ECE326

PROGRAMMING LANGUAGES

Lecture 16 : Python Decorator

Kuei (Jack) Sun

ECE

University of Toronto

Fall 2019

Decorator

- A way to augment an existing function or class
  - E.g. Do something before and/or after a function call
  - E.g. modify a class in some ways

```
@function_decorator
def foo():
    pass
```

```
@class_decorator
class Foo:
    pass
```

- Decorator is a syntactic sugar
  - A callable that returns a callable object (e.g. function or class)

```
# equivalent to the definitions above
foo = function_decorator(foo)
Foo = class_decorator(Foo)
```

Callable

- An entity that can be called
  - Accepts some argument(s) and returns some value(s)
- Functor
  - A function object
  - In Python, an instance of a class that implements __call__
  - In C++, an instance of a class that implements operator()
  - Has a different meaning in mathematics, do not confuse!

```
class Foo:                      >> f = Foo()
    def __call__(self, arg):    >> f(5)       # works just like
        return arg + 2          7             # a function
```

Inner Function

- A function defined inside a function
- Also known as a nested function
- Have access to variables in enclosing function
  - Known as outer variables
  - Requires nonlocal statement to reassign outer variables

```
def outside():
  outsideList = [1, 2]
  def nested():
    outsideList.append(3)
  nested()
  print(outsideList)
  # [1, 2, 3]
```

```
def outer():
  stuff = None
  def inner():
    nonlocal stuff
    stuff = 5
  inner()
  return stuff    # returns 5
```

Closure

- An inner function that is *returned* by the outer function that uses outer variables
  - Accessible even after the outer function finishes!
  - Retain values they had at the time the closure was defined

```
def adder(outer_argument):                   # outer function
    def inner(inner_argument):               # inner function
        # ** notice outer_argument **
        return outer_argument + inner_argument
    return inner

add5 = adder(5)  # a function that adds 5 to its argument
add7 = adder(7)  # a function that adds 7 to its argument
print add5(3)    # prints 8
print add7(3)    # prints 10
```

Decorator

- Wraps a function and modify its behaviour

```
def time_me(func):
    def wrapper(*args, **kwargs):
        start = datetime.now()
        ret = func(*args, **kwargs)
        diff = datetime.now() - start
        print("Took”, diff, “to run”, func.__name__))
        return ret
    return wrapper

@time_me
def non_recursive_expensive_function(n):
    … # does some expensive computation

>> non_recursive_expensive_function(100)
Took 0:01:02.423414 to run non_recursive_expensive_function
```

Decorator

- Decorator permanently alters the function
  - All later calls are also “decorated”
  - Make sure that’s the behaviour you want

```
@time_me
def fib(n):
    return 1 if n < 2 else fib(n-1) + fib(n-2)
```

```
>> fib(5)
Took 0:00:00.000004 to run fib
Took 0:00:00.000006 to run fib
Took 0:00:00.000281 to run fib
…
```

```
@time_me
def fibonacci(n):    # fixed version
    return fib(n)    # assume fib(n) no longer decorated
```

Pros and Cons

- Benefits
    - Explicit syntax
        - easy to notice augmentation on function/class
    - Code reuse
        - Same decorator can be used for many functions/classes
- Drawbacks
    - Performance
        - Decorator requires additional function call(s)
    - Type change
        - Function/class now exists inside a wrapper, may break existing usage

Decorator with Arguments

- Decorators can take arguments

```
def repeat(ntimes):
    def decorator(func):
        def repeater(*args, **kwargs):
            for __ in range(ntimes):
                func(*args, **kwargs)     # no return value
        return repeater
    return decorator

# same as foo = repeat(5)(foo)
@repeat(5)
def foo(msg):
    print(msg, end=“ ”)
    return len(msg)
```

```
>> a = foo(“hi”)
hi hi hi hi hi
>> a
None
```

Decorator can “eat” your return value, beware

Caveat

- Decorator breaks some introspection attributes

```
def foo(msg):
    “foo prints msg without new line”
    print(msg, end=“ ”)
    return len(msg)

>> foo.__name__, foo.__doc__
(‘foo’, ‘foo prints msg without new line’)

@repeat(5)
def foo(msg):
    …

>> foo.__name__, foo.__doc__
(‘repeater’, None)
```

This is called a doc string, which allows you to use help(object) during interactive use .

wraps Decorator

- If it matters to you, fix it via wraps decorator

```
import functools
def twice(func):
    ntimes = 2
    @functools.wraps(func)           # fixes introspection
    def repeater(*args, **kwargs):
        for __ in range(ntimes):
            func(*args, **kwargs)
    return repeater

@twice
def foo(msg):
    …

>> foo.__name__, foo.__doc__
(‘foo’, ‘foo prints msg without new line’)
```

Decorator Example

- Decorator does not have to alter function
  - Can be used to register certain functions for other use

```
EXPORTS = {}
def register(func):
    EXPORTS[func.__name__] = func
    return func

@register
def mangle(text):
    … # scrambles text and returns it

def manipulate_text(transformer, text):
    if transformer in EXPORTS:
        return EXPORTS[transformer](text)
    return text
```

Decorator Example

- Modifying output of function

```
def add_unit(unit):
    def decorator(func):
        def transform(*args, **kwargs):
            # assume this class keeps magnitude and unit
            return Quantity(func(*args, **kwargs), unit)
        return transform
    return decorator

# assume we know this function returns speed in km/hr
@add_unit(“km/hr”)
def speed(distance, time):
    …

>> print(speed(10, datetime.timedelta(minutes=5))
120 km/hr
```

Input Type Checking

```
def type_check(**argchecks):
    def decorator(func):
        code = func.__code__
        allargs = list(code.co_varnames[:code.co_argcount])
        def on_call(*pargs, **kargs):
            positionals = allargs[:len(pargs)]
            for argname, argtype in argchecks.items():
                if argname in kargs:
                    assert(isinstance(kargs[argname], argtype))
                elif argname in positionals:
                    pos = positionals.index(argname)
                    assert(isinstance(pargs[pos], argtype))
            return func(*pargs, **kargs)
        return on_call
    return decorator
```

__code__

- Contains information about the function’s code
  - `code.co_varnames`: local variables, first *N* are arguments
  - `code.co_argcount`: number of arguments

```
@type_check(a=int, b=float)
def foo(a, b):              # co_varnames = (‘a’, ‘b’, ‘c’)
    c = 5                   # co_argcount = 2
    return a + b + c

>> foo(2, 2.3)          # OK
>> foo(5, b=3.3)        # OK
>> foo(3, 4)            # FAIL – b is not a float
AssertionError
```

Built-in Decorators

- Decorators we have seen (or used)

```
class Foo:
    default = 0

    @property
    def value(self):
        return getattr(self, ‘_v’, self.default)

    @value.setter
    def value(self, v):     # method name MUST be the same!
        self._v = v

    @classmethod
    def setDefault(cls, v):
        cls.default = v
```

Decorator with Functor

```
class counter:
    def __init__(self, func):
        self.func = func
        self.count = 0

    def __call__(self, *args, **kwargs):
        self.count += 1
        return self.func(*args, **kwargs)

@counter
def fibonacci(n):
    return 1 if n < 2 else fib(n-1) + fib(n-2)

>> fibonacci(10)
>> fibonacci.count
177
```

Problem with Method

- Decorator made with functor requires self in __call__
  - When decorating method, which self does it refer it?
    - The decorator, or the instance of the decorated method?

```
class Foo:
    @counter            # this doesn’t work
    def add(self, n):
        self.x += n
```

- Solution
  - Must use a proxy object

Workaround

```
class proxy:
    def __init__(self, desc, inst):
        self.desc, self.inst = desc, inst

    def __call__(self, *args, **kwargs):
        return self.desc(self.inst, *args, **kwargs)

class counter:
    def __init__(self, func):
        self.func = func
        self.count = 0

    # from descriptor!
    def __get__(self, instance, owner):
        return proxy(self, instance)
```

```
>> f = Foo()
>> f.add(2)
>> f.add(6)
# must go through desc
>> f.add.desc.count
2
```

Class Decorator

- Example: Singleton

```
def singleton(aClass):
    instance = None
    def onCall(*args, **kwargs):     # On instance creation
        nonlocal instance
        if instance == None:
            instance = aClass(*args, **kwargs)
        else:
            raise RuntimeError(“Cannot instantiate ” + \
                aClass.__name__ + “ more than once”)
        return instance
    return onCall

@singleton
class Calculator:
    …
```

```
>> a = Calculator()
>> b = Calculator()
RuntimeError: Cannot instantiate Calculator
more than once
```

Class Wrapper

```
def tracer(Cls): # On @ decorator
    class Tracer:
        def __init__(self, *args, **kargs):
            self.fetches = defaultdict(int)
            self.wrapped = Cls(*args, **kargs)
        def __getattr__(self, name):
            self.fetches[name] += 1
            return getattr(self.wrapped, name)
    return Tracer

@tracer
class Employee:
    …
    def pay(self):
        return self.hours * \
            self.rate
```

```
>> sue = Employee(“Sue”, 40, 39.99)
>> print(sue.name)
Sue
>> sue.pay()
1599.6
>> sue.fetches
{'name': 1, 'pay': 1}
```

Limitation

- __getattr__ does not work for special built-in methods
- Workaround
  - Use a mixin that to force routing through __getattr__

```
class BuiltinMixin:
    def __init__(self):
        self._getattr = self.__class__.__getattr__

    def __add__(self, other):
        return self._getattr(self, '__add__')(other)

    def __str__(self):
        return self._getattr(self, '__str__')()

    def __getitem__(self, index):
        return self._getattr(self, '__getitem__')(index)
```

Updated Tracer

```
def tracer(Cls):
    class Tracer(BuiltinMixin):
        def __init__(self, *args, **kargs):
            # initialize super class
            super().__init__()
            self.fetches = defaultdict(int)
            self.wrapped = Cls(*args, **kargs)
        def __getattr__(self, name):
            self.fetches[name] += 1
            return getattr(self.wrapped, name)
    return Tracer
```

```
>> sue = Employee(“Sue”, 40, 39.99)
>> print(sue)     # calls __str__
…
>> sue.fetches
{'name': 1, 'pay': 1, '__str__': 1}
```