File-based logging library

Preamble

Let's do this ! First, as expected:

```
>>> import logfile
```

The `logfile` module outputs the log messages to `sys.stderr` by default; for testing purposes, we redirect the logs to `sys.stdout`.

```
>>> import sys
>>> logfile.config.output = sys.stdout
```

Logfile

Logfiles are named files that also behave like integers:

```
>>> import logfile
>>> warning = logfile.warning
>>> hasattr(warning, "write")
True
>>> warning.name == str(warning) == "warning"
True
>>>
>>> int(warning)
0
```

Five standard logfiles are defined:

```
>>> names = "critical error warning info debug".split()
>>> logfiles = [getattr(logfile, name) for name in names]
>>> for lf in logfiles:
...     format = "{0:>8}: {1:>2}".format
...     print format(str(lf), int(lf))
critical: -2
   error: -1
 warning:  0
    info:  1
   debug:  2
```

The logfile integer value measure the importance of the channel ; the lower the number, the more important the messages sent to the logfile. Concretely, a message will *effectively* be logged only if it targets a logfile whose integer value is less or equal to the current verbosity (`config.level`) ; the other messages are filtered out.

The default logfile verbosity is set to `warning` (or `0`), so only message that target the `critical`, `error` and `warning` logfiles will logged.

Let's check that in the default setting, information messages get lost:

```
>>> info = logfile.info
>>> logfile.config.level < info
True
>>> print >> info, "testing info logfile ..."
```

As expected, nothing gets printed. But if we set the logfile level to `info` (or higher), information messages don't get lost anymore:

```
>>> logfile.config.level = info
>>> print >> info, "testing info logfile again"
testing info logfile again
```

Writing into LogFiles

Messages can be send to logfiles in two ways: either using the file interface, as in:

```
>>> print >> warning, "I felt a disturbance in the Force."
I felt a disturbance in the Force.
```

or using the callable interface:

```
>>> logfile.warning("I felt a disturbance in the Force.")
I felt a disturbance in the Force.
```

The former call is handy to quicky enable `logfile` support in a code full of `print` statements, but we advise you to adopt the second style to benefit from all features of `logfile`.

Message Variables

The message send to logfiles often depends on the value of local variables. To simplify the definition of messages, we may refer to local variables in messages with the same curly brace syntax use by the string.format method. The value of the variable will be automatically substituted into the message. For example:

```
>>> jedi = "Luke"
>>> logfile.info("Use the Force, {jedi}.")
Use the Force, Luke.
```

Standard LogFile Hooks

LogFile hooks are an easy way to automatically trigger an action when a message is sent to a logfile. Only the `error` and `critical` logfiles use hooks.

A message send to `error` triggers an exception (so that you don't have to):

```
>>> try:
...     logfile.error("I felt a great disturbance in the Force.\n")
... except Exception as e:
...     print repr(e)
I felt a great disturbance in the Force.
Exception('I felt a great disturbance in the Force.',)
```

The type of the exception defaults to `Exception` ; it may be controlled by the keyword argument `type`:

```
>>> class ForceFluctuation(Exception):
...     pass
>>> try:
...     message = "I felt a great disturbance in the Force.\n"
...     logfile.error(message, type=ForceFluctuation)
... except Exception as e:
...     print repr(e)
I felt a great disturbance in the Force.
ForceFluctuation('I felt a great disturbance in the Force.',)
```

A message send to `error` triggers a call to `sys.exit`:

```
>>> try:
...     logfile.critical("This is the end for you ... my former master.\n")
... except SystemExit as e:
...     print repr(e)
This is the end for you ... my former master.
SystemExit('This is the end for you ... my former master.',)
```

Custom Output Format

```
>>> from datetime import datetime as _datetime
```

```
>>> class mock_datetime(object):
...     @staticmethod
...     def now():
...         return _datetime(2014, 1, 1, 12, 00, 00)
>>> logfile.datetime.datetime = mock_datetime

>>> def format(**kwargs):
...     kwargs["date"] = kwargs["date"].strftime("%Y/%m/%d %H:%M:%S")
...     kwargs["logfile"] = str(kwargs["logfile"]).upper()
...     return "{date} | {logfile} from {tag} | {message}".format(**kwargs)

>>> logfile.config.format = format

>>> def Yoda():
...     info("Do. Or do not. There is no try.")
>>> Yoda()
2014/01/01 12:00:00 | INFO from Yoda | Do. Or do not. There is no try.
```

TODO: Explain tag scheme, and how more control can be achieved. **TODO:** Think a little more about this scheme, this is *a lot* of magic.

```
>>> def Yoda():
...     logfile.tag("Dagobah")
...     info("Do. Or do not. There is no try.")
>>> Yoda()
2014/01/01 12:00:00 | INFO from Dagobah | Do. Or do not. There is no try.
```