

[sandeepsuryaprasad](#) / [python\\_tutorials](#) Private[Code](#) [Issues](#) [Pull requests](#) [Actions](#) [Projects](#) [Wiki](#) [Security](#)

master ▾

[python\\_tutorials / 6\\_decorators /](#)  
[\\_decorators.py](#) / [Jump to ▾](#)[Go to file](#)

...

**Sandeep Suryaprasad** added deco...

Latest commit 9dc75ff on 27 Aug

[History](#)[0 contributors](#)

383 lines (330 sloc) | 9.84 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 # Decorator that allows positional only arguments
68 # Solution:1
69 def positional_only(func):
70     def wrapper(*args, **kwargs):
71         if len(kwargs) == 0:
72             result = func(*args, **kwargs)
```

```
73         return result
74         raise Exception("Only Positional Arguments are allowed")
75     return wrapper
76
77 # Solution:2
78 def positional_only(func):
79     def wrapper(*args):
80         result = func(*args)
81         return result
82     return wrapper
83
84 # Caches the argument and its result in a dictionary.
85 # If the function is called with the same argument, decorator will not re-exe
86 # It looks up for the result in dictionary and returns the result.
87 def cache(func):
88     _cache = {}
89     def wrapper(*args, **kwargs):
90         if args not in _cache:
91             result = func(*args, **kwargs)
92             _cache[args] = result
93             return result
94         print('returning cached result')
95         return _cache[args]
96     return wrapper
97
98 @cache
99 def add(a, b):
100     sleep(10)
101     return a+b
102 # =====
103 # Using inbuilt lru_cahce decorator
104 from functools import lru_cache
105 @lru_cache
106 def is_prime(number):
107     print('calling is_prime function')
108     for n in range(2, number):
109         if number % n == 0:
110             return False
111     return True
112
113 @lru_cache
114 def add(a, b):
115     print('calling add function')
116     return a+b
117 # =====
```

```
118 # Repeats the function 'n' times
119 def _repeat(n):
120     def repeat(func):
121         def wrapper(*args, **kwargs):
122             for _ in range(n):
123                 result = func(*args, **kwargs)
124             return result
125         return wrapper
126     return repeat
127
128 # Counting Number of Function Calls.
129 from collections import defaultdict
130 _count = defaultdict(int)
131 def func_count(func):
132     def wrapper(*args, **kwargs):
133         _count[func.__name__] += 1
134         return func(*args, **kwargs)
135     return wrapper
136
137 @func_count
138 def add(a, b):
139     return a+b
140
141 @func_count
142 def sub(a, b):
143     return a-b
144 # =====
145 # Alternate Method
146 # =====
147 def func_count(func):
148     func.count = 0
149     def wrapper(*args, **kwargs):
150         func.count += 1
151         print(f"function {func.__name__} was called {func.count} times!")
152         return func(*args, **kwargs)
153     return wrapper
154 # =====
155 # Alternate Method
156 # =====
157 # Below decorator just attaches an attribute "count" to the decorated function
158 # and returns the same function back
159 def count(func):
160     func.count = 0
161     return func
162
```

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

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

```
253         return func(*args, **kwargs)
254     return wrapper
255     return _validate
256
257 @validate(a=int, b=int)
258 def add(a, b):
259     print("Executing Add")
260     return a+b
261
262 @validate(a=int, b=int)
263 def sub(a, b):
264     return a-b
265
266 @validate(name=str, age=int, pay=float)
267 def greet(name, age, pay):
268     print(f"Hello {name} You are {age} years of age and you have {pay}")
269
270 # This decorator re-executes the function as long as there is a ValueError
271 def retry(func):
272     def wrapper(*args, **kwargs):
273         while True:
274             try:
275                 return func(*args, **kwargs)
276             except ValueError:
277                 print("Retrying")
278     return wrapper
279
280 import random
281 @retry
282 def dice():
283     number = random.randint(1, 10)
284     if number != 8:
285         raise ValueError
286     else:
287         return number
288
289 # Decorator that executes a function for 3 times.
290 def retry(func):
291     def wrapper(*args, **kwargs):
292         max_tries = 3
293         while max_tries > 0:
294             try:
295                 max_tries -= 1
296                 return func(*args, **kwargs)
297             except ValueError:
```

```
298             print(f'Invalid Creds, Attempts left {max_tries}')
299             if max_tries == 0:
300                 print('Your account is locked')
301         return wrapper
302
303
304 @retry
305 def login():
306     username = input('Enter Username: ')
307     password = input('Enter Passowrd: ')
308     if username == "admin" and password == "Password123":
309         return "Log in successfull"
310     else:
311         raise ValueError('Invalid Credentials')
312
313 # Memory Decorator
314 def _memory(func):
315     def wrapper(*args, **kwargs):
316         tracemalloc.start()
317         result = func(*args, **kwargs)
318         print(f"Memory Usage: {tracemalloc.get_traced_memory()}")
319         tracemalloc.stop()
320         return result
321     return wrapper
322
323 # Handles any kind of exception
324 def _exception(func):
325     def wrapper(*args, **kwargs):
326         try:
327             result = func(*args, **kwargs)
328         except Exception as e:
329             print(e)
330         else:
331             return result
332     return wrapper
333
334 @_memory
335 def read_csv():
336     with open('data/covid_data.csv') as f:
337         records = []
338         rows = csv.reader(f)
339         headers = next(rows)    # Skip Headers
340         for row in rows:
341             records.append((row[2], row[3], row[5]))
342     return records
```



```
343
344 @_memory
345 def test_list():
346     a = []
347     for i in range(1000000):
348         a.append(i)
349     return a
350
351
352 @_memory
353 def test_tuple():
354     a = tuple(list(range(1000000)))
355     return a
356
357 # Closures
358 """
359 When a function is passed as to other function, the callback function carries
360 related to the environment in which the function was defined.
361 """
362 def add(a, b):
363     name = "sandeep"
364     def do_add():
365         print(f"hello {name}")
366         return a+b
367     return do_add
368
369 def delay(seconds, func):
370     sleep(seconds)
371     return func()
372
373 # the value of variables "a", "b" and "name" will be carried by function "add
374 delay(5, add)
375
376 # Few function attributes
377 """
378 1. __name__
379 2. __qualname__
380 3. __doc__
381 4. __annotations__
382 5. __closure__
383 """
```

[Give feedback](#)