Python Logging
(It really is better than using print statements)

Inland Empire Python Users Group
November 19, 2019

John Sheehan

Logging Basics - Video

- Purpose: Record progress and problems
- Levels: Debug, Info, Warning, Error, Critical
- Create and configure logger
    - Set file location `logging.basicConfig()`
    - Set logging level
    - Set logging format
    - Get logging instance `logger = logging.getLogger()`

Logging Basics - Why?

- Logging can be used as a diagnostic tool or for audit purposes
- Adds comments to code - debug statements can be left in
- Avoids leaving `print()` statements strewn about your code
- Gives you control over how much prints out
- Keeps a history of code operation
- Provides a way of performing forensics when code fails
- Provides a way to track code performance
- More professional and cleaner way to code
- Good habit to get into, even for small projects

- ***Important:*** Always assume log messages can become public!

  (So don't log credit card numbers, passwords, SSNs, encryption keys, etc.)

Intermediate Logging

Use logging to replace print() function calls

Instead of doing this:

```
    print("debug message")
```

Do this:

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

    logging.debug("debug message")
```

Just change this to INFO to silence all of your debug messages!

Note: Because it sets up the 'root' logger, sometimes using `logging.basicConfig()` can result in unintended consequences when logging from multiple modules.

Intermediate Logging

Logger Naming: root logger vs. named logger

Initializing this way you get the ‘root’ logger:

```
    log = logging.getLogger()   # log.name == 'root’
```

Using a named logger gives you more control:

```
    log = logging.getLogger(“MyApp”)   # log.name == 'MyApp’
```

Intermediate Logging

Unify logging across modules

Initialize the main entry point module off the root logger:

```
    log = logging.getLogger()
```

Then initializing module loggers this way uses the name of the current module:

```
    log = logging.getLogger(__name__)
```

- Logging control (log level, file location, formatting, etc.) for your entire application can then be handled in one place.
- You can still change log level of specific modules without affecting logging for the main module.

Intermediate Logging

Logging in exceptions - saving traceback output

When logging is used in an except block, the stack trace can be written to the log as well by setting the `exc_info` parameter:

```
    Except Exception:
        logging.error(“error msg”, exc_info=True)
```

Logging also has a convenience method for this purpose as well:

```
    Except Exception:
        logging.exception(“error message”)
```

Intermediate Logging

Built-in Logging Handlers

StreamHandler is the default and logs output to stdout:

```
    logging.basicConfig()
```

FileHandler is the most common way to persist logs to a file:

```
    logging.basicConfig(filename='app.log')
```

RotatingFileHandler is commonly used in production environments:

```
    log = logging.getLogger(__name__)
    f_hdlr = logging.handlers.RotatingFileHandler(
        'app.log', maxBytes=999999, backupCount=8)
    f_hdlr.setLevel(logging.ERROR)
    log.addHandler(f_hdlr)
```

Intermediate Logging

Logging Formatters

Each logging handler can have its own format:

```
logging.basicConfig(format=%(levelname)s - %(message)s)
log_fmt = %(asctime)s - %(levelname)s - %(message)s
file_hdlr.setFormatter(log_fmt)
```

Some common items to include in a log entry:

asctime - time when the LogRecord was created

levelname - text logging level for the message

funcName - name of function containing the log call

module - module (name portion of filename)

lineno - source line number where log call was issued

message - the logged message

Advanced Logging

Custom Handler: Logging to a Tk textbox

```
class WidgetLogger(logging.Handler):
    def __init__(self, widget):
        logging.Handler.__init__(self)
        self.setLevel(logging.INFO)
        self.widget = widget
        self.widget.config(state='disabled')

    def emit(self, record):
        self.widget.config(state='normal')
        # Append message (record) to the widget
        self.widget.insert(tk.END, self.format(record) + '\n')
        self.widget.see(tk.END)  # Scroll to the bottom
        self.widget.config(state='disabled')

txt_log = tk.Text(self.mainframe, name='txt_log',
yscrollcommand=scrollbar.set)
tk_handler = WidgetLogger(self.mainframe.children['txt_log'])
log.addHandler(tk_handler)
```

Advanced Logging

Custom Handler: Remote logging to AWS

Set up AWS config and credentials using ***awscli*** from PyPI:

```
    $ pip install awscli
    $ aws configure
```

Once the AWS keys are in place, adding remote logging is easy!

```
import logging
import watchtower   # pip install watchtower

log_fmt = %(asctime)s - %(levelname)s - %(message)s
logging.basicConfig(format=log_fmt)
log = logging.getLogger()
formatter = logging.Formatter(log_fmt)

aws_hdlr = watchtower.CloudWatchLogHandler(log_group='MyApp', stream_name='v1')
aws_hdlr.setFormatter(formatter)
log.addHandler(aws_hdlr)
```

Resources

- Logging in Python - Socratica Video:

  https://www.youtube.com/watch?v=g8nQ90Hk328

- Python Logging Docs:

  https://docs.python.org/3/library/logging.html

  https://docs.python.org/3/howto/logging.html

- Watchtower: Python CloudWatch Logging:

  https://watchtower.readthedocs.io/en/latest/

  https://pypi.org/project/watchtower/

Discussion