Introduction to Logging

David Beazley
Copyright (C) 2008
http://www.dabeaz.com

Note: This is a supplemental subject component to Dave's Python training classes. Details at:
http://www.dabeaz.com/python.html

Last Update: March 22, 2009

Application Logging

• Complex systems are often instrumented with some kind of logging capability
  • Debugging and diagnostics
  • Auditing
  • Security
  • Performance statistics
• Example: Web server logs
Error Handling

• Even in small programs, you often run into subtle issues related to exception handling

```python
for line in open("portfolio.dat"):
    fields = line.split()
    try:
        name = fields[0]
        shares = int(fields[1])
        price = float(fields[2])
    except ValueError:
        ????
Do you print a warning?
print "Bad line", line
Do you ignore the error?
pass
```

• Short answer: It depends

The Logging Problem

• Logging is a problem that seems simple, but usually isn't in practice.

• Problems:
  • Figuring out what to log
  • Where to send log data
  • Log data management
  • Implementation details
Homegrown Solutions

• Observation: Most significant applications will implement some kind of logging facility.
• Homegrown solutions tend to grow into a huge tangled hack.
• No one actually wants to work on logging---it's often added as a half-baked afterthought.

logging Module

• A module for adding a logging capability to your application.
• A Python port of log4j (Java/Apache)
• Highly configurable
• Implements almost everything that you could possibly want for a logging framework
Logger Objects

- logging module relies on "Logger" objects
- A Logger is a target for logging messages
- Routes log data to one or more "handlers"

Logging Big Picture

- Creating Logger objects
- Sending messages to a Logger object
- Attaching handlers to a Logger
- Configuring Logger objects
Getting a Logger

• To create/fetch a Logger object

```python
log = logging.getLogger("logname")
```

• If this is the first call, a new Logger object is created and associated with the given name

• If a Logger object with the given name was already created, a reference to that object is returned.

• This is used to avoid having to pass Logger objects around between program modules.

Logging Messages

• The logging module defines 5 logging "levels"

<table>
<thead>
<tr>
<th>Level</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRITICAL</td>
<td>50</td>
<td>Critical errors</td>
</tr>
<tr>
<td>ERROR</td>
<td>40</td>
<td>Error messages</td>
</tr>
<tr>
<td>WARNING</td>
<td>30</td>
<td>Warning messages</td>
</tr>
<tr>
<td>INFO</td>
<td>20</td>
<td>Information messages</td>
</tr>
<tr>
<td>DEBUG</td>
<td>10</td>
<td>Debugging messages</td>
</tr>
</tbody>
</table>

• A logging message consists of a logging level and a logging message string

```python
(ERROR,"message")
```
How to Issue a Message

• Methods for writing to the log

```python
log.critical(fmt [, *args ])
log.error(fmt [, *args ])
log.warning(fmt [, *args ])
log.info(fmt [, *args ])
log.debug(fmt [, *args ])
```

• These are always used on some Logger object

```python
import logging
log = logging.getLogger("logname")
log.info("Hello World")
log.critical("A critical error occurred")
```

Logging Messages

• Logging functions work like printf

```python
log.error("Filename '%s' is invalid", filename)
log.error("errno=%d, %s", e.errno,e.strerror)
log.warning("'%s' doesn't exist. Creating it", filename)
```

• Here is sample output

Filename 'foo.txt' is invalid
errno=9, Bad file descriptor
'out.dat' doesn't exist. Creating it
Logging Exceptions

- Messages can optionally include exception info
  
  ```python
  try:
  ...  
  except RuntimeError:
  log.error("Update failed", exc_info=True)
  ```

- Will include traceback info from the current exception (if any)

- Sample output with exception traceback
  
  ```
  Update failed
  Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
  RuntimeError: Invalid user name
  ```

Example Usage

- Logging often gets added as an optional feature
  
  ```python
def read_portfolio(filename, log=None):
    for line in open(filename):
      fields = line.split()
      try:
        name = fields[0]
        shares = int(fields[1])
        price = float(fields[2])
      except ValueError:
        if log:
          log.warning("Bad line: %s", line)
  ```

- By doing this, the handling of the log message becomes user-configurable

- More flexible than just hard-coding a print
Log Handlers

- A Logger object only receives messages
- It does not produce any output
- Must use a handler to output messages

Attaching a Handler

- Example of attaching a handler to a Logger

```python
import logging, sys

# Create a logger object
log = logging.getLogger("logname")

# Create a handler object
stderr_hand = logging.StreamHandler(sys.stderr)

# Attach handler to logger
log.addHandler(stderr_hand)

# Issue a log message (routed to sys.stderr)
log.error("An error message")
```
Attaching Multiple Handlers

- Sending messages to stderr and a file

```python
import logging, sys

# Create a logger object
log = logging.getLogger("logname")

# Create handler objects
stderr_hand = logging.StreamHandler(sys.stderr)
logfile_hand = logging.FileHandler("log.txt")

# Attach the handlers to logger
log.addHandler(stderr_hand)
log.addHandler(logfile_hand)

# Issue a log message. Message goes to both handlers
log.error("An error message")
```

Handler Types

- There are many types of handlers

```python
logging.StreamHandler
logging.FileHandler
logging.handlers.RotatingFileHandler
logging.handlers.TimedRotatingFileHandler
logging.handlers.SocketHandler
logging.handlers.DatagramHandler
logging.handlers.SMTPHandler
logging.handlers.SysLogHandler
logging.handlers.NTEventLogHandler
logging.handlers.MemoryHandler
logging.handlers.HTTPHandler
```

- Consult a reference for details.
- More examples later
Level Filtering

• All messages have numerical "level"
  
  50  CRITICAL
  40  ERROR
  30  WARNING
  20  INFO
  10  DEBUG
  0   NOTSET

• Each Logger has a level filter
  
  log.setLevel(logging.INFO)

• Only messages with a level higher than the set level will be forwarded to the handlers

Level Filtering

• All Handlers also have a level setting

  stderr_hand = logging.StreamHandler(sys.stderr)
  stderr_hand.setLevel(logging.INFO)

• Handlers only produce output for messages with a level higher than their set level

• Each Handler can have its own level setting
Level Filtering

- Big picture: Different objects are receiving the messages and only responding to those messages that meet a certain level threshold.
- Logger level is a "global" setting.
- Handler level is just for that handler.

Level Filtering Example

```python
# Create handler objects
stderr_hand = logging.StreamHandler(sys.stderr)
stderr_hand.setLevel(logging.CRITICAL)
logfile_hand = logging.FileHandler("log.txt")
logfile_hand.setLevel(logging.DEBUG)
```

Copyright (C) 2008, http://www.dabeaz.com

Level Filtering

Logger

Handler

Output

messages

level

messages

level

level

level

level

level
Advanced Filtering

- All log messages can be routed through a filter object.

```python
# Define a filter object. Must implement .filter()
class MyFilter(logging.Filter):
    def filter(self, logrecord):
        # Return True/False to keep message
        ...

# Create a filter object
myfilt = MyFilter()

# Attach it to a Logger object
log.addFilter(myfilt)

# Attach it to a Handler object
hand.addFilter(myfilt)
```

- Filter objects receive a LogRecord object

  ```python
class MyFilter(logging.Filter):
    def filter(self, logrecord):
        ...
  ```

- LogRecord attributes

  ```python
  logrecord.name             Name of the logger
  logrecord.levelno          Numeric logging level
  logrecord.levelname        Name of logging level
  logrecord.pathname         Path of source file
  logrecord.filename         Filename of source file
  logrecord.module           Module name
  logrecord.lineno           Line number
  logrecord.created          Time when logging call executed
  logrecordasctime           ASCII-formatted date/time
  logrecord.thread           Thread-ID
  logrecord.thraddename     Thread name
  logrecord.process          Process ID
  logrecord.message          Logged message
  ```
Filtering Example

- Only produce messages from a specific module

```python
class ModuleFilter(logging.Filter):
    def __init__(self, modname):
        logging.Filter.__init__(self)
        self.modname = modname
    def filter(self, logrecord):
        return logrecord.module == self.modname

log = getLogger("logname")
modfilt = ModuleFilter("somemod")
log.addFilter(modfilt)
```

Multiple Filters

- Multiple Filters may be added
  ```
  log.addFilter(f)
  log.addFilter(g)
  log.addFilter(h)
  ```

- Messages must pass all to be output
- Filters can be removed later
  ```
  log.removeFilter(f)
  ```
Log Message Format

• By default, log messages are just the message
  
  ```python
  log.error("An error occurred")
  ```

  An error occurred

• However, you can add more information
  
  • Logger name and level
  
  • Thread names
  
  • Date/time

Customized Formatters

• Create a Formatter object
  
  ```python
  # Create a message format
  msgform = logging.Formatter(
      "%(levelname)s:%(name)s:%(asctime)s:%(message)s"
  )

  # Create a handler
  stderr_hand = logging.StreamHandler(sys.stderr)
  
  # Set the handler's message format
  stderr_hand.setFormatter(msgform)
  ```

  • Formatter determines what gets put in output

  ```python
  ```
**Message Format**

- **Special format codes**
  
  ```
  %(name)s                Name of the logger
  %(levelname)s           Name of logging level
  %(pathname)s            Path of source file
  %(filename)s            Filename of source file
  %(module)s              Module name
  %(lineno)d              Line number
  %(asctime)s             ASCII-formated date/time
  %(created)f             Time when logging call executed
  %(asctime)s             ASCII-formated date/time
  %(thread)d              Thread-ID
  %(threadName)s          Thread name
  %(process)d             Process ID
  %(message)s             Logged message
  ```

- **Information is specific to where logging call was made (e.g., source file, line number, etc)**

---

**Message Formatting**

- Each Handler has a single message formatter
- Use `setFormatter()` to set it
  ```python
  hand.setFormatter(form)
  ```
- Different handlers can have different message formats
Multiple loggers

• An application may have many loggers
• Each is identified by a logger name

```python
netlog = logging.getLogger("network")
guilog = logging.getLogger("gui")
thrlog = logging.getLogger("threads")
```

• Each logger object is independent
• Has own handlers, filters, levels, etc.

Hierarchical Loggers

• Loggers can be organized in a name hierarchy

```python
applog = logging.getLogger("app")
netlog = logging.getLogger("app.network")
conlog = logging.getLogger("app.network.connection")
guilog = logging.getLogger("app.gui")
thrlog = logging.getLogger("app.threads")
```

• Messages flow up the name hierarchy
Hierarchical Loggers

- With a hierarchy, filters and handlers can be attached to every single Logger involved
- The level of a child logger is inherited from the parent unless set directly
- To prevent message forwarding on a Logger:
  
  ```python
  log.propagate = False
  ```
- Commentary: Clearly it can get quite advanced

The Root Logger

- Logging module optionally defines a "root" logger to which all logging messages are sent
- Initialized if you use one of the following:

  ```python
  logging.critical()
  logging.error()
  logging.warning()
  logging.info()
  logging.debug()
  ```
The Root Logger

- Root logger is useful for quick solutions and short scripts

```python
import logging
logging.basicConfig(
    level    = logging.INFO,
    filename = "log.txt",
    format   = "%(levelname)s:%(asctime)s:%(message)s"
)
logging.info("My program is starting")
...
```

Putting it Together

- Adding logging to your program involves two steps
  - Adding support for logging objects and adding statements to issue log messages
  - Providing a means for configuring the logging environment
  - Let's briefly talk about the second point
Logging Configuration

• Logging is something that frequently gets reconfigured (e.g., during debugging)

• To configure logging for your application, there are two approaches you can take
  • Isolate it to a well-known module
  • Use config files (ConfigParser)

A Sample Configuration

• A sample configuration module

```python
# logconfig.py
import logging, sys

# Set the message format
format = logging.Formatter("%(levelname)-10s %(asctime)s %(message)s")

# Create a CRITICAL message handler
crit_hand = logging.StreamHandler(sys.stderr)
crit_hand.setLevel(logging.CRITICAL)
crit_hand.setFormatter(format)

# Create a handler for routing to a file
applog_hand = logging.FileHandler('app.log')
applog_hand.setFormatter(format)

# Create a top-level logger called 'app'
app_log = logging.getLogger("app")
appl_log.setLevel(logging.INFO)
appl_log.addHandler(applog_hand)
appl_log.addHandler(crit_hand)
```
A Sample Configuration

- To use the previous configuration, you import
  ```python
  # main.py
  import logconfig
  import otherlib
  ...
  ```

- Mainly, you just need to make sure the logging gets set up before other modules start using it

- In other modules...
  ```python
  import logging
  log = logging.getLogger('app')
  ...
  log.critical("An error occurred")
  ```

Using a Config File

- You can also configure with an INI file
  ```ini
  ; logconfig.ini
  [loggers]
  keys=root,app
  [handlers]
  keys=crit,applog
  [formatters]
  keys=format
  [logger_root]
  level=NOTSET
  handlers=
  [logger_app]
  level=INFO
  propagate=0
  qualname=app
  handlers=crit,applog
  [handler_crit]
  class=StreamHandler
  level=CRITICAL
  formatter=format
  args=(sys.stderr,)
  [handler_applog]
  class=FileHandler
  level=NOTSET
  formatter=format
  args=('app.log',)
  [formatter_format]
  format=%(levelname)-10s %(asctime)s %(message)s
  datefmt=
  ```
Reading a Config File

- To use the previous configuration use this

  # main.py
  import logging.config
  logging.config.fileConfig('logconfig.ini')
  ...

- The main advantage of using an INI file is that you don't have to go in and modify your code

- Easier to have a set of different configuration files for different degrees of logging

- For example, you could have a production configuration and a debugging configuration

Summary

- There are many more subtle configuration details concerning the logging module

- However, this is enough to give you a small taste of using it in order to get started