IDL Quick Reference/Cheat/Crib Sheet

Remember: IDL is NOT case sensitive.

Introduction

IDL is a programming language similar to Fortran 90. Important things to note are:

- IDL can perform array arithmetic operations.
- All keywords can be abbreviated, if unique (plot,x,y,linestyle=1 is equivalent to plot,x,y,line=1).
- Different from C and Fortran, IDL has a dynamic allocation of variable types.
- You can personalize your IDL command file which is executed every time IDL is started. In Unix `export IDL_STARTUP=name_of_startup_file` In Windows go to the Control Panel,System Properties,Environment tab and create a user variable named IDL_STARTUP whose value is the filename with full path.
- Spaces are ignored.
- IDL is Not compiled into an executable.
- `Control-C` Stops the active procedure.

Procedures and Functions

Procedure, *arg1*[,*arg2*][,keyword1=*value*][,/keyword2].

result=**Function**(*arg1*[,*arg2*][,keyword1=*value*][,/keyword2]).

- Filenames have to be the same as the name of the procedure or function with the extension `.pro`.
- Functions must return a result. Procedures do not.
- Arguments in a procedure or function are passed by their position in the call, where as keywords are not.
- Keywords can be set true by a backslash ie. \keyword.
- Functions are declared by using the word `Function` in the declaration. Whereas Procedures have the word `pro`.

System Variables

Constant System Variables

`!PI`	Single precision π
`!DPI`	Double precision π
`!DTOR`	Converts Degrees to Radians
`!RADEG`	Converts Radians to Degrees
`!VALUE`	Single and Double precision NaN and infinity values.

Error Handling/Information System Variables

`!WARN`	Reports use of obsolete routines.
`!ERROR_STATE`	Structure containing last error information.
`!EXCEPT`	Controls when IDL checks for math error conditions.
`!MOUSE`	Status from the last cursor read operations.

Environment Variables

`!D`	Structure of Device information.
`!P`	Structure of Plotting information. All of which have corresponding keywords in the `plot` procedure.
`!PATH`	The path for IDL routines.
`!CPU`	Variable that supplies information about the state of the system processor, and of IDL's use of it.
`!DIR`	Location of the main IDL directory.
`!DIR_PATH`	Indicates where IDL looks for Dynamically loadable modules when started.
`!JOURNAL`	100 if journal is set to on. 0 if set off
`!QUIET`	Suppresses informational messages if set to zero.
`!VERSION`	Structure of the type, architecture, and version of IDL.

Special Characters

`Ampersand (&)`	Separates multiple commands on a single line.
`Apostrophe (')`	•Delimits string constants •Indicates part of octal or hex constants
`Asterisk (*)`	•Pointer Dereference •Muliplication operator •Array subscript range
`At Sign (@)`	•Include character: Used at the beginning of a line to cause the IDL compiler to substitute the contents of file whose names after the @ symbol for the line. •In interactive mode, @ is used to execute a batch file.
`Colon (:)`	•Ends label identifiers. •Separates start and end subscript ranges
`Dollar Sign ($)`	•Continue current command on the next line. •In interactive mode. If it is the first symbol in a line the following text is executed as an operating system command. (in command mode right?)
`Exclamation (!)`	•First character of system variable names and font-positioning commands.
`Period (.)`	•First character of executive commands. •Indicates floating point numbers. •Indicates fields in a structure, such as structure.*field*
`Question Mark (?)`	•Invokes online help when entered at the command line •Part of the ?: ternary operator used in conditional expressions.
`Semicolon (;)`	First character of a comment field.
`Plus (+)`	•Number Addition •Concatenates a string.

Data Types

Type	Size	Scalar	Conversion	Array
byte	8	0B	byte()	bytearr()
integer	16		fix(), nint()	intarr()
long	32	0L	long()	lonarr()
float	32	0.0	float()	fltarr()
double	64	0.0D	double(0)	dblarr()
complex	32,32		complex()	complexarr()
dcomplex	64,64		dcomplex()	dcomplexarr()
string	16	'xyz'	string()	strarr()

Types of Data Storage

Type	Scope	Dynamic Data Type	Data type
Arrays	Local	Yes	Must all be the same
Named Structures	Global	No	Can vary
Annoymas Structures	Local	No	Can vary
Table	Local	No	Can vary
Lists	Local	Yes	?
Associative Arrays	Local	Yes	Can vary
System variables	Global	Depends	?

Flow Control

```
if (expression) then a=1 else begin
   b=2
endelse

for i=init,limit,stride do begin
  print, array[i]
endfor

if(expression) then begin
  A=1
endif

repeat begin
  A = A * 2
endrep until (A gt B)

while eof(1) do begin
  readf, 1, a, b, c
endwhile

label1:begin
  print, a
end

case name of
 'Moe': begin
          print, 'Stooge'
        end
endcase
```

```
switch name of
 'Moe': begin
          print, 'Stooge'
        end
endswitch
```

Jump Statements

`break`	Immediately exits from a loop (`for`,`while`,`repeat`),`case`, or `switch` statements without resorting to statements.
`continue`	Immediately starts the next iteration of the enclosing `for`, `while` or `repeat` loop.
`goto,`*`label`	Transfers program control to point specified by label. `goto` statements are generally seen as bad programming practice.

Arrays

- Arrays can be upto eight dimensions.
- IDL allows the use of arithmetic array operations without the use of loops.
- Unsubscripted arrays are passed by reference. Subscripted arrays are passed by value.
- Prior to IDL version 5 parentheses () were used to create and index arrays. However this caused confusion with the use of functions. In IDL 5.0 the square bracket array syntax is used (although the parentheses were kept for backwards compatibility).
- Array indexing starts at zero like in C. This is different to Fortran where array indexing can start at any number.
- Arrays are stored in column major format the same as Fortran. This is not the same as C which is stored in row major format.

Array Constructors

Concatenation through the use of brackets ie. arr = [arr1,arr2,]

`indgen(n)`	arr=Integer indexed array elements from 0 to n-1 . Analogous are `findgen`, `dindgen`, `cindgen`, `sindgen`.
`intarr(n)`	Creates an n element array all set to zero. Analogous are `fltarr`, `dblarr`, `complexarr`, `strarr`.
`make_array(n,m)`	Creates an array N columns wide and M rows long. Keywords can be used in the function call to specifiy the data type.

Array Subscripts

`arr(i,j)`	Subscript for a single value
`arr(i,*)`	Column i of a two dimensional array.
`arr(i:k,j:l)`	A 2D subarray of columns i to k, rows j to l.
`arr(i:k:m)`	A 1D array starting at subscript i and finishing at subscript k with a stride of m.
`arr1(arr2)`	The elements of Array whose subscripts are the values of Array2

Functions for Determing Array Properties

`where`(arr lt 50)	Returns subscripts of nonzeo array elements which match expression.
`nelements`(*arr*)	Returns the total number of elements.
`size`(*arr*)	Returns array size and type information.
`min`(*arr*)	Smallest element of an array.
`max`(*arr*)	Largest element of an array.
`mean`(*arr*)	Mean value of all the elements of an array.
`variance`(*arr*)	Variance of all the elements of an array.
`stddev`(*arr*)	Standard deviation of all the elements of an array.
`moment`(*arr*)	Mean, variance, skew, kurtois, standard deviation, mean absolute deviation
`total`(*arr*)	Total sum of all the elements of an array.

Array Reordering Functions

`reform`(*arr*)	Changes array dimensions without changing contents.
`reverse`(*arr*)	Reverses order of array elemets.
`rotate`(*arr*)	Rotates array.
`transpose`(*arr*)	Takes the transpose of the array.
`shift`(*arr*)	Shifts array elements.
`sort`(*arr*)	Gets indices of sorted array elements.
`uniq`(*arr*)	Gets indices of unique elements in a sorted array.
`PM`(*arr*)	Prints matrix ie. as row major instead of column major.

Array Resizing Functions

`rebin`(*arr*)	Resize an n-dimensional array by an integer multiple or factor.
`congrid`(*arr*)	Resize a one-,two- or three- dimensional array to an arbitrary size (user-friendly version)
`interpolate`(*arr*)	Resize a one-,two- or three- dimensional array to an arbitrary size (generalized version)

Structures

Unlike arrays structures allow different data types to be packaged together into one entity. They are similar to Structures in C and Derived Data types in Fortran.

Named structures

`var={Structure_name,Tag_name1:Tag_def1}`

Unnamed structures

Useful for dynamic structures.
`var={Tag_name1:Tag_def1}`

Functions for working with structures

`n_tags(`*struc*`)`	Returns the number of variables (tags) within a structure.
`tag_names(`*struc*`)`	Returns the name of each variable (tag) within a structure.
`create_struct('`*name*`',`*var*`)`	Create a structure, or append variables to a structure.
`struct_assign()`	Merges the contents of two existing structures.

Pointers

Pointers in IDL are the same as, in C and Fortran.

ptr = `ptr_new`(*var*)	Creates *ptr* which points to *var*.
ptr = `ptr_new`()	Creates a null pointer called *ptr*.
`ptr_free`, *ptr*	Procedure to release the memory referenced by *ptr*.
`ptr_valid`(*ptr*)	Returns true if a pointer is actually pointing to something. False if not.
`*`*ptr*	Dereferences *ptr*.

Method of Passing Arguments

Passed by Reference	Passed by Value
Scalars	Constants
Arrays	Indexed subarrays
Structures	Structured elements
Undefined variables	System variables
	Expressions

Input/Output

StdIn has lun of 0, StdOut has a lun of 1 and StdErr has a lun of 2

`print,var`	Prints variables to StdOut.
`read,var`	Reads variables from StdIn.
`openr,`*lun*`,'file.txt'`	Opens an existing file for read access.
`openw,`*lun*`,'file.txt'`	Opens a new file for read and write access.
`openu,`*lun*`,'file.txt'`	Opens an existing unformatted file for read and write.
`readf,`*lun*`,var`	Formatted read into *var*
`readu,`*lun*`,var`	Reads unformatted files
`printf,`*lun*`,var`	Formatted write of *var*
`writeu,`*lun*`,var`	Writes *var* to an unformatted file.
`close,`*lun*	Closes *lun*
`close, /all`	Closes all luns
`get_lun,`*lun*	Procedure to allocate a free unit number
`free_lun,`*lun*	Procedure to free a unit number that was allocated by `get_lun`.
`tplate=ascii_template()`	Starts GUI interface to setup a template for reading in ascii data.
`tplate=binary_template()`	Starts GUI interface to setup a template for reading in binary data.
`read_ascii(file_name, Template=tplate)`	Reads ascii data using an existing ascii template.
`read_binary(file_name, Template=tplate)`	Reads binary data using an existing binary template.
`assoc`	Associates an array structure with a file, allowing random access input and output.

Format Codes

`n`	Repeat count, specifying the number of times the format code should be processed.
`+`	An optional flag that specifies that positive numbers should be output with a + prefix.
`-`	An optional flag that specifies that string or numeric values are left justified.
`w`	Optional width specification. Width specifications and default values are format-code specific.
`d`	The number of digits after the decimal point.
`m`	Minimum number of nonblank digits to be shown on output.

`[n]A[-][w]`	Transfers character values.
`[n]F[±][w][.d]`	Single precision floating-point value
`[n]D[±][w][.d]`	Double precision floating-point value
`[n]E[±][w][.d][Ee]`	Floating point exponential format. The width should include all characters. The number of digits in the exponent is platform specific.
`[n]G[±][w][.d][Ee]`	The G format code uses the F output style when reasonable and E for other values, but displays exactly d significant digits rather than d digits following the decimal point.
`[n]I[±][w][.m]`	Transfer integer values
`Q`	Returns the number of characters that remain to be transfered during a read operation
`'str', H`	Output string values directly.
`Tn`	Specifies the the absolute position
`TLn`	Moves the position with a record to the left.
`TRn,nX`	Moves the position within a record to the right.
`:`	Terminates processing.
`$`	Suppresses newlines in output.
`/`	Starts a new line

Executive Commands

Executive commands must be entered at the command prompt. They cannot be used in programs.

`.compile`	Compiles program without running it
`.go`	Executes previously compiled main program.
`.return`	Continues execution until the RETURN statement.
`.run`	Compiles and executes IDL commands from files or keyboard.
`.rnew` [*File1,File2*]	Erases main program variables and then does .RUN
`.reset_session`	Resets much of the state of an IDL session without requiring the user to exit and restart the session.
`.step` [*n*]	Executes one or *n* statements from the previous position. (Useful for debugging)
`.trace`	Similar to .CONTINUE, but displays each line of code before execution. (Useful for debugging)

Error Handling

`catch,..`	Procedure which intercepts and processes error messages and continues program execution.
`message,..`	Procedure which issues error and informational messages.
`on_error,n`	Designates the error recovery behavior. The default is to stop in the routine which has caused the error.
`on_ioerror,label`	Declares I/O error exception handler.
`strmessage(Err)`	Returns the text of a given error number.

Help

`help,var`	Displays information on *var* (data type, size, memory usage (`\mem`), details of a structure (`\struc`) etc) via the appropriate keywords.
`?`	starts built-in help tool.
`idlhelp &`	Starts help tool from the OS shell all IDL-included functions and procedures described in detail.
`idlman`	To display pdf manuals for IDL.

File Management Functions

`file_delete`, 'file'	Deletes the associated file.
`file_search(*.pro)`	Returns an array of strings containing all the file names that match.
`file_mkdir`,	Makes a new directory.
`file_test`	Tests a file or directory for existence and other specific atributes.
`file_info`	Returns status information about a file.
`Fush, lun`	Forces, data from memory buffers to disk.
`point_lun,lun,pos`	Sets file pointer to position in bytes.
`eof(lun)`	Tests a file unit to see if the pointer is at the end of a file. Returns true or false.
`fstat(lun)`	Returns a structure of status information on the file unit. (To view it use `help,/struc`)
`cd, '../dir'`	Changes directory
`ls`	List files in current directory.

Time Management Functions

`timegen()`	Generates an array of time values.
`caldat()`	Converts Julian to Calendar.
`julday()`	Converts Calendar to Julian.
`systime()`	The current system time.

Debugging

`breakpoint,index`	Sets and clears breakpoints for debugging.
`shmdebug(enable)`	Print debugging information when a variable loses reference to an underlying shared merory segment.

String Manipulation

`+`	Concatonates strings
`strtrim(str,flag)`	Removes blanks from string, Flag - 0 front, 1- back, 2 - both
`strcompress()`	Removes blanks spaces within a string.
`strmid()`	Extracts a substring from a string.
`strpos()`	Position of a substring within a string.
`strput()`	Inserts the contents of one string into another.
`strjoin()`	Concatenates all elements into a scalar string.
`strsplit()`	Splits string into an array of substrings.
`strmatch()`	Compares search string against input string expression.

Code Optimization

1. Use of arrays rather then loops. As IDL is optimized for array arithmetic operations.
2. Always free memory associated with a pointer when it is not needed anymore to avoid memory leaks.
3. Use IDL intrinsic procedures where you can rather the writing your own or using a spawn procedure. As they are optimized for speed.
4. When creating a large array which is going to be filled with data straight away. Use the "nozero" keyword in the array creation function.
5. Keep the memory requirement of your application smaller than the physical memory to avoid performance penalties by swapping.
6. Write expressions in an order which minimizes the about of computation needed to evaluate it.
7. Access large array data in machine address order.(IDL is column-major)

`amoeba()`	Minimizes a function using the downhill simplex method.
`constrained_min,..`	Minimizes a function using the Generalized Reduced Gradient Method.
`dfpmin,..`	Minimizes a function using the David-Fletcher-Powell method.
`powell,..`	Minimizes a function using the Powell method.
`simplex()`	Function which uses the simplex method to solve linear programming problems.
`temorary(`*var*`)`	Returns a temporay copy of a variable, and sets the original variable to undefined. Conserves memory when performing operations on large arrays.
`delvar, var`	Deletes a variable from the main IDL program level freeing any memory used.
`memory()`	Returns a vector of information about the amount of dynamic memory currently in use by the IDL session.
`profiler,..`	Procedure to accesses the IDL Code Profiler used to analyze performance of applications.
`time_test2,..`	Procedure to perform speed benchmarks for IDL.

Session Commands

`save,var1,var2..,file_name='file.dat '`
`restore,'file.dat'`
`journal, 'name.jou'` Creates journal of output.
`spawn` Starts OS shell

Differences between PV-Wave and IDL

`help` in IDL is the same as `info` in PV-Wave
epoch of the PV-Wave julian day is 2361220.5 days behind IDL julian day.

Useful Development Environment Shortcuts

`Ctrl + Space` - Content assist

Useful Links

`www.ittvis.com`
`www.ittvis.com/services/techtip.asp?ttid=1799` - Article on code optimization.
`www.ittvis.com/ProductServices/IDL/ProductDocumentation/tabid/206/language/en-US/Default.aspx` - Product documentation
`www.astro.washington.edu/deutsch/idl/htmlhelp` - Searchable index of publicly available IDL procedures or functions.
`comp.lang.idl-pvwave` - Usenet group.
