

 sandeepsuryaprasad / python_tutorials Private[Code](#) [Issues](#) [Pull requests](#) [Actions](#) [Projects](#) [Security](#) [Insights](#)

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



```
1  import time
2  import csv
3  import tracemalloc
4  from time import sleep
5
6  # Decorators:
7
8  '''
9  1. Decorator is a function! Which adds an extra functionality to the existing
10 without modifying the original function or existing function!
11
12 2. First Class Functions are the one which is treated as any other object in
13 You can pass a function to another function, you can return a function from a
14 A Decorator is a function, which takes another function as an argument, adds
15 and returns another function without altering the source code of original fun
16 '''
17
18
19 # Log Decorator
20 def logging(msg="Hello World", debug=True):
21     def log(func):
22         def wrapper(*args, **kwargs):
23             if debug:
24                 print(msg, func.__name__)
25             return func(*args, **kwargs)
26         return wrapper
27     return log
```

```
28
29
30 # Delay Decorator
31 def _delay(_time_delay):
32     def delay(func):
33         def wrapper(*args, **kwargs):
34             time.sleep(_time_delay)
35             return func(*args, **kwargs)
36         return wrapper
37     return delay
38
39 # Reverse Decorator
40 def reverse(func):
41     def wrapper(*args, **kwargs):
42         result = func(*args, **kwargs)
43         if isinstance(result, str):
44             return result[::-1]
45         return result
46     return wrapper
47
48
49 # Time Decorator
50 def _time(func):
51     def wrapper(*args, **kwargs):
52         start = time.time()
53         result = func(*args, **kwargs)
54         end = time.time()
55         print(f'Exe Time for {func.__name__} : {end-start}')
56         return result
57     return wrapper
58
59
60 # Positive Decorator
61 def positive(func):
62     def wrapper(*args, **kwargs):
63         result = func(*args, **kwargs)
64         return abs(result)
65     return wrapper
66
67 # Caches the argument and its result in a dictionary.
68 # If the function is called with the same argument, decorator will not re-exe
69 # It looks up for the result in dictionary and returns the result.
70 def cache(func):
71     _cache = {}
72     def wrapper(*args, **kwargs):
```

```
73         if args not in _cache:
74             result = func(*args, **kwargs)
75             _cache[args] = result
76             return result
77         print('returning cached result')
78         return _cache[args]
79     return wrapper
80
81 @cache
82 def add(a, b):
83     sleep(10)
84     return a+b
85
86 # =====
87 # Using inbuilt lru_cache decorator
88 from functools import lru_cache
89 @lru_cache
90 def is_prime(number):
91     print('calling is_prime function')
92     for n in range(2, number):
93         if number % n == 0:
94             return False
95     return True
96
97 @lru_cache
98 def add(a, b):
99     print('calling add function')
100     return a+b
101
102 # =====
103 # Repeats the function 'n' times
104 def _repeat(n):
105     def repeat(func):
106         def wrapper(*args, **kwargs):
107             for _ in range(n):
108                 result = func(*args, **kwargs)
109             return result
110         return wrapper
111     return repeat
112
113 # Counting Number of Function Calls.
114 from collections import defaultdict
115 _count = defaultdict(int)
116 def func_count(func):
117     def wrapper(*args, **kwargs):
118         _count[func.__name__] += 1
119     return func(*args, **kwargs)
```

```
118         return wrapper
119
120 @func_count
121 def add(a, b):
122     return a+b
123
124 @func_count
125 def sub(a, b):
126     return a-b
127 # =====
128 # Alternate Method
129 # =====
130 def func_count(func):
131     func.count = 0
132     def wrapper(*args, **kwargs):
133         func.count += 1
134         print(f"function {func.__name__} was called {func.count} times!")
135         return func(*args, **kwargs)
136     return wrapper
137 # =====
138 # Alternate Method
139 # =====
140 # Below decorator just attaches an attribute "count" to the decorated function
141 # and returns the same function back
142 def count(func):
143     func.count = 0
144     return func
145
146 @count
147 def add(a, b):
148     add.count += 1
149     return a+b
150
151 @count
152 def sub(a, b):
153     sub.count += 1
154     return a-b
155
156 @count
157 def mul(a, b):
158     mul.count += 1
159     return a*b
160 # =====
161 # decorator to restrict the number of calls to 5
162 def max_calls(func):
```

```
163     func = 0
164     def wrapper(*args, **kwargs):
165         func.count += 1
166         if func.count > 5:
167             raise ValueError(f"Cannot call {func.__name__} more than 5 times")
168         return func(*args, **kwargs)
169     return wrapper
170
171 @max_calls      # greet = max_calls(greet)  "greet" will be pointing to "wrap
172 def greet():
173     return "hello world"
174
175 # decorator to prefix +91 to the phone number
176 numbers = [ 1234567890, 9988776655, 1122334455, 910099887766 ]
177
178 def add_prefix(number):
179     if len(str(number)) == 12 and str(number).startswith("91"):
180         return "+" + str(number)[:2] + "-" + str(number)[2:]
181     elif len(str(number)) == 10:
182         return "+91-" + str(number)
183     else:
184         return number
185
186 def prefix_country_code(func):
187     def wrapper(*args, **kwargs):
188         numbers, = args
189         prefix_numbers = [ add_prefix(number) for number in numbers ]
190         return func(prefix_numbers)
191     return wrapper
192
193 @prefix_country_code
194 def print_numbers(numbers):
195     for number in numbers:
196         print(number)
197
198 # Type validator decorator for function arguments.
199 def validate(*types):
200     def _validate(func):
201         def wrapper(*args, **kwargs):
202             for _arg, _type in zip(args, types):
203                 if not isinstance(_arg, _type):
204                     raise TypeError(f'Invalid Type passed for {_arg}')
205             return func(*args, **kwargs)
206         return wrapper
207     return _validate
```

```
208
209 @validate(int, int)
210 def add(a, b):
211     print("Executing Add")
212     return a+b
213
214 @validate(int, int)
215 def sub(a, b):
216     return a-b
217
218 @validate(str, int, float)
219 def greet(name, age, pay):
220     print(f"Hello {name} You are {age} years of age and you have {pay}")
221
222
223 # Separate function for checking type
224 def type_check(actual_values, exp_types):
225     for _type, _value in zip(exp_types, actual_values):
226         if not isinstance(_value, _type):
227             raise TypeError
228
229 # Alternate Solution using Keyword arguments
230 def validate(**types):
231     def _validate(func):
232         def wrapper(*args, **kwargs):
233             _actual_values = list(args)
234             _expected_types = list(types.values())
235             type_check(_actual_values, _expected_types)
236             return func(*args, **kwargs)
237         return wrapper
238     return _validate
239
240 @validate(a=int, b=int)
241 def add(a, b):
242     print("Executing Add")
243     return a+b
244
245 @validate(a=int, b=int)
246 def sub(a, b):
247     return a-b
248
249 @validate(name=str, age=int, pay=float)
250 def greet(name, age, pay):
251     print(f"Hello {name} You are {age} years of age and you have {pay}")
252
```

```
253 # This decorator re-executes the function as long as there is a ValueError
254 def retry(func):
255     def wrapper(*args, **kwargs):
256         while True:
257             try:
258                 return func(*args, **kwargs)
259             except ValueError:
260                 print("Retrying")
261         return wrapper
262
263 import random
264 @retry
265 def dice():
266     number = random.randint(1, 10)
267     if number != 8:
268         raise ValueError
269     else:
270         return number
271
272 # Decorator that executes a function for 3 times.
273 def retry(func):
274     def wrapper(*args, **kwargs):
275         max_tries = 3
276         while max_tries > 0:
277             try:
278                 max_tries -= 1
279                 return func(*args, **kwargs)
280             except ValueError:
281                 print(f'Invalid Creds, Attempts left {max_tries}')
282                 if max_tries == 0:
283                     print('Your account is locked')
284         return wrapper
285
286
287 @retry
288 def login():
289     username = input('Enter Username: ')
290     password = input('Enter Passowrd: ')
291     if username == "admin" and password == "Password123":
292         return "Log in successfull"
293     else:
294         raise ValueError('Invalid Credentials')
295
296 # Memory Decorator
297 def _memory(func):
```

```
298     def wrapper(*args, **kwargs):
299         tracemalloc.start()
300         result = func(*args, **kwargs)
301         print(f"Memory Usage: {tracemalloc.get_traced_memory()}")
302         tracemalloc.stop()
303         return result
304     return wrapper
305
306 # Handles any kind of exception
307 def _exception(func):
308     def wrapper(*args, **kwargs):
309         try:
310             result = func(*args, **kwargs)
311         except Exception as e:
312             print(e)
313         else:
314             return result
315     return wrapper
316
317 @_memory
318 def read_csv():
319     with open('data/covid_data.csv') as f:
320         records = []
321         rows = csv.reader(f)
322         headers = next(rows)    # Skip Headers
323         for row in rows:
324             records.append((row[2], row[3], row[5]))
325     return records
326
327 @_memory
328 def test_list():
329     a = []
330     for i in range(1000000):
331         a.append(i)
332     return a
333
334
335 @_memory
336 def test_tuple():
337     a = tuple(list(range(1000000)))
338     return a
339
340 # Closures
341 """
342 When a function is passed as to other function, the callback function carries
```



```
343 related to the environment in which the function was defined.
344 """
345 def add(a, b):
346     name = "sandeep"
347     def do_add():
348         print(f"hello {name}")
349         return a+b
350     return do_add
351
352 def delay(seconds, func):
353     sleep(seconds)
354     return func()
355
356 # the value of variables "a", "b" and "name" will be carried by function "add
357 delay(5, add)
358
359 # Few function attributes
360 """
361 1. __name__
362 2. __qualname__
363 3. __doc__
364 4. __annotations__
365 5. __closure__
366 """
```