IDL Tutorial 2:
Writing an IDL Script

To start IDL, type:
> sswidl
at the Linux prompt. This should start a program and lots of stuff should print to the screen, then you should see an idl prompt, like this:
IDL>

To complete the tutorial below, read the text below and enter the commands that are given by the IDL prompt. Sometimes, you will be asked to enter something at the LINUX prompt (">"), instead of the command prompt. To do this, open another terminal.

A note about IDL programs and functions. IDL has two types of programs, or scripts: functions and procedures. Throughout this first tutorial, you were using several IDL procedures and functions. Examples of procedures are "help", "print", and "plot". Procedures have the command name first, followed by a list of arguments and keywords separated by commas. Examples of functions include "findgen" and "rd_tfile". Functions have a variable first, then an equal sign, then the function name with the arguments and keywords inside parenthesis.

For this tutorial all IDL scripts will be written in the working directory (usually /home/yourlastname). The name of the file needs to be the same as the name of the function or procedure, meaning if you want to make a procedure called "make_plot", the name of the file you write should be "make_plot.pro".

IDL is a very easy computer language to learn to program because each command that you can enter into the command line can also be used in a program. All the commands you learned in the first tutorial can (and will) be used in your programs. To write an IDL program, open a new window to get a Linux prompt. Make sure you are in the same directory as your IDL session. (To determine what directory you are in for your IDL session, go back to the IDL window and type:
IDL> $pwd
Note that the "$" is needed because you are using a Linux command.) After you have verified you are in the home directory, go to the Linux prompt and type
> emacs first.pro &
(or you can use your own favorite editor.) Note the ".pro" file extension which indicates an IDL program. The first line of the program should be
PRO first
where the "PRO" tells IDL this is, in fact, a procedure while "first" is the program's name. It should always match the name of the file. The last line of the program should be
END
which lets IDL know the program is over. In between the first line and the last line, you can enter any other commands. For instance, enter
x=findgen(50)

y=x^2
plot,x,y
between the first and last line and save the file. Go back to your IDL window and type
IDL> .r first
which compiles the program. You can now run the program by simply:
IDL> first
Did it make the plot? You are now an IDL programmer!

Now let's make a slightly more complicated program. Go back to your leafpad window and replace the 2 in the line:
y=x^2
with an "n", so it now reads:
y=x^n
Save the program, compile and run it.
IDL> .r first
IDL> first
You should have gotten an error saying that the variable "n" was undefined. It also should give you the line the error was on.

Here is a good time to learn a very important IDL command called retall (stands for return all). When you run a program with an error, IDL will stop in the middle of the program (wherever the error is.) To get IDL back to the main level type:
IDL> retall
Believe me, you will use retall a lot!

Go back and add
n=3
before this line. Save, compile and run again.
IDL> .r first
IDL> first

In this way you have hardwired "n" inside the program, but what if "n" needs to be changed every time the program is run? It would be annoying to have to edit, save, compile and run the program every time just to change the value of n. To get around this, we make n an argument to the program. Erase the line you just added (n=3) and make your first line:
PRO first,n
This tells the program that n is an argument (in this case, an input) into the program. Save the file in leafpad and compile the file in IDL.
IDL> .r first
Now you must supply a value for n every time you run the file. Try to run it without supplying n:
IDL> first
You should get the same error as before saying "n" was undefined. Try again, but this time supply a value:
IDL> first, 2
Does this plot look familiar? Try
IDL> first, 3

IDL> first,.5

Now you can change "n" every time you run the program. However, having arguments to your programs is often a double-edged sword. You can change them anytime, but you always must supply them. A good habit to get into is to hardwire numbers that will never change and make the numbers that change often arguments.

In your first program, you learned that you can supply input to the argument list. You can also have outputs in the argument list as well. Let's right a new program called "read_data.pro". Open a new file in your leafpad session. The first line should be
PRO read_data, year, tot,dem,rep
where year, tot, dem, and rep are going to be outputs of the program. The last line should be
END

In between these lines, go back and add some of the commands we used yesterday:
file='partydiv.txt'
r=rd_tfile(file,/nocomment,/auto)
year = reform(float(r[1,*]))
tot=reform(float(r[2,*]))
dem = reform(float(r[3,*]))
rep = reform(float(r[4,*]))

Save the program in leafpad and go back to your IDL window. At the prompt, type:
IDL> .r read_data
IDL> read_data,year,tot,dem,rep
The program should have read in the file, extracted the information from the array, and passed out the information. To verify, do
IDL> help,year,tot,dem,rep
Are they what you expected? Note, however, that you could have used any names for the variables for year, tot, dem, and rep, they need not match the names inside the programs. For instance, do
IDL> read_data,a,b,c,d
IDL> help,a,b,c,d
The year information should be saved as the variable a, the tot array should be b, etc. Also, you could have supplied no variable names:
IDL> read_data
The program will run, but there will be no output because you have assigned no names to the output. In this way, outputs are different than inputs because they are almost always optional, while inputs are almost always required (and the code will crash if they are not supplied.)

Now lets change our program so we have both inputs and outputs in the argument line. Erase the first line of your read_data program (the line that defines the file) and add file to the argument list, meaning the first line of your program should be
PRO read_data, file, year, tot,dem,rep
Note that file could have come after the other arguments, but it is customary to put inputs

before outputs in the variable list. Now re-compile the program and try to run:
IDL> .r read_data
IDL> read_data,year,tot,dem,rep
You should have gotten an error because the program is now expecting you to send a filename as the first argument and you didn't. Do a
IDL> retall
and try again
IDL> file='partydiv.txt'
IDL> read_data,file ,year,tot,dem,rep
Now it should work properly. The first thing you have provided is a filename (an input), all other arguments are output. As long as you provide the input, the program will run. Note that you didn't have to create a variable called file to use in the argument list. You could have just as easily put the filename directly into the call of the program
IDL> read_data,'partydiv.txt',year,tot,dem,rep

Now lets try programming a function. Functions are fundamentally different than procedures, both in how you call them from the IDL command line, but also in philosophy. We think of functions as operating on something.

A simple function could just return the square of a number. Open a new file in leafpad called squared.pro. Inside the file, write
FUNCTION squared,x
y = x^2
return, y
END
x is in the input into the function, y is the output. Unlike the procedure, you have to include a "return,y" statement at the end of the function. Save this file, then go to the IDL window and type
IDL> .r squared
IDL> a = squared(2.)
IDL> print, a
You could also just print the results of the function to the screen, i.e.
IDL> print, squared(3)
Check this for different numbers and verify it works.

Like procedures, functions can have arguments as well. Let's change the above function so that instead of automatically squaring the number, it gives the user a chance to raise the input number to a different power.
FUNCTION squared,x,n
y = x^n
return, y
END

Now when we compile the program and run, we have to supply a power, i.e.,
IDL> .r squared
IDL> print,squared(2.,3)
IDL> print,squared(2,4)
Another option is to make squaring the number (n=2) a default value and the user has the

option of changing the n to another number. In this case, n would be an optional keyword and the first line of the program would be
FUNCTION squared,x,n=n
Now we need to check and see if n is set. If not, we set n = 2. We use the IDL function keyword_set inside an IF statement, as in
IF keyword_set(n) THEN…
Ah, but now we have a problem. If n is set, we don't want to change it. We only want to set n if the keyword is not set. Then we just use the not function, i.e.
IF not(keyword_set(n)) THEN n = 2
Add this to the second line of your program, just after the FUNCTION line. Compile and run. Try setting n and not setting n. Does it work?
IDL> print,squared(2.,n = 3)
IDL> print,squared(2.)

Often in functions and procedures, we want to do things multiple times. We do this with either FOR or WHILE statements. For instance, let's re-write our squared program allowing that the input (x) will be an array instead of single number.

If our input x is an array of numbers, then the output, y, also needs to be an array of numbers. Normally IDL doesn't require you to declare variables, but in this case you have to declare y. The first think we need to do is to determine how big x is so we know how big y needs to be. We can do this using the n_elements() function. Type into your IDL window:
IDL> x=findgen(50)
IDL> nx=n_elements(x)
IDL> print,nx
It should have said that nx = 50, the number of elements in the x array. We then want to create a variable, y, that has the same number of elements, i.e.,
IDL> y=fltarr(nx)
IDL> help,y

Let's implement this in our function. It should look something like this:
```
FUNCTION squared,x,n=n
IF not(keyword_set(n)) THEN n = 2
nx=n_elements(x)
y=fltarr(nx)
```

Now we want to make each elements in y (or y[i]) equal to x[i]^n. We do this with a FOR loop.
```
FOR i=0,nx-1 DO BEGIN
        y[i]=x[i]^n
ENDFOR
```

The return,y and END should remain unchanged. Does this work?
IDL> x=findgen(50)
IDL> .r squared
IDL> print,squared(x,n=3)
IDL> print,squared(x)

(The above example was actually unnecessary. IDL can handle array operations, meaning even if x was an array, you still could have done y=x^n and the answer would have been the same.)

Now you have written a procedure and a program used inputs, outputs, and optional inputs. You have used and IF statement and a FOR loop. Can you write a real science program?

TASK: Write a program to generate a histogram

A histogram is a way of sorting and describing date. For instance, if there are 10 people in the room and you sort them into ages (3 20 year olds, 4 21 year olds, and 3 22 year olds) you have made a histogram. The input to the program would be the ages of people, i.e. ages= [20, 21, 22, 21, 20, 20, 21, 22, 22, 21], the output would be the array that describes the ages (bin= [20,21,22]) and the numbers of people in each age bin (hist =[3,4,3]). In your program, you should have at least 1 input (like ages) and at least 2 outputs (like bin and hist). The other thing you have to consider is the size of the bins. For instance, what if your bin was in 10 year increments, bin = [10, 20, 30]? All the ages would be in the 20 bin (h = [0,10,0]) and your histogram would be pretty boring. Alternatively, the bin size could be too small, you could have increments of days or even minutes. In that case, the bin array would be huge and there would only be one person in each one bin so h would be mostly 0 with 10 1s. So binsize could also be an input into your program or you should figure out a way to make the program to calculate the correct binsize.

Once you have written you histogram program, try it out on some real data. Make a histogram of the sunspot number data we looked at in the first tutorial. Make a histograms of the number of months as a function of sunspot number. Write the plot into a ps file and send it to amy.winebarger@nasa.gov along with your histogram program.