Plogpro

Wouter Heyvaert

Sep 06, 2021

GETTING STARTED

Plogpro is a simple and versatile Python package that can be used for logging, profiling and more. On top of that, all its functionality is easily customizable and extendable.

CHAPTER

ONE

INSTALLATION

1.1 Using PyPI

Plogpro is available on PyPI, which means that it can easily be installed with `pip` using:

```
pip install plogpro
```

1.2 Using Anaconda

When using Anaconda, you can install Plogpro with:

```
conda install -c wohe plogpro
```

1.3 From source

You can also clone the Plogpro project from Github and install it locally with `pip` using:

```
git clone git@github.com:wohe157/plogpro.git
cd plogpro
pip install .
```

Alternatively, you can use `setup.py` directly:

```
git clone git@github.com:wohe157/plogpro.git
cd plogpro
python setup.py install
```

1.4 Developers

Developers should clone the Plogpro project from Github and install it locally:

```
git clone git@github.com:wohe157/plogpro.git
cd plogpro
pip install --editable .
```

CHAPTER

TWO

USER GUIDE

CHAPTER

THREE

LOGGING

3.1 Logger implementations

The default implementations of loggers are listed here, these are the classes that should be used by users.

Note: See `Logger` for more info on how to use its implementations.

class ConsoleLogger
A logger that writes messages to the console

The messages are written with the following syntax:

```
[  <Type>  ] <Date> <Time> - <Message>
```

class TextLogger(*fname*, *overwrite=False*)
A logger that writes messages to a text file

The messages are written with the following syntax:

```
[  <Type>  ] <Date> <Time> - <Message>
```

Parameters

- **fname** (*str*) – The name of the output file
- **overwrite** (*bool, optional*) – Whether to overwrite the contents of the output file if it already exists or to append the messages to the end of the file (default: *False*)

3.2 Logger Interface

These are the classes that should be used when creating a custom logger.

class LogMessage(*msg*, *msg_type*)
Container class for log messages

This class contains all the necessary information about a log message and is used to send this information from the base class `Logger` to an actual implementation of a logger. More specifically, when a user calls the `log()` method of a logger, a `LogMessage` will be created and passed on to the `write_message()` implementation.

Parameters

- **msg** (*str*) – The message

- **msg_type** (LogType, *optional*) – The message type that indicates its severity, this should be one of the options given by the enumeration LogType (default: LogType.INFO)

msg
The text message given to the log() function

Type str

type
The type of the log message

Type *LogType*

time
The time of the log message

Type datetime

timestring
A formatted string with the date and time of the message

Type str

class Logger
Base class that provides an interface for different loggers

To be able to log messages using a logger of your choice, e.g. TextLogger, create an instance of that logger. You can then use the method log(msg, msg_type) to actually write a log message to a file.

To implement a new logger, create a class that inherits from Logger and at least has a method write_message(self, msg) that accepts one argument: an instance of the LogMessage class. If you need to do anything once in the beginning or the and, you can override the setup() and/or teardown() methods respectively.

> **Warning:** The log() method should **not** be overwritten.

log(*msg*, *msg_type=LogType.INFO*)
Write a log message

Parameters

- **msg** (*str*) – The message
- **msg_type** (LogType, *optional*) – The message type that indicates its severity, this should be one of the options given by the enumeration LogType (default: LogType.INFO)

CHAPTER

FOUR

PROFILING

4.1 Profiler implementations

The default implementations of profilers are listed here, these are the classes that should be used by users.

Note: See `Profiler` for more info on how to use its implementations.

class TracingProfiler(*fname*)

A profiler that outputs a JSON file that can be read by Chrome Tracing

The results will be stored in a JSON file. The contents of this file, i.e. the profiling results, can be visualized using Chrome Tracing. To open Chrome Tracing, open a window in Google Chrome and type `chrome://tracing` in the address bar.

Parameters **fname** (*str*) – The name of the file in which the results will be written

4.2 Profiler Interface

These are the classes that should be used when creating a custom profiler.

class Profiler

Base class for profilers

To profile a program, create an instance `p` of one of the `Profiler` implementations and apply the `@p.profile` decorator to all functions that should be investigated.

Example:

```
import plogpro
p = plogpro.TracingProfiler("results.json")

@p.profile
def func()
    # do something ...
    pass

func()
```

To create a custom profiler, create a subclass of `Profiler` and implement the method `write(self, name, start_time, end_time)` that accepts 3 arguments:

- `name`: the name of the decorated function

- `start_time`: the start time in seconds since Epoch
- `end_time`: the end time in seconds since Epoch

If you need to do anything once in the beginning or the and, you can override the `setup()` and/or `teardown()` methods respectively.

> **Warning:** The `profile()` method should **not** be overwritten.

`profile`(*func*)
Decorator to use for profiling a function

CHAPTER

FIVE

PROGRESS BARS

5.1 Progress bar implementations

The default implementations of progress bars are listed here, these are the classes that should be used by users.

Note: See `ProgressBar` for more info on how to use its implementations.

class ConsoleProgressBar(*nsteps*, *width=70*)

A text-based progress bar for in a terminal or console

Parameters

- **nsteps** (*int*) – The number of steps that the progressbar will go through
- **width** (*int, optional*) – The width of the progress bar (default: 70)

5.2 Progress bar Interface

These are the classes that should be used when creating a custom progress bar.

class ProgressBar(*nsteps*)

Base class that provides an interface for progress bars

To create a progress bar, create an instance of one of its implementations. The progressbar can then be updated by calling the method `update()`.

To implement a new progress bar, create a class that derives from `ProgressBar` and implement the method `draw()`. This method should draw the progress bar based on the accessible member variables. If you need to do anything once in the beginning or the and, you can override the `setup()` and/or `teardown()` methods respectively.

Warning: The `update()` method should **not** be overwritten.

Parameters **nsteps** (*int*) – The number of steps that the progressbar will go through

nsteps

The number of steps that the progressbar will go through

Type int

`step`
The current step of the iteration, going from `0` to `nsteps`

Type int

`start_time`
The start time of the progress bar in seconds since Epoch

Type float

`current_time`
The current time of the progress bar in seconds since Epoch

Type float

`progress()`
Get the progress as a number between 0 and 1

Returns The progress

Return type float

`update`(*step=None*)
Update the progressbar

Parameters `step` (*int, optional*) – The current step of the operation, the progress will be increased with one step if not provided (default: *None*)

CHAPTER

SIX

SETTINGS AND ENUMERATIONS

class Config

A dict-like object that contains the settings for Plogpro

This class is a singleton, which means that every instance contains the same info and changes to any instance will also apply to every other instance. Use the standard instance `config` to avoid any confusion.

Settings can be accessed using square brackets, e.g.:

```
print(config['release'])
```

will print the type of release to the console. To change a setting, use the same syntax:

```
config['release'] = ReleaseType.DEBUG
```

will change the release type to `DEBUG`, which is the most verbose type. The list of attributes below shows the possible cofiguration settings.

> **Warning:** Only change the settings at the beginning of your program. Changing the settings in a later stage can result in unexpected errors. For example, the `setup()` or `teardown()` methods that are available in most base classes will not be called if `config['release'] == ReleaseType.RELEASE_QUIET`, therefore a file that needs to be opened and closed in those methods will not be available is the type of release changes from `ReleaseType.RELEASE_QUIET` to a more verbose type after creating e.g. a logger or profiler.

release

The state of the software that uses Plogpro

Type *ReleaseType*

class LogType(*value*)

An enumeration for indicating the severity of a log message

DEBUG

Useful information for debugging only

INFO

General information

WARNING

A warning to the user

ERROR

Information about an error that has occurred

FATAL

Information about an error that resulted in a crash

class ReleaseType(*value*)

An enumeration for indicating the state of the software using Plogpro

DEBUG

Everything is enabled and as verbose as possible

VERBOSE

Debugging log messages (`LogType.DEBUG`) are disabled, but other messages will still be shown and profilers still work

RELEASE

Only log messages indicating an error (`LogType.ERROR` or higher) are shown and profilers are disabled

RELEASE_QUIET

All loggers and profilers are disabled

CHAPTER

SEVEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

p

INDEX

W