`!logging.config` --- Logging configuration

- Configuration functions
- Security considerations
- Configuration dictionary schema
  - Dictionary Schema Details
  - Incremental Configuration
  - Object connections
  - User-defined objects
  - Handler configuration order
  - Access to external objects
  - Access to internal objects
  - Import resolution and custom importers
  - Configuring QueueHandler and QueueListener
- Configuration file format

---

logging.config

Source code: `Lib/logging/config.py`

Important

This page contains only reference information. For tutorials, please see

- `Basic Tutorial <logging-basic-tutorial>`
- `Advanced Tutorial <logging-advanced-tutorial>`
- `Logging Cookbook <logging-cookbook>`

This section describes the API for configuring the logging module.

Configuration functions

The following functions configure the logging module. They are located in the `!logging.config` module. Their use is optional --- you can configure the logging module using these functions or by making calls to the main API (defined in `logging` itself) and defining handlers which are declared either in `logging` or `logging.handlers`.

dictConfig(config)

Takes the logging configuration from a dictionary. The contents of this dictionary are described in `logging-config-dictschema` below.

If an error is encountered during configuration, this function will raise a `ValueError`, `TypeError`, `AttributeError` or `ImportError` with a suitably descriptive message. The following is a (possibly incomplete) list of conditions which will raise an error:

- A `level` which is not a string or which is a string not corresponding to an actual logging level.
- A `propagate` value which is not a boolean.
- An id which does not have a corresponding destination.
- A non-existent handler id found during an incremental call.
- An invalid logger name.
- Inability to resolve to an internal or external object.

Parsing is performed by the `DictConfigurator` class, whose constructor is passed the dictionary used for configuration, and has a `configure` method. The `!logging.config` module has a callable attribute `dictConfigClass` which is initially set to `DictConfigurator`. You can replace the value of `dictConfigClass` with a suitable implementation of your own.

`dictConfig` calls `dictConfigClass` passing the specified dictionary, and then calls the `configure` method on the returned object to put the configuration into effect:

```
def dictConfig(config):
    dictConfigClass(config).configure()
```

For example, a subclass of `DictConfigurator` could call `DictConfigurator.__init__()` in its own `__init__`, then set up custom prefixes which would be usable in the subsequent `configure` call. `dictConfigClass` would be bound to this new subclass, and then `dictConfig` could be called exactly as in the default, uncustomized state.

3.2

fileConfig(fname, defaults=None, disable_existing_loggers=True, encoding=None)

Reads the logging configuration from a `configparser`-format file. The format of the file should be as described in `logging-config-fileformat`. This function can be called several times from an application, allowing an end user to select from various pre-canned configurations (if the developer provides a mechanism to present the choices and load the chosen configuration).

It will raise `FileNotFoundError` if the file doesn't exist and `RuntimeError` if the file is invalid or empty.

param fname
A filename, or a file-like object, or an instance derived from `~configparser.RawConfigParser`. If a `!RawConfigParser`-derived instance is passed, it is used as is. Otherwise, a `~configparser.ConfigParser` is instantiated, and the configuration read by it from the object passed in `fname`. If that has a `readline` method, it is assumed to be a file-like object and read using `~configparser.ConfigParser.read_file`; otherwise, it is assumed to be a filename and passed to `~configparser.ConfigParser.read`.

param defaults
Defaults to be passed to the `!ConfigParser` can be specified in this argument.

param disable_existing_loggers
If specified as `False`, loggers which exist when this call is made are left enabled. The default is `True` because this enables old behaviour in a backward-compatible way. This behaviour is to disable any existing non-root loggers unless they or their ancestors are explicitly named in the logging configuration.

param encoding
The encoding used to open file when *fname* is filename.

3.4 An instance of a subclass of `~configparser.RawConfigParser` is now accepted as a value for `fname`. This facilitates:

- Use of a configuration file where logging configuration is just part of the overall application configuration.
- Use of a configuration read from a file, and then modified by the using application (e.g. based on command-line parameters or other aspects of the runtime environment) before being passed to `fileConfig`.

3.10 Added the *encoding* parameter.

3.12 An exception will be thrown if the provided file doesn't exist or is invalid or empty.

listen(port=DEFAULT_LOGGING_CONFIG_PORT, verify=None)

Starts up a socket server on the specified port, and listens for new configurations. If no port is specified, the module's default `DEFAULT_LOGGING_CONFIG_PORT` is used. Logging configurations will be sent as a file suitable for processing by `dictConfig` or `fileConfig`. Returns a `~threading.Thread` instance on which you can call `~threading.Thread.start` to start the server, and which you can

`~threading.Thread.join` when appropriate. To stop the server, call `stopListening`.

The `verify` argument, if specified, should be a callable which should verify whether bytes received across the socket are valid and should be processed. This could be done by encrypting and/or signing what is sent across the socket, such that the `verify` callable can perform signature verification and/or decryption. The `verify` callable is called with a single argument - the bytes received across the socket - and should return the bytes to be processed, or `None` to indicate that the bytes should be discarded. The returned bytes could be the same as the passed in bytes (e.g. when only verification is done), or they could be completely different (perhaps if decryption were performed).

To send a configuration to the socket, read in the configuration file and send it to the socket as a sequence of bytes preceded by a four-byte length string packed in binary using `struct.pack('>L', n)`.

> Because portions of the configuration are passed through `eval`, use of this function may open its users to a security risk. While the function only binds to a socket on `localhost`, and so does not accept connections from remote machines, there are scenarios where untrusted code could be run under the account of the process which calls `listen`. Specifically, if the process calling `listen` runs on a multi-user machine where users cannot trust each other, then a malicious user could arrange to run essentially arbitrary code in a victim user's process, simply by connecting to the victim's `listen` socket and sending a configuration which runs whatever code the attacker wants to have executed in the victim's process. This is especially easy to do if the default port is used, but not hard even if a different port is used. To avoid the risk of this happening, use the `verify` argument to `listen` to prevent unrecognised configurations from being applied.

3.4 The `verify` argument was added.

> If you want to send configurations to the listener which don't disable existing loggers, you will need to use a JSON format for the configuration, which will use `dictConfig` for configuration. This method allows you to specify `disable_existing_loggers` as `False` in the configuration you send.

stopListening()

Stops the listening server which was created with a call to `listen`. This is typically called before calling `join` on the return value from `listen`.

Security considerations

The logging configuration functionality tries to offer convenience, and in part this is done by offering the ability to convert text in configuration files into Python objects used in logging configuration - for example, as described in `logging-config-dict-userdef`. However, these same mechanisms (importing callables from user-defined modules and calling them with parameters from the configuration) could be used to invoke any code you like, and for this reason you should treat configuration files from untrusted sources with *extreme caution* and satisfy yourself that nothing bad can happen if you load them, before actually loading them.

Configuration dictionary schema

Describing a logging configuration requires listing the various objects to create and the connections between them; for example, you may create a handler named 'console' and then say that the logger named 'startup' will send its messages to the 'console' handler. These objects aren't limited to those provided by the `logging` module because you might write your own formatter or handler class. The parameters to these classes may also need to include external objects such as `sys.stderr`. The syntax for describing these objects and connections is defined in `logging-config-dict-connections` below.

Dictionary Schema Details

The dictionary passed to `dictConfig` must contain the following keys:

- *version* - to be set to an integer value representing the schema version. The only valid value at present is 1, but having this key allows the schema to evolve while still preserving backwards compatibility.

All other keys are optional, but if present they will be interpreted as described below. In all cases below where a 'configuring dict' is mentioned, it will be checked for the special `'()'` key to see if a custom instantiation is required. If so, the mechanism described in `logging-config-dict-userdef` below is used to create an instance; otherwise, the context is used to determine what to instantiate.

- *formatters* - the corresponding value will be a dict in which each key is a formatter id and each value is a dict describing how to configure the corresponding `~logging.Formatter` instance.
  The configuring dict is searched for the following optional keys which correspond to the arguments passed to create a `~logging.Formatter` object:
  - `format`
  - `datefmt`
  - `style`
  - `validate` (since version >=3.8)
  - `defaults` (since version >=3.12)

An optional `class` key indicates the name of the formatter's class (as a dotted module and class name). The instantiation arguments are as for `~logging.Formatter`, thus this key is most useful for instantiating a customised subclass of `~logging.Formatter`. For example, the alternative class might present exception tracebacks in an expanded or condensed format. If your formatter requires different or extra configuration keys, you should use `logging-config-dict-userdef`.

- *filters* - the corresponding value will be a dict in which each key is a filter id and each value is a dict describing how to configure the corresponding Filter instance.
  The configuring dict is searched for the key `name` (defaulting to the empty string) and this is used to construct a `logging.Filter` instance.
- *handlers* - the corresponding value will be a dict in which each key is a handler id and each value is a dict describing how to configure the corresponding Handler instance.
  The configuring dict is searched for the following keys:
  - `class` (mandatory). This is the fully qualified name of the handler class.
  - `level` (optional). The level of the handler.
  - `formatter` (optional). The id of the formatter for this handler.
  - `filters` (optional). A list of ids of the filters for this handler.
    3.11 `filters` can take filter instances in addition to ids.

  All *other* keys are passed through as keyword arguments to the handler's constructor. For example, given the snippet:

```
handlers:
  console:
    class : logging.StreamHandler
    formatter: brief
    level   : INFO
    filters: [allow_foo]
    stream  : ext://sys.stdout
  file:
    class : logging.handlers.RotatingFileHandler
    formatter: precise
    filename: logconfig.log
    maxBytes: 1024
    backupCount: 3
```

  the handler with id `console` is instantiated as a `logging.StreamHandler`, using `sys.stdout` as the underlying stream. The handler with id `file` is instantiated as a `logging.handlers.RotatingFileHandler` with the keyword arguments `filename='logconfig.log', maxBytes=1024, backupCount=3`.
- *loggers* - the corresponding value will be a dict in which each key is a logger name and each value is a dict describing how to configure the corresponding Logger instance.
  The configuring dict is searched for the following keys:
  - `level` (optional). The level of the logger.
  - `propagate` (optional). The propagation setting of the logger.

    - `filters` (optional). A list of ids of the filters for this logger.
      3.11 `filters` can take filter instances in addition to ids.
    - `handlers` (optional). A list of ids of the handlers for this logger.

    The specified loggers will be configured according to the level, propagation, filters and handlers specified.
- *root* - this will be the configuration for the root logger. Processing of the configuration will be as for any logger, except that the `propagate` setting will not be applicable.
- *incremental* - whether the configuration is to be interpreted as incremental to the existing configuration. This value defaults to `False`, which means that the specified configuration replaces the existing configuration with the same semantics as used by the existing `fileConfig` API.
  If the specified value is `True`, the configuration is processed as described in the section on `logging-config-dict-incremental`.
- *disable_existing_loggers* - whether any existing non-root loggers are to be disabled. This setting mirrors the parameter of the same name in `fileConfig`. If absent, this parameter defaults to `True`. This value is ignored if *incremental* is `True`.

Incremental Configuration

It is difficult to provide complete flexibility for incremental configuration. For example, because objects such as filters and formatters are anonymous, once a configuration is set up, it is not possible to refer to such anonymous objects when augmenting a configuration.

Furthermore, there is not a compelling case for arbitrarily altering the object graph of loggers, handlers, filters, formatters at run-time, once a configuration is set up; the verbosity of loggers and handlers can be controlled just by setting levels (and, in the case of loggers, propagation flags). Changing the object graph arbitrarily in a safe way is problematic in a multi-threaded environment; while not impossible, the benefits are not worth the complexity it adds to the implementation.

Thus, when the `incremental` key of a configuration dict is present and is `True`, the system will completely ignore any `formatters` and `filters` entries, and process only the `level` settings in the `handlers` entries, and the `level` and `propagate` settings in the `loggers` and `root` entries.

Using a value in the configuration dict lets configurations to be sent over the wire as pickled dicts to a socket listener. Thus, the logging verbosity of a long-running application can be altered over time with no need to stop and restart the application.

Object connections

The schema describes a set of logging objects - loggers, handlers, formatters, filters - which are connected to each other in an object graph. Thus, the schema needs to represent connections between the objects. For example, say that, once configured, a particular logger has attached to it a particular handler. For the purposes of this discussion, we can say that the logger represents the source, and the handler the destination, of a connection between the two. Of course in the

configured objects this is represented by the logger holding a reference to the handler. In the configuration dict, this is done by giving each destination object an id which identifies it unambiguously, and then using the id in the source object's configuration to indicate that a connection exists between the source and the destination object with that id.

So, for example, consider the following YAML snippet:

```
formatters:
  brief:
    # configuration for formatter with id 'brief' goes here
  precise:
    # configuration for formatter with id 'precise' goes here
handlers:
  h1: #This is an id
   # configuration of handler with id 'h1' goes here
   formatter: brief
  h2: #This is another id
   # configuration of handler with id 'h2' goes here
   formatter: precise
loggers:
  foo.bar.baz:
    # other configuration for logger 'foo.bar.baz'
    handlers: [h1, h2]
```

(Note: YAML used here because it's a little more readable than the equivalent Python source form for the dictionary.)

The ids for loggers are the logger names which would be used programmatically to obtain a reference to those loggers, e.g. `foo.bar.baz`. The ids for Formatters and Filters can be any string value (such as `brief`, `precise` above) and they are transient, in that they are only meaningful for processing the configuration dictionary and used to determine connections between objects, and are not persisted anywhere when the configuration call is complete.

The above snippet indicates that logger named `foo.bar.baz` should have two handlers attached to it, which are described by the handler ids `h1` and `h2`. The formatter for `h1` is that described by id `brief`, and the formatter for `h2` is that described by id `precise`.

User-defined objects

The schema supports user-defined objects for handlers, filters and formatters. (Loggers do not need to have different types for different instances, so there is no support in this configuration schema for user-defined logger classes.)

Objects to be configured are described by dictionaries which detail their configuration. In some places, the logging system will be able to infer from the context how an object is to be instantiated, but when a user-defined object is to be instantiated, the system will not know how to do this. In order to provide complete flexibility for user-defined object instantiation, the user needs to provide a 'factory' - a callable which is called with a configuration dictionary and which returns the instantiated object. This is signalled by an absolute import path to the factory being made available under the special key `'()'`. Here's a concrete example:

```
formatters:
  brief:
    format: '%(message)s'
  default:
    format: '%(asctime)s %(levelname)-8s %(name)-15s %(message)s'
    datefmt: '%Y-%m-%d %H:%M:%S'
  custom:
      (): my.package.customFormatterFactory
      bar: baz
      spam: 99.9
      answer: 42
```

The above YAML snippet defines three formatters. The first, with id `brief`, is a standard `logging.Formatter` instance with the specified format string. The second, with id `default`, has a longer format and also defines the time format explicitly, and will result in a `logging.Formatter` initialized with those two format strings. Shown in Python source form, the `brief` and `default` formatters have configuration sub-dictionaries:

```
{
  'format' : '%(message)s'
}
```

and:

```
{
  'format' : '%(asctime)s %(levelname)-8s %(name)-15s %(message)s',
  'datefmt' : '%Y-%m-%d %H:%M:%S'
}
```

respectively, and as these dictionaries do not contain the special key `'()'`, the instantiation is inferred from the context: as a result, standard `logging.Formatter` instances are created. The configuration sub-dictionary for the third formatter, with id `custom`, is:

```
{
  '()' : 'my.package.customFormatterFactory',
  'bar' : 'baz',
  'spam' : 99.9,
  'answer' : 42
}
```

and this contains the special key `'()'`, which means that user-defined instantiation is wanted. In this case, the specified factory callable will be used. If it is an actual callable it will be used directly - otherwise, if you specify a string (as in the example) the actual callable will be located using normal import mechanisms. The callable will be called with the **remaining** items in the configuration sub-dictionary as keyword arguments. In the above example, the formatter with id `custom` will be assumed to be returned by the call:

```
my.package.customFormatterFactory(bar='baz', spam=99.9, answer=42)
```

> The values for keys such as `bar`, `spam` and `answer` in the above example should not be configuration dictionaries or references such as `cfg://foo` or `ext://bar`, because they will not be processed by the configuration machinery, but passed to the callable as-is.

The key `'()'` has been used as the special key because it is not a valid keyword parameter name, and so will not clash with the names of the keyword arguments used in the call. The `'()'` also serves as a mnemonic that the corresponding value is a callable.

3.11 The `filters` member of `handlers` and `loggers` can take filter instances in addition to ids.

You can also specify a special key `'.'` whose value is a mapping of attribute names to values. If found, the specified attributes will be set on the user-defined object before it is returned. Thus, with the following configuration:

```
{
  '()' : 'my.package.customFormatterFactory',
  'bar' : 'baz',
  'spam' : 99.9,
  'answer' : 42,
  '.' : {
    'foo': 'bar',
    'baz': 'bozz'
  }
}
```

the returned formatter will have attribute `foo` set to `'bar'` and attribute `baz` set to `'bozz'`.

> The values for attributes such as `foo` and `baz` in the above example should not be configuration dictionaries or references such as `cfg://foo` or `ext://bar`, because they will not be processed by the configuration machinery, but set as attribute values as-is.

Handler configuration order

Handlers are configured in alphabetical order of their keys, and a configured handler replaces the configuration dictionary in (a working copy of) the `handlers` dictionary in the schema. If you use a construct such as `cfg://handlers.foo`, then initially `handlers['foo']` points to the configuration dictionary for the handler named `foo`, and later (once that handler has been configured) it points to the configured handler instance. Thus, `cfg://handlers.foo` could resolve to either a dictionary or a handler instance. In general, it is wise to name handlers in a way such that dependent handlers are configured *after* any handlers they depend on; that allows something like `cfg://handlers.foo` to be used in configuring a handler that depends on handler `foo`. If that dependent handler were named `bar`, problems would result, because the configuration of `bar` would be attempted before that of `foo`, and `foo` would not yet have been configured. However, if the dependent handler were named `foobar`, it would be configured after `foo`, with the result that `cfg://handlers.foo` would resolve to configured handler `foo`, and not its configuration dictionary.

Access to external objects

There are times where a configuration needs to refer to objects external to the configuration, for example `sys.stderr`. If the configuration dict is constructed using Python code, this is straightforward, but a problem arises when the configuration is provided via a text file (e.g. JSON, YAML). In a text file, there is no standard way to distinguish `sys.stderr` from the literal string `'sys.stderr'`. To facilitate this distinction, the configuration system looks for certain special prefixes in string values and treat them specially. For example, if the literal string `'ext://sys.stderr'` is provided as a value in the configuration, then the `ext://` will be stripped off and the remainder of the value processed using normal import mechanisms.

The handling of such prefixes is done in a way analogous to protocol handling: there is a generic mechanism to look for prefixes which match the regular expression `^(?P<prefix>[a-z]+)://(?P<suffix>.*)$` whereby, if the `prefix` is recognised, the `suffix` is processed in a prefix-dependent manner and the result of the processing replaces the string value. If the prefix is not recognised, then the string value will be left as-is.

Access to internal objects

As well as external objects, there is sometimes also a need to refer to objects in the configuration. This will be done implicitly by the configuration system for things that it knows about. For example, the string value `'DEBUG'` for a `level` in a logger or handler will automatically be converted to the

value `logging.DEBUG`, and the `handlers`, `filters` and `formatter` entries will take an object id and resolve to the appropriate destination object.

However, a more generic mechanism is needed for user-defined objects which are not known to the `logging` module. For example, consider `logging.handlers.MemoryHandler`, which takes a `target` argument which is another handler to delegate to. Since the system already knows about this class, then in the configuration, the given `target` just needs to be the object id of the relevant target handler, and the system will resolve to the handler from the id. If, however, a user defines a `my.package.MyHandler` which has an `alternate` handler, the configuration system would not know that the `alternate` referred to a handler. To cater for this, a generic resolution system allows the user to specify:

```
handlers:
  file:
    # configuration of file handler goes here

  custom:
    (): my.package.MyHandler
    alternate: cfg://handlers.file
```

The literal string `'cfg://handlers.file'` will be resolved in an analogous way to strings with the `ext://` prefix, but looking in the configuration itself rather than the import namespace. The mechanism allows access by dot or by index, in a similar way to that provided by `str.format`. Thus, given the following snippet:

```
handlers:
  email:
    class: logging.handlers.SMTPHandler
    mailhost: localhost
    fromaddr: my_app@domain.tld
    toaddrs:
      - support_team@domain.tld
      - dev_team@domain.tld
    subject: Houston, we have a problem.
```

in the configuration, the string `'cfg://handlers'` would resolve to the dict with key `handlers`, the string `'cfg://handlers.email` would resolve to the dict with key `email` in the `handlers` dict, and so on. The string `'cfg://handlers.email.toaddrs[1]` would resolve to `'dev_team@domain.tld'` and the string `'cfg://handlers.email.toaddrs[0]'` would resolve to the value `'support_team@domain.tld'`. The `subject` value could be accessed using either `'cfg://handlers.email.subject'` or, equivalently, `'cfg://handlers.email[subject]'`. The latter form only needs to be used if the key contains spaces or non-alphanumeric characters. Please note that the characters `[` and `]` are not allowed in the keys. If an index value consists only of decimal digits, access will be attempted using the

corresponding integer value, falling back to the string value if needed.

Given a string `cfg://handlers.myhandler.mykey.123`, this will resolve to `config_dict['handlers']['myhandler']['mykey']['123']`. If the string is specified as `cfg://handlers.myhandler.mykey[123]`, the system will attempt to retrieve the value from `config_dict['handlers']['myhandler']['mykey'][123]`, and fall back to `config_dict['handlers']['myhandler']['mykey']['123']` if that fails.

Import resolution and custom importers

Import resolution, by default, uses the builtin `__import__` function to do its importing. You may want to replace this with your own importing mechanism: if so, you can replace the `importer` attribute of the `DictConfigurator` or its superclass, the `BaseConfigurator` class. However, you need to be careful because of the way functions are accessed from classes via descriptors. If you are using a Python callable to do your imports, and you want to define it at class level rather than instance level, you need to wrap it with `staticmethod`. For example:

```
from importlib import import_module
from logging.config import BaseConfigurator


BaseConfigurator.importer = staticmethod(import_module)
```

You don't need to wrap with `staticmethod` if you're setting the import callable on a configurator *instance*.

Configuring QueueHandler and QueueListener

If you want to configure a `~logging.handlers.QueueHandler`, noting that this is normally used in conjunction with a `~logging.handlers.QueueListener`, you can configure both together. After the configuration, the `QueueListener` instance will be available as the `~logging.handlers.QueueHandler.listener` attribute of the created handler, and that in turn will be available to you using `~logging.getHandlerByName` and passing the name you have used for the `QueueHandler` in your configuration. The dictionary schema for configuring the pair is shown in the example YAML snippet below.

```
handlers:
  qhand:
    class: logging.handlers.QueueHandler
    queue: my.module.queue_factory
    listener: my.package.CustomListener
```

```
    handlers:
      - hand_name_1
      - hand_name_2
      ...
```

The `queue` and `listener` keys are optional.

If the `queue` key is present, the corresponding value can be one of the following:

- An object implementing the `Queue.put_nowait <queue.Queue.put_nowait>` and `Queue.get <queue.Queue.get>` public API. For instance, this may be an actual instance of `queue.Queue` or a subclass thereof, or a proxy obtained by `multiprocessing.managers.SyncManager.Queue`. This is of course only possible if you are constructing or modifying the configuration dictionary in code.
- A string that resolves to a callable which, when called with no arguments, returns the queue instance to use. That callable could be a `queue.Queue` subclass or a function which returns a suitable queue instance, such as `my.module.queue_factory()`.
- A dict with a `'()'` key which is constructed in the usual way as discussed in `logging-config-dict-userdef`. The result of this construction should be a `queue.Queue` instance.

If the `queue` key is absent, a standard unbounded `queue.Queue` instance is created and used.

If the `listener` key is present, the corresponding value can be one of the following:

- A subclass of `logging.handlers.QueueListener`. This is of course only possible if you are constructing or modifying the configuration dictionary in code.
- A string which resolves to a class which is a subclass of `QueueListener`, such as `'my.package.CustomListener'`.
- A dict with a `'()'` key which is constructed in the usual way as discussed in `logging-config-dict-userdef`. The result of this construction should be a callable with the same signature as the `QueueListener` initializer.

If the `listener` key is absent, `logging.handlers.QueueListener` is used.

The values under the `handlers` key are the names of other handlers in the configuration (not shown in the above snippet) which will be passed to the queue listener.

Any custom queue handler and listener classes will need to be defined with the same initialization signatures as `~logging.handlers.QueueHandler` and `~logging.handlers.QueueListener`.

3.12

Configuration file format

The configuration file format understood by `fileConfig` is based on `configparser` functionality. The file must contain sections called `[loggers]`, `[handlers]` and `[formatters]` which identify by name the entities of each type which are defined in the file. For each such entity, there is a separate section which identifies how that entity is configured. Thus, for a logger named `log01` in the `[loggers]` section, the relevant configuration details are held in a section `[logger_log01]`. Similarly, a handler called `hand01` in the `[handlers]` section will have its configuration held in a section called `[handler_hand01]`, while a formatter called `form01` in the `[formatters]` section will have its configuration specified in a section called `[formatter_form01]`. The root logger configuration must be specified in a section called `[logger_root]`.

> The `fileConfig` API is older than the `dictConfig` API and does not provide functionality to cover certain aspects of logging. For example, you cannot configure `~logging.Filter` objects, which provide for filtering of messages beyond simple integer levels, using `fileConfig`. If you need to have instances of `~logging.Filter` in your logging configuration, you will need to use `dictConfig`. Note that future enhancements to configuration functionality will be added to `dictConfig`, so it's worth considering transitioning to this newer API when it's convenient to do so.

Examples of these sections in the file are given below.

```
[loggers]
keys=root,log02,log03,log04,log05,log06,log07

[handlers]
keys=hand01,hand02,hand03,hand04,hand05,hand06,hand07,hand08,hand09

[formatters]
keys=form01,form02,form03,form04,form05,form06,form07,form08,form09
```

The root logger must specify a level and a list of handlers. An example of a root logger section is given below.

```
[logger_root]
level=NOTSET
handlers=hand01
```

The `level` entry can be one of `DEBUG, INFO, WARNING, ERROR, CRITICAL` or `NOTSET`. For the root logger only, `NOTSET` means that all messages will be logged. Level values are `evaluated <func-eval>` in the context of the `logging` package's namespace.

The `handlers` entry is a comma-separated list of handler names, which must appear in the `[handlers]` section. These names must appear in the `[handlers]` section and have corresponding sections in the configuration file.

For loggers other than the root logger, some additional information is required. This is illustrated by the following example.

```
[logger_parser]
level=DEBUG
handlers=hand01
propagate=1
qualname=compiler.parser
```

The `level` and `handlers` entries are interpreted as for the root logger, except that if a non-root logger's level is specified as `NOTSET`, the system consults loggers higher up the hierarchy to determine the effective level of the logger. The `propagate` entry is set to 1 to indicate that messages must propagate to handlers higher up the logger hierarchy from this logger, or 0 to indicate that messages are **not** propagated to handlers up the hierarchy. The `qualname` entry is the hierarchical channel name of the logger, that is to say the name used by the application to get the logger.

Sections which specify handler configuration are exemplified by the following.

```
[handler_hand01]
class=StreamHandler
level=NOTSET
formatter=form01
args=(sys.stdout,)
```

The `class` entry indicates the handler's class (as determined by `eval` in the `logging` package's namespace). The `level` is interpreted as for loggers, and `NOTSET` is taken to mean 'log everything'.

The `formatter` entry indicates the key name of the formatter for this handler. If blank, a default formatter (`logging._defaultFormatter`) is used. If a name is specified, it must appear in the `[formatters]` section and have a corresponding section in the configuration file.

The `args` entry, when `evaluated <func-eval>` in the context of the `logging` package's namespace, is the list of arguments to the constructor for the handler class. Refer to the constructors for the relevant handlers, or to the examples below, to see how typical entries are constructed. If not provided, it defaults to `()`.

The optional `kwargs` entry, when `evaluated <func-eval>` in the context of the `logging` package's namespace, is the keyword argument dict to the constructor for the handler class. If not provided, it defaults to `{}`.

```
[handler_hand02]
class=FileHandler
level=DEBUG
```

```
formatter=form02
args=('python.log', 'w')

[handler_hand03]
class=handlers.SocketHandler
level=INFO
formatter=form03
args=('localhost', handlers.DEFAULT_TCP_LOGGING_PORT)

[handler_hand04]
class=handlers.DatagramHandler
level=WARN
formatter=form04
args=('localhost', handlers.DEFAULT_UDP_LOGGING_PORT)

[handler_hand05]
class=handlers.SysLogHandler
level=ERROR
formatter=form05
args=(('localhost', handlers.SYSLOG_UDP_PORT), handlers.SysLogHandler.LOG_USER)

[handler_hand06]
class=handlers.NTEventLogHandler
level=CRITICAL
formatter=form06
args=('Python Application', '', 'Application')

[handler_hand07]
class=handlers.SMTPHandler
level=WARN
formatter=form07
args=('localhost', 'from@abc', ['user1@abc', 'user2@xyz'], 'Logger Subject')
kwargs={'timeout': 10.0}

[handler_hand08]
class=handlers.MemoryHandler
level=NOTSET
formatter=form08
target=
args=(10, ERROR)
```

```
[handler_hand09]
class=handlers.HTTPHandler
level=NOTSET
formatter=form09
args=('localhost:9022', '/log', 'GET')
kwargs={'secure': True}
```

Sections which specify formatter configuration are typified by the following.

```
[formatter_form01]
format=F1 %(asctime)s %(levelname)s %(message)s %(customfield)s
datefmt=
style=%
validate=True
defaults={'customfield': 'defaultvalue'}
class=logging.Formatter
```

The arguments for the formatter configuration are the same as the keys in the dictionary schema `formatters section <logging-config-dictschema-formatters>`.

The `defaults` entry, when `evaluated <func-eval>` in the context of the `logging` package's namespace, is a dictionary of default values for custom formatting fields. If not provided, it defaults to `None`.

> Due to the use of `eval` as described above, there are potential security risks which result from using the `listen` to send and receive configurations via sockets. The risks are limited to where multiple users with no mutual trust run code on the same machine; see the `listen` documentation for more information.

Module `logging`
API reference for the logging module.

Module `logging.handlers`
Useful handlers included with the logging module.

□□□ #2
□ □□□ □□□ 2026-10-31 17:14:00 UTC
□ □□□ □□□ 2026-10-31 17:14:04 UTC