# Beginning IDL

**Table of Contents**

## Introduction

IDL is one of the programming languages commonly used in astronomy. The long use of IDL in the astronomy community has resulted in the development of many astronomy-related **libraries**.

**Libraries** are essentially collections of programs and functions written in a language that people can download and use. Usually these codes have a unifying purpose. For example, someone might make a library of code that has to do with astro statistics, or one that does cosmological calculations. There are libraries specifically developed for X-ray analysis and stellar photometry. The possibilities are endless!

For the purposes of this class, we will be starting simple, but don't be surprised if you end up using some user-made libraries for your research projects (your mentors will help you, or course). What we will do here is just the beginning, the foundation that will allow you to use (and perhaps develop!) more specialized software.

## The Interactive Environment and Basic Operations

Ok, first thing's first. IDL has an *interactive environment*, which is pretty convenient to use. Simply type `idl` into your terminal. You will get something that looks like this:

```
IDL Version 7.0 (linux x86_64 m64). (c) 2007, ITT Visual Information
Solutions
Installation number: 97443-1.
Licensed for use by: University of Washington

% Compiled module: SXPAR.
IDL>
```

To Exit, just type "exit" and hit return.
Now you are ready to start doing stuff!

## Some Basic Operations, Understanding Variable types

| **IDL Statements** | **Notes and Questions** |
|---|---|
| `> print, 3*5` | 1. What happens if you don't use a comma? |
| `> a=3*5` | *This equal sign is an assignment. Read this as "a gets the value of 3 times 5."* |
| `> print, 3*5 ; Hello there!` | 2. What does the ";" do? (Try it without the ";") |
| `> help,a`<br>`> help,A` | 3. What does help do? What's the difference between using a and A? *"INT" means integer.* |
| `> d=32767`<br>`> print,d+1` | *Integers run from -32768 to + 32768.*<br>4. What happened? |
| `> print,d+1.` | 5. What is the difference between using d+1 and d+1.? |
| `> x = d+1.`<br>`> help, x` | Different types of Variables<br>*Float: Floating-point, with six significant places.*<br>*Double: Floating-point with approximately sixteen decimal digits.*<br>*String: A sequence of 0 to 32,767 characters.*<br>*Long: A float that has space for bigger numbers* |
| `> print,3/5`<br>`> print,3/5.` | What is the difference?<br>*One floating number makes the result a float.* |
| `> width = 20`<br>`> height = 5*9`<br>`> print, width * height` | 6. Explain what happened in these three lines. |
| `> print, 2^2`<br>`> a = 5`<br>`> print, 2^a, 2^2^a` | 7. What does the ^ do? |

## Using Some Math Functions

Functions in idl are called in a way similar to the syntax in your math classes. They take input values and apply mathematical operations to those inputs to produce a result (or set of results). A function called "`Function`" that takes an input, `x` is called by writing `Function(x)`. If `Function` takes more than one variable, then it is called by writing `Function(x,y,z,...)`.

There are many useful functions that are already recognized by IDL right when you first start it up!

| **IDL Statements** | **Notes and Questions** |
|---|---|
| `> print, cos(180), cos(90)` | 8.  Is this in **degrees** or **radians**? |
| `> print, !PI`<br><br>`> print, !pi, !Pi, !pI` | *!PI is an **internal variable** in IDL.  IDL knows what you mean without you having to define it! Again, IDL doesn't care about upper or lower case letters. We will see **internal variables** more later....* |
| `> print, cos(!PI), cos(!PI/2)`<br><br>`> print, sin(!PI/2), tan(!PI/2)` | *Aha! Much better!* |
| `> print, acos(0), asin(0)`<br><br>`> print, acos(0)*180/!PI`<br><br>`> print, asin(0)*180/!PI`<br><br>`> print, acos(1), asin(1)`<br><br>`> print, acos(1)*180/!PI`<br><br>`> print, asin(1)*180/!PI` | 9. What do acos and asin do?   What does multiplying by 180/!PI do? |
| `> a = 5`<br><br>`> print, a, alog10(a), alog10(10^a)`<br><br>`> print, exp(a), alog(exp(a))` | 10. What does alog10() do?<br><br>11. What do exp() and alog() do? |
| `> print, 10 mod 4`<br><br>`> print, 10 mod 3`<br><br>`> print, 9 mod 3`<br><br>`> print, 10 mod 2, 10 mod 5` | 12. What does "mod" do?  Hint: think about division and *remainders!* |

## Working with Arrays

Arrays are convenient ways to store data. Every array is essentially a *matrix* of values. Recall what a matrix is:

$$\begin{array}{cccc} & \text{2x2} & \text{2x3} & \text{3x2} \\ \text{Rows} \longrightarrow & \begin{bmatrix}1 & 2\\ 3 & 4\end{bmatrix} & \begin{bmatrix}1 & 2 & 3\\ 4 & 5 & 6\end{bmatrix} & \begin{bmatrix}1 & 2\\ 3 & 4\\ 5 & 6\end{bmatrix} \\ & \uparrow & & \\ & \text{Columns} & & \end{array}$$

You can also have one dimension matrices (vectors are examples of this!) or may more dimensions (though those are harder to visualize)

Each value is given a particular *address* based on where it is in the matrix. Take this matrix, for example:

*a* is in position 1,1

*b* is in position 1, 2

*f* is in position 3, 2

$$\begin{bmatrix}a & b\\ c & d\\ e & f\end{bmatrix}$$

Note that all of the “addresses” above have two numbers. This is because the matrix is a *two dimensional* matrix. If it was, say, 5 dimensional then it would have 5 numbers in its “address”

## Defining And Manipulating Arrays in IDL

| **IDL Statements** | **Notes and Questions** |
|---|---|
| `> a = intarr(5)`<br>`> print, a`<br>`> b = fltarr(5)`<br>`> print, b`<br>`> c = dblarr(5)`<br>`> print, c`<br>`> d = dblarr(5,3)`<br>`> print, d`<br>`> e = dblarr(5,4,3)`<br>`> print, e` | intarr, fltarr, *and* dblarr *allow you to define arrays of integers, floats, and double type numbers respectively. You can make your arrays have different shapes.*<br><br>*You will need to get used to working with arrays of more than two dimensions. Look at how IDL "visualizes" these arrays. Try to make a 4 dimensional array; a 5 dimensional array. The visualization is not so good now...* |
| `> a = intarr(100)`<br>`> print, a`<br>`> a = a+2`<br>`> print, a`<br>`> print, a^2`<br>`> print, a*5`<br>`> b = exp(a)`<br>`> print, b`<br>`> print, 10^b` | *Now we are manipulating arrays! Here is why arrays are so useful. They allow you to do the* ***same operation on an entire set of data.***<br><br>*Inputting an array into a function gives you a new array.* |
| `> print, a[0], a[2]`<br>`> a[0] = 10`<br>`> print, a[0]`<br>`> c = [0,3,5,2]`<br>`> print, a[c]`<br>`> print, a[0], a[3], a[5], a[2]`<br>`> print, c[11]` | *You can also very easily select parts of data from arrays, whether it be a single element or a range of elements. This is called* <u>*indexing*</u> *an array.* ***Note: array counting starts at 0***<br><br>*You can also index an array with another array. It will output the elements of the array in the order given to it.*<br><br>*Notice what happens when we try to index an array with a number greater than its size.* |

| | |
|---|---|
| `> a[0:9] = 100`<br>`> a[40:49] = 200`<br>`> print, a`<br>`> a[*] = 0`<br>`> print, a` | 13. What does the ':' do here?<br><br>14. What does the '*' do? |
| `> b = dblarr(10,10,3)`<br>`> print, b`<br>`> b[0,0,0] = 10`<br>`> print, b`<br>`> b[*,0,0] = 1`<br>`> print, b`<br>`> b[0:4,0,0] = 2`<br>`> print, b` | *Now, get some practice manipulating a higher dimensional array* |
| `> a = dblarr(10)`<br>`> b = dblarr(10,10)`<br>`> a[0] = 2`<br>`> a[1:9] = 3`<br>`> b[0,*] = 2`<br>`> print, b[0,*]*a`<br>`> c = dblarr(3,5)`<br>`> print, b*c` | *Arrays can be multiplied together! However, note that this is NOT the same as matrix multiplication!*<br><br>***Make sure you can explain, in words, what each of these commands do***<br><br>15. What happens when you multiply arrays with **different** dimensions? Multiplying arrays that have different sizes is usually not a good idea. |

## Useful built-in IDL functions for Arrays.

| IDL Statements | Notes and Questions |
|---|---|
| `> a = dblarr(3,5)`<br>`> print, a`<br>`> print, transpose(a)` | 16. What does transpose do? |
| `> b = indgen(100)`<br>`> print, b`<br>`> print, n_elements(b), n_elements(a)`<br>`> b = b + 3`<br>`> print, where(b gt 10)`<br>`> print, where(b lt 10)`<br>`> print, where(b eq 11)`<br>`> print, b[where(b eq 11)]`<br>`> o = where(b lt 10)`<br>`> print, b[o]`<br>`> print, b[where(b lt 10)]`<br>`> print, where(b gt 5 and b lt 10)`<br>`> print, where(b lt 10 and b gt 10)`<br>`> print, where(b eq 8 or b eq 10)`<br>`> print, b[where(b eq 8 or b eq 10)]` | 17. What does indgen do? What does n_elements do?<br>18. What does where do? what do ‘gt’, ‘lt’ and ‘eq’ indicate?<br>*the “and” can be very useful*<br>*“ -1 “ indicates that your statement is true for* ***none*** *of the elements in the array. This can also be very useful. Remember it!*<br>*“or“ is also useful!* |
| `> c = [1,6,3,4,9,8,7]`<br>`> print, c`<br>`> o = sort(c)`<br>`> print, o`<br>`> print, c[o]`<br>`> print, max(c), min(c)`<br>`> d = [3,5,2,7,4,8,10]`<br>`> print, d[where(c eq max(c))]` | *This is another useful way of defining an array. Just make a list of numbers in brackets!*<br>19. What does sort do?<br>20. What do max and min do?<br>21. What is happening in his line? |

## If Statements and For/While Loops

**“If” Statements** allow you to tell IDL to do something only if a certain logical statement is true.

The Syntax is rather simple.

```
> if [logical true/false statement] then [do something]
```

The logical statements used in if statements will have similar syntax to the “where” statements used above. Examples of good logical statements would be:

```
> if var lt 0 then print, "hello"
> if var gt 0 then print, "hello"
> if var eq 0 then print, "hello"
```

Assuming that “var” is just some variable (NOT an array)

Your if statement must be a statement that is true or false. So, you have to be careful using arrays. For example, the following would cause problems.

```
> a = [1,2,3,4,5,6]
> if a eq 6 then print, "hello
```

The above if statement is both true and false, since one element of a is equal to 6. In other words, it makes no sense and IDL will yell at you!

```
> if a lt 10 then print, "hello"
```

Even though all of the elements in a are less than 10, IDL still won’t like that you are leaving the possibility for both true and false statements.

Of course, there are ways to use arrays in “if” statements. For example:

```
> if n_elements(a) eq 6 then print, "hello"
```

### “For” and “While” Loops

Sometimes, you want to run a piece of code many times. Do do this this, “for” and “while” loops are very useful. The general syntax for a for loop is as follows

```
> for i=num1,num2 do [something]
```

In words, this code loops through values of “i” from num1 to num2. Of course, “i” is just an example. You can name the variable anything you want. It is important to realize that “i” is a real variable that IDL is updating every step.

For example:

```
> for i = 0,10 do print, i
```

The output from the above statement will be:

```
       0
       1
       2
       3
       4
       5
       6
       7
       8
       9
      10
```

While loops, on the other hand, will loop continuously until the “while” statement is no longer met. In other words, you should give IDL a logical statement that is initially true, but will become false eventually.  For example:

```
> a = 0
> while a lt 10 do a = a +1
> print, a
```

What happened above?  First, we initialized the variable a to be 0, which is obviously less than 10.  So, the while loop kept adding 1 to our variable.  After each iteration, it checks to see if a is still less than 10.  If it is, then it adds another 1 to it. Once a = 10, IDL checks once more and the logical statement is now false, so it stops.  At the end of the loop, a is 10.

We can combine our loops with if statements.  For example:

```
> a = intarr(100)
> a[55] = 1
> for i=0,99 do if a[i] eq 1 then print, i
```

What happened above? We first initialized “a” to be an array where one of the elements was 1 and the rest 0.  We then loop through the elements of the array using “i” . For each of these elements, we check to see if it equals 1.  If it does, then we print out the value of “i”.

Loops will get much more interesting once we start making our own functions and procedures.  For now, you are limited to one line statements (because IDL’s interactive environment is not as good as python’s, sadly)

## File Input/Output

Most of the time, you will be working with data. That data is going to be located in files. Thus, reading files is a must! Luckily it is (relatively) straight forward!

Often, the files you read will be columns of data, where each column represents a certain type of data and each row represents a different object (like the BrightStars.dat file we worked with already).

To read in data like this, it is easiest to use the `readcol` command. It works like this:

```
> readcol, “filename”, var1,var2,...,varN,format = ‘...’
```

This will read in each of the N columns in “filename” into the variables var1 through varN. The name of the file must always be input with quotation marks (either single or double). Each of the variables will be an ***array*** of length equal to the number of rows in the file. The i^th^ element of each array corresponds to data from the i^th^ object in the file.

The “format” variable is crucial here. It will tell IDL what kind of data it is reading in. The syntax for this is a string (i.e. bracketed by quotations) of letters separated by commas. For example:

```
> readcol,”filename”, v1,v2,v3,v4,v5,v6, format = ‘a,a,i,f,f,f’
```

In most cases, the format string should have the same number of elements as the number of variables you are reading in.

**IMPORTANT**: *IDL assumes that you will be reading in variables from the left to the right and that different columns of data are separated by a space.*

If you put less variables than there are columns, then IDL will read in the first N columns of data, where N is the number of variables you give it.

If you want to skip a column, put an ‘x’ in the format sequence. For example, `format = ‘a,i,i,x,i,f’` will read in the first, second, third, fifth, and sixth columns of data, but will skip the fourth (assuming you gave readcol five variables). Here is a list of some of the format strings and what they mean:
`a` -- string
`i` -- integer
`l` -- long (just a bigger integer)
`ll` -- 64 bit long (even bigger integer)

`f` -- float
`d` -- double
`x` -- skip the column
Now, onto writing stuff to files.  First, you have to open the file.

```
>openw,1,”filename”
```

The ‘w’ at the end of oepen tells IDL that you want to open “filename” for *writing.* The ‘1’ is just a marker so that you can open multiple files and IDL will be able to tell them apart.  You can choose any number you like.

Now, that our file is open, we can write stuff to it.

```
> a = [1,2,3,4,5]
> printf, 1, a
```

The printf statement works similar to the normal print statement, but the first variable it takes is not output, but rather the marker for the file you want to write to.  Say I open a new file and want to write to that one, I would do something like this...

```
> openw, 2, “file2”
> printf, 2, a
> printf, 2, “this is the second file”
```

**NOTE:** that print statements automatically make a new line.

When you are done with your files, close them.

```
> close, 1
> close, 2
```

This last step might not seem important, but it really is.  *You always want to close files that you are opening.*

Now, one important thing to know about `openw` is that if you use it on an existing file it will *automatically overwrite* any file you already have with the same name in your working directory.  If you would like to *add stuff* to a file use the “append” keyword.

```
> openw, 3, “file_that_exists”,/append
```

You will be seeing this keyword format in the future.  Optional inputs to idl procedures are either written as `Keyword = [stuff]` or `/Keyword`. The latter is for when the keyword is just like an on/off switch.  Either you do something or you don’t (like `Append`).  The former is used when there are many options (like `Format`)

## Defining Procedures and Functions

IDL files end in ".pro". Simply enter `emacs filename.pro` into the command line. Now we are editing an IDL file!

This is the general format for functions

```
function FunName, var1, var2, ... , varN
     [do some stuff]
     return, [something]
end
```

The above defines a function called "FunName" which takes N variables as input. To call the function and collect the value it returns, first make sure you **saved** filename.pro. Now, open up the IDL environment and compile filename.pro. You **need a return statement** in a function!

```
> .compile filename.pro
```

when you do this, you will see something like this: `% Compiled module: FUNNAME.`

Now, you can call the function from the interactive environment. Again, notice that capitol/lowercase letters don't matter to IDL.

```
> output = funname([num1],[num2],...[numN])
```

In order to collect the output from your function, you must either `print` the output or store it in a variable. The value stored in `output` will be whatever value was in your `return` statement above.

**Functions** can only return **one value.** If you want to save a piece of code, but allow it to store more than one value permanently, then use **procedures**.

The general format for procedures is the following:

```
pro ProName, var1, var2, ..., varN
     [do some stuff]
end
```

Now, this looks very similar to functions... so what is the difference?

Well, for one, notice that you **do not need a return statement.** Because of this, we can just run procedures straight from the command line.

```
> .compile filename.pro
> ProName, var1, var2, ..., varN
```

Thus, **procedures** are nice when all we want to do is run a series of commands. **Functions** are nice when we are actively manipulating the input to get a single result.

Notice that we have already been using procedures and functions.

Examples of Procedures we've used: `print, openw, readcol`
Examples of Functions we've used: `cos(), exp(), n_elements()`

As a rule of thumb, functions should be simpler statements that do very specific things, while procedures are for more complicated series of code. Procedures where you would utilize many functions and procedures to do a number of different things.

The other cool thing that *both* functions and procedures can do is change input variables. For example, say I have a function that does the following:

```
function testfun, a, b
     a = a +10
     b = b - 3
     return, a+b
end
```

I would use the function like this

```
> a = 10
> b = 20
> c = testfun(a,b)
> print, a, b, c
20      17     37
```

The function `testfun` still returns a new value (because it has to!) but it can also be used to manipulate already existing variables. Procedures can do this as well.

You can also input undefined variables into functions and procedures, as long as they are defined within the function/procedure itself.

You can have any number of procedures and functions defined in your .pro file. However, you cannot call a function before you define it in the file (order goes top-down). Once you run `.compile` on your .pro file, you will have access to all user-defined procedures and functions that exist within that file.

## For/While Loops and If Statments In Functions and Procedures

Using .pro files opens up a lot of options in IDL.  For one, we can now do multiple line if statements and loops.
The syntax is the following

```
if [true/false statement] then begin
     [do stuff]
     [do more stuff]
endif

for var = 0, 99 do begin
     [do stuff]
     [do more stuff]
endfor

while [true/false statement] then begin
     [do stuff]
     [do more stuff]
endwhile
```

So many more options!

## Intro to Plotting

One advantage to languages like IDL is that they are (relatively) easy to plot with.

| **IDL Statements** | **Notes and Questions** |
|---|---|
| `> x = findgen(100)/4`<br>`> y = findgen(100)/2`<br>`> plot, x,y`<br>`> plot, y`<br>`> z = findgen(100)/3`<br>`> oplot,x,y`<br>`> plot,x,y,xtitle='x-axis',$`<br>`ytitle='y-axis',thick=2,$`<br>`linestyle=1, charsize=1.5,$`<br>`title='plotty-plot'`<br>`> plot, x, y, psym=1` | *The plot command will automatically open a plot window.*<br>22. What happens when you only specify one value?<br>*oplot allows you to plot something without overwriting the first plot.*<br>*There are many options to define in the plotting environment. Note: the '$' just allows you to continue a single line onto the next line. In other words, IDL sees lines with '$' as a single line.*<br>23. Try different values for linestylem(0-5), thick, and charsize. What do these do?<br>24. Try different values for psym (0-7). What does it do? |
| `> !p.multi = [4,2,2,0]`<br>`> plot, [0,1],[0,1]`<br>`> plot, [0,1],[0,1]`<br>`> plot, [0,1],[0,1]`<br>`> oplot, [0,1],[0,0.5]`<br>`> plot, [0,1],[0,1]`<br>`> plot, [0,1],[0,1]` | *No let's display multiple plots on the same page!*<br>*This defines a global variable p.multi. The general form should be p.multi = [A,B,C,D] where*<br>*A = total number of plots ( = BxC[xD])*<br>*B = number of columns*<br>*C = number of rows*<br>*D = number of plots stacked in z-dir (I would always leave this at 0)*<br>*Every new plot command you do will make a plot in another space. oplot will keep you in your current plotting space.*<br>*If you plot too many times, it will reset.* |

## Making Your Plots Look Fancier

**`PSYM` lets you set the point type (you explored this a bit earlier)**
0 = no point (just connected by lines)
1 = +
2 = *
3 = points
4 = diamond
5 = triangle
6 = square
7 = X
10 = histogram mode
Negative values will make it so lines are used to connect your points. No lines will be drawn for positive values

**`Color` lets you set the color**
The values you input for this will depend on the color table you use. Check out some of the possible color tables here (#40 is a popular one):
http://www.exelisvis.com/docs/LoadingDefaultColorTables.html

Sadly, the numbers aren't very intuitive. Higher numbers move you to the right on whatever color table you loaded. You will need to do a lot of guess and checking to get the color you really want.

To load a color table, just type the command `loadct, <table number>`
Now, be careful. Setting the color will set EVERYTHING to that color... even the axes! One way to get around this is to use the `/NoData` option.

First plot your axes:
```
plot, [xmin,xmax],[xmin,xmax],/nodata
```
Then plot your data
```
oplot, datax,datay,color=100
```

**Labels and Legends** are also very important, especially if you want people to understand what you are plotting!

Use the following keywords to label your plot!

```
Title = "Plot Title"

XTitle = "x-axis title"
YTitle = "y-axis title"
ZTitle = "z-axis title"
```

To make a legend, you will use the LEGEND function. The general syntax is the following:

```
legend,
['label1','label2',etc.],psym=[…],linestyle=[...],color=[...]
```

So, you give the legend function a list of labels, point styles, linestyles, and colors and it will make you a legend, placing the first label with the first psym and the first linestyle, etc.

Example:

```
legend,['Plus sign','Asterisk','Period'],psym=[1,2,3]
produces:
                ----------------
               |                |
               |  + Plus sign   |
               |  * Asterisk    |
               |  . Period      |
               |                |
                ----------------
```

Place the legend on your plot with `/right`,`/bottom`, or `/center`

To be really fancy, you can place it with `position=[x,y]`, where x and y denote a spatial coordinate *on the figure* (this is NOT a coordinate on your plot). Use this keyword with `/norm` so that (0,0) is the bottom left edge of the figure and (1,1) is the top right edge and (0.5,0.5) is in the middle.

**Saving plots**

| | |
|---|---|
| `> set_plot, 'ps'`<br><br>`> device, filename='output.ps'`<br><br>`> plot, x,y, xrange=[0,20],yrange=[50,-10]`<br><br>`> device,/close`<br><br>`> set_plot, 'x'` | *Here we will plot to a file. First, we must tell IDL what kind of file we will write to. "eps" is another common file type. Then, we must tell it the file we want to write to with the "device" command.*<br><br>*Note: you can control the axes ranges with the x/yrange = [x/ymin,x/ymax] keyword. This also allows you to flip axes by going high to low rather than low to high (i.e. having ymin>ymax)*<br><br>*When you are done, make sure to close your file with* `device,/close`<br><br>*If, when you are done, you want to go back to the real time plotting we had before, you must reset set_plot.* |

## Histograms

In IDL, you use the function called HISTOGRAM. The output corresponds to the number of elements in each bin specified by the function.

general syntax is like any other function, where [optional key words] represents any list or combination of keywords shown below.

```
> result = HISTOGRAM(inputarray,[optional key words])
```

Optional keywords:

`binsize`: sets the size of your bins
`nbins`: sets the number of bins
`locations`: sets the name of a different variable to which HISTOGRAM will output the bin edges
`max`: maximum value to consider (defaults to the maximum value of your data)
`min`: minimum value to consider (defaults to minimum value of your data)

Note:
if neither `nbins` nor `binsize` is given, then `binsize = 1`.
If `nbins` is given, then by default `binsize = (max - min)/(nbins - 1)`

Now... onto plotting! Simply plot using the plot command we've already learned, but with psym = 10. Your x-value will just be the bin edges that the `location` keyword lets you save.

for example:

```
> hist=histogram(data,location=bins,min=0,nbins=10,binsize=0.3)
> plot, bins, hist, psym=10
```

## Additional Resources

The online help is reasonably good.  Type “idlhelp” on the command line (not in IDL).

Here’s an IDL Tutorial: http://nstx.pppl.gov/nstx/Software/IDL/idl_intro.html
Another one: http://www.msi.umn.edu/software/idl/tutorial/
Tips and Tricks: http://www.dfanning.com/documents/tips.html
The IDL astronomy library: http://idlastro.gsfc.nasa.gov/homepage.html
David Fanning's Coyote's Guide: http://www.dfanning.com/

David Fanning’s page is an *excellent* resources for supplemental code to make your life easier.