Functions_decorators_and_properties

November 24, 2019

0.1 Regular Function

```
def first_name(name):
    """Get first name"""
    return name.split()[0]
```

0.2 Anonymous Function (Lambda)

```
lambda name: name.split()[0]
```

Regular Functions	Anonymous Functions
Statement which defines a function and binds it to a name	*Expression* which evaluates to a function
Must have a name	Anonymous
Arguments delimited by parentheses, separated by commas	Argument list terminated by colon, separated by commas
Body is an indented block of statements	Body is a single *expression*
A `return` statement is required to return anything other than `None`	The return value is given by the body *expression*. No `return` statement is permitted
Regular functions can have docstrings	Lambdas cannot have docstrings
Easy to access for testing	Awkward or impossible to test

0.3 Decorators

Are functions which modify/add functionality of other functions

[38]:
```
def my_decorator(func):
    def wrapper():
        print("Before")
        func()
        print("After")
    return wrapper

@my_decorator
def say_whee():
    print("Whee!")
```

[39]:
```
say_whee()
```

```
Before
Whee!
After
```

[40]:
```
class CallCount:
    def __init__(self, f):
        self.f = f
        self.count = 0

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

@CallCount
def hello(name):
    print("Hello, {}".format(name))
```

[41]:
```
hello("Oscar")
hello("Oscar")
hello("Oscar")
hello.count
```

```
Hello, Oscar
Hello, Oscar
Hello, Oscar
```

[41]:
```
3
```

[42]:
```
class Tracer:
    def __init__(self):
        self.enabled = True

    def __call__(self, f):
        def wrap(*args, **kwargs):
            if self.enabled:
                print("Calling {}".format(f))
            return f(*args, **kwargs)
        return wrap

tracer = Tracer()

@tracer
def rotate_list(l):
    return l[1:] + [l[0]]
```

[43]:
```
l = [1, 2, 3]
l = rotate_list(l)
```

```
tracer.enabled = False
l = rotate_list(l)
tracer.enabled = True
l = rotate_list(l)
```

```
Calling <function rotate_list at 0x7f3b2076b0e0>
Calling <function rotate_list at 0x7f3b2076b0e0>
```

[44]:
```
import functools

def noop(f):
    @functools.wraps(f)
    def noop_wrapper():
        return f()
    return noop_wrapper

@noop
def hello():
    "Print a well-known message."
    print("Hello, world!")
```

[45]:
```
help(hello)
```

```
Help on function hello in module __main__:

hello()
    Print a well-known message.
```

[46]:
```
def check_non_negative(index):
    def validator(f):
        def wrap(*args, **kwargs):
            if args[index] < 0:
                raise ValueError("Argument {} must be non-negative".
 ↪format(index))
            return f(*args, **kwargs)
        return wrap
    return validator

@check_non_negative(1)
def create_list(value, size):
    return [value] * size
```

[47]:
```
create_list('a', 3)
```

[47]:
```
['a', 'a', 'a']
```

[48]:
```
create_list(123, -6)
```

```
      ␣
↪---------------------------------------------------------------------------

       ValueError                                Traceback (most recent call␣
↪last)

       <ipython-input-48-aa2ae22b4c92> in <module>
   ----> 1 create_list(123, -6)


       <ipython-input-46-d0019037d0a7> in wrap(*args, **kwargs)
         3         def wrap(*args, **kwargs):
         4             if args[index] < 0:
   ----> 5                 raise ValueError("Argument {} must be non-negative".
↪format(index))
         6             return f(*args, **kwargs)
         7         return wrap


       ValueError: Argument 1 must be non-negative
```

0.4 Properties

Are a type of decorator used to define in a Pythonic way getters and setter for object attributes

@property: to define the getter method for an attribute

@propertyname.setter: to define the setter method for an attribute

[1]:
```
class Person:
    def __init__(self, name):
        self._name = name

    @property
    def name(self):
        return self._name

    @name.setter
    def name(self, name):
        self._name = name
```

[2]:
```
p = Person("Oscar")
print("My name is {}".format(p.name))
p.name = "Maitesin"
```

```
print("My nickname is {}".format(p.name))
```

```
My name is Oscar
My nickname is Maitesin
```