sandeepsuryaprasad / **python_tutorials** Private

Code Issues Pull requests Actions Projects Security Ins

master python_tutorials / 6_decorators / **_decorators.py** / Jump to

Go to file ...

Sandeep Suryaprasad testing Latest commit 8d2eb85 on 4 Feb **History**

0 contributors

366 lines (315 sloc) | 9.41 KB

Raw Blame

```python
import time
import csv
import tracemalloc
from time import sleep

# Decorators:

'''
1. Decorator is a function! Which adds an extra functionality to the existing
without modifying the original function or existing function!

2. First Class Functions are the one which is treated as any other object in
You can pass a function to another function, you can return a function from a
A Decoretor is a function, which takes another function as an argument, adds
and returns another function without altering the source code of original fun
'''



# Log Decorator
def logging(msg="Hello World", debug=True):
    def log(func):
        def wrapper(*args, **kwargs):
            if debug:
                print(msg, func.__name__)
            return func(*args, **kwargs)
        return wrapper
    return log
```

```


# Delay Decorator
def _delay(_time_delay):
    def delay(func):
        def wrapper(*args, **kwargs):
            time.sleep(_time_delay)
            return func(*args, **kwargs)
        return wrapper
    return delay

# Reverse Decorator
def reverse(func):
    def wrapper(*args, **kwargs):
        result = func(*args, **kwargs)
        if isinstance(result, str):
            return result[::-1]
        return result
    return wrapper


# Time Decorator
def _time(func):
    def wrapper(*args, **kwargs):
        start = time.time()
        result = func(*args, **kwargs)
        end = time.time()
        print(f'Exe Time for {func.__name__} : {end-start}')
        return result
    return wrapper


# Positive Decorator
def positive(func):
    def wrapper(*args, **kwargs):
        result = func(*args, **kwargs)
        return abs(result)
    return wrapper

# Caches the argument and its result in a dictionary.
# If the function is called with the same argument, decorator will not re-exe
# It looks up for the result in dictionary and returns the result.
def cache(func):
    _cache = {}
    def wrapper(*args, **kwargs):
```

```python
        if args not in _cache:
            result = func(*args, **kwargs)
            _cache[args] = result
            return result
        print('returning cached result')
        return _cache[args]
    return wrapper

@cache
def add(a, b):
    sleep(10)
    return a+b
# =======================================================
# Using inbuilt lru_cahce decorator
from functools import lru_cache
@lru_cache
def is_prime(number):
    print('calling is_prime function')
    for n in range(2, number):
        if number % n == 0:
            return False
    return True

@lru_cache
def add(a, b):
    print('calling add function')
    return a+b
# =======================================================
# Repeats the function 'n' times
def _repeat(n):
    def repeat(func):
        def wrapper(*args, **kwargs):
            for _ in range(n):
                result = func(*args, **kwargs)
            return result
        return wrapper
    return repeat

# Counting Number of Function Calls.
from collections import defaultdict
_count = defaultdict(int)
def func_count(func):
    def wrapper(*args, **kwargs):
        _count[func.__name__] += 1
        return func(*args, **kwargs)
    return wrapper

@func_count
def add(a, b):
    return a+b

@func_count
def sub(a, b):
    return a-b
# =======================================================
# Alternate Method
# =======================================================
def func_count(func):
    func.count = 0
    def wrapper(*args, **kwargs):
        func.count += 1
        print(f"function {func.__name__} was called {func.count} times!")
        return func(*args, **kwargs)
    return wrapper
# =======================================================
# Alternate Method
# =======================================================
# Below decorator just attaches an attribute "count" to the decorated functio
# and returns the same function back
def count(func):
    func.count = 0
    return func

@count
def add(a, b):
    add.count += 1
    return a+b

@count
def sub(a, b):
    sub.count += 1
    return a-b

@count
def mul(a, b):
    mul.count += 1
    return a*b
# =======================================================
# decorator to restrict the number of calls to 5
def max_calls(func):
    func = 0
    def wrapper(*args, **kwargs):
        func.count += 1
        if func.count > 5:
            raise ValueError(f"Cannot call {func.__name__} more than 5 times"
        return func(*args, **kwargs)
    return wrapper

@max_calls      # greet = max_calls(greet)  "greet" will be pointing to "wrap
def greet():
    return "hello world"

# decorator to prefix +91 to the phone number
numbers = [ 1234567890, 9988776655, 1122334455, 910099887766 ]

def add_prefix(number):
    if len(str(number)) == 12 and str(number).startswith("91"):
        return "+" + str(number)[:2] + "-" + str(number)[2:]
    elif len(str(number)) == 10:
        return "+91-" + str(number)
    else:
        return number

def prefix_country_code(func):
    def wrapper(*args, **kwargs):
        numbers, = args
        prefix_numbers = [ add_prefix(number) for number in numbers ]
        return func(prefix_numbers)
    return wrapper

@prefix_country_code
def print_numbers(numbers):
    for number in numbers:
        print(number)

# Type validator decorator for function arguments.
def validate(*types):
    def _validate(func):
        def wrapper(*args, **kwargs):
            for _arg, _type in zip(args, types):
                if not isinstance(_arg, _type):
                    raise TypeError(f'Invalid Type passed for {_arg}')
            return func(*args, **kwargs)
        return wrapper
    return _validate
```

```python

@validate(int, int)
def add(a, b):
    print("Executing Add")
    return a+b

@validate(int, int)
def sub(a, b):
    return a-b

@validate(str, int, float)
def greet(name, age, pay):
    print(f"Hello {name} You are {age} years of age and you have {pay}")


# Separate function for checking type
def type_check(actual_values, exp_types):
    for _type, _value in zip(exp_types, actual_values):
        if not isinstance(_value, _type):
            raise TypeError

# Alternate Solution using Keyword arguments
def validate(**typs):
    def _validate(func):
        def wrapper(*args, **kwargs):
            _actual_values = list(args)
            _expected_types = list(typs.values())
            type_check(_actual_values, _expected_types)
            return func(*args, **kwargs)
        return wrapper
    return _validate

@validate(a=int, b=int)
def add(a, b):
    print("Executing Add")
    return a+b

@validate(a=int, b=int)
def sub(a, b):
    return a-b

@validate(name=str, age=int, pay=float)
def greet(name, age, pay):
    print(f"Hello {name} You are {age} years of age and you have {pay}")

```

```python
# This decorator re-executes the function as long as there is a ValueError
def retry(func):
    def wrapper(*args, **kwargs):
        while True:
            try:
                return func(*args, **kwargs)
            except ValueError:
                print("Retrying")
    return wrapper

import random
@retry
def dice():
    number = random.randint(1, 10)
    if number != 8:
        raise ValueError
    else:
        return number

# Decorator that executes a function for 3 times.
def retry(func):
    def wrapper(*args, **kwargs):
        max_tries = 3
        while max_tries > 0:
            try:
                max_tries -= 1
                return func(*args, **kwargs)
            except ValueError:
                print(f'Invalid Creds, Attempts left {max_tries}')
                if max_tries == 0:
                    print('Your account is locked')
    return wrapper


@retry
def login():
    username = input('Enter Username: ')
    password = input('Enter Passowrd: ')
    if username == "admin" and password == "Password123":
        return "Log in successfull"
    else:
        raise ValueError('Invalid Credentials')

# Memory Decorator
def _memory(func):
    def wrapper(*args, **kwargs):
        tracemalloc.start()
        result = func(*args, **kwargs)
        print(f"Memory Usage: {tracemalloc.get_traced_memory()}")
        tracemalloc.stop()
        return result
    return wrapper

# Handles any kind of exception
def _exception(func):
    def wrapper(*args, **kwargs):
        try:
            result = func(*args, **kwargs)
        except Exception as e:
            print(e)
        else:
            return result
    return wrapper

@_memory
def read_csv():
    with open('data/covid_data.csv') as f:
        records =[]
        rows = csv.reader(f)
        headers = next(rows)    # Skip Headers
        for row in rows:
            records.append((row[2], row[3], row[5]))
        return records

@_memory
def test_list():
    a = []
    for i in range(1000000):
        a.append(i)
    return a


@_memory
def test_tuple():
    a = tuple(list(range(1000000)))
    return a

# Closures
"""
When a function is passed as to other function, the callback function carries
```

```python
related to the environment in which the function was defined.
"""
def add(a, b):
    name = "sandeep"
    def do_add():
        print(f"hello {name}")
        return a+b
    return do_add

def delay(seconds, func):
    sleep(seconds)
    return func()

# the value of variables "a", "b" and "name" will be carried by function "add
delay(5, add)

# Few function attributes
"""
1. __name__
2. __qualname__
3. __doc__
4. __annotations__
5. __closure__
"""
```