IDLdoc 3.3 Tutorial

Michael Galloy

Abstract This tutorial attempts to provide a friendly guide to start using IDLdoc. See the companion reference guide for a more detailed listing of all the options that IDLdoc provides.

Introduction

IDLdoc generates nicely formatted API documentation directly from source code. If the documentation is close to the code it is much more likely to be kept up-to-date. As much as possible is generated directly from the source code itself.

IDLdoc goals are to:

1. provide browsable API documentation for both developers and users of the code base

2. indicate portions of the API that are not documented

3. analyze code for aspects like complexity

There are similar tools for other languages, notably:

1. Javadoc for Java

2. Doxygen for multiple Languages

3. Sphinx for Python

IDLdoc can easily do code introspection because it is written in IDL.

The rst format and markup styles are inspired by the reStructuredText and Markdown projects, though the syntax is not exactly the same as either one.

This tutorial intends to get a new user up to speed in using IDLdoc in the simplest way using the newer, more modern style of IDLdoc commenting. Don't worry, though, IDLdoc still supports legacy commenting styles so you don't have to go changing existing documentation (unless you want to make use of some of the cool, new features!). Experienced users will probably learn some new things too, since documentation for IDLdoc has been spotty in the past.

Installation

To install IDLdoc, simply unzip and place the IDLdoc *.sav* file, or the *src/* directory if you are installing the IDLdoc source distribution, in your IDL path.

Do not separate the contents of the distribution; the code looks for files in locations relative to itself.

Basics

The basic calling sequence for IDLdoc specifies the path to a directory (or hierarchy of directories) with the *ROOT* keyword and, optionally, the path to where the output should be created:

```
IDL> idldoc, root='path/to/code', output='path/to/output'
```

If *OUTPUT* is not specified, output will be created inside the *ROOT* hierarchy next to the source files. By default, IDLdoc creates HTML documentation, though it can be customized to create other types of documentation. To view the documentation produced from the above command, open *path/to/output/index.html* in a web browser.

IDLdoc will generate useful documentation from any valid IDL code, showing the calling sequence of routines even if there are no comments, or no comments in a format that IDLdoc recognizes, in the source code. But for more useful documentation, comments formatted in a simple manner that IDLdoc recognizes can be parsed and placed into the output documentation.

IDLdoc examines files with *.pro*, *.sav*, *.dlm*, and *.idldoc* extensions. Source code and specially formatted comments in *.pro* files are parsed. Save files, of either the code or data varieties, are examined for their contents and listing of the routines or variables contained are produced. DLM files produce a similar, but more limited, output as normal source code files. Finally, *.idldoc* files are a way of including documentation that is common to several items, such as an overview of a directory or a page describing a topic that several routines refer to.

Two keywords you most likely will want to specify are *TITLE* and *SUBTITLE*. Set these to string values to be displayed prominently on your documentation. For example, the command used to generate the API documentation for the IDLdoc project itself is:

```
idldoc, root='src', output='api-docs', $
        title='API documentation for IDLdoc ' + idldoc_version(), $
        subtitle='IDLdoc ' + idldoc_version(/full), /statistics, $
        index_level=1, overview='overview', footer='footer', /embed, $
        format_style='rst', markup_style='rst'
```

This places the IDLdoc version information into the title/subtitle of the documentation. We'll talk about some of the other options in the following sections.

Note: By default, IDLdoc 3.0 copies source code into the output directory, so placing the output directory in your *!PATH* can cause IDL to choose the (possibly outdated) copy in the doc output directory over the correct source file. It is recommended to either place your docs outside your *!PATH* or use the *NOSOURCE* keyword.

Comment format

IDLdoc parses comment headers which start with `;+` and end with `;`, like:

```
;+
; This is a comment header.
;-
```

If this comment header is immediately before or after a routine declaration header, then the comment is considered a routine-level comment and associated with the routine in the documentation. If not, then it is considered a file-level comment and associated with the file. For example, some file-level and routine-level comments are shown below:

```
;+
; File-level comments
;-


;+
; Routine-level comments
;-
pro my_routine1
...
end

pro my_routine2
;+
; Routine-level comments
;-
...
end
```

The comments inside the comment headers can just be free-form comments describing the routine, but they can also use various tags recognized by IDLdoc to document a particular aspect of the routine or file. For example, there is a *Keywords* tag allowing specifics about the keywords to be documented.

A typical comment header for a routine is shown below. Don't worry about all syntax in the header, we'll get into more details later. Just notice the general remarks followed by tags like `:Examples:`, `:Uses:`, `:Returns:`, etc.:

```
;+
; Get an RGB color value for the specified color name. The available colors
; are:
;
; .. image:: vis_colors.png
;
; :Examples:
;    For example::
;
;        IDL> print, vis_color('black')
;           0   0   0
;        IDL> print, vis_color('slateblue')
;         106  90 205
;        IDL> c = vis_color('slateblue', /index)
;        IDL> print, c, c, format='(I, Z)'
;           13458026     CD5A6A
;        IDL> print, vis_color(['blue', 'red', 'yellow'])
;           0 255 255
;           0   0 255
;         255   0   0
;        IDL> print, vis_color(/names)
;        aliceblue antiquewhite aqua aquamarine azure beige ...
;
;    These commands are in the main-level example program::
;
;        IDL> .run vis_color
;
; :Uses:
;    vis_src_root, vis_index2rgb
;
; :Returns:
;    Returns a triple as a `bytarr(3)` or `bytarr(3, n)` by default if a
;    single color name or `n` color names are given. Returns a decomposed
;    color index as a long or `lonarr(n)` if `INDEX` keyword is set.
;
;    Returns a string array for the names if `NAMES` keyword is set.
;
; :Params:
;    colorname : in, required, type=string/strarr
;        case-insensitive name(s) of the color; note that both "grey" and
;        "gray" are accepted in all names that incorporate them
;
```

```
; :Keywords:
;    names : in, optional, type=boolean
;       set to return a string of color names
;    index : in, optional, type=boolean
;       set to return a long integer with the RGB decomposed into it
;    xkcd : in, optional, type=boolean
;       set to use xkcd color survey color names instead of the HTML color
;       names (see `xkcd color survey <http://xkcd.com/color/rgb/>`)
;-
function vis_color, colorname, names=names, index=index, xkcd=xkcd
```

There are multiple formats for comments to make them understandable by IDLdoc, i.e., format styles. The three format styles allowed in IDLdoc comments: "rst" (shown above), "IDLdoc", and "IDL." The "rst" format style is the modern style in current versions of IDLdoc. The "IDLdoc" and "IDL" format styles are provided for legacy documentation headers; they are described in the reference manual. Because it was the first format style, the "IDLdoc" style is the default, but new comments are recommended to be written in the "rst" format style.

Indentation and spacing is significant in the *rst* format style. There must be a blank line before a tag, like `:Examples:` or `:Uses:`. Comments after a tag can start on the same line as the tag, but further comments must be indented at least one space and end with a blank line. Some tags, like `:Params:` and `:Keywords:` take arguments which are indented and comments for the arguments are then further indented

Note there is also a "rst" *markup style* that can be used to annotate comments in one of the locations that are specified by the "rst" format style. We'll give more details about the markup style in the "Comment markup" section later on.

Source code files

Source code files, i.e., *.pro* files, can contain file or routine-level comments. Some common tags for file-level comments are *Examples*, *Author*, *Copyright*, and *History* (the full list is in the reference manual). For example, the *mg_h5_getdata.pro* file contains multiple helper routines followed by the main *MG_h5_GETDATA* routine. The file-level comment at the beginning of the file looks something like:

```
;+
; Routine for extracting datasets, slices of datasets, or attributes from
; an HDF 5 file with simple notation.
;
; :Categories:
;    file i/o, hdf5, sdf
;
; :Examples:
;    An example file is provided with the IDL distribution::
;
```

```
;          IDL> f = filepath('hdf5_test.h5', subdir=['examples', 'data'])
;
;     A full dataset can be easily extracted::
;
;          IDL> fullResult = mg_h5_getdata(f, '/arrays/3D int array')
;
;     Slices can also be pulled out::
;
;          IDL> bounds = [[3, 3, 1], [5, 49, 2], [0, 49, 3]]
;          IDL> res1 = mg_h5_getdata(f, '/arrays/3D int array', bounds=bounds)
;          IDL> help, res1
;          RESULT1         LONG       = Array[1, 23, 17]
;
;     This example is available as a main-level program included in this file::
;
;          IDL> .run mg_h5_getdata
;
; :Author:
;     Michael Galloy
;
; :Copyright:
;     This library is released under a BSD-type license.
;-
```

Common routine-level tags are *Returns*, *Params*, *Keywords*, *Examples*, *Uses*, *Requires*, *Author*, *Copyright*, and *History*. For example, the *MG_H5_GETDATA* routine's comment header is:

```
;+
; Pulls out a section of a HDF5 variable.
;
; :Returns:
;    data array
;
; :Params:
;    filename : in, required, type=string
;       filename of the HDF5 file
;    variable : in, required, type=string
;       variable name (with path if inside a group)
;
; :Keywords:
;    bounds : in, optional, type="lonarr(3, ndims) or string"
;       gives start value, end value, and stride for each dimension of the
;       variable
```

```
;       error : out, optional, type=long
;          error value
;-
function mg_h5_getdata, filename, variable, bounds=bounds, error=error
```

Source code files documented in different styles can be placed in the same directory hierarchy. The default IDLdoc styles, or those provided by the *FORMAT_STYLE* and *MARKUP_STYLE* keywords, can be overridden for a single file by placing a special comment on the first line of the file:

```
; docformat = 'rst'
```

This indicates that the rst format style should be used for this file. Since the rst markup style is the default when using the rst format style, it will also be used. To use the verbatim markup style with the rst format style for a particular file, place the following on the first line of the file:

```
; docformat = 'rst verbatim'
```

It is a good idea to place the *docformat* line on the beginning of every file that is shared with others, then IDLdoc will always use the correct styles even if the file is placed in another library.

The overview file

The overview file, specified with the *OVERVIEW* keyword to IDLdoc, contains comments describing the entire directory hierarchy. It is displayed near the front of the documentation, e.g., in the HTML documentation it is shown on the first page of the output.

For the most part, the file is just a comment block describing the directory hierarchy, but after that it can contain *Author*, *Copyright*, *History*, *Version*, and *Dirs* tags. For example, the overview file my library starts off with:

```
Personal IDL library of Michael Galloy. This is code that doesn't have
enough "meat" on it to be it's own package.

:Author:
  Michael Galloy

:Dirs:
  ./
    Main utility routines
  analysis/
    Various algorithms (sorting, sampling, etc.) and math helper routines
  animation/
    Classes to produce animations using object graphics.
  collection/
    Objects implementing various types of collections.
```

.idldoc files

Special documentation files, with extension *.idldoc*, can be placed into the output. There are no special tags in *.idldoc* files; the entire file is just one big comment block. The one special syntax for *.idldoc* files is the *title* directive described in the markup section. Headings can be used in any comment block, but are particularly useful in *.idldoc*, overview, and directory overview files.

NOTE: "*.idldoc* files" refers to files with an *.idldoc* extension, like *cptcity-catalog.idldoc*. Files named *.idldoc* are directory overview files, described below.

Directory overview files

Directory overview files are special *.idldoc* files that describe the contents of a particular directory. They are named *.idldoc* and placed in the corresponding directory. *Private*, *Hidden*, *Author*, *Copyright*, and *History* tags are allowed in a directory overview file.

For example, the *collection/* directory of the IDLdoc source contains the following *.idldoc* file:

```
The collection framework defines classes to provide various types of
containers, primarily list ('MGcoArrayList') and hash table
('MGcoHashTable') implementation. These containers are more general than
'IDL_Container', in that they allow elements of any IDL type instead of
just objects.

:Author:
   Michael Galloy

:Copyright:
   BSD licensed
```

The comments from the above directory overview file, along with a listing of the files in the directory, appear somewhere near the beginning of the documentation for the directory. In the HTML output, the link from the main overview page or the link in the lower-left navigation window when the directory has been selected in the upper-right navigation window lead to the directory overview page.

Comment markup

The comment markup style defines how text can be annotated. Once the format style has defined a place for putting comments for a particular item, the markup style describes the syntax of those comments.

Several markup styles are available to annotate comment text with typesetting instructions. The "verbatim" and "preformatted" markup styles are the simplest, the comments are copied straight to the documentation with the "preformatted" style displaying the comments as monospaced, plain text also. The more modern

"rst" markup style defines a simple syntax for annotating the comment text with links, images, or code samples. While the "verbatim" and "preformatted" markup styles can be useful for legacy code comments, the "rst" markup style is easier to read and is recommended for all new comments.

The *rst* markup style attempts to make its format definition similar to what someone would do normally for readability in a text document. For example, paragraphs are created by simply skipping a line:

```
; Merges a string array into a single string separated by carriage
; return/linefeeds.
;
; Defaults to use just linefeed on UNIX platforms and both carriage returns
; and linefeeds on Windows platforms unless the UNIX or WINDOWS keywords are
; set to force a particular separator.
```

There is other special syntax for some annotations that are common when documenting code. To place a block of code into the documentation, end a line with ::, skip a line, indent the block of code, and skip another line:

```
; Set the decomposed mode, if available in the current graphics device i.e.
; equivalent to::
;
;    device, get_decomposed=oldDec
;    device, decomposed=dec
;
; The main advantage of this routine is that it can be used with any graphics
; device; it will be ignored in devices which don't support it.
```

Another common annotation is to place a link in the documentation. For example, to link "http://michaelgalloy. com" to the phrase "my website", simply do:

```
; Check out 'my website <http://michaelgalloy.com>'.
```

But often, links are to other items in the documentation. For example, the comments for a routine, might briefly mention some of its keywords and it would be convenient to link to the documentation for these keywords. In this case, just put the method names in backticks like:

```
; :Returns:
;    Returns a triple as a 'bytarr(3)' or 'bytarr(3, n)' by default if a single
;    color name or n color names are given. Returns a decomposed color index
;    as a long or lonarr(n) if 'INDEX' keyword is set.
;
;    Returns a string array for the names if 'NAMES' keyword is set.
```

IDL will search for a name matching the quoted string and link to the closest one it finds. If the name is not found, as in bytarr(3) above, it will simply be displayed in a monospace space font as code.

Different level headers can be added to comments, particularly useful for *.idldoc* files. Just underline with -, =, or ~. For example, the following beginning to an *.idldoc* file, creates a level 1 header "TxDAP API Introduction", with a level 2 header "Basic Use" immediately after:

```
TxDAP API Introduction
=======================


Basic Use
---------
```

The order of use of the underlining determines the level of the header: the first underlined header is assumed to be level 1. The second, unless it is the same as the first, is assumed to be level 2, etc. From then on, titles underlined with "=" are level 1 headers and those underlined with "-" are level 2 headers.

Directives provide a more general markup syntax. Currently, there are three directives defined:

1. image directive

2. embed directive

3. title directive

The "image" directive allows images to be placed into comments. To use, put the following on the end of a line:

```
.. image:: filename
```

where *filename* is any image file format read by *READ_IMAGE*. The *filename* specified will be copied into the output directory.

The "embed" directive allows *.svg* files to be embedded in the documentation. To use, put the following on the end of a line:

```
.. embed:: filename
```

The "title" directive is available to provide a title for *.idldoc* files:

```
.. title:: cpt-city color tables
```

This title is used for the *.idldoc* file in the table of contents of available documentation.

IDLdoc options

The keywords used when IDLdoc is run provide some options in the type of output produced.

The *USER* keyword specifies whether "user" or "developer" documentation is produced. User documentation is appropriate for users of a library. Directories, files, routines, and keywords/parameters can be marked

to not show up in user documentation by using the "Private" tag. For example, the *MG_H5_DUMP* routine has a few helper routines that are not intended for end users to call:

```
;+
; Return a string representing an IDL declaration of the given item
; (attribute or dataset).
;
; :Private:
;
; :Returns:
;    string
;
; :Params:
;    typeId : in, required, type=long
;       type identifier
;    spaceId : in, required, type=long
;       dataspace identifier
;-
function mg_h5_dump_typedecl, typeId, spaceId
```

Individual keywords or parameters use a attribute to mark it as private. For instance, the *MG_STREPLACE* has a private keyword *START* that is not intended for users of the library routine, but is used by internal calls to the routine. The keyword's documentation is:

```
;    start : out, optional, type=integral, default=0, private
;       index into string of where to start looking for the pattern
```

Developer documentation is the default and will show items marked as private (though there is a "Hidden" tag for not showing an item in any documentation).

When producing HTML documentation, there are often two cases that need to be handled:

1. documentation served on a web site and intended to be served as a full collection

2. documentation pages intended to be handed out individually, e.g., giving someone a *.pro* file and its generated HTML documentation file

In the later case, it is often useful to set the *EMBED* and *NONAVBAR* keywords. The *EMBED* keyword embeds the, rather large, CSS file into each HTML page. This is inefficient for a full documentation set on a web site because in that situation, each page can just refer to a common *.css* file. The *NONAVBAR* keyword simply omits the navigation bar at the top of the page which is not needed when only one HTML page is given but useful to navigate a full documentation set.

The *FOOTER* keyword can specify a file to include at the bottom of each page of output. This file is included verbatim in the output, so it should be already be in the format of the output.

By default, IDLdoc will copy the source code and put a link to it in the output. Use the *NOSOURCE*

keyword to indicate that source code should not be copied or linked to. If the source code should be linked to, but not copied use *SOURCE_LINK* to specify relative (`SOURCE_LINK=1`) or absolute (`SOURCE_LINK=2`) links.

If the *STATISTICS* keywords is set, IDLdoc will compute certain measures of the code's complexity like the number of lines in a routine and the cyclomatic complexity. Use the *COMPLEXITY_CUTOFFS* and *ROUTINE_LINE_CUTOFFS* to specify to 2-element arrays which specify the warning and flagged levels. The defaults are `[10, 20]` for *COMPLEXITY_CUTOFFS* and `[75, 150]` for *ROUTINE_LINE_CUTOFFS*.

References

The project site for IDLdoc, idldoc.idldev.com, contains more information about IDLdoc including the FAQ, the mailing list, ticket system, and downloads of all versions along with their release notes.