the joy of logging

Mike Pirnat

Clepy: 9/11/2006

Ob. Monty Python

I'm a lumberjack and I'm okay,
I sleep all night and I work all day!

Logging

- What is your program is doing when you're not looking?
- Especially helpful for long-running processes
- You might log:
  - Events
  - Errors
  - Warnings
  - Debugging information

Primitive Logging

```
...
print time.localtime() + \
    “\thello world”
...

$ python foo.py > foo.log 2>&1
```

Primitive Logging

```
...
log = open('foo.log', 'a')
log.write(time.localtime() + \
    '\thello world\n')
log.close()
...

$ python foo.py &
```

DIY Logging Frameworks

- Primitive logging sucks
- Might inspire a DIY solution...
- ...but it probably already inspired at least two or three others in your organization!
- Almost as embarassing as writing your own web framework

Desirable Features

- Filtering/levels
- Formatting
- Output management
- Output options (more than stdout and file logging please!)

Python standard library to the rescue!

- logging.py
- New in Python 2.3
- Super-flexible
- Features galore (too many to cover them all here)

Log Levels

- Five levels of information:

– `CRITICAL`	`50`
– `ERROR`	`40`
– `WARNING`	`30`
– `INFO`	`20`
– `DEBUG`	`10`
– `NOTSET`	`0`

- Convenience methods correspond to logging at these levels
- Plus you can define your own levels

Basic Logging

- Logging messages are sent to a special `Logging` object called the *root logger*
- By default, only handles `WARNING` and above
- Messages sent to `sys.stderr` or writen to a file

Basic Logging Methods

- critical(*fmt* [, **args* [, exc_info]])
  - Logs at the `CRITICAL` level on the root logger
  - `fmt` is a format string
  - Any remaining args apply to format specifiers in the format string
  - If kwarg `exc_info` is `True`, or an exception tuple (`sys.exc_info`), exception info is also logged

Basic Logging Methods

- error(*fmt*[, **args* [, exc_info]])
- exception(*fmt*[, *args])
  - Includes exception info
  - Can only be used inside an exception handler
- warning(*fmt*[, **args* [, exc_info]])
- info(*fmt*[, **args* [, exc_info]])
- debug(*fmt*[, **args* [, exc_info]])
- log(*level*, *fmt*[, **args* [, exc_info]])

Basic Configuration

- basicConfig([**kwargs])
  - `filename`
  - `filemode`
  - `format`
  - `datefmt`
  - `level`
  - `stream`

Basic Formatting

- Lots of stuff you can use in your format string:
  - `%(name)s`
  - `%(levelno)s`
  - `%(levelname)s`
  - `%(pathname)s`
  - `%(filename)s`
  - `%(module)s`
  - `%(lineno)d`

Basic Formatting

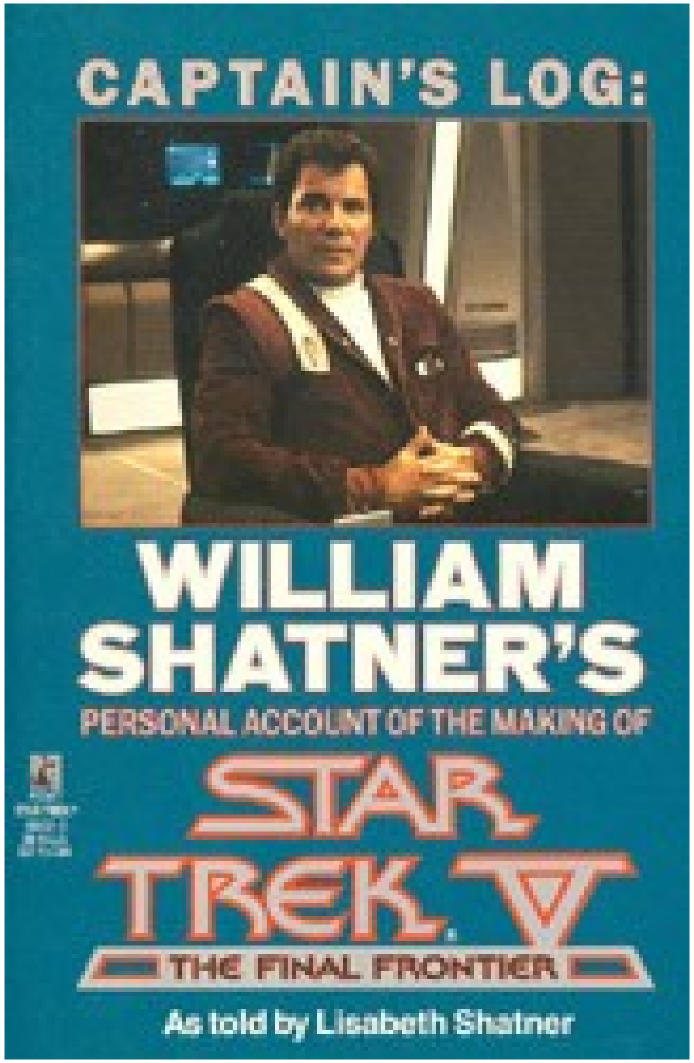
- More stuff for your format string:
  - `%(created)f`
  - `%(asctime)s`
  - `%(msecs)s`
  - `%(thread)d`
  - `%(threadName)s`
  - `%(process)d`
  - `%(message)s`

Basic Example

```
import logging
logging.basicConfig(
    filename="foo.log",
    format="%(levelname)-10s %(asctime)s"\
    "%(message)s",
    level=logging.DEBUG)
logging.debug("Debugging info")
logging.info("Something wonderful is about to" \
    "happen...")
logging.critical("I have a bad feeling about this")
```

```
DEBUG      2006-08-06 23:09:43,570 Debugging info
INFO       2006-08-06 23:09:43,574 Something
wonderful is about to happen...
CRITICAL   2006-08-06 23:09:44,348 I have a bad
feeling about this
```

Ob. Shatner


CAPTAIN'S LOG:
WILLIAM SHATNER'S
PERSONAL ACCOUNT OF THE MAKING OF
STAR TREK V
THE FINAL FRONTIER
As told by Lisabeth Shatner

Customized Logging

- Create a `Logger` object and customize it as needed
- Call methods on this `Logger` instance instead of the `logging` module

Customized Logging

- `getLogger(`*`logname`*`)`
  - `logname` specifies a name or series of names (`'foo'` or `'foo.bar.spam'`)
  - If `logname` is `''`, you get the root logger
  - If no `Logger` named *`logname`* exists, creates and returns a new logger
  - If a `Logger` named *`logname`* already exists, returns the existing instance

Logger Methods

- L.critical(_fmt_ [, _*args_ [, exc_info]])
- L.error(_fmt_ [, _*args_ [, exc_info]])
- L.exception(_fmt_ [, _*args_])
- L.warning(_fmt_ [, _*args_ [, exc_info]])
- L.debug(_fmt_ [, _*args_ [, exc_info]])
- L.log(_level_, _fmt_ [, _*args_
  [, exc_info]])

Logger Attrs & Methods

- `L.propagate`
  - If `L` has the name `'foo.bar.spam'`, and this is `True`, messages to `L` will also be sent to the logger with name `'foo.bar'`
- `L.setLevel(`*`level`*`)`
- `L.isEnabledFor(`*`level`*`)`
- `L.getEffectiveLevel()`
  - Level set by `setLevel()`
  - Or parent logger's `getEffectiveLevel()`
  - Or root logger's effective level

Logger Methods

- `L.addFilter(`*`filter`*`)`
- `L.removeFilter(`*`filter`*`)`
- `L.filter(`*`record`*`)`
  - `record` is a `LogRecord` instance
  - Returns `True` if the message would be processed

Logger Methods

- `L.addHandler(`*`handler`*`)`
- `L.removeHandler(`*`handler`*`)`
- `L.handle(`*`record`*`)`
  - Dispatch a `LogRecord` to all handlers registered with this `Logger`
- `L.findCaller()`
  - Returns a tuple:

```
    (filename, lineno)
```

LogRecord

- Internal implementation of the contents of a logging message
- `LogRecord(`*`name`*`, `*`level`*`,`
  `pathname``, `*`line`*`, `*`msg`*`,`
  `args``, `*`exc_info`*`)`
- `r.getMessage()`
- `makeLogRecord(`*`attrdict`*`)`

Handlers

- Process log messages
- Attach to your `Logger` using its `addHandler()`
- Attach any number of different handlers
- All are `Handler` subclasses
- Some default handlers are in `logging`; others in `logging.handlers`

Handlers

- `handlers.DatagramHandler(`*`host`*`, `*`port`*`)`
  - Sends log messages to a UDP server as pickled `LogRecords`
  - Delivery is not guaranteed
- `FileHandler(`*`filename`*` [, `*`mode`*`])`
  - Write log messages to a file
- `handlers.HTTPHandler(`*`host`*`, `*`url`*` [, `*`method`*`])`
  - Upload log messages to an HTTP server using `GET` or `POST`

Handlers

- `handlers.MemoryHandler(` *`capacity`* `[,` *`flushLevel`* `[,` *`target`*`]])`
  - Collect messages in memory and flush them to another handler (*`target`*) when we hit *`capacity`* (bytes) or see a message of level *`flushLevel`*
- `handlers.NTEventLogHandler( appname [, dllname [, logtype]])`
  - Only available if Win32 extensions for Python have been installed

Handlers

- `handlers.RotatingFileHandler(` *`filename`* `[,` *`mode`* `[,` *`maxBytes`* `[,` *`backupCount`*`]]])`
- `handlers.SMTPHandler(`*`mailhost`*`,` *`fromaddr`*`,` *`toaddrs`*`,` *`subject`*`)`
- `handlers.SocketHandler(` *`host`*`,` *`port`*`)`
  - The TCP version of `DatagramHandler`
  - Delivers reliably

Handlers

- `StreamHandler([`*`fileobj`*`])`
  - Default handler for the root logger
- `handlers.SysLogHandler(`
  `[`*`address`*` [, `*`facility`*`]])`
  - *`address`* is a tuple `(host, port)`; defaults to `('localhost', 514)`
  - *`facility`* is an integer facility code (see `SysLogHandler`'s code for a full list)

Handlers

- `handlers.TimedRotatingFileHandler(` *`filename`* `[,` *`when`* `[,` *`interval`* `[,` *`backupCount`*`]]])`
  - Like `RotatingFileHandler`, but time-based
  - *`interval`* is a number
  - *`when`* is a string: `'S'`econds, `'M'`inutes, `'H'`ours, `'D'`ays, `'W'`eeks, `'midnight'`

Handler Methods

- For threaded environments:
  - `h.createLock()`
  - `h.acquire()`
  - `h.release()`
- `h.setLevel(`*`level`*`)`
- `h.setFormatter(`*`formatter`*`)`

Handler Methods

- `h.addFilter(`*`filter`*`)`
- `h.removeFilter(`*`filter`*`)`
- `h.filter(`*`record`*`)`
- `h.handle(`*`record`*`)`
  - Applies filters, deals with locking, and emits the message
- `h.handleError(`*`record`*`)`
  - Used when an error occurs during normal handling; does nothing by default

Handler Methods

- `h.format(`*`record`*`)`
- `h.emit(`*`record`*`)`
  - Unlike `handle()`, just emits without locking
- `h.flush()`
- `h.close()`

Filters

- Filter messages using a method other than log level
- Basic logname-based filter is provided by the logging module:
  - `Filter([`*`name`*`])`
  - `f.filter(`*`record`*`)`

Formatters

- Perform the formatting of log messages
- Attach to a handler using its `setFormatter()`
- Subclass and modify if special formatting is required

Formatter Methods

- `Formatter([`*`fmt`*` [, `*`datefmt`*`]])`
- `f.format(`*`record`*`)`
- `f.formatTime(`*`record`*` [, `*`datefmt`*`])`
- `f.formatException(`*`exc_info`*`)`

Utility Functions

- `disable(`*`level`*`)`
  - Globally disable logging of all messages below *`level`*
- `addLevelName(`*`level, levelName`*`)`
  - Create a new logging level
- `getLevelName(`*`level`*`)`
  - Returns the name of the level associated with the numeric level
- `shutdown()`
  - Flush and shut down all logging objects

Custom Logging Examples

- Four basic steps:
  - Use `getLogger()` to create a `Logger` and establish a name
  - Create a `Handler` object
  - Create a `Formatter` and attach it to the `Handler`
  - Attach the `Handler` to the `Logger`

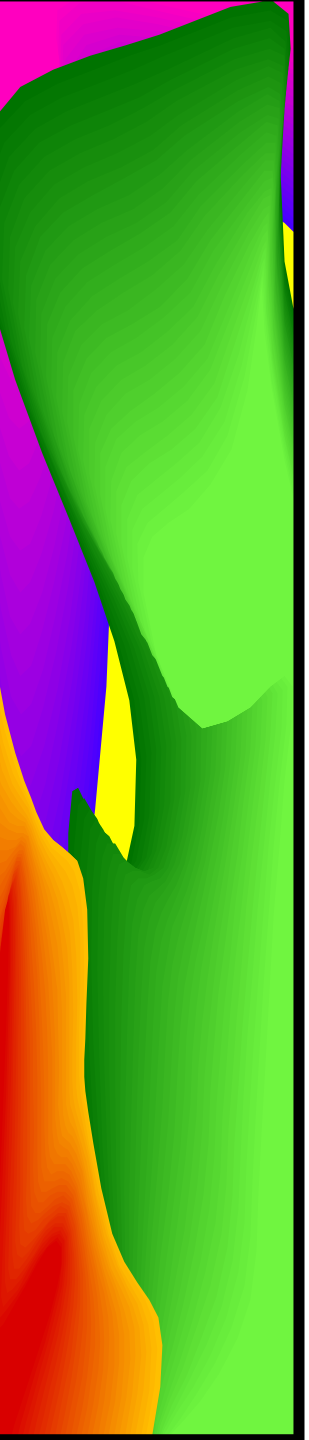
Example: Logging to Rotating Files

```
import logging
import logging.handlers

log1 = logging.getLogger('mondo')
log1.setLevel(logging.INFO)

h = logging.handlers.RotatingFileHandler(
    'mondo.log', 'a', 100000, 4)
f = logging.Formatter('%(levelname)-10s '\
    '%(name)-12s %(asctime)s %(message)s')
h.setFormatter(f)
log1.addHandler(h)

log1.info('MONDO application starting up')
log1.warning('MONDO flag not set')
```

Example: Multiple Destinations

- We might want to handle critical errors specially...

```
crithand = logging.StreamHandler(sys.stderr)
crithand.setLevel(logging.CRITICAL)
crithand.setFormatter(f)
log1.addHandler(crithand)
```



Example: Multiple Loggers and Message Propagation

- Does our app have many components? Might want to divide logging into multiple loggers...

```
netlog = logging.getLogger('mondo.net')
netlog.info(“Networking on port %d”, port)
```

- Logging messages issued on 'mondo.net' will propagate up to loggers defined for 'mondo'... So the `mondo.log` will have:

```
CRITICAL   mondo          2006-08-07 00:50:17,900
   MONDO OVERLOAD!
INFO       mondo.net      2006-08-07 00:50:17,905
   networking on port 31337
```

Multiple Loggers & Message Propagation

- We can define more handlers for 'mondo.net'; eg, if we wanted to log network messages to a file:

```
nethand = logging.FileHandler('mondo.net.log')
nethand.setLevel(logging.DEBUG)
nethand.setFormatter(f)
netlog.addHandler(nethand)
```

- Now messages sent to `netlog` will be written to `'mondo.net.log'` and to `'mondo.log'`; critical messages will go to both places and be displayed on `sys.stderr`

Logging Tips

- There are a lot more customization options – check out the online docs
- Use `getLogger()` to avoid having to pass log objects around
- In earlier versions of Python 2.3, `findCaller()` is lightly broken
  - Only unwinds the stack by 1 level instead of as many as needed
  - Chance to practice your monkeypatching skills

Ob. Ren & Stimpy

It's log, it's log,
It's big, it's heavy, it's wood!
It's log, it's log,
It's better than bad, it's good!

I'm sold! What now?

- Read the documentation:

  http://docs.python.org/lib/module-logging.html

- Adapted from

  Python Essential Reference, Third Edition by David M. Beazley

- This presentation available at:

  http://www.pirnat.com/geek/