decorating Documentation

Release 0.6.1

Manoel Vilela

Dec 27, 2017

Table of Contents

CHAPTER 1

Decorating: A Meta Repo To Decorators

CHAPTER 2

Abstract

This project encourages an exploration into the limits of decorators in `Python`. While decorators might by new to beginners, they are an extremely useful feature of the language. They can be similar to Lisp Macros, but without changes to the AST. Great decorators from this packages are `@animated` and `@writing`. This repository is made from scratch, just using Python's Standard Library, no dependency!

CHAPTER 3

Examples

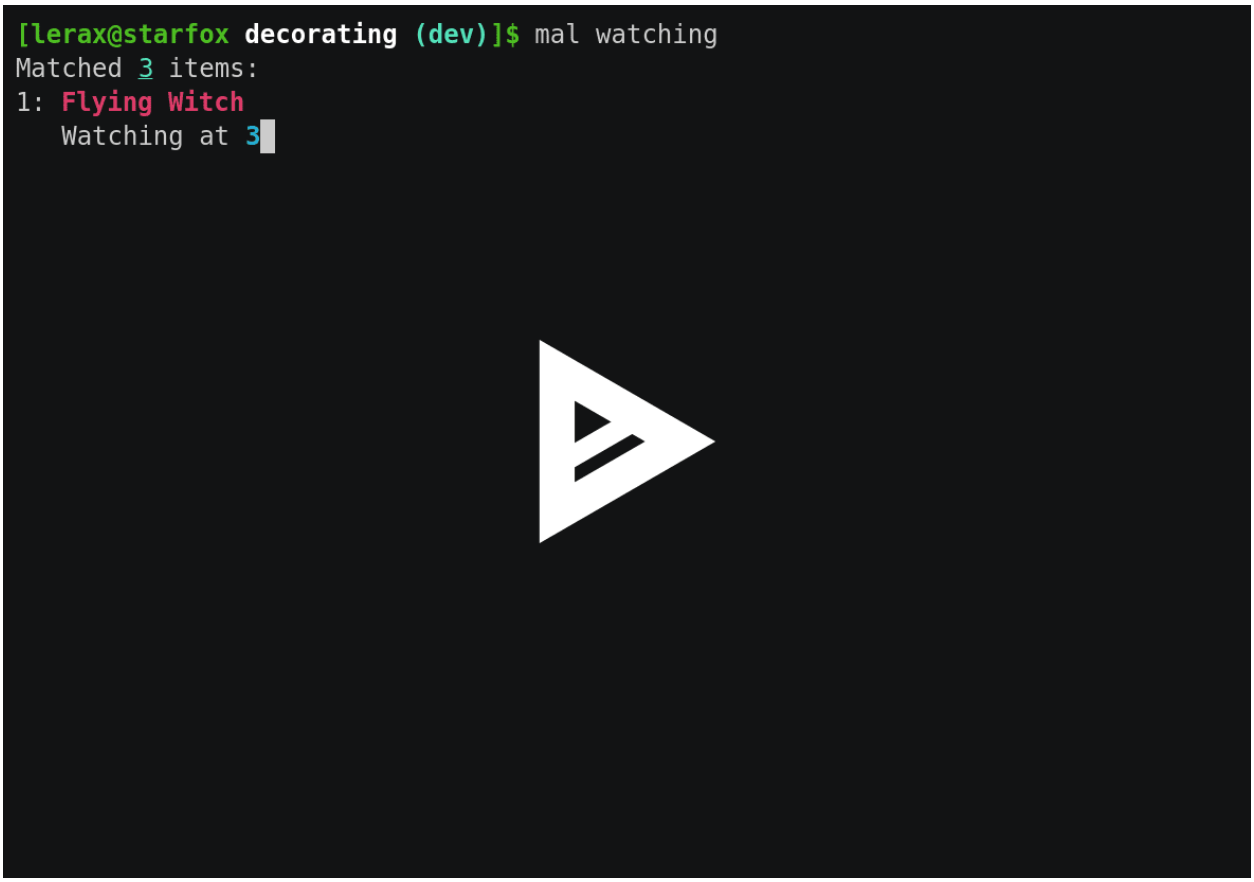
3.1 Animated

Using as decorator and mixed with context-managers

Using with nested context-managers

3.2 Writing

Another project mine called MAL uses the decorating package —- basically a command line interface for MyAnimeList. The decorator @writing can be used by just adding 3 lines of code! The behavior is a retro typing-like computer. Check out the awesome effect:



More examples are covered on my personal blog post about decorating.

CHAPTER 4

Decorators & Usage

Currently public decorators on the API of decorators `decorating`:

- **decorating.debug**
- **decorating.cache**
- **decorating.counter**
- **decorating.count_time**
- **decorating.animated**
- **decorating.writing**

Mostly decorators has a pretty consistent usage, but for now only `animated` and `writing` has support to use as `contextmanagers` using the `with` syntax.

CHAPTER 5

Installation

Supported Python versions:

- Python3.4+
- Python2.7

You can install the last release on PyPI by calling:

```
pip install --user decorating
```

If you want get the last development version install directly by the git repository:

```
pip install --user git+https://www.github.com/ryukinix/decorating
```

We have a published package on Arch Linux,which you can install using your favorite AUR Helper, like `pacaur` or `yaourt`:

```
yaourt -S python-decorating
```

Though since the version `0.6` we have support for Python2.7, an AUR package for Python2 was not made yet. Fill a issue if you have interest on that :). Thanks to *Maxim Kuznetsov <https://github.com/mkuznets>* which implemented the necessary changes to make compatible with Python2!

5.1 License

MIT

Because good things need to be free.

CHAPTER 6

decorating package

6.1 Submodules

6.2 decorating.animation module

This module was be done to handle the beautiful animation using the sin function (whose cause a pulse in the stdout).

Some examples of using is here:

@animated def slow():

heavy_stuff()

As well with custom messages @animated('WOOOOW') def download_the_universe():

while True: pass

with animated('loool'): stuff_from_hell()

@writing def printer():

lot_of_messages()

with writing(delay=0.5): print("L O L => IS NOT THE STUPID GAME LOL, LOL.")

6.3 decorating.asciiart module

This is another LOL-zone

LOOOOOOOOOOOOOL ART

6.4 decorating.base module

Abstract Classes to do composition by inheterince and some other utitities from base clases

- Stream: Abstract Class for implementation of a Stream
- Decorator: Abstract Class for creating new decorators

class decorating.base.**DecoratorManager**
Bases: object

Decorator-Context-Manager base class to keep easy creating more decorators

argument: can be empty or a callable object (function or class)

start()
You active here your pre-fucking crazy feature

stop()
You can deactivate any behavior re-writing your method here

class decorating.base.**Stream**(*stream*, ***kargs*)
Bases: object

A base class whose is specify a Stream is

We need at least a stream on init and a message param on write method

write(*message*, *optional=None*)
a write method interfacing sys.stdout or sys.stderr

6.5 decorating.color module

Module focused in termcolor operations

If the exection is not attatched in any tty, so colored is disabled

decorating.color.**colorize**(*printable*, *color*, *style='normal'*, *autoreset=True*)
Colorize some message with ANSI colors specification

Parameters

- **printable** – interface whose has __str__ or __repr__ method
- **color** – the colors defined in COLOR_MAP to colorize the text

Style can be 'normal', 'bold' or 'underline'

Returns the 'printable' colorized with style

6.6 decorating.debugging module

An collection of usefull decorators for debug and time evaluation of functions flow

decorating.debugging.**count_time**(*function*)
Function: count_time Summary: get the time to finish a function

print at the end that time to stdout

Examples: <NONE> Attributes:

@param (function): function

Returns: wrapped function

decorating.debugging.**counter** (*function*)

Function: counter Summary: Decorator to count the number of a function is executed each time Examples: You can use that to had a progress of heally heavy

computation without progress feedback

Attributes: @param (function): function

Returns: wrapped function

decorating.debugging.**debug** (*function*)

Function: debug Summary: decorator to debug a function Examples: at the execution of the function wrapped,

the decorator will allows to print the input and output of each execution

Attributes: @param (function): function

Returns: wrapped function

6.7 decorating.decorator module

The base class for creating new Decorators

- Decorator: A base class for creating new decorators

class decorating.decorator.**Decorator**

Bases: *decorating.base.DecoratorManager*

Decorator base class to keep easy creating more decorators

triggers: self.start self.stop

context_manager: self.__enter__ self.__exit__

Only this is in generall necessary to implement the class you are writing, like this:

class Wired(Decorator):

def __init__(self, user='Lain') self.user = user

def start(self): self.login()

def stop(self): self.logoff()

def login(self): print('Welcome to the Wired, {user}!'.format(user=self.user))

def logoff(self): print('Close this world, open the next!'.)

And all the black magic is done for you behind the scenes. In theory, you can use the decorator in these way:

@Wired('lain') def foo():

pass

@Wired(argument='banana') def bar():

pass

@Wired def lain():

pass

@Wired() def death():

pass

And all are okay! As well, natively, you have support to use as context managers.

So that you can handle that way:

with Wired: print("Download the Knight files...")

with Wired(): print("Underlying bugs not anymore")

with Wired("Lerax"): print("I'm exists?")

with Wired(user="Lerax"): print("I don't have the real answer.")

And all occurs be fine like you thinks this do.

classmethod check_arguments(*passed*)
Put warnings of arguments whose can't be handle by the class

classmethod default_arguments()
Returns the available kwargs of the called class

instances = []

classmethod recreate(**args*, ***kwargs*)
Recreate the class based in your args, multiple uses

6.8 decorating.general module

An collection of usefull decorators for debug and time evaluation of functions flow

decorating.general.**cache**(*function*)
Function: cache Summary: Decorator used to cache the input->output Examples: An fib memoized executes at O(1) time

instead O(e^n)

Attributes: @param (function): function

Returns: wrapped function

TODO: Give support to functions with kwargs

decorating.general.**with_metaclass**(*meta*, **bases*)
Create a base class with a metaclass.

6.9 decorating.stream module

This module have a collection of Streams class used to implement:

- Unbuffered(Stream) :: stream wrapper auto flushured
- Animation(Unbuferred) :: stream with erase methods
- Clean(Unbuffered) :: stream with handling paralell conflicts
- Writing(Unbuffered) :: stream for writing delayed typing

class decorating.stream.**Animation**(*stream*, *interval=0.05*)
Bases: *decorating.stream.Unbuffered*

A stream unbuffered whose write & erase at interval

After you write something, you can easily clean the buffer and restart the points of the older message. stream = Animation(stream, delay=0.5) self.write('message')

ansi_escape = re.compile('\\x1b[^m]*m')

erase(*message=None*)
Erase something whose you write before: message

last_message = ''

write(*message*, *autoerase=True*)
Send something for stdout and erased after delay

class decorating.stream.**Clean**(*stream*, *paralell_stream*)
Bases: *decorating.stream.Unbuffered*

A stream wrapper to prepend ' ' in each write

This is used to not break the animations when he is activated

So in the start_animation we do: sys.stdout = Clean(sys.stdout, <paralell-stream>)

In the stop_animation we do: sys.stdout = sys.__stdout__Whose paralell_stream is a Animation object.

write(*message*, *flush=False*)
Write something on the default stream with a prefixed message

class decorating.stream.**Unbuffered**(*stream*)
Bases: *decorating.base.Stream*

Unbuffered whose flush automaticly

That way we don't need flush after a write.

lock = <unlocked _thread.lock object>

write(*message*, *flush=True*)
Function: write Summary: write method on the default stream Examples: >>> stream.write('message')

'message'

Attributes: @param (message): str-like content to send on stream @param (flush) default=True: flush the stdout after write

Returns: None

class decorating.stream.**Writting**(*stream*, *delay=0.08*)
Bases: *decorating.stream.Unbuffered*

The Writting stream is a delayed stream whose simulate an user Writting something.

The base class is the AnimationStream

write(*message*, *flush=True*)

6.10 Module contents

DECORATING: A MODULE OF DECORATORS FROM HELL

You have a collection of decorators, like thesexg:

- animated: create animations on terminal until the result's returns
- cache: returns without reprocess if the give input was already processed
- counter: count the number of times whose the decorated function is called
- debug: when returns, print this pattern: @function(args) -> result
- count_time: count the time of the function decorated did need to return

decorating.**cache** (*function*)

Function: cache Summary: Decorator used to cache the input->output Examples: An fib memoized executes at O(1) time

instead O(e^n)

Attributes: @param (function): function

Returns: wrapped function

TODO: Give support to functions with kwargs

decorating.**counter** (*function*)

Function: counter Summary: Decorator to count the number of a function is executed each time Examples: You can use that to had a progress of heally heavy

computation without progress feedback

Attributes: @param (function): function

Returns: wrapped function

decorating.**debug** (*function*)

Function: debug Summary: decorator to debug a function Examples: at the execution of the function wrapped,

the decorator will allows to print the input and output of each execution

Attributes: @param (function): function

Returns: wrapped function

decorating.**count_time** (*function*)

Function: count_time Summary: get the time to finish a function

print at the end that time to stdout

Examples: <NONE> Attributes:

@param (function): function

Returns: wrapped function

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

Python Module Index

d

Index