What's Logging For?

Logging provides useful data about what your application is doing.

It can be as detailed or coarse as you want, changeable at any time.

- Gives insight into **business logic** (unlike automated tracing tools)
- Valuable **telemetry** for monitoring
- Critical for **troubleshooting** and debugging

The larger your application, the more important logging becomes.

Two Ways

You can use Python's `logging` module in two broad ways.

The Basic Interface - Simpler. Useful for scripts and some mid-size applications.

Logger Objects - More complex to set up. But FAR more powerful, and invaluable with larger apps. Scales to any size.

We'll focus on the first.

The Basic Interface

The easiest way:

```
import logging
logging.warning('Look out!')
```

Save this in a script and run it, and you'll see this printed to standard error:

```
WARNING:root:Look out!
```

You call `logging.warning()`, and the output line starts with `WARNING`. There's also `error()`:

```
logging.error('Look out!')
```

```
ERROR:root:Look out!
```

Log Level Spectrum

debug: Detailed information, typically of interest only when diagnosing problems.

info: Confirmation that things are working as expected.

warning: 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.

Thresholds

Run this as a program:

```
import logging
logging.debug("Small detail. Useful for troubleshooting.")
logging.info("This is informative.")
logging.warning("This is a warning message.")
logging.error("Uh oh. Something went wrong.")
logging.critical("We have a big problem!")
```

And here's the output:

```
WARNING:root:This is a warning message.
ERROR:root:Uh oh. Something went wrong.
CRITICAL:root:We have a big problem!
```

What's missing? Why?

Log Levels

Python loggers have a *logging threshold*. And the default threshold is `logging.WARNING`.

You can change it with `logging.basicConfig()`.

```
import logging
logging.basicConfig(level=logging.INFO)
logging.info("This is informative.")
logging.error("Uh oh. Something went wrong.")
```

Run this new program, and the `INFO` message gets printed:

```
INFO:root:This is informative.
ERROR:root:Uh oh. Something went wrong.
```

Two Meanings

The phrase "log level" means two different things:

1) The **severity** of a message.

The order is `debug()`, `info()`, `warning()`, `error()` and `critical()`, from lowest to highest severity.

2) It can mean the **threshold** for ignoring messages.

Ignores everything less severe than `logging.INFO`:

```
logging.basicConfig(level=logging.INFO)
```

And this means "show me everything":

```
logging.basicConfig(level=logging.DEBUG)
```

Log Destination

By default, log messages are written to stderr.

Use `filename` to write them to a file instead:

```
# Appends messages to the file, one at a time.
logging.basicConfig(filename="log.txt")
logging.error("oops")
```

You can make your program *clobber* the log file each time by setting `filemode` to "w":

```
# Wipes out previous log entries when program restarts
logging.basicConfig(filename="log.txt", filemode="w")
logging.error("oops")
```

Or set it to "a" for "append". That's the default.

Prod Vs. Dev

```
log_file = 'myapp.log'
# mode can be set by an environment variable, command-line option, etc.
if mode == 'development':
    log_level = logging.DEBUG
    log_mode = 'w'
else:
    log_level = logging.WARNING
    log_mode = 'a'

logging.basicConfig(
    level = log_level,
    filename = log_file,
    filemode = log_mode)

logging.debug('debug message')
logging.warning('look out!')
logging.critical('we have a problem here')
```

What does this print out if mode is "development"? What if it's "production"?

Practice: `basiclog.py`

```
# Create a new file name `basiclog.py`. Type in this program:
import logging
mode = 'development'
log_file = 'myapp.log'
if mode == 'development':
    log_level = logging.DEBUG
    log_mode = 'w'
else:
    log_level = logging.WARNING
    log_mode = 'a'
logging.basicConfig(level=log_level, filename=log_file, filemode=log_mode)
logging.debug('debug message')
logging.warning('look out!')
logging.critical('we have a problem here')
```

Run to verify `myapp.log` contains `DEBUG`, `WARNING` and `CRITICAL`. Then change `mode` to "production", and re-run several times. Verify it appends only `WARNING` and `CRITICAL` to `myapp.log` each run.

EXTRA CREDIT: Make `mode` controlled by an environment variable.

Environment Switch

```
import os
mode = os.environ.get('MODE', 'production')
print('Running in mode: ' + mode)
if mode == 'development':
    # ...
```

```
$ python3 myprogram.py
Running in mode: production
```

```
$ export MODE=development
$ python3 myprogram.py
Running in mode: development
```

More here: http://powerfulpython.com/blog/nifty-python-logging-trick/

Common variation: Use a local config file.

Why not use print()?

If logging is new you to you, you may wonder why we don't just sprinkle the code with `print()` statements.

Essentially: some of those `print()` statements are more important than others. Python's logging module makes that hierarchy explicit.

And: you can change the threshold as needed.

Also: `print()` only writes to stdout. The logging module allows other destinations.

Efficiency

Freely write as many logging statements as you want, without affecting performance. Any lines below the threshold are cheaply ignored.

```
# Only log INFO messages or higher
logging.basicConfig(log_level = logging.INFO)
# Python essentially skips over this statement.
logging.debug("Received a message from X")
```

Often you will introduce many `debug()` statements while troubleshooting, and leave them in after the issue is resolved.

Interlude: String Formatting

"String Formatting" means inserting parameters into a template string at runtime, to get a final, calculated string.

Every language has it. C does it with `snprintf()`, `sprintf()`, `printf()`, etc.:

```c
int main() {
    char message[BUFFER_SIZE];
    char* template = "Your %s costs $%0.2f.";
    snprintf(message, BUFFER_SIZE, template, "cup of coffee", 1.75);
    printf("%s\n", message);
}
/*
  Output:
  Your cup of coffee costs $1.75.
 */
```

In Python, the situation with string formatting is complicated.

Modern String Formatting

In all modern versions of Python (2 and 3), you can use `str.format()`.

```
>>> template = "Your {} costs ${:0.2f}."
>>> output = template.format("cup of coffee", 1.75)
>>> print(output)
Your cup of coffee costs $1.75.
```

In 3.6 and later, you can use f-strings:

```
>>> item = "cup of coffee"
>>> price = 1.75
>>> print(f"Your {item} costs ${price:0.2f}.")
Your cup of coffee costs $1.75.
```

Percent Formatting

Python's original string interpolation. Inspired by C.

Uses `%s` for string; `%f` for float; `%d` for integer.

```
>>> template = "Your %s costs $%0.2f."
>>> output = template % ("cup of coffee", 1.75)
>>> print(output)
Your cup of coffee costs $1.75.

>>> # Or as a dictionary:
... data = {"item": "burger", "price": 6.95}
>>> print("Your %(item)s costs $%(price)0.2f." % data)
Your burger costs $6.95.
```

Largely outdated now, but still important in logging. Even in the latest Python 3.

Log Message Parameters

In many (most) log messages, you'll want to inject runtime data.

There's a good way and a bad way. The good way uses a *variant* of percent formatting:

```
logging.info("Your %s costs $%0.2f.", item, price)
```

- First argument: a percent-style template string
- 2nd and subsequent arguments: the parameters

The Bad Way

The good way again:

```
logging.info("Your %s costs $%0.2f.", item, price)
```

The bad way:

```
logging.info("Your %s costs $%0.2f." % (item, price))
```

Why is that bad?

Avoid Unnecessary Calculations

Imagine the log threshold is set to `logging.WARNING`. Then neither of these will emit any message:

```
logging.info("Your %s costs $%0.2f.", item, price)
logging.info("Your %s costs $%0.2f." % (item, price))
```

The second incurs a computational cost, to calculate the log message that's thrown away. But the first is very cheap.

This removes any hesitation to introducing log statements. Especially for INFO and lower, more tends to be better.

Why Percent Formatting?

Basically: Legacy constraints.

Original plan was to remove percent formatting completely. But it's not practical to convert old logging code to `str.format()` or f-strings.

Rather than render large groups of Python programs practically impossible to upgrade to Python 3, percent formatting is here to stay.

It's less painful to use the alternatives in NEW logging code. But I recommend you just use percent formatting.

Log Formats

Each call to `logging.debug()`, `.warning()`, etc. creates one log record - one line in the log file.

What info that record includes, and in what order, is determined by the *log format*.

```
logging.basicConfig(
    format="Log level: %(levelname)s, msg: %(message)s")
logging.warning("Collision imminent")
```

Run this as a program, and you get the following log line:

```
Log level: WARNING, msg: Collision imminent
```

Contrast with the default format:

```
WARNING:root:Collision imminent
```

Log Format Attributes

Attribute	Format	Description
asctime	`%(asctime)s`	Human-readable date/time
funcName	`%(funcName)s`	Function containing the logging call
lineno	`%(lineno)d`	The line number of the logging call
message	`%(message)s`	The log message
pathname	`%(pathname)s`	Full pathname of the source file of the logging call
levelname	`%(levelname)s`	Text logging level for the message (*DEBUG*, *INFO*, *WARNING*, *ERROR*, *CRITICAL*)
name	`%(name)s`	The logger's name

Lab: Basic Logging

Lab file: `logging/logging_basic.py`

- In labs/py3 for 3.x; labs/py2 for 2.7
- When you are done, give a thumbs up...
- ... and then do `logging/logging_basic_extra.py`

Instructions: `LABS.txt` in courseware.

Logger Objects

Richer interface to logging.

For details, read PythonLogging.pdf.

```
import logging
logger = logging.getLogger()
# StreamHandler defaults to using sys.stderr
console_handler = logging.StreamHandler()
logger.addHandler(console_handler)
# Now the file handler:
logfile_handler = logging.FileHandler("log.txt")
logger.addHandler(logfile_handler)
logger.warning("This goes to both the console, AND log.txt.")
```

Benefits of Logging

"How Logging Made me a Better Developer", by Erik Hazzard

http://vasir.net/blog/development/how-logging-made-me-a-better-developer

- Visibility into code helps manage complexity
- Visibility after shipping
- Visibility while developing
- Communication
- Explicit comments

And I'll add: Dramatically accelerates troubleshooting.

Benefits of Logging

"How Logging Made me a Better Developer", by Erik Hazzard

http://vasir.net/blog/development/how-logging-made-me-a-better-developer

- Visibility into code helps manage complexity
- Visibility after shipping
- Visibility while developing
- Communication
- Explicit comments

And I'll add: Dramatically accelerates troubleshooting.