Programming IDL for Astronomy
September 6, 2004

Marshall Perrin[1]

1. Introduction

This is not a programming course, but nonetheless it will involve a lot of programming. This is true of astronomy as a whole: Very few astronomers are trained as computer scientists (though some are!) but nearly every astronomer spends the majority of their time using computers to analyze data, compute simulations, and write papers. It's been said that we're living in a 'Golden Age' of astronomy today, because of the combination of electronic detectors to measure photons at all wavelengths, and computers to analyze the mountain of data that results. Sure, astronomers putting eye to eyepiece and gazing upon the sky from remote mountaintops sounds romantic - but would you really want to determine the spectral type of 250,000 stars *by hand*?[2]

1.1. A Digression: the Virtues of the Programmer

Given that you have to learn to program, you might as well learn to program *well.* So what are the characteristics of a good programmer? Larry Wall, the creator of the Perl programming language, famously identified the three great virtues of the programmer:

- **Laziness**
- **Impatience**
- **Hubris**

A virtuous programmer is **lazy**. He wants the computer to do as much of the work as possible. A virtuous programmer will write labor-saving routines and shortcuts that save vast amounts of time. A virtuous programmer will learn time-saving shortcuts from others. Anything you have to do more than twice is worth writing a program to automate. Any sequence of commands you enter many times should be made into a procedure.

[1]Based in part upon some earlier tutorials by Carl Heiles

[2]Annie Jump Cannon, one of the foremost early female astronomers, did just that. It took her and half a dozen assistants (all female!) half a decade from 1910-1914 to complete the Henry Draper Catalog . It remains to this day one of the cornerstone databases on which much of modern astronomy is built.

A virtuous programmer is **impatient**. She will write programs which are quick and efficient. If a program can figure out something on its own without prompting the user, it should. Programs should notice and react to errors on their own, if at all possible. A virtuous programmer will quickly get frustrated at any dull task; she'd rather spend her time learning a more clever way to do the task than typing in the same command over and over again. (Even if the more clever way takes longer to learn now, it'll surely save time in the long run!) Any time you find yourself doing something repetitive and boring, you should figure out how to make the computer do it for you.

A virtuous programmer has **hubris**: excessive pride. A virtuous programmer wants others to be impressed upon seeing his programs. Algorithms should be beautiful and elegant. Source code should be clean and well-commented. Any particularly tricky or clever bits should be especially-well commented, so that others may understand rather than be confused.

Laziness, impatience, and hubris: strive to cultivate these virtues and you will surely become a master programmer yourself!

2. Ways of Entering IDL commands

There are a number of different ways to enter commands into IDL. Unfortunately, there are some differences in IDL syntax depending on which way you are using.

- **The IDL command line**. The simplest way to enter commands is to type them in directly at the IDL prompt. This works well for simple commands, but becomes something of a hassle for multi-line loop commands: You'll need to add a $ or &$ at the end of each line of text that's part of a multi-line command.
- **batch files**. You can run a text file containing a list of commands by entering `@filename` at the IDL prompt. This is essentially identical to entering commands directly at the IDL command line, and requires the same set of punctuation.
- **script files**. This is very similar to a batch file, but is somewhat superior. You can *also* run a text file containing a list of commands by entering `.run filename`. To use this syntax, your script file must end with the `end` command on its last line. This method is better than the batch file approach because it handles multi-line commands without a problem. You don't need to enter in any $s.
- **Procedures and Functions**. You can define your own procedures and functions which you can then use in the rest of your code. This is the most powerful and flexible way to write IDL code.

I generally recommend creating a script or procedure file for any substantial amount of code you're going to write. The command line is great for interactive use, but for anything complicated, which you may want to run many times as you debug or improve it, a script or procedure is the way to go!

2.1. Making a script file

Create a text file named `myscript.pro` (or whatever) using your favorite text editor. Here's the contents of my `myscript.pro`:

```
for i=0,9 do begin
  print,i, i^2
  print, string(i)+" factorial is "+ strcompress(string(long(factorial(i))),/rem)
endfor

pisquared = !pi^2
print,"pies aren't square!", pisquared

end
```

Note that the script ends with the word ”end”. Now I can run this script at the IDL prompt:

```
IDL> .run myscript
% Compiled module: $MAIN$.
       0       0
% Compiled module: FACTORIAL.
       0 factorial is 1
       1       1
       1 factorial is 1
       2       4
       2 factorial is 2
       3       9
       3 factorial is 6
       4      16
       4 factorial is 24
       5      25
       5 factorial is 120
       6      36
       6 factorial is 720
       7      49
       7 factorial is 5040
       8      64
       8 factorial is 40320
       9      81
       9 factorial is 362880
pies aren't square!      9.86961
```

2.2. Making your own procedures

You can create your own procedures and functions. What's the difference between the two? Nominally, functions return values and procedures don't. In reality, there's not much difference other than a little bit of syntax. First, let's discuss procedures.

A procedure is a chunk of code with a name (let's name it ”foo”), saved in a file called `foo.pro`, which you can call by name from the IDL prompt (`IDL> foo`). When a procedure is run in IDL, IDL enters the procedure, runs the code in that procedure, and then returns to the IDL prompt. The highest level IDL prompt is considered the $MAIN$ procedure.

Procedures contain their own sets of variables inside them (this is referred to as ”variable scope” in many programming languages). Procedures only know about variables that are given to them from the $MAIN$ procedure or procedures that call them. As an example, we'll create a procedure that adds together two numbers. We first create a file called `adder.pro` in a text editor. Inside `adder.pro` , we write

```
pro adder, a, b ; This declares the program and what variables get passed to it.
    sum = a+b ; create a new variable containing the sum
    print, sum ; print to screen
return ; Go back to the program that called adder
end
```

(Remember the semi-colon is the comment indicator in IDL. This tells IDL to ignore everything after the semicolon on this line.) The program `adder` only knows about the variables `a` and `b`. It adds them together and prints the result to the screen. You use `adder` just like `readcol` or any other IDL procedure:

```
IDL> adder, 5, 3
    8
IDL>
```

If you change the`adder.pro` file on disk, you must tell IDL to re-compile the new version.

This is done with the `.comp` command. Let's modify `adder` so that it returns the sum to the calling routine.

```
pro adder, a, b, sum  ; This declares the program and what variables get passed to it.
    sum = a+b ; create a new variable containing the sum
    print, sum ; print to screen
return ; Go back to the program that called adder
end
```

The only change is on the first line! We put the `sum` variable there as an argument. In IDL, unlike many programming languages, arguments to procedures are bidirectional - data may be passed both in and out. Making `sum` an argument lets us pass it back out to the calling procedure. Don't forget to recompile your procedure. You need to do this once after each time you edit it.

```
IDL> .comp adder
% Compiled module: ADDER
IDL>  adder, 5, 3, s
    8
IDL> print,s
    8
IDL>
```

Now the variable `s` contains the sum of 5 and 3. This is a pretty silly example of a procedure, but much more complicated ones are possible ("compute_histogram.pro"? "create_pretty_plot.pro"? "reduce_data_and_write_lab_report.pro"? The sky's the limit!)

2.2.1. Where to save your procedures

When you try to call a procedure from the command line, IDL has to figure out which file on disk it should compile. When you type in `IDL> foo`, IDL has a default list of directories it looks through to find `foo.pro`. This list of directories is set with the IDL_PATH *environment variable*[3] which is set in the `.idlenv` file in your home directory. By default, the IDL path contains the directory `/home/username/idl` and all its subdirectories, your current working directory, and a number of system-wide library directories containing useful functions. If you save your .pro files

[3]An environment variable is a variable which is assocated with your UNIX login session and is set in the shell, rather than an IDL variable. Many environment variables are set in your .cshrc, .login, and .idlenv files, which are all executed as soon as you log in.

in your `/home/username/idl` directory, or subdirectories thereof, IDL will always be able to find them.

If you want to save them in some other directory, that's fine too. You'll need to either change to that directory before starting IDL, or add that other directory to your IDL_PATH environment variable by editing your `.idlenv` file.

2.2.2. Only one procedure per file?

Yes, you really do want to only put one procedure in each file. If you start putting multiple procedures into one file, then that messes up IDL's ability to automatically determine which file is associated with which procedure. There are rare circumstances when you do want to put multiple procedures in one .pro file, but only in moderately advanced situations. For now, stick to the "one procedure = one .pro file" rule!

Make sure to give your procedures unique names! If you have multiple files named `foo.pro` in different directories, IDL may compile a different one than the one you wanted. Similarly, don't give your procedures the same name as any of the library routines, like `readcol` or `total`, because then you can't call the library version any more.

2.3. Functions

Functions are just like procedures, except not. Like procedures, functions must be saved as .pro files with the same name as the function. Unlike procedures, functions are called with parenthesis and return arguments directly.

Let's create a file called `square.pro`:

```
function square,a
    return, a^2
end
```

Note that the return command now has an argument, which is the value that will be returned. We run our new function like this:

```
IDL> print,square(2)
       4
IDL> print,square(3.14159)
      9.86959
```

2.4. Keyword Arguments

Procedures and functions can also take named keyword arguments in addition to regular arguments. Keyword arguments are optional, and can be specified in any order. You've already encountered keyword arguments with the `plot` command: `title, charsize, psym` and all the rest are keyword arguments.

```
; This function counts how many elements in data
; are between min and max

function count_between,data,min=min,max=max
    if not(keyword_set(min)) then min=0
    if not(keyword_set(max)) then max=max(data)

    between = where(data gt min and data lt max,between_count)

    return, between_count
end
```

Now let's use our new function.

```
IDL> d = findgen(100)
IDL> print,count_between(d,min=50,max=70)
          19
```

Note how the function checks to see if the keywords are set or not, and if they are not set, it provides default values.

2.5. Documenting your Procedures; and Reading Other's Documentation

The IDL help files only contain information on functions and procedures which are a built-in part of IDL. But what about documenting all the code that you've written yourself?

IDL provides an easy way for you to document any procedure or function that you write. To see how, look at **doc_library** in IDL's help. This command lets you see the documentation for any procedure for which documentation has been provided; for example, all of the Goddard library's procedures are documented in this way.

```
IDL> doc_library,"readcol"
```

If you write a procedure and don't document it, you might as well forget it—because you *WILL* forget it! Make sure to document your code as soon as you write it, and be double-sure to document any code that someone else will have to use or read!