Lecture #11: logging

Presented by Jamal Bouajjaj

2023–10–02

For University of New Haven's Fall 2023 CSCIxx51 Course

Logging Stuff

Let's say you have a functional application. Cool!

Logging Stuff

Let's say you have a functional application. Cool!

Now what if during run-time an error happended? Wouldn't you like to know what occured? Both during debugging and deployment?

Naive Logging

The following is a simple way to "log" stuff to the terminal. The simple print statement.

```
a = some_function(5, 2)
print(f"a is {a}")
try:
  do_something_else(a, 50)
except ValueError as e:
  print("Exception here! ", e)
```

logging Module

While the above could be sufficient, there are a couple of issues/downsides:

- If this is a CLI application, the user application will have print-outs that aren't relavent to what the user needs.
- There is no distunction between important errors and just debug logs.
- What is printed does not get sent to a file, allowing loss of information with no way of getting it.
- If a GUI application, there will be no console. Same as above
- Exception printing is limited

Solutions

Enough rambling, what is a viable solution?

Solutions

Enough rambling, what is a viable solution?

- Force the user to open a command-line when running the application, and requiring them to copy the output if there is an error

Solutions

Enough rambling, what is a viable solution?

- Force the user to open a command-line when running the application, and requiring them to copy the output if there is an error
- Give up!

Solutions

Enough rambling, what is a viable solution?

- Force the user to open a command-line when running the application, and requiring them to copy the output if there is an error
- Give up!
- Ignore any errors, they aren't important.

Solutions

Enough rambling, what is a viable solution?

- Force the user to open a command-line when running the application, and requiring them to copy the output if there is an error
- Give up!
- Ignore any errors, they aren't important.
- Use the *logging* module

logging

logging

logging is a standard module that is designed to handle logging stuff in your application.

Logger

A Logger is an object that handles logging. There is always a root logger (top level), and you can create a logging handler. Each logger can have it's own "settings", as in level and handlers.

Getting a logger with the same name will return the same logger.

A logger name is hierarchal, so root is the top level, then all other loggers are child loggers. A dot indicates a child of the logger name.

```
l = logging.getLogger() # root logger
l = logging.getLogger('sensor')   # a logger with name
l = logging.getLogger('sensor.ser')  # a logger with
↪  name, child of sensor
```

Log Levels

There are several levels of log message withing the module[1]:

- NOSET (placeholder for nothing set, not a real level)
- DEBUG
- INFO
- WARNING
- ERROR
- CRITICAL

Any level less than the set level is ignored, either by the logger object itself or a handler.

A logger object by default assumes a level of WARNING.

[1] `https://docs.python.org/3/howto/logging.html`

Level Example

This is how to log with levels. The logger itself must have a level set unless you want WARNING or above to be logged.

```
l = logging.getLogger() # root logger
l.setLevel(logging.DEBUG) # set logger level
l.debug("Message")
l.info("Message")
l.warning("Message")
l.error("Message")
l.critical("Message")
# calls to logging (the module) directly will just log
↪  with the root logger
logging.debug("Message")
```

Exception Logging

You can also log exceptions, which will also record the traceback! The level for exception logs is ERROR.

```
l = logging.getLogger() # root logger
try:
  a = 5 / 0
except:
  logging.exception("Error!")
```

Log Handlers

Once a log is emmited (i.e sent), where does it get sent? That is determined by what handlers exist for the logger.

The handler just handles what to do with a log.

Each handler can have a level: anything below the set level is ignored for the handler.

The handler is applied to a logger on a hierarchal basis. So a handler for the root logger will get called for all child loggers, etc.

Log Handlers

```
l = logging.getLogger()
l.setLevel(logging.DEBUG) # set the logger's level
hs = logging.StreamHandler()
hs.setLevel(logging.WARNING)
l.addHandler(hs)
fs = logging.FileHandler("file.log")
hs.setLevel(logging.DEBUG)
l.addHandler(fs)
```

Log Format

You can change what each log handler formats it's output. You can for example have a log output the function name, time, message, etc.

```
l = logging.getLogger()
l.setLevel(logging.DEBUG) # set the logger's level
hs = logging.StreamHandler()
hs.setLevel(logging.WARNING)
f = logging.Formatter("%(asctime)s
↪  %(levelname)s:%(filename)s:%(funcName)s:%(lineno)s
↪  %(message)s")
hs.setFormatter(f)
l.addHandler(hs)
```

Logging Without Setup!

logging can have a decent amount of setup for it. If you just want something to just log in the strem, and maybe file, you can have logging setup a lot for you by using the following function:

```
logging.basicConfig(level=logging.DEBUG)
logging.basicConfig(level=logging.DEBUG,
↪  filename="program.log")
```

End

The end