greenhouse

Version

Table of Contents

Contents:

Welcome to Greenhouse's Documentation!

This is the documentation for the **Greenhouse** project. This project provides tools for monitoring and managing a greenhouse environment. It includes:

- A custom logging module (*school_logging*) for flexible and colored logging.
- Functionality to process and store measurements in db (*greenhouse/measurements2db.py*).

Installation

To install the Greenhouse project, follow these steps:

1. Clone the repository:

```
git clone <your_repository_url> #todo
```

2. Install the required packages:

```
pip install -r requirements.txt
```

Usage Example

Here's a quick example of how to use the *ColoredLogger* from the *school_logging* module:

```python
from school_logging.log import ColoredLogger

log = ColoredLogger(__name__)
log.info("This is an informational message.")
```

school_logging

This module defines a custom logger class, *ColoredLogger* , that provides colored log output to the console. It uses ANSI escape codes to color the log messages based on their severity level (DEBUG, INFO, WARNING, ERROR, CRITICAL).

Additionally, the *ColoredLogger* is designed to terminate the program immediately when a message with a CRITICAL log level is emitted. This behavior is implemented by raising a custom exception, *CriticalError* , in the *critical()* method.

The module also includes a custom logging handler, *CriticalExitHandler* , which can be used as an alternative to raising an exception. This handler will terminate the program when a CRITICAL log message is received.

Usage:

> from logger import ColoredLogger
>
> # Create a logger instance log = ColoredLogger(__name__)
>
> # Log messages with different levels log.debug("This is a debug message.") log.info("This is an info message.") log.warning("This is a warning message.") log.error("This is an error message.") log.critical("This is a critical message. The program will terminate.")

The module can also be used directly as a script. In this case, it will parse command-line arguments to set the logging level and demonstrate the logger's usage.

class school_logging.log. ColoredFormatter (*fmt = None* , *datefmt = None* , *style = '%'* , *colored : bool = True*) [source]

Bases: `Formatter`

format (*record : LogRecord*) → str [source]

Format the specified record as text.

The record’s attribute dictionary is used as the operand to a string formatting operation which yields the returned string. Before formatting the dictionary, a couple of preparatory steps are carried out. The message attribute of the record is computed using LogRecord.getMessage(). If the formatting string uses the time (as determined by a call to usesTime(), formatTime() is called to format the event time. If there is exception information, it is formatted using formatException() and appended to the message.

class school_logging.log. ColoredLogger (*name : str* , *verbose_level_str : str = 'INFO'*) [source]

Bases: `object`

A custom logger class that provides colored output and terminates the program on critical errors.

critical (*msg : str* , ** args : Any* , *** kwargs : Any*) → None [source]

Logs a critical message and raises a CriticalError.

debug (*msg : str* , ** args : Any* , *** kwargs : Any*) → None [source]

Logs a debug message.

error (*msg : str* , ** args : Any* , *** kwargs : Any*) → None [source]

Logs an error message.

info (*msg : str* , ** args : Any* , *** kwargs : Any*) → None [source]

Logs an info message.

warning (*msg : str* , ** args : Any* , *** kwargs : Any*) → None [source]

Logs a warning message.

class school_logging.log. CriticalExitHandler (*level = 0*) [source]

Bases: `Handler`

Custom handler that terminates the program when a CRITICAL-level log is emitted.

emit (*record : LogRecord*) → None [source]

Handles the log record. If the record’s level is CRITICAL or higher, terminates the program.

Parameters :

record (*logging.LogRecord*) – The log record.

school_logging.log. parse_args ( ) → Namespace [source]

Parses command-line arguments.

Returns :

The parsed arguments.

Return type :

argparse.Namespace

DatabaseOperations

This module provides a set of operations for interacting with an SQLite database to store and manage sensor data, specifically temperature and humidity measurements. It includes functionalities to establish a database connection, create tables, save sensor readings, and retrieve the current time from an NTP server.

The module uses a *ColoredLogger* for logging messages, providing clear and color-coded output to the console. It is designed to be used as part of a larger application, such as a greenhouse monitoring system, where real-time sensor data needs to be logged and stored in a persistent manner.

Classes:

DatabaseOperations: Encapsulates the database operations.

Functions:

parse_args: Parses command-line arguments.

class greenhouse.measurements2db. DatabaseOperations [source]

Bases: `object`

Provides methods to interact with an SQLite database.

This class encapsulates database operations such as creating a database, saving measurements, and handling the database connection. It is designed to work with an SQLite database and uses a ColoredLogger instance for logging.

DATABASE_FILE

The path to the SQLite database file.

Type :

str

log

Logger instance for logging messages.

Type :

ColoredLogger

conn

Database connection object.

Type :

Optional[sqlite3.Connection]

DATABASE_FILE *: str = 'measurements.db'*

TIME_SERVER *: str = '216.239.35.0'*

close_connection ( ) → None [source]

Closes the database connection.

connect_to_database ( ) → None [source]

Establishes a connection to the SQLite database.

create_database ( ) → None [source]

Creates the database table if it doesn't exist.

get_ntp_time (*ip_address : str*) → datetime | None [source]

Fetches the server time from the given IP address and converts it to the local time zone.

Parameters: ip_address (str): The IP address of the NTP server.

Returns: datetime

print_database ( ) → None [source]

Prints the contents of the 'measurements' table to the console without brackets, quotes, or other special characters.

read_sensor ( ) → None [source]

Simulates reading sensors and returns temperature and humidity. #todo replace with real data

save_measurement (*temp : str* , *hum : float*) → None [source]

Saves a measurement to the database.

Parameters :

- **sensor_name** (*str*) – The name of the sensor.
- **value** (*float*) – The measured value.

greenhouse.measurements2db. parse_args ( ) → Namespace [source]

Parses command-line arguments for the database operations script.

This function defines and parses the command-line arguments required to perform database operations. Modify the argument definitions as needed for your specific database operations.

Returns: argparse.Namespace: Parsed command-line arguments.

Indices and tables

- Index
- Module Index
- Search Page

© Copyright 2025, Alexey Obukhov.
Created using Sphinx 8.1.3.