas" tied to the window onto which we may draw lines (and other things as well). Since the canvas is dimensioned 400 pixels wide and high, the window expands when the canvas is "packed" into it.

```
>>> canvas = Canvas(win,height=400,width=400)
>>> canvas.pack()
```

Finally we create a line from the origin (0,0) in the upper left corner to the center of the canvas (and the window) at (200,200).

```
>>> canvas.create_line(0,0,200,200)
```

Now, if you are using Windows you will need one more command before seeing the window on the screen. The window must be "run" with the following

```
>>> win.mainloop()
```

This will unfortunately keep your program (or the interactive mode) from proceeding until the window is closed. That means that, unlike the Linux (and other Unix) users, you won't be able to stack several windows on the screen.

That's really all we need to know to build our plot method. Let's look at it now.

```
def plot (self, title="??", pixHeight=None, maxY=None, others=[]) :
  if not pixHeight : pixHeight = self.points*2/3   # Pleasant ratio
  pixWidth = self.points
  # find max and min data to scale
  if not maxY :
    maxY = max (max(self.data), -min(self.data))
    offset = pixHeight/2
    scale = offset/maxY
```

This first part determines the scaling factor for the vertical "y" axis. If maxY is supplied the window scales pixHeight to it, actually to twice it's value, half above and half below the zero line. If maxY is not specified, it is determined by the maximum positive (or minimum negative) value in the data points.

Next a window and canvas are created and the zero line is drawn.

```
...
  win = Tk()
  win.title (title)
  canvas = Canvas(win,width=pixWidth,height=pixHeight)
  # create zero line
  canvas.create_line(0,offset,pixWidth,offset)
  canvas.pack()
```

Next "plotOne" is called to plot the wave. The parameters "others" may be a list of other waves to plot as well with the same scale factor.

```
...
  self.plotOne (canvas, pixWidth, scale, offset)
  for i in range(len(others)) :
    others[i].plotOne (canvas, pixWidth, scale, offset)
    if sys.platform == "win32" : win.mainloop()
```

Finally the method "plotOne" draws lines between each of the data points scaled to pixels. Notice that the first line drawn is for the second data point when x==1.

```
def plotOne (self, canvas, pixWidth, scale, offset) :
  for x in range(pixWidth) :
    y = offset - self.data[x] * scale
    if x : canvas.create_line(x-1,yprev,x,y)
    yprev = y
```

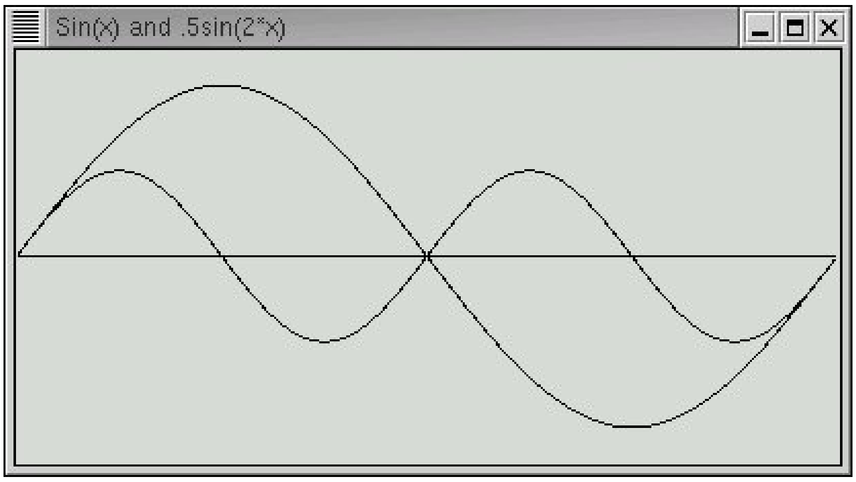
Complex Waveforms

Let's start making waves and see what they look like. We'll generate a simple sine wave and its first harmonic (twice the frequency) at half the amplitude.

```
>>> import wave
>>> a = wave.wave(formula="sin(x)")       # fundamental
>>> b = wave.wave(formula=".5*sin(2*x)")   # harmonic
```

Next we'll plot both of them onto the same window.

```
>>> a.plot(maxY=1.2, pixHeight=200, title="Sin(x) and .5sin(2*x)", others=[b])
```



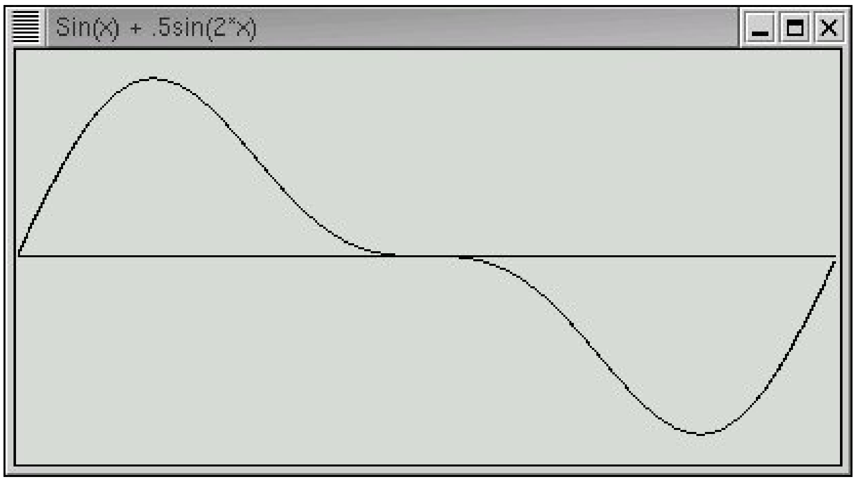
Next let's compute and plot the sum of the two simple waves. Notice that since both "a" and "b" are wave objects, their sum, computed by the "__add__" method above is a seperate wave with its own data points.

```
>>> c = a + b
>>> c.plot(maxY=1.5, pixHeight=200, title="Sin(x) + .5sin(2*x)")
```

Or more consisely.

```
>>> (a+b).plot(maxY=1.5, pixHeight=200, title="Sin(x) + .5sin(2*x)")
```

And the result is.



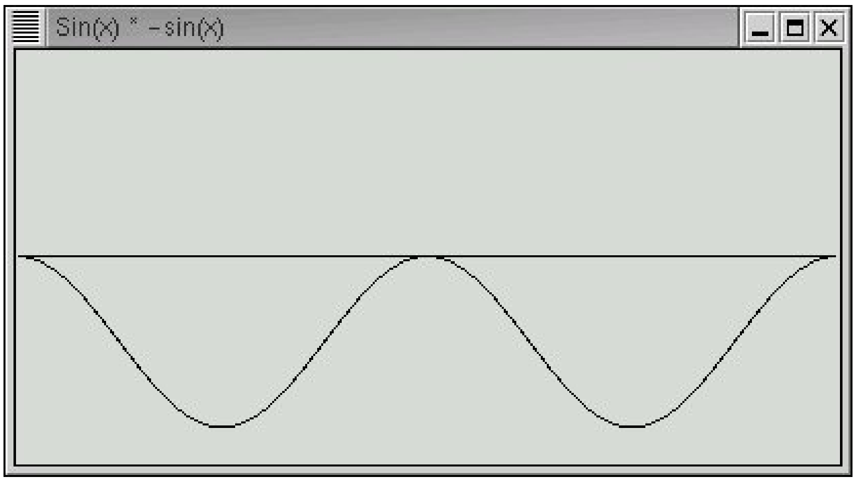
Now it's pretty simple to generate a complex waveform from simple waveforms. Much trickier is to extract the simple waves from the complex. But that is exactly what we would need to do in order to see the pattern of harmonic frequencies that distinguish a violin from a piano or, going back to japanese vowels, the "ah" sound from the "ee" sound.

Fourier provided an ingeneous method for this. To understand it we need to consider the integrals of our simple waves.

```
>>> print a.integral(), b.integral(), c.integral()
-1.143366892e-16 -1.63003354337e-16 -3.61866356668e-17
```

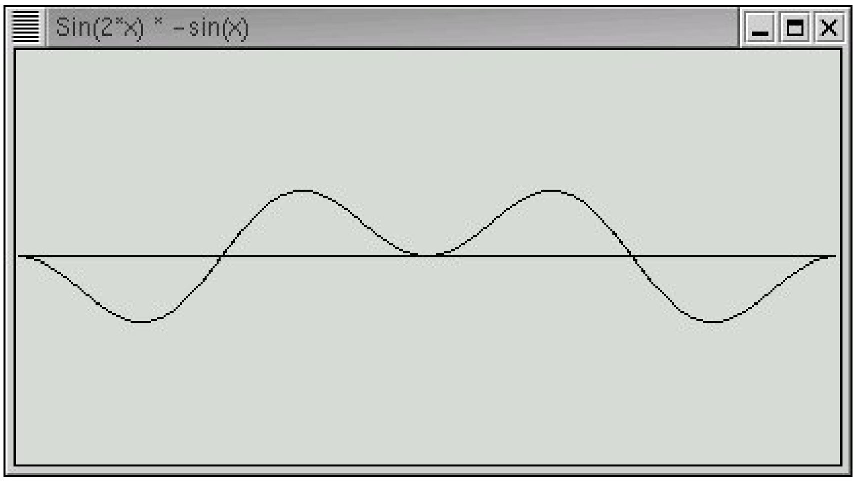
The integrals are basically zero. But we must expect that since in each wave there is just as much area below the center line as above. The waves are symetrical around the zero line. But supposing we multiply both "a" and "b" by "-sin(x)"

```
>>> mx = wave.wave(formula="-sin(x)")
>>> c = a*mx
>>> c.plot(maxY=1.2, pixHeight=200, title="Sin(x) * -sin(x)")
>>> print c.integral()
-3.14159265359
```



Notice that for our fundamental frequency this multiplication takes the curve below the centerline and yields an area of -pi. Let's try the same thing for our first harmonic.

```
>>> mx = wave.wave(formula="-sin(2*x)")
>>> c = b*mx
>>> c.plot(maxY=1.2, pixHeight=200, title="Sin(2*x) * -sin(x)")
>>> print c.integral()
-1.74286132071e-16          (basically zero)
```



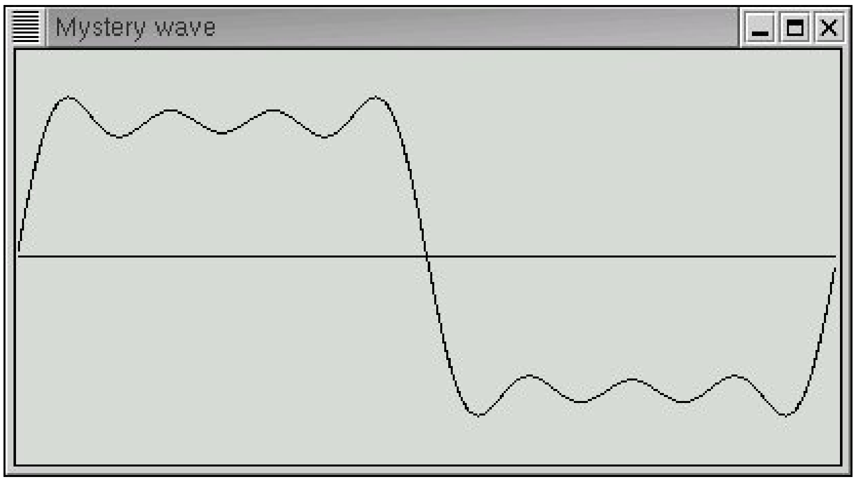
We can see that there are still equal areas above and below the center line. You can try this on other harmonics and see that this always holds true. Only the fundamental frequency will yield a non-zero area of -pi times its amplitude.

This leads to the following technique to "remove" first the fundamental frequency followed by the successive harmonics. We multiply the complex wave by "-sin(n*x)" where n is 1,2,3... and then compute the integral of the result. The integral divided by -pi is the amplitude of the nth harmonic. We then form a wave with that amplitude and frequency and subtract it from the complex wave leaving the remaining harmonics. When all of the harmonics have been extracted our complex wave will be reduced to a flat line of zero amplitude. But we will have the necessary information to reconstruct the complex wave anytime we want.

The recombination of simple waves to complex is how music synthesizers work and is also the basis for MIDI music format.

Fourier Analysis of a Mystery Wave

Let's apply the above technique to the following wave form



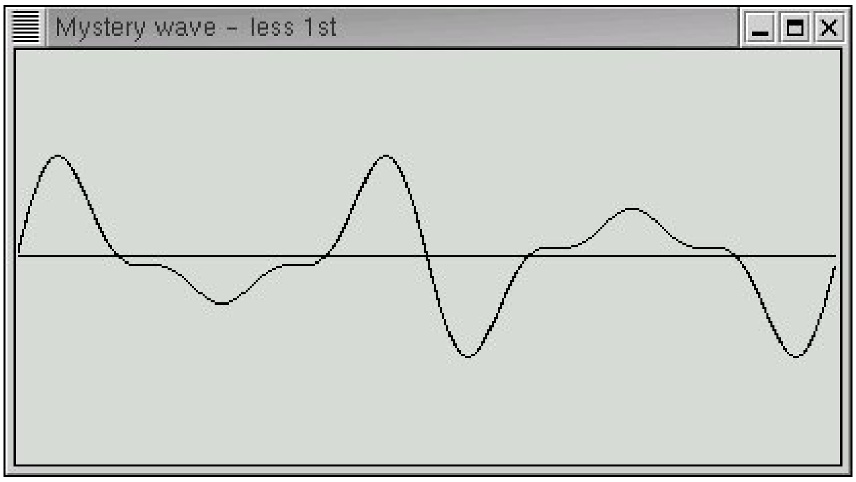
which we have referenced by the variable "mystery". How we got this wave is not important. It might be captured from a microphone to a wave file. Let's start with the fundamental frequency and integrate its product with "-sin(x)"

```
>>> d = mystery
>>> (d*wave.wave(formula="-sin(x)")).integral()
-3.1415926535897931
>>>
```

The result is -pi which means the amplitude of the fundamental frequency must be one. Let's now subtract that from "d" leaving only the harmonics.

```
>>> d = d - wave.wave(formula="sin(x)")
>>> d.plot(maxY=1.2, pixHeight=200, title="Mystery wave - without fundamental")
>>>
```

The result looks like this

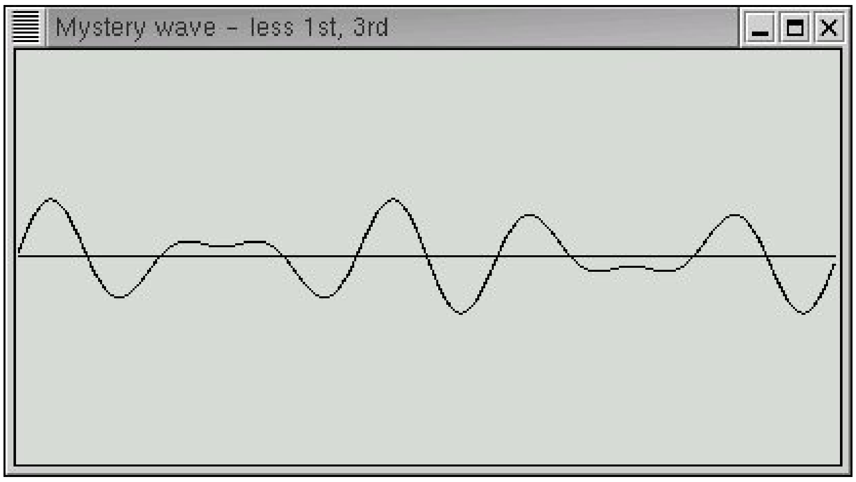


That certainly looks simpler. Let's now try the harmonic at twice the fundamental frequency.

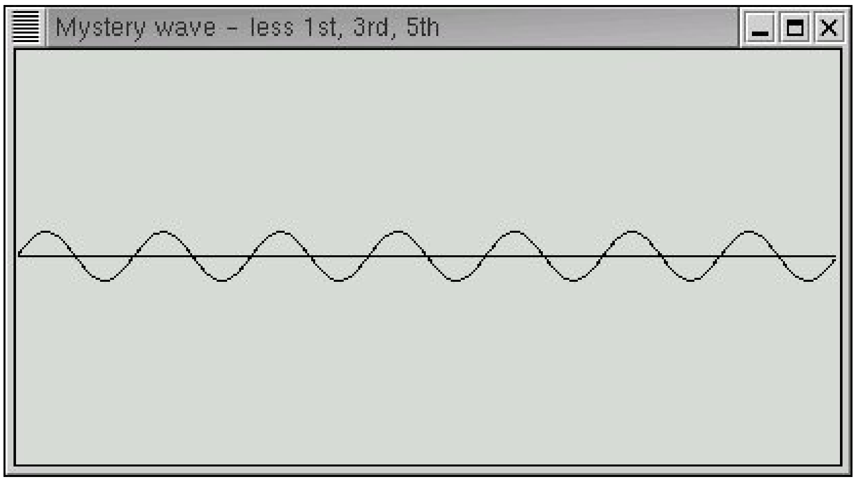
```
>>> (d*wave.wave(formula="-sin(2*x)")).integral()
-7.9727087825085978e-17
>>>
```

With e-17, this is basically zero. Let's keep going, subtracting out harmonics as we find them.

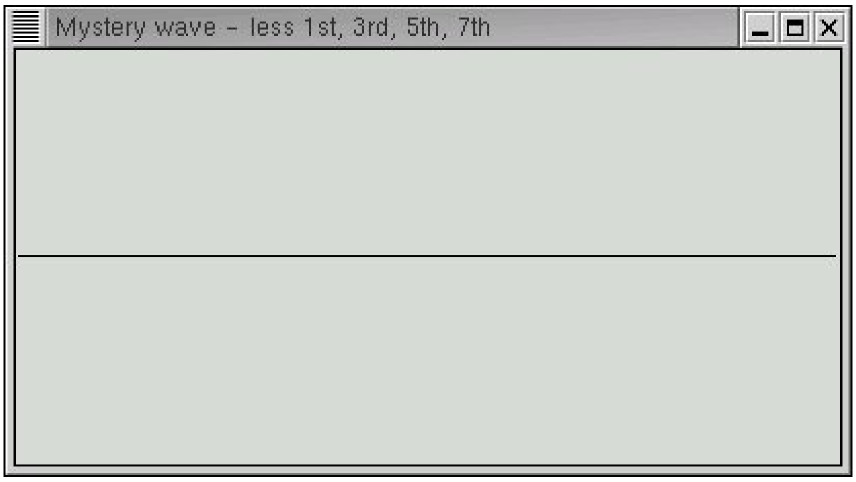
```
>>> (d*wave.wave(formula="-sin(3*x)")).integral()
-1.0471975511965983
>>> d = d - wave.wave(formula="-1.0471975511965983*sin(3*x)/-pi")
>>> d.plot(maxY=1)
```



```
>>> (d*wave.wave(formula="-sin(4*x)")).integral()
-2.5293858853722756e-17
>>> (d*wave.wave(formula="-sin(5*x)")).integral()
-0.62831853071795829
>>> d = d - wave.wave(formula="-0.62831853071795829*sin(5*x)/-pi")
>>> d.plot(maxY=1)
```



```
>>> (d*wave.wave(formula="-sin(6*x)")).integral()
-4.8554263124133835e-17
>>> (d*wave.wave(formula="-sin(7*x)")).integral()
-0.44879895051282731
>>> d = d - wave.wave(formula="-0.44879895051282731*sin(7*x)/-pi")
>>> d.plot(maxY=1)
```



As the final plot shows, all harmonics are basically gone.

Our mystery wave was the begining of a square wave with just the first few harmonics. The square has an interesting set of harmonics. Only the odd harmonics are present. Their amplitudes also decrease in an orderly way.

```
sin(x) + sin(3*x)/3 + sin(5*x)/5 + sin(7*x)/7 + sin(9*x)/9 + ...
```

If you check the non-zero integrals above you'll find that they are in fact -pi/3, -pi/5, and -pi/7 except for some very small error due to the approximations discussed above.

This is encapsulated in the wave.fft method at the bottom of the wave class. A test function demonstrates its use

```
def test() :
    p1 = wave(formula="sin(x)/1")
    p3 = wave(formula="sin(3*x)/3")
    p5 = wave(formula="sin(5*x)/5")
    mys = p1+p3+p5
    mys.fft()

if __name__ == "__main__" : test()
```

```
$ python wave.py
Harmonic= 1  Amplitude=1.0000
Harmonic= 3  Amplitude=0.3333
Harmonic= 5  Amplitude=0.2000
```

Where from here?

Here are a few things you might try.

Write a Python function to automate the steps above until all datapoints are below some small noise factor. Your function takes a wave object as input and returns a list of the coeffients.

The formula "sin(x)-sin(2*x)/2+sin(3*x)/3-sin(4*x)/4+ ..." creates triangular sawtooth waves. Write a function to create a sawtooth wave which takes the number of terms as an input parameter and returns the wave. Plot the wave.