callerframe Documentation

Release 1.0.3

Simone Campagna

October 06, 2015

Contents

The *callerframe* decorator adds to the decorated function information about it's caller. This information can be accessed through the `__caller_frame__` attribute, which is inserted into the function's globals. The information is the namedtuple `FrameInfo` containing:

- `frame`: the caller's frame
- `filename`: the filename
- `line_number`: the line number
- `function_name`: the function name
- `context`: a list of context lines
- `index`: the index of the line in the `context` where the call is done

CHAPTER 1

Motivation

Suppose you want to define a `log()` function:

```
>>> def log(kind, message):
...     print("{}: {}".format(kind, message))
...
>>> log("error", "lost connection")
error: lost connection
>>>
```

You may want to automatically add to the log some information about the caller, for instance:

```
>>> def log(kind, message):
...     print("{}: function {}: {}".format(kind, function_name, message))
```

so you need to obtain the callers' function name, and eventually the filename or the line number.

Using `inspect` you can easily obtain such information:

```
>>> import inspect
>>> def log(kind, message):
...     frame, filename, line_number, function_name, context, index = inspect.getouterframes(inspect
...     print("{}: function {}: {}".format(kind, function_name, message))
...
>>> def foo():
...    log("error", "lost connection")
...
>>> foo()
error: function foo: lost connection
```

But what if you want to define also an `error` function using the log one? In this cast `log` should show information about the `error`'s caller, and not about it's direct caller:

```
>>> def error(message):
...     log("error", message)
...
>>> def bar():
...    error("lost connection")
...
>>> bar()
error: function error: lost connection
```

Notice that the `log()` function should behave as above when called directly, showing its direct caller's name. But it should show the `error()` caller's name when called through `error()`.

CHAPTER 2

Solution

The `callerframe` decorator can solve these problems. First of all, it adds to the decorated function's globals a `__caller_frame__` attribute containing all the information about the caller:

```
>>> @callerframe
... def log(kind, message):
...     print("{}: function {}: {}".format(kind, __caller_frame__.function_name, message))
...
>>> def foo():
...    log("error", "lost connection")
...
>>> foo()
error: function foo: lost connection
```

Moreover, it is possible to use the same decorator for the `error` function too: in this case, when called through `error`, `log` will receive the `error`'s caller:

```
>>> @callerframe
... def error(message):
...     log("error", message)
...
>>> def bar():
...    error("lost connection")
...
>>> bar()
error: function bar: lost connection
```

In general, the first decorated function in a call stack sets the caller's information.

It is possible to change the name of the added attribute:

```
>>> @callerframe("CALLERFRAME")
... def show_caller():
...     print(CALLERFRAME.function_name)
...
>>> def foo():
...     show_caller()
...
>>> foo()
foo
```

Contents:

2.1 callerframe package

2.1.1 Module contents

The callerframe decorator adds a __caller_frame__ global attribute to the decorated function's globals; this attribute refers to a FrameInfo object containing information about the caller function:

- frame: the caller's frame;
- filename: the name of the file where the function has been called;
- line_number: the line number of the call in filename;
- function_name: the name of the caller function;
- context: a list of source line containing the call;
- index: the index of the line in context where the function has been called.

```
>>> @callerframe
... def log(kind, message):
...     print("{}: function {}: {}".format(kind, __caller_frame__.function_name, message))
...
>>> def foo():
...     log("error", "lost connection")
...
>>> def main():
...     return foo()
...
>>> main()
error: function foo: lost connection
```

The log function receives information about it's direct caller; but what if we want to have an error() function based on log()?

```
>>> def error(message):
...     log("error", message)
...
>>> def foo():
...     error("lost connection")
...
>>> def main():
...     return foo()
...
>>> main()
error: function error: lost connection
```

This is correct, since error() is the direct caller of the log() function; nevertheless we would like to show the information about the error()'s caller instead. In this case it is possible to decorate error() too (no modification is needed in log):

```
>>> @callerframe
... def error(message):
...     log("error", message)
...
>>> def foo():
...     error("lost connection")
...
>>> def main():
...     return foo()
...
```

```
>>> main()
error: function foo: lost connection
```

In other words, the first decorated function found in the call stack sets the caller information. This information is not overwritten by nested calls to decorated functions.

The attribute name, by default __caller_frame__, can be choosen:

```
>>> @callerframe("CALLERFRAME")
... def show_caller():
...     print(CALLERFRAME.function_name)
...
>>> def foo():
...     show_caller()
...
>>> foo()
foo
```

class callerframe.**FrameInfo**(*frame*, *filename*, *line_number*, *function_name*, *context*, *index*)

Bases: tuple

__getnewargs__()

Return self as a plain tuple. Used by copy and pickle.

__getstate__()

Exclude the OrderedDict from pickling

static __new__(*_cls*, *frame*, *filename*, *line_number*, *function_name*, *context*, *index*)

Create new instance of FrameInfo(frame, filename, line_number, function_name, context, index)

__repr__()

Return a nicely formatted representation string

context

Alias for field number 4

filename

Alias for field number 1

frame

Alias for field number 0

function_name

Alias for field number 3

index

Alias for field number 5

line_number

Alias for field number 2

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

C

Symbols

C

F

I

L