Overview

Reasons to Log

- **Better than `'print'`:** Using Python `'print'` is suited to development phase, not production code
- **Diagnostics:** Only reasonable way to understand problems with operational code in production

Types of Python Logging

- **Basic:** Use the simple 'logging' object for basic logging functionality.
- **Advanced:** Create individual 'logger' objects for more detailed and complex logging tasks.

Logging Basics

```
import logging
```

Logging levels:

- Debug: *debugging phase*
- Info: *notable event*
- Warning: *irregular event*
- Error: *failure of the operation*
- Critical: *failure of system*

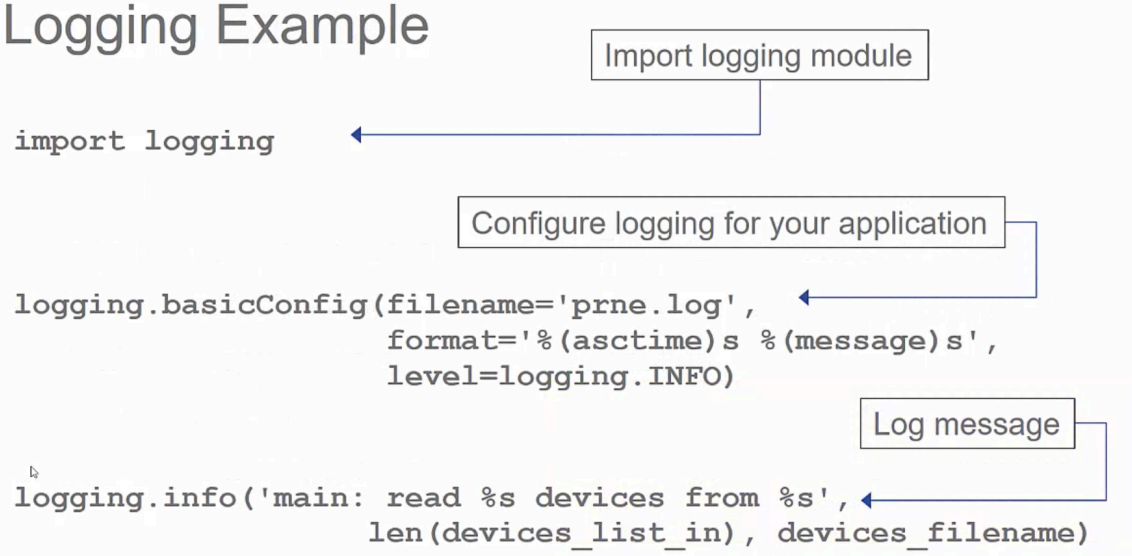
Configuration

- Filename: *log file name*
- Format: *date, module, message*
- Level: *minimum log level*

Execution

- **`logging.info(`*`string`*`)`**
- Message contents, can include data, e.g. **`('...%s',value)`**

Logging Example



Advanced Logging

```
import logging
import logging.handlers
```

Logger:

- Individual logger per module
- Inherits attributes from parent ('main') logger
- Logger name must follow specific 'dot' notation for inheritance

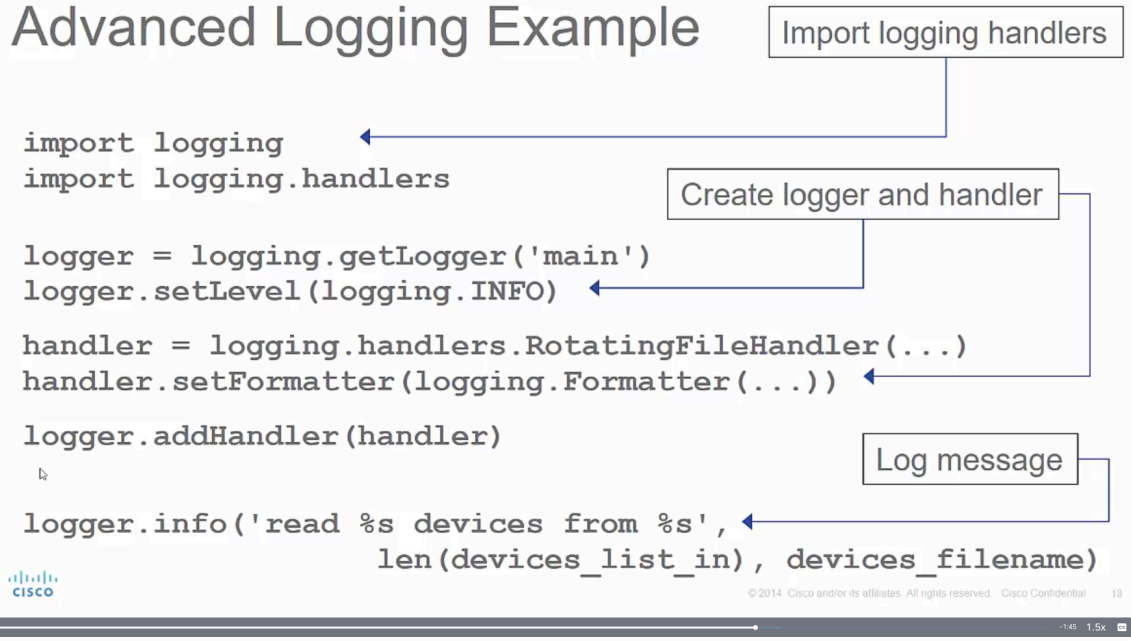
Components:

- *File handler*: E.g. rotating
- *Formatter*: E.g. module name

Execution:

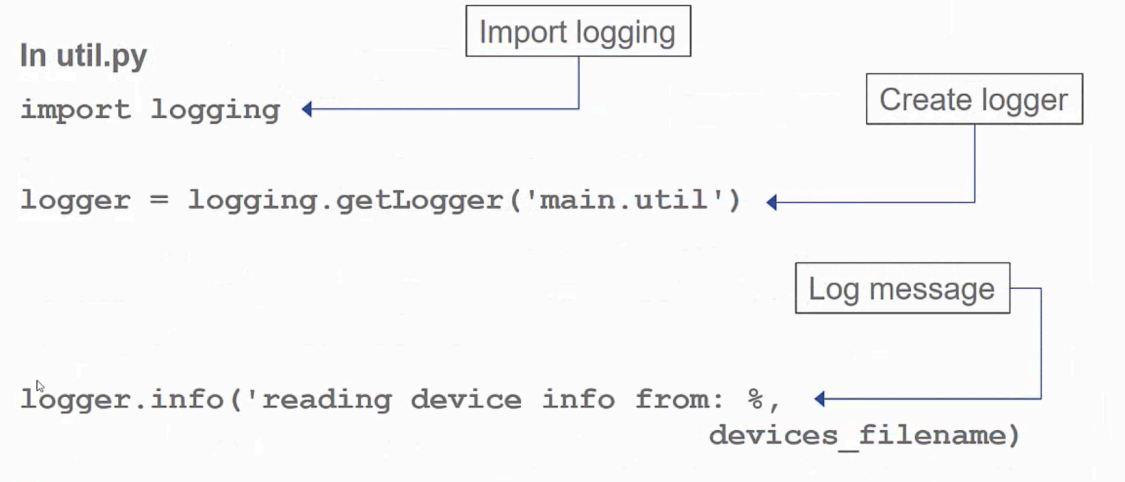
- `log=logging.getLogger()`
- `log.info(`*`string`*`)`

Advanced Logging Example




```
import logging
import logging.handlers

logger = logging.getLogger('main')
logger.setLevel(logging.INFO)

handler = logging.handlers.RotatingFileHandler(...)
handler.setFormatter(logging.Formatter(...))

logger.addHandler(handler)

logger.info('read %s devices from %s',
                len(devices_list_in), devices_filename)
```


Advanced Logging Example (continued)



In util.py

```
import logging

logger = logging.getLogger('main.util')


logger.info('reading device info from: %,
                                devices_filename)
```

If you have used print statements to display information about what is going on in your application, then you are familiar with the general idea of *logging*. The main differences between printing and logging are:

- Printing is appropriate for the development phase of creating your application, when you can read items that are printed to the console.
- Logging is appropriate for other phases, especially when your application is deployed and operating in a production environment. When in production your application will not be printing data to the console, but you will be logging information to a log file.
- Logging messages are sent to files – sometimes with a rotating filename scheme – so it can always be on, and can store information for later examination.
- Logging has a richer set of parameters, for example the logging *level*, which identifies whether the message is merely information, for example for auditing purpose; or whether it indicates a serious or critical error.

Python provides basic logging services which provide reasonable functionality for moderately sized applications. For larger applications, perhaps involving multiple packages, modules, or developers, it may be helpful to utilize some of the advanced logging features.

Logging Basics

There are a few important items related to the use of the logging facility in Python:

- **Import:** You will need to import `logging` in order to use logging functionality in your application.

```
import logging
```

- **Logging level:** You will need to set the logging level which specifies the level at which messages will be logged. If the logging level is set to 'info', then informational messages and messages of greater severity (warning, error, critical) will be logged. The supported logging levels are:
  - **Debug:** debugging phase
  - **Info:** notable event
  - **Warning:** irregular event
  - **Error:** failure of the operation
  - **Critical:** failure of system

- **Configuration:** You will want to specify some configuration parameters for logging in your application. Some examples include setting the filename for the log file, specifying the format of your message (for example, logging date, module name and so on), and setting the logging level (as described above).

When you wish to log a message you use the logging module, as follows:

```
logging.level(string)
```

In the example above, the *level* will either be debug, info, warning, error, or critical. The *string* will be the actual message that will be output. The *message* itself can be a literal string, or it may include your own variable data, such as specific values of strings, lists, or objects that will be of interest to whomever reads the log file**.**

Here is a more complete example of configuring logging in an application:

```
import logging
```

The `import logging` statement above tells Python that you will be using the logging module in your code.

You will then want to set configuration parameters for logging:

```
logging.basicConfig(filename='prne.log',
                    format='%(asctime)s %(message)s',
                    level=logging.INFO)
```

The configuration statement above sets up the important configuration aspects for logging by your application. The configuration file is set to `prne.log`, the format will include the timestamp at the beginning of the message, and the logging level will be set to INFO.

Actually logging messages involves using the logging library as follows:

```
 logging.info('main: read %s devices from %s',
                  len(devices_list_in), devices_filename)
```

In the logging statement above, the actual output will include the timestamp (from the earlier configuration of the 'format' for log file output), followed by the string `main: read...`. The locations where you see `%s` are locations where variable data will be substituted. In the example, there are two items of variable data. The first variable data is the number of items in `devices_list_in`. The second variable data will be the filename of the devices file.

The output of such a logging statement would look similar to:

```
2016-02-05 11:00:03,416 main: read 3 devices from csv-devices
```

The first part of the log line is the date and time. The second part is the message, which contains both the string literal values (`main: read ...`) and the variable data (the number of devices and the filename).

In this most simple form of logging, it is your responsibility to include your module name ('main' in this case) in the output to the log file. Including the module name, and other tasks, require using logger functionality, described next.

Advanced Logging

More advanced logging functionality is available, and involves using loggers, which are derived from the main 'root' logger. You will import `logging` as before and if you intend to utilize any of the various handlers for dealing with log files and creating more advanced messages, you will also need to import `logging.handlers`.

Two of the logging components that are of interest in more advanced logging are file handlers and formatters:

- **Handlers**: There are several file handlers available from the `logging.handlers` library, including a rotating file handler for rotating log files among a group of files, and a datagram socket handler for sending logging messages to a datagram socket.
- **Formatting**: The formatting capabilities provide the opportunity to customize the output format of your log records, including fields such as time of day, module name, logging level, and so on.

In order to use this more advanced version of logging, you call the `getLogger()` function in order to receive a logging object. Next, you attach the appropriate handlers for the log file and formatting. Logging takes place using your new logger object, and messages are logged using the standard methods such as `info()`, `warning()`, `error()` and so on.

Following is an advanced logging example. The first step is to import the required modules:

```
import logging
import logging.handlers
```

You will then create your logger from the logging module, passing in the name of your logger. Note that the name you specify is extremely important, and in order for your logging functionality to be linked across modules, the logger names must be related to one another. Notice the following example:

In `main.py`:

```
logger = logging.getLogger('main')
```

In your modules that hold utility functionality (`util.py`) and device objects (`devclass.py`), the names are related, using dot notation:

In `util.py`:

```
import logging
logger = logging.getLogger('main.util')
```

In `devclass.py`:

```
import logging
logger = logging.getLogger('main.devclass')
```

By specifying the logger names in this way, Python knows that the logging *handlers* set up in `main.py` are going to be shared with the *descendent* loggers in `util.py` and `devclass.py`.

In the `main.py` file, you will set the logging level, and configure your handlers. The following code example shows the setting of these values:

```
logger.setLevel(logging.INFO)
handler = logging.handlers.RotatingFileHandler('main.log',
                                               maxBytes=20000,backupCount=4)
handler.setFormatter(logging.Formatter(
                '%(asctime)s - %(name)s - %(levelname)s - %(message)s'))
logger.addHandler(handler)
```

The first line sets logging level. In the example, the level is set to log all items of level INFO or greater.

The second line sets up the file handler. The `RotatingFileHandler` is used, which means that log entries will be written into the main log file until it is full (`maxBytes`) at which time the current log file gets renamed with a numerical suffix depending on the value of `backupCount`, and new entries get put into the main log file, starting at the beginning of that file.

The actual log entry is using a formatter, which is includes the timestamp (`asctime`), name of the module (`name`), logging level (`levelname`), and the actual message itself (`message`). Every entry in the file, for all modules, will have this format.

The last line takes the handler, which includes the rotating file handler and the formatter, and attaches it to the logger. This configuration will be used for all loggers – `main.py`, and those modules that inherit these characteristics, which in this case would be `util.py` and `devclass.py`. Because the loggers for `util.py` and `devclass.py` are related to `main.py` logger due to the name used during creation of those loggers (the `getLogger` call), they do not have to specify handlers – they are using what has already been created in `main.py`.

Actually logging the messages looks very much like basic logging, with the difference being that you use the logger object that was created, rather than using the logging module directly.

```
logger.info('read %s devices from %s', len(devices_list_in), devices_filename)
```