2024-10-30-Card-Tech-Python-logging

#Card #Tech/Python #Tech/Logging

Context

Object: update the logic of code to show logs better

Chat

Usage of python logging

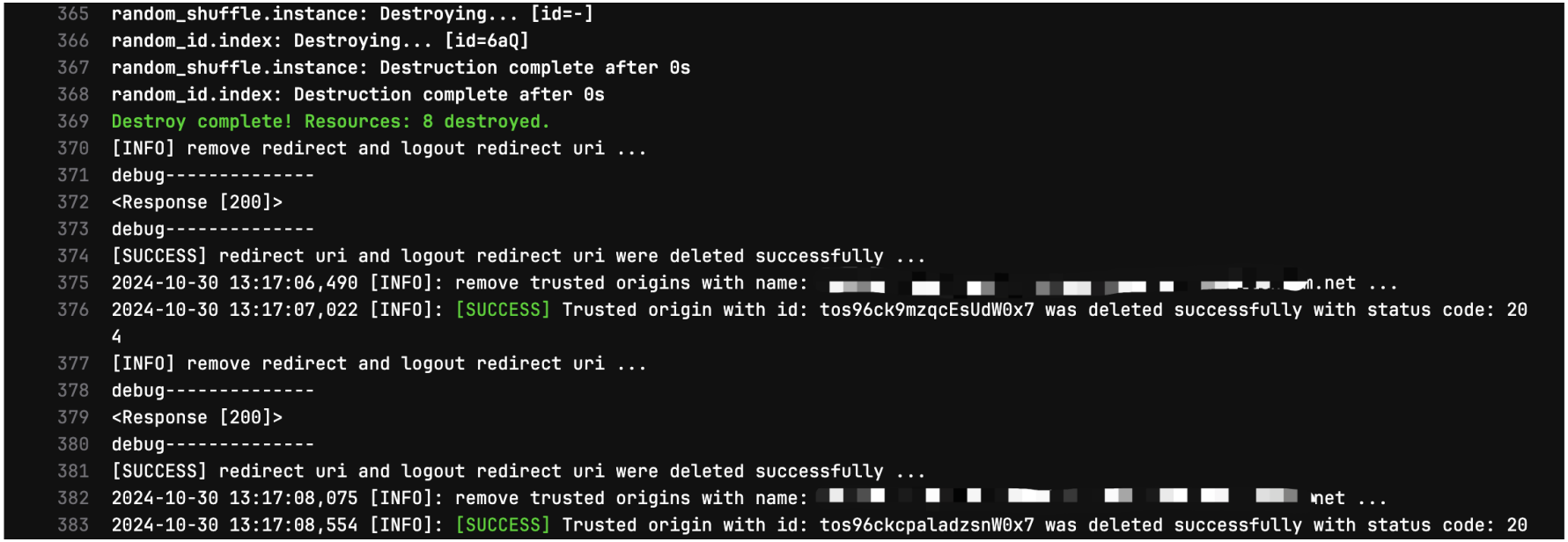
Python's logging module provides a flexible framework for generating log messages from Python programs. It's a built-in module that allows you to track events that happen when your software runs. Here's an overview of how to use Python logging:

1. Basic Usage:

```python
import logging

# Configure the logging system
# logging.basicConfig(level=logging.INFO)
logging.basicConfig(level=logging.INFO, format='%(asctime)s [%(levelname)s]: %
(message)s')

# Create a logger with the name of the module
logger = logging.getLogger(__name__)

# Log messages
logger.debug("This is a debug message")
logger.info("This is an info message")
```

```
logger.warning("This is a warning message")
logger.error("This is an error message")
logger.critical("This is a critical message")
```

The log with basicConfig here



2. Logging Levels:
   There are five standard logging levels, indicating the severity of events:
   - **DEBUG**: Detailed information, typically of interest only when diagnosing problems.
   - **INFO**: Confirmation that things are working as expected.
   - **WARNING**: An indication that something unexpected happened, or indicative of some problem in the near future.
   - **ERROR**: Due to a more serious problem, the software has not been able to perform some function.
   - **CRITICAL**: A serious error, indicating that the program itself may be unable to continue running.
3. Configuring Logging:
   You can configure logging using `basicConfig()`:

```
logging.basicConfig(filename='app.log', filemode='w', format='%(name)s - %(levelname)s - %(message)s')
```

4. Formatting Log Messages:
   You can specify the format of log messages:

```
logging.basicConfig(format='%(asctime)s - %(name)s - %(levelname)s - %(message)s', datefmt='%d-%b-%y %H:%M:%S')
```

5. Logging to Files:
   To log to a file instead of the console:

```
logging.basicConfig(filename='app.log', filemode='w', level=logging.DEBUG)
```

6. Using Handlers:
   Handlers send the log records to the appropriate destination:

```
import logging

# Create a custom logger
logger = logging.getLogger(__name__)

# Create handlers
c_handler = logging.StreamHandler()
f_handler = logging.FileHandler('file.log')
c_handler.setLevel(logging.WARNING)
f_handler.setLevel(logging.ERROR)

# Create formatters and add it to handlers
```

```
c_format = logging.Formatter('%(name)s - %(levelname)s - %(message)s')
f_format = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s')
c_handler.setFormatter(c_format)
f_handler.setFormatter(f_format)

# Add handlers to the logger
logger.addHandler(c_handler)
logger.addHandler(f_handler)

logger.warning('This is a warning')
logger.error('This is an error')
```

7. Using Configuration Files:
   You can also configure logging using a configuration file:

```
import logging
import logging.config

logging.config.fileConfig('logging.conf')

# Create a logger
logger = logging.getLogger('simpleExample')

logger.debug('debug message')
logger.info('info message')
```

These are the basics of using Python's logging module. It provides much more functionality, including log rotation, filtering, and integration with other Python libraries and frameworks.

Why we recommend to use pyhton logging but not print ?

Using Python's logging module instead of print statements for debugging and monitoring applications is recommended for several important reasons:

1. Flexibility:
   - Logging allows you to easily change the verbosity of output without modifying code.
   - You can redirect logs to different outputs (console, file, network) without changing the logging calls.
2. Severity Levels:
   - Logging provides different severity levels (DEBUG, INFO, WARNING, ERROR, CRITICAL), allowing for more nuanced output.
   - You can filter logs based on severity, showing only errors in production while seeing all debug messages during development.
3. Formatting:
   - Logging offers built-in formatting options, making it easy to include timestamps, line numbers, and other contextual information.
4. Thread Safety:
   - The logging module is thread-safe, making it suitable for use in multi-threaded applications.
5. Performance:
   - Logging can be configured to have minimal impact on performance, especially when disabled.
   - Print statements always execute, while logging can be completely turned off in production.
6. Integration:
   - Many Python libraries and frameworks use the logging module, allowing for consistent logging across your entire application ecosystem.
7. Configuration:
   - Logging can be configured via code or external configuration files, allowing for easy changes without code modification.
8. Structured Logging:
   - Advanced logging setups can output structured data (e.g., JSON), which is easier to parse and analyze.
9. Production Readiness:
   - Logging is designed for use in production environments, whereas print statements are generally considered a development tool.

10. Hierarchical Loggers:
    - Logging supports hierarchical loggers, allowing for fine-grained control over logging in different parts of your application.
11. Easy to Disable:
    - In production, you can easily disable or reduce logging output without removing code.
12. Standardization:
    - Using logging promotes a standardized approach to outputting information across your project and team.
13. Debugging:
    - Logging provides more context (like timestamps and line numbers) which is crucial for debugging, especially in production environments.
14. Separation of Concerns:
    - Logging separates the concern of outputting information from the main logic of your code, leading to cleaner, more maintainable code.

While print statements are quick and easy for simple scripts or during initial development, logging is a more robust, flexible, and maintainable solution for any non-trivial application, especially those intended for production use. It provides a professional, standardized way to output information from your application, which is critical for monitoring, debugging, and maintaining software systems.

Solution

More robust, flexible, and maintainable solution for all cases, all ENVs of the software product.

Resource

- Link (option)
- Glossary (option)
- Relevant notes (option)
    - 2024-10-29-Emeria-task-22-RA-optimize OKTA python script
- Relevant query (option)

History

- 2024.10.30 11:36, created by xiaoka: first version
- Template: 2024-09-10-模板-card-<subject>
- Reference: What is the general format for citing articles?