△ sandeepsuryaprasad / python_tutorials (Private)

```
11 Pull requests
                                         Actions
<> Code
           (•) Issues
                python_tutorials / 6_decorators /
   master •
                                                                   Go to file
                _decorators.py / <> Jump to ▼
     Sandeep Suryaprasad testing
                                          Latest commit 8d2eb85 on 4 Feb (S) History
 ৪३ 0 contributors
                                                                       ſΩ
 366 lines (315 sloc) 9.41 KB
                                                    Raw
                                                          Blame
       import time
   2
       import csv
   3
       import tracemalloc
   4
       from time import sleep
   5
       # Decorators:
   6
   7
       1 1 1
   8
   9
       1. Decorator is a function! Which adds an extra functionality to the existing
       without modifying the original function or existing function!
  10
  11
  12
       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
  13
       A Decoretor is a function, which takes another function as an argument, adds
  14
       and returns another function without altering the source code of original fun
  15
       1.1.1
  16
  17
  18
  19
       # Log Decorator
  20
       def logging(msg="Hello World", debug=True):
           def log(func):
  21
  22
               def wrapper(*args, **kwargs):
  23
                   if debug:
                       print(msg, func.__name__)
  24
                   return func(*args, **kwargs)
  25
  26
               return wrapper
  27
           return log
```

```
28
29
30
     # Delay Decorator
     def _delay(_time_delay):
31
         def delay(func):
32
             def wrapper(*args, **kwargs):
33
34
                 time.sleep(_time_delay)
                 return func(*args, **kwargs)
35
36
             return wrapper
37
         return delay
38
39
     # Reverse Decorator
     def reverse(func):
40
41
         def wrapper(*args, **kwargs):
             result = func(*args, **kwargs)
42
43
             if isinstance(result, str):
                 return result[::-1]
44
45
             return result
         return wrapper
46
47
48
     # Time Decorator
49
     def time(func):
50
         def wrapper(*args, **kwargs):
51
52
             start = time.time()
             result = func(*args, **kwargs)
53
             end = time.time()
54
             print(f'Exe Time for {func.__name__} : {end-start}')
55
56
             return result
57
         return wrapper
58
59
60
     # Positive Decorator
     def positive(func):
61
62
         def wrapper(*args, **kwargs):
             result = func(*args, **kwargs)
63
             return abs(result)
64
65
         return wrapper
66
67
     # Caches the argument and its result in a dictionary.
     # If the function is called with the same argument, decorator will not re-exe
68
     # It looks up for the result in dictionary and returns the result.
69
     def cache(func):
70
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
             print('returning cached result')
 77
 78
             return _cache[args]
 79
         return wrapper
 80
 81
     @cache
 82
     def add(a, b):
 83
         sleep(10)
         return a+b
 84
     85
 86
     # Using inbuilt lru_cahce decorator
     from functools import lru cache
 87
     @lru cache
 88
     def is_prime(number):
 89
 90
         print('calling is_prime function')
 91
         for n in range(2, number):
 92
             if number % n == 0:
                 return False
 93
 94
         return True
 95
     @lru_cache
 96
 97
     def add(a, b):
         print('calling add function')
 98
 99
         return a+b
     100
101
     # Repeats the function 'n' times
102
     def _repeat(n):
103
         def repeat(func):
104
             def wrapper(*args, **kwargs):
105
                for _ in range(n):
                    result = func(*args, **kwargs)
106
107
                return result
108
             return wrapper
109
         return repeat
110
     # Counting Number of Function Calls.
111
112
     from collections import defaultdict
     _count = defaultdict(int)
113
     def func count(func):
114
         def wrapper(*args, **kwargs):
115
116
             count[func. name ] += 1
117
             return func(*args, **kwargs)
```

```
118
       return wrapper
119
120
    Ofunc 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
          print(f"function {func.__name__} was called {func.count} times!")
134
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
    # and returns the same function back
141
142
    def count(func):
       func.count = 0
143
       return func
144
145
146
    @count
147
    def add(a, b):
148
       add.count += 1
149
       return a+b
150
    @count
151
152
    def sub(a, b):
       sub.count += 1
153
       return a-b
154
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
                      # greet = max_calls(greet) "greet" will be pointing to "wrap
171
      @max_calls
172
      def greet():
173
          return "hello world"
174
      # decorator to prefix +91 to the phone number
175
176
      numbers = [ 1234567890, 9988776655, 1122334455, 910099887766 ]
177
178
      def add prefix(number):
          if len(str(number)) == 12 and str(number).startswith("91"):
179
180
              return "+" + str(number)[:2] + "-" + str(number)[2:]
          elif len(str(number)) == 10:
181
182
              return "+91-" + str(number)
183
          else:
184
              return number
185
      def prefix_country_code(func):
186
187
          def wrapper(*args, **kwargs):
188
              numbers, = args
              prefix numbers = [ add prefix(number) for number in numbers ]
189
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):
                      if not isinstance(_arg, _type):
203
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(**typs):
          def validate(func):
231
232
              def wrapper(*args, **kwargs):
233
                  _actual_values = list(args)
234
                  _expected_types = list(typs.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")
          return a+b
243
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
```

```
# This decorator re-executes the function as long as there is a ValueError
253
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
      def dice():
265
266
          number = random.randint(1, 10)
          if number != 8:
267
              raise ValueError
268
269
          else:
270
              return number
271
272
      # Decorator that executes a function for 3 times.
273
      def retry(func):
          def wrapper(*args, **kwargs):
274
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():
          username = input('Enter Username: ')
289
          password = input('Enter Passowrd: ')
290
          if username == "admin" and password == "Password123":
291
292
              return "Log in successfull"
293
          else:
              raise ValueError('Invalid Credentials')
294
295
296
      # Memory Decorator
297
      def _memory(func):
```

```
def wrapper(*args, **kwargs):
298
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:
                  records.append((row[2], row[3], row[5]))
324
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):
          name = "sandeep"
346
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
      # the value of variables "a", "b" and "name" will be carried by function "add
356
357
      delay(5, add)
358
359
      # Few function attributes
      0.00
360
361
      1. __name__
362
      2. __qualname__
363
      3. __doc__
      4. __annotations__
364
365
      5. __closure__
      0.00
366
```