Decorators - A Deep Dive

Solutions

July 17, 2023

Prague, Czech Republic

Author:	Dr.-Ing. Mike Müller
E-Mail:	mmueller@python-academy.de
Twitter:	pyacademy
Version:	1.0

Trainer:	Dr.-Ing. Mike Müller
E-Mail:	mmueller@python-academy.de

Current Training Modules - Python Academy

As of 2023

Module Topic	**Length (days)**	**In-house**	**Remote**	**Open Training**
Python for Programmers	4	yes	yes	yes
Python for Non-Programmers	5	yes	yes	yes
Advanced Python	3	yes	yes	yes
Python for Scientists and Engineers	3	yes	yes	yes
Python for Data Analysis	3	yes	yes	yes
Machine Learning with Python	3	yes	yes	yes
Professional Testing with Python	3	yes	yes	yes
High Performance Computing with Python	3	yes	yes	yes
Cython in Depth	2	yes	yes	yes
Introduction to Django	4	yes	yes	yes
Advanced Django	3	yes	yes	yes
Image Processing with Python	3	yes	yes	no
SQLAlchemy	3	yes	yes	no
High Performance XML with Python	1	yes	yes	no
Optimizing Python Programs	1	yes	yes	no
Python Extensions with Other Languages	1	yes	yes	no
Data Storage with Python	1	yes	yes	no
Introduction to Software Engineering with Python	1	yes	yes	no
Introduction to PySide/PyQt	1	yes	yes	no
Overview of the Python Standard Library	1	yes	yes	no
Threads and Processes in Python	1	yes	yes	no
Network Programming with Python	1	yes	yes	no

We offer on-site and open courses world-wide. We customize and extend training modules for your needs. We also provide consulting services such as code review, customized programming, and workshops. Open courses are offered as in-person training in Leipzig, Germany as well as remote trainer-led training. Individuals participants can enroll in open courses.

More information: www.python-academy.com

Python Tech Radar

Do you want to stay ahead of the curve and be the first to know about new developments in the Python ecosystem? Look no further! The Python Tech Radar has the answers you seek.

With the Python Tech Radar, you'll get unique insights from a community of Python experts. Our resident team of specialists have shared their expertise on the most cutting-edge developments in the Python world—with an additional guest contribution from Łukasz Langa, CPython Developer in Residence at the Python Software Foundation. On top of that, we have also compiled a report based on data collected from a survey of 100+ Python developers of various levels of advancement. All in all, the result is over 100 pages of in-depth observations on the current state of Python and its future.

Python Tech Radar

Contents

1 Basic Decorators

1.1 Wrapping Functions and Classes

1.2 Examples from the Core Language

1.3 Examples from Other Libraries

1.4 Closures

1.5 Writing a Simple Decorator

1.6 Best Practice

1.7 Use cases

1.8 Exercises

1.8.1 Exercise 1

Write a function decorator that can be used to measure the run time of a function. Use `timeit.default_timer()` to get time stamps.

Solution

```
"""Decorator for measuring run times.
"""

import functools
import time
import timeit

run_times = {}


def measure_time(func):
    """Decorator to measure function run times.
    """
    @functools.wraps(func)
    def _proxy_func(*args, **kwargs):
```

(continues on next page)

(continued from previous page)

```
        """Function that will replace the original function.
        """
        start = timeit.default_timer()
        res = func(*args, **kwargs)
        end = timeit.default_timer()
        key = '.'.join((func.__module__, func.__name__))
        run_times[key] = end - start
        return res
    return _proxy_func


@measure_time
def add(arg1, arg2):
    """Test function.
    """
    return arg1 + arg2


@measure_time
def run_long():
    """Long running function.
    """
    time.sleep(1)


if __name__ == '__main__':
    add(10, 20)
    run_long()
    print(run_times)
```

1.8.2 Exercise 2

Use `functools.wraps()` to preserve the function attributes including the docstring that you wrote.

Solution

Already done in previous exercises.

2 Advanced Decorators

2.1 Parameterized Decorators

2.2 Chaining Decorators

2.3 Callable Instances

2.4 Use Cases

2.5 Class Decorators

2.6 Use Cases

2.7 Exercises

2.7.1 Exercise 3

Modify your solution from exercise 2. Measure the average run time for multiple runs of the function. To achieve this, make the decorator parameterized. It should take an integer that specifies how often the function has to be run. Make sure you divide the resulting run time by this number.

Solution

```
"""Decorator for measuring run times.

Paramterized version.
"""

import functools
import time
import timeit

run_times = {}


def measure_time(repeat=1):
    """Decorator to measure function run times.
    """
    def _measure_time(func):
        """Function that takes the function that is to be wrapped.
```

(continues on next page)

(continued from previous page)

```
        """
        @functools.wraps(func)
        def _proxy_func(*args, **kwargs):
            """Function that will replace the original function.
            """
            start = timeit.default_timer()
            for _ in range(repeat):
                res = func(*args, **kwargs)
            end = timeit.default_timer()
            key = '.'.join((func.__module__, func.__name__))
            run_times[key] = (end - start) / repeat
            return res
        return _proxy_func
    return _measure_time


# Run the function a hundred time..
@measure_time(100)
def add(arg1, arg2):
    """Test function.
    """
    return arg1 + arg2


# Ones is enough.
@measure_time()
def run_long():
    """Long running function.
    """
    time.sleep(1)


if __name__ == '__main__':
    add(10, 20)
    run_long()
    print(run_times)
```

2.7.2 Exercise 4

Make the time measurement optional by using a global switch in the module that can be set to `True` or `False` to turn time measurement on or off.

Solution

```
"""Decorator for measuring run times.

Turn measurement on/off.
"""

import functools
import time
import timeit

MEASURE = True
RUN_TIMES = {}
```

(continues on next page)

(continued from previous page)

```python
def measure_time(repeat=1, run_times=None):
    """Decorator to measure function run times.
    """
    if run_times is None:
        run_times = RUN_TIMES

    def _measure_time(func):
        """Function that takes the function that is to be wrapped.
        """
        if MEASURE:
            @functools.wraps(func)
            def _proxy_func(*args, **kwargs):
                """Function that will replace the original function.
                """

                start = timeit.default_timer()
                for _ in range(repeat):
                    res = func(*args, **kwargs)
                end = timeit.default_timer()
                key = '.'.join((func.__module__, func.__name__))
                run_times[key] = (end - start) / repeat
                return res
            return _proxy_func
        return func
    return _measure_time


# Run the function a hundred time..
@measure_time(100)
def add(arg1, arg2):
    """Test function.
    """
    return arg1 + arg2


# Ones is enough.
@measure_time()
def run_long():
    """Long running function.
    """
    time.sleep(1)


if __name__ == '__main__':
    add(10, 20)
    run_long()
    print('RUN_TIMES:', RUN_TIMES)
```

2.7.3 Exercise 5

Write another decorator that can be used with a class and registers every class that it decorates in a dictionary. Use a string consisting of the module name (`cls.__module__`) and the class name (`cls.__name__`) as key for each class.

Solution

```python
"""Decorator for classes
"""

classes = {}


def register(cls):
    """Register classes.
    """
    classes['.'.join((cls.__module__, cls.__name__))] = cls
    return cls


if __name__ == '__main__':

    #  pylint: disable-msg=too-few-public-methods

    @register
    class Sample1:
        """A sample class.
        """

    @register
    class Sample2:
        """A sample class.
        """

    print(classes)
```