Logging and Exception Handling for Django

Ryan J. Sullivan
Wharton Research Data Services (WRDS)
rsull@wharton.upenn.edu
@rgs258

Objectives

- Configuring Logging in Django
- Getting a logger and writing messages
- Creating, Raising, Chaining, and Catching exceptions
- Format and direct output

Audience

- Django and/or Logging Beginners
- Experienced Djangonauts
- Python Experts
- Everyone Else

About Me

- Wanted to know what was happening in my apps
- Dislike disparate message formats, particularly emails
- Found log4j docs so engaging, I left a suitcase on BART
- I, and I think you'll, prefer to control at runtime, of how log messages are presented

Success Criteria

- Python’s logging module
- Configure Django logging
- Use the logger
- Format and direct output
- Exceptions

Why We Don’t Log

- At the very beginning, it takes more time than print()
- Intimidating Looking API
- Not a business requirement (usually); doesn’t make money

How Can Logs Help

- Know what’s happening
- Troubleshooting made easier
- Debugging during development
- Flexibility of when you need it

Terms

Term	Definition
Package	A collection of Modules, frequently a directory
Module	Something that you can Import, frequently a Python file
__name__	A variable set by the importer; The dotted namespace of a module
Logger	Instance of the Logger class representing a single logging channel
Exception	Instance of an Exception or subclass of Exception; An exceptional event

Exploring Logging Through Examples

Without and With Logging

Persisting Logs

Logging a message

Out of the Box Django Logging

…Not a lot of logs ☺

Configuring Django Logging

mysite/settings.py

```
CONSOLE_LOGGING_FORMAT = '%(hostname)s %(asctime)s %(levelname)-8s %(threadName)-14s ' \
                         '(%(pathname)s:%(lineno)d) %(name)s.%(funcName)s: %(message)s'
```

```
'loggers': {
  # root logger
  '': {
    'level': os.getenv('ROOT_LOG_LEVEL',
'INFO'),
    'handlers': ['console', 'mail_admins'],
  },
  'django': {
    'handlers': ['console', 'mail_admins'],
    'level': os.getenv('DJANGO_LOG_LEVEL',
'INFO'),
    'propagate': False,
  },
  'django.server': {
    'propagate': True,
  },
},
```

```
'handlers': {
  'mail_admins': {
    'level': 'ERROR',
    'filters': ['require_debug_false', ],
    'class':
      'django.utils.log.AdminEmailHandler',
    'include_html': True,
  },
  'console': {
    'class': 'logging.StreamHandler',
    'formatter': 'my_formatter',
  },
  'file': {
    # ... Upcoming slide
  },
},
```

Configuring Django Logging (cont.)

- Django looks for LOGGING in settings.py
- LOGGING is a dict
- Django uses dictConfig
- Other ways of configuring logging

Messages Lost After Restart

- Logging to Console
- Logs don’t survive restart

Configuring File Handler

mysite/settings.py

```
CONFIG_FILE = os.path.dirname(__file__)
CONSOLE_LOGGING_FILE = os.path.join(
    CONFIG_FILE.split(f'config{os.sep}settings')[0],
    'django-wrds.log'
)
'handlers': {
  'file': {
    'class': 'logging.handlers.RotatingFileHandler',
    'filename': CONSOLE_LOGGING_FILE_LOCATION,
    'mode': 'a',
    'encoding': 'utf-8',
    'formatter': 'my_formatter',
    'backupCount': 5,
    'maxBytes': 10485760,
  },
},
```

Add Handler to Logger

mysite/settings.py

```
'loggers': {
  # root logger
  '': {
    'level': os.getenv('ROOT_LOG_LEVEL', 'INFO'),
    'handlers': ['console', 'mail_admins', 'file'],
  },
  'django': {
    'handlers': ['console', 'mail_admins', 'file'],
    'level': os.getenv('DJANGO_LOG_LEVEL', 'INFO'),
    'propagate': False,
  },
  'django.server': {
    'propagate': True,
  },
},
```

Print Messages

- Formatting is written into code
- Control output through commenting
- Doesn’t play nice with others

```
print(f'{datetime.now()}Start of some method')
print(f'{datetime.now()}Doing some work')
...
print(f'{datetime.now()}Result from work: {status_code}')
print(f'{datetime.now()}End of some method’)
```

Log Messages

- Messages to the console, just like print()
- Easy to turn off or send elsewhere
- Formatting defined in config

```
logger.info('Start of some method')
logger.debug('Doing some work')
...
logger.debug(f'Result from work: {status_code}')
logger.info('End of some method’)
```

- Find and Replace print\((.*)\) with logger.info(\1)

Python Logging

logging — Logging facility for Python

LogRecord

- An object created each time a logger is asked to log a message, etc.

  e.g. logger.info() and logger.debug()

- Has many attributes, many of which may be used by formatters

- Passed around until it is handled

LogRecord Attributes (abbreviated)

https://docs.python.org/3/library/logging.html#logrecord-attributes

Attribute name	Description
asctime	Human-readable time when the LogRecord was created. By default this is of the form '2003-07-08 16:49:45,896' (the numbers after the comma are millisecond portion of the time).
filename	Filename portion of pathname.
funcName	Name of function containing the logging call.
levelname	Text logging level for the message ('DEBUG', 'INFO', 'WARNING', 'ERROR', 'CRITICAL').
lineno	Source line number where the logging call was issued (if available).
message	The logged message, computed as msg % args. This is set when Formatter.format() is invoked.
module	Module (name portion of filename).
name	Name of the logger used to log the call.
threadName	Thread name (if available).
exc_info	Exception tuple (à la sys.exc_info) or, if no exception has occurred, None.

Logging Levels

Level	
CRITICAL	**50**
Problems that crash the application	
ERROR	**40**
Problems that break the current function	
WARNING	**30**
Unexpected or undesirable events	
INFO	**20**
Interesting runtime events. Notice that things are working	
DEBUG	**10**
Detail for debugging in development and diagnosing problems	

Four Basic Classes

- Loggers
- Handlers
- Filters
- Formatters

Loggers

- Your interface to the Python logging framework
- Instantiated with **logging.getLogger()**
- Receive messages that are sent to them
- Are hierarchical
- Named to facilitate accessing them later

Getting a ‘logger’

- Recommended construction:

  logger = logging.getLogger(__name__)

- **__name__** is the module’s name in Python

- Module’s name is dot separated and so is your logger

Logger Hierarchy

- “logger” is instance of Logger Class
- Represents a single logging channel
- Is identified by a unique string
- Can be nested
- Are organized into a namespace hierarchy
- Hierarchy levels are separated by periods

Logger Hierarchy by Example

mysite/settings.py

```
'loggers': {
  '': {                          # root logger
    'level': os.getenv('ROOT_LOG_LEVEL', 'INFO'),
    'handlers': ['console', 'mail_admins'],
  },

  'django’: {                    # Django logger
    'handlers': ['console', 'mail_admins'],
    'level': os.getenv('DJANGO_LOG_LEVEL', 'INFO'),
    'propagate': False,
  },
  'django.server’: {             # Django’s server logger
    'propagate': True,
  },
  'django.db.backends’: {        # A third party logger – Django’s db backends
    'propagate': False,
    'level': 'DEBUG’,
    'handlers': ['console', 'file'],
  },
},
```

Handlers

- Deliver messages to destinations via their code
- The console handler:
  logging.StreamHandler

- Django's handler for emailing admins:
  django.utils.log.AdminEmailHandler

- You can write your own, just extend
  logging.Handler
  …and override the method:
  emit(self, record)

Filters

- Used by `Handlers` and `Loggers` for more sophisticated filtering than levels
- When filter returns False, Log Record stops processing

Filters Example

mysite/logging_helpers.py

```
class SomethingFilter(logging.Filter):
  def filter(self, record):
    if 'something' in record.message:
      return False
    else:
      return True
```

mysite/settings.py

```
'filters': {
  'something_filter': {
    '()':
'mysite.log_helper.SomethingFilter',
  },
},


'handlers': {
  'console': {
    'class': 'logging.StreamHandler',
    'filters': ['something_filter', ],
    'formatter': 'wrds_formatter',
  },
},
```

Filters Example

django/utils/log.py

```
class RequireDebugFalse(logging.Filter):
  def filter(self, record):
    return not settings.DEBUG
```

mysite/settings.py

```
'filters': {
  'require_debug_false': {
    '()':
'django.utils.log.RequireDebugFalse',
  },
},


'handlers': {
  'mail_admins': {
    'level': 'ERROR',
    'filters': ['require_debug_false',
],
    'class':

'django.utils.log.AdminEmailHandler',
  },
},
```

Formatters

- Convert logRecord objects to a string (usually) that can be written by a handler
- Make use of the attributes of logRecord to create the string
- Accept a format string to understand how to construct the string

Make Your Own Formatter

```
import logging
from socket import gethostname

class HostnameAddingFormatter(logging.Formatter):

    def __init__(self, fmt=None, datefmt=None, style='%'):
        super().__init__(fmt, datefmt, style)

    def format(self, record):
        # Try to add a hostname attribute to every log record
        try:
            record.__dict__['hostname'] = gethostname()
        except:
            record.__dict__['hostname'] = 'exception-getting-hostname'
        s = super().format(record)
        return s
```

Formatter Example

mysite/settings.py

```
FORMAT = '%(hostname)s %(asctime)s %(levelname)-8s ' \
         '%(threadName)-14s (%(pathname)s:%(lineno)d) ‘ \
         '%(name)s.%(funcName)s: %(message)s’

'formatters': {
  'my_formatter': {
    'format': FORMAT,
    'style': '%’,
    'class': 'mysite.logging_helpers.HostnameAddingFormatter',
  },
},

'handlers': {
  'console': {
    'class': 'logging.StreamHandler',
    'formatter': 'my_formatter',
  },
},
```

Exceptions

Disrupt the normal flow of the application

Exceptions

- Exceptional Events
- Raising an Exception
- Except Clauses Handle Exceptions of Matching Type
- Propagate Until Exception is Handled or Unhandled

Raising Exceptions

mysite/polls/views.py

mysite/polls/views.py

```
def vote(request, question_id):
  question = get_object_or_404(Question, pk=question_id)
```

django/shortcuts.py

```
def get_object_or_404(klass, *args, **kwargs):
  try:
    return queryset.get(*args, **kwargs)
  except queryset.model.DoesNotExist:
    raise Http404('No %s matches the given query.' % queryset.model._meta.object_name)
```

django/http/response.py

```
class Http404(Exception):
    pass
```

Handling Exceptions

- Example: decide to do something else:

```
try:
  user = User.objects.get(username=username)
except User.DoesNotExist:
  user = User.objects.create(username=username, email=email_address)
  logger.debug("User didn't exist; created new user.")
```

- Example: catch exception and log it:

```
try:
  user = User.objects.get(username=username)
except User.DoesNotExist:
  logger.error("User doesn't exist and the user attribute remains unset!")
```

Handling Exceptions (cont.)

- Example: catch exception and log it with Exception information

```
try:
  user = User.objects.get(username=username)
except User.DoesNotExist:
  logger.exception("User doesn't exist and the user attribute remains unset!")
```

- Example: catch exception and re-raise it:

```
try:
  user = User.objects.get(username=username)
except User.DoesNotExist as exc:
  raise User.DoesNotExist("User doesn't exist in this system!") from exc
```

Django's Exception Handler

- Unhandled exceptions engage Django's exception handler

```
def convert_exception_to_response(get_response):
    @wraps(get_response)
    def inner(request):
        try:
            response = get_response(request)
        except Exception as exc:
            response = response_for_exception(request, exc)
        return response
    return inner
```

…or else sys.excepthook

Getting to Django's Exception Handler

django.core.handlers.wsgi.py and django.core.handlers.base.py

```
class WSGIHandler(base.BaseHandler):
  def __init__(self, *args, **kwargs):
    # ...
    self.load_middleware()

  def __call__(self, environ, start_response):
    # ...
    response = self.get_response(request)

class BaseHandler:
  def load_middleware(self):
    # ...
    handler = convert_exception_to_response(self._get_response)
    # ...
    self._middleware_chain = handler

  def get_response(self, request):
    # ...
    response = self._middleware_chain(request)
```

Invisible Exception Handlers

- Anti Pattern
- Bury the code to handle exceptional cases in the business code
- Simpler than using exceptions
- Conflates edge cases with business code
- No re-use

Making Your Own Exceptions

- Example from the Python ldap3 library:

```
class LDAPException(Exception):
    pass

class LDAPExceptionError(LDAPException):
    pass

class LDAPConfigurationError(LDAPExceptionError):
    pass
```

Re-Raising and Chaining Exceptions

- In Python 3 you can chain Exceptions and preserve tracebacks
- Example from manage.py

```
try:
  from django.core.management import execute_from_command_line
except ImportError as exc:
  raise ImportError(
    "Couldn't import Django. Are you sure it's installed and "
    "available on your PYTHONPATH environment variable? Did you "
    "forget to activate a virtual environment?"
  ) from exc
```

More Examples

Third Party Packages

Log Aggregator

Exception Hook

Third Party Packages

- ORM did something
- What did it do?
- You want to see the SQL!

- So long as the third party is logging and you find out to where…

Third Party Loggers

- Add django.db.backends to loggers
- Set log level and handler

```
'loggers': {
    …
    # Django's db backend logger
    'django.db.backends': {
        'propagate': False,
        'level': 'DEBUG',
        'handlers': ['console', 'file'],
    },
},
```

Log Aggregation with Rollbar

- Or Sentry, Airbrake, Elastic, Etc.
- Rollbar becomes a handler
- Forward as much as you'd like to pay for
- Store and understand your log records
- Set up alerts

Log Aggregation with Rollbar - Set Up

- Sign up for rRollbar
- pip install rollbar
- A Little Config

Log Aggregation with Rollbar - Config

```
MIDDLEWARE.insert(0, 'rollbar.contrib.django.middleware.RollbarNotifierMiddlewareOnly404')
MIDDLEWARE.append('rollbar.contrib.django.middleware.RollbarNotifierMiddlewareExcluding404')
```

```
# Rollbar Settings
ROLLBAR = {
  'access_token': '…',
  'access_token_frontend': '…',
  'environment': 'dev',
  'branch': 'ask_me',
  'root': BASE_DIR,
}
```

```
'handlers': {
  'rollbar': {
    'level': 'WARNING',
    'class': 'rollbar.logger.RollbarHandler'
  },
},
'loggers': {
  '': {
    'handlers': ['console', 'mail_admins', 'file', 'rollbar'],
    'level': os.getenv('ROOT_LOG_LEVEL', 'INFO'),
  },
  'django': {
    'handlers': ['console', 'mail_admins', 'file', 'rollbar'],
    'level': os.getenv('DJANGO_LOG_LEVEL', 'INFO'),
    'propagate': False,
  },
},
```

Application Crash Not Reported

- You wrote a manage.py command
- You put the command in cron
- The command crashed
- Wouldn’t it be nice to get an email about said crash?

The Exception Hook

- Handles uncaught exceptions
- Fires just before app crashes

```
def exception_hook(type, value, traceback):
    logging.getLogger('*excepthook*').critical(
        'Uncaught Exception!',
        exc_info=(type, value, traceback))

sys.excepthook = exception_hook
```

Next Steps

- Configure
  - A root logger
  - A format you like
  - A file handler
- Add an exception hook
- Import logging and assign logger at the top of your packages
- FIND and REPLACE all your print() with logger.info()
- Make a root exception for your application

Additional Resources

- Python's Logging Documentation
  https://docs.python.org/3/library/logging.html
- Django's Logging Documentation
  https://docs.djangoproject.com/en/2.2/topics/logging/
- Django's Base Logging Configuration
  https://github.com/django/django/blob/master/django/utils/log.py
- Peter Baumgartner's Discussion of Django Logging
  https://lincolnloop.com/blog/django-logging-right-way/

Thank You

https://github.com/rgs258/logging_in_django/

Ryan J. Sullivan
Wharton Research Data Services (WRDS)
rsull@wharton.upenn.edu
@rgs258