logging: structured logging for gretl

Artur Tarassow

July 8, 2021

1 Introduction

This addon provides structured logging for gretl; that is, a means of recording the history and progress of a computation as a log of events.

There are in principle two roles involved in logging—the *coder* (the writer of a hansl script or function package) and the *user* (the person running the script or package)—although one person may play both roles.

The *coder* gets to decide which events will be logged, and the importance or "level" to be assigned to each event. This is done by means of the logging functions `Debug`, `Info`, `Warn`, `Error` and `Critical` (in increasing order of importance). Each of these functions requires a single string argument and offers no return value. For example, the signature of `Info` is

```
void Info (const string msg)
```

The coder must include the following statement prior to calling these functions:

```
include logging.gfn
```

The *user* determines which log messages will be shown, by selecting a threshold: print only messages of a specified level or above. This is done via the command

```
set loglevel <level>
```

where `<level>` can be given by number or name, as shown below.

number	name	associated function
0	`debug`	`Debug`
1	`info`	`Info`
2	`warn`	`Warn`
3	`error`	`Error`
4	`critical`	`Critical`

The default level is 2 or `warn`, so messages set via the `Debug` and `Info` functions will not be printed unless the user specifies a lower threshold. A user who does not care to see warning messages can raise the threshold to 3 or `error`.

2 Remarks

Using structured logging provides some advantages over using `print` or `printf` statements:

1. It gives control over the visibility and presentation of messages without editing the source code. For example, the code

```
Debug("This is a debugging message")
```

will produce no output by default; such messages are printed only if the user selects a verbose level of logging.

2. It's cheap to leave debugging statements like this in the source code: the program evaluates the message only if it is currently called for.
3. Log messages can have timestamps, and can be written to a separate file which can be analysed afterwards. More on these points below.

Note that the message passed to a logging function does not have to a fixed piece of text. You can incorporate current state information by means of the `sprintf` function, as in this example

```
Warn(sprintf("The matrix X looks funny:\n%12g\n", X))
```

which prints the elements of `X` following the message.

The table below may be helpful in determining which level of logging to use for which purpose.[1]

Debug	Detailed information, typically of interest only when diagnosing problems.
Info	Confirmation that things are working as expected.
Warn	An indication that something unexpected happened, or indicative of some problem in the near future (e.g. "disk space low"). The software is still working as expected.
Error	Due to a more serious problem, the software has not been able to perform some function.
Critical	A serious error, indicating that the program itself may be unable to continue running.

3 Timestamps

Optionally, the user can arrange for each logging message to show a timestamp. This is achieved via the command

```
set logstamp on
```

And timestamps can be turned off via "`set logstamp off`".

Suppose a function contains the following statement, triggered when an argument `x` is negative:

```
Warn("x is negative")
```

Without a timestamp the output will be

```
WARNING: x is negative
```

With a timestamp it will resemble the following, showing date, time and time-zone:

```
WARNING 2021-07-08 10:26:44 EDT: x is negative
```

4 Logging to file

By default log messages are printed to the same place (window, file, or whatever) as regular program output. But the `set` variable `logfile` can be used to redirect logging output. For example, if you specify

```
set logfile "mylog.txt"
```

logging output will go `mylog.txt`. Note that when a simple filename is given, as above, the file will be written in the user's working directory. To take control over its location you can supply a full path. You can also specify the "file" as `stdout` or `stderr` (without quotes) to send logging to the standard output or standard error streams, respectively.

[1] It is borrowed from https://docs.python.org/3/howto/logging.html.

5 A simple example

Listing 1 illustrates usage on the part of both coder (in the function `testlog`) and user. You can try uncommenting the "`set`" lines in the main script to see their effect.

```
include logging.gfn

function void testlog (scalar x)
   Debug("Here in function testlog")
   Info(sprintf("testlog: x = %g", x))
   if missing(x)
       Error("x value is invalid")
   elif x < 0
       Warn("x is negative")
   endif
end function

/* main script */

# set loglevel info
# set logstamp on
# set loglevel debug
testlog(3)
testlog(-1)
testlog(NA)
```

Listing 1: Sample usage of logging functionality

6 Accessors

The settings of `loglevel`, `logstamp` and `logfile` can be accessed via `$loglevel`, `$logstamp` and `$logfile`, respectively. The first two accessors return a numerical value (0/1 for `logstamp`); `$logfile` returns an empty string if redirection is not set. However, these accessors are basically internals of the addon, unlikely to be of interest to its users.

7 Changelog

2021-07-08 Initial version.