pytimers

Release 3.0

Michal Filippi

Apr 12, 2023

CONTENTS

Welcome to the Pytimers documentation page.

Pytimers is a python micro library that allows you to quickly measure time to run functions, methods or individual blocks of codes. Here's a quick demo.

```
import logging
from time import sleep

from pytimers import timer

logging.basicConfig(level=logging.INFO)


@timer
def foo() -> None:
    print("Some heavy lifting.")
    sleep(2)


if __name__ == "__main__":

    print("Let's call the function decorated with timer.")
    foo()

    print("Use timer context manager.")
    with timer.label("sleeping block"):
        print("Some more work to do.")
        sleep(1)
```

```
Let's call the function decorated with timer.
Some heavy lifting.
INFO:pytimers.triggers.logger_trigger:Finished foo in 2s 0.494ms [2.0s].
Use timer context manager.
Some more work to do.
INFO:pytimers.triggers.logger_trigger:Finished sleeping block in 1s 1.247ms [1.001s].
```

For details go through the following pages.

CHAPTER

ONE

GETTING STARTED

1.1 Requirements

Pytimers require Python 3.7+ for `contextvars.ContextVar` and decorator>=4.0.0 library.

1.2 Installation

You can use PyPi to install the library directly using:

```
pip install pytimers
```

CHAPTER

TWO

USAGE

The whole purpose of this micro library is to provide easy and quick access to measuring the time it takes to run a piece of code without populating the codebase with reoccurring and unnecessary variables. The library allows you to measure the run time of your code in two ways. Using decorators for callables and using context manager to measure run time of any code block using the `with` statement.

The timer on it's own does not do anything unless provided with triggers (see *Triggers*). In the following examples we will be using `pytimers.timer` which is a provided instance of *`pytimers.Timer`* containing single trigger that logs measured time to std output using standard logging library `logging` using a trigger instance of *`pytimers.LoggerTrigger`*.

2.1 Timer Decorator

The timer decorator can be applied to both synchronous and asynchronous functions and methods. PyTimers leverage python library decorator to make sure decorating will preserve the function/method signature, name and docstring.

Note: Decorating classes is currently not supported and will raise `TypeError`.

```
import logging
from time import sleep

from pytimers import timer


logging.basicConfig(level=logging.INFO)

@timer
def func(*args: int):
    print("Hello from func.")
    sleep(1)
    return sum(args)


if __name__ == "__main__":
    func(1, 2, 3)
```

```
Hello from func.
INFO:pytimers.triggers.logger_trigger:Finished func in 1s 1.061ms [1.001s].
```

2.1.1 Class Methods and Static methods

To combine `timer` decorator with decorators `staticmethod()` and `classmethod()` you have to first apply `timer` decorator. Applying the decorators the other way around will result in `TypeError` exception.

```
import logging
from time import sleep

from pytimers import timer


logging.basicConfig(level=logging.INFO)

class Foo:
    @staticmethod
    @timer
    def method(*args: int):
        print("Hello from static method.")
        sleep(1)
        return sum(args)


if __name__ == "__main__":
    foo = Foo()
    foo.method(1, 2, 3)
```

```
Hello from static method.
INFO:pytimers.triggers.logger_trigger:Finished Foo.method in 1s 1.025ms [1.001s].
```

2.2 Timer Context Manager

To measure time of any piece of code not enclosed in a callable object you can use `timer` context manager capabilities.

```
import logging
from time import sleep

from pytimers import timer


logging.basicConfig(level=logging.INFO)

if __name__ == "__main__":
    with timer:
        print("Hello from code block.")
        sleep(1)
```

```
Hello from code block.
INFO:pytimers.triggers.logger_trigger:Finished code block in 1s 1.143ms [1.001s].
```

Entering the context manager actually returns an instance of a *pytimers.clock.Clock*. This allows you to access the current duration from inside of the code block but also the measured duration after the context manager is closed.

```
import logging
from time import sleep

from pytimers import timer


logging.basicConfig(level=logging.INFO)

if __name__ == "__main__":
    with timer as t:
        sleep(1)
        print(f"We want to run this under 5s and so far it took {t.current_duration}.")
        sleep(1)
    print(f"We still had {5 - t.duration}s remaining.")
```

```
We want to run this under 5s and so far it took 1.0001475979988754.
INFO:pytimers.triggers.logger_trigger:Finished code block in 2s 1.384ms [2.001s].
We still had 2.998615708000216s remaining.
```

Block of code can also be named to increase log readability.

```
import logging
from time import sleep

from pytimers import timer

logging.basicConfig(level=logging.INFO)

if __name__ == "__main__":
    with timer.label("data processing pipeline"):
        print("Hello from code block.")
        sleep(1)
```

```
Hello from code block.
INFO:pytimers.triggers.logger_trigger:Finished data processing pipeline in 1s 0.625ms [1.
↪001s].
```

Timer context manager also allows you to stack context managers freely without a worry of interference.

```
import logging
from time import sleep

from pytimers import timer

logging.basicConfig(level=logging.INFO)

if __name__ == "__main__":
    with timer.label("data collecting pipeline"):
        print("Hello from code block n.1.")
        sleep(1)
        with timer:
            print("Hello from code block n.2.")
            sleep(1)
```

(continues on next page)

(continued from previous page)

```
            with timer.label("data processing pipeline"):
                print("Hello from code block n.3.")
                sleep(1)
```

```
Hello from code block n.1.
Hello from code block n.2.
Hello from code block n.3.
INFO:pytimers.triggers.logger_trigger:Finished data processing pipeline in 1s 1.207ms [1.
↪001s].
INFO:pytimers.triggers.logger_trigger:Finished code block in 2s 2.895ms [2.003s].
INFO:pytimers.triggers.logger_trigger:Finished data collecting pipeline in 3s 4.176ms [3.
↪004s].
```

Note: Timer context manager fully supports async code execution using `contextvars.ContextVar`.

2.3 Triggers

Triggers are an abstraction for the action performed after each timer is finished. The simplest trigger can just log the measured time using standard `logging` library. Trigger doing just that is already provided in the library as *pytimers.LoggerTrigger*.

Triggers can be implemented in two ways. Either using a function with keywords arguments `duration_s: float`, `decorator: bool`, `label: str` or by defining a *pytimers.BaseTrigger* subclass.

The following two examples shows how to implement a trivial custom trigger using both methods.

2.3.1 Function Based Trigger

```
import logging
from time import sleep

from pytimers import Timer


def custom_trigger(duration_s: float, decorator: bool, label: str):
    print(f"Measured duration is {duration_s}s.")

if __name__ == "__main__":
    timer = Timer([custom_trigger])

    with timer:
        sleep(1)
```

```
Measured duration is 1.0010350150005252s.
```

2.3.2 BaseTrigger Subclass Trigger

```
import logging
from time import sleep
from typing import Optional

from pytimers import Timer, BaseTrigger


class CustomTrigger(BaseTrigger):
    def __call__(
        self,
        duration_s: float,
        decorator: bool,
        label: Optional[str] = None,
    ) -> None:
        print(f"Measured duration is {duration_s}s.")


if __name__ == "__main__":
    timer = Timer([CustomTrigger()])

    with timer:
        sleep(1)
```

```
Measured duration is 1.0010350150005252s.
```

CHAPTER

THREE

API REFERENCE

If you are looking for information on a specific class or method, this part of the documentation is for you.

3.1 Timers

class pytimers.**Timer**(*triggers: Optional[Iterable[*BaseTrigger | *Callable[[float, bool, Optional[str]], Any]]] = None*)

Initializes Timer object with a set of triggers to be applied after the timer finishes.

Parameters

triggers – An iterable of callables to be called after the timer finishes. All triggers should accept keywords arguments `duration_s: float, decorator: bool, label: str`. PyTimers also provide an abstract class *BaseTrigger* to help with trigger interface implementation. See the *BaseTrigger* for more details. Any instance of *BaseTrigger* subclass is a valid trigger and can be passed to the argument `triggers`.

label(*text: str*) → *Timer*

Sets label for the next timed code block. This label propagates to all triggers once the context managers is closed.

Parameters

text – Code block label text.

Returns

Returns `self`. This makes possible to call the method directly inside context manager with statement.

named(*name: str*) → *Timer*

This method only ensures backwards compatibility. Use *pytimers.Timer.label()* instead.

Deprecated since version 3.0.

class pytimers.clock.**Clock**(*label: Optional[str]*)

current_duration(*precision: Optional[int] = None*) → float

Calculates the current duration elapsed since the clock was started. This property can be used inside a timed code block.

Parameters

precision – Number of decimal places of the returned time. If set to `None` the full precision is returned.

Returns

Measured time in seconds between start of the clock and the method call.

duration(*precision: Optional[int] = None*) → float

Exposes measured time of the clock. You can use this method to access the measured time even after the context manager is closed. This property should never be used directly inside a timed code block as it would raise an *pytimers.exceptions.ClockStillRunning* exception.

Parameters

precision – Number of decimal places of the returned time. If set to None the full precision is returned.

Returns

Measured time in seconds between start and stop of the clock.

Raises

pytimers.exceptions.ClockStillRunning – Clock has to be stopped before accessing elapsed time.

stop() → None

Stops the running clock.

exception pytimers.exceptions.**ClockStillRunning**

Custom exception to be raised while accessing properties of *pytimers.clock.Clock* before being stopped.

3.2 Triggers

class pytimers.**BaseTrigger**

This class provides timer trigger abstraction. Custom triggers can be implemented using simple functions but subclassing this abstract class is the preferred way. Any custom implementation has to override *pytimers.BaseTrigger.__call__()* method where the trigger logic should be provided.

abstract __call__(*duration_s: float*, *decorator: bool*, *label: Optional[str] = None*) → None

This is a trigger action entrypoint. This method is called in *pytimers.Timer* once the timer stops.

Parameters

- **duration_s** – The measured duration in seconds.
- **decorator** – True if the timer was used as a decorator for callable. False if used as a context manager for timing code blocks.
- **label** – The label of the measured code block provided by client before entering the context manager. For decorator usage this value is set to the callable name.

static humanized_duration(*duration_s: float*, *precision: int = 0*) → str

This method provides formatter for human-readable duration with hours being the highest level of the format.

Parameters

- **duration_s** – The duration in seconds to be formatted.
- **precision** – Number of decimal places for milliseconds.

Returns

Human-readable duration as a string.

class pytimers.**LoggerTrigger**(*level: int = 20*, *template: str = 'Finished ${label} in ${humanized_duration} [${duration}s].'*, *precision: int = 3*, *humanized_precision: int = 3*, *default_code_block_label: str = 'code block'*)

Provided trigger class for logging the measured duration using std logging library.

Parameters

- **level** – Log level (as understood by the standard logging library `logging`) used for the message.
- **template** – Message template string containing placeholders for label, duration and/or humanized_duration.
- **precision** – Number of decimal places for the message duration in seconds.
- **humanized_precision** – Number of decimal places for milliseconds in human-readable duration in the message.
- **default_code_block_label** – Label used for code blocks with missing label.

CHAPTER

FOUR

CHANGELOG

4.1 Release 3.1

- Added `py.typed` file to mark full type checking compliance.
- Created manually triggered release pipeline to Test PyPi.

4.2 Release 3.0

- Implemented support for async context manage usage using `contextvars.ContextVar`.
- Context manager now returns *`pytimers.clock.Clock`* to expose *`pytimers.clock.Clock.duration()`* and *`pytimers.clock.Clock.current_duration()`*.
- Logging moved outside of the *`pytimers.Timer`* class and implemented as a separate trigger.
- Deprecating *`pytimers.Timer.named()`* for *`pytimers.Timer.label()`*.
- Sphinx documentation added.

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search

INDEX

Symbols

B

C

D

H

L

N

S

T