Decorators

Contents

1. Getting started with decorators
2. Additional decorator techniques
3. Parameterized decorators

Demo folder: **AppxB-Decorators**

1. Getting Started with Decorators

- Overview
- Defining a decorator function
- Applying a decorator function manually
- Applying a decorator function properly

Overview

- Python allows you do decorate functions and classes using the `@decorator` syntax
  - The decorator enhances the function/class with extra capabilities

```
@someDecorator
def someFunction(…) :
   …
```

```
@someDecorator
class SomeClass :
   …
```

- This chapter shows how to define decorators
  - We'll see how to define decorator functions
  - We'll also describe how Python applies decorator functions

Defining a Decorator Function (1 of 2)

- Define a function that takes a function pointer as an argument
  - The pointer indicates the target function you want to decorate
- Inside the decorator function, implement a nested function
  - The nested function should call the target function
  - And should also perform the desired decoration behaviour
- At the end of the decorator function, return a pointer to the nested function
- See example on next slide…

Defining a Decorator Function (2 of 2)

- Here's a simple example of a decorator function
  - Invokes the specified target function
  - Decorates the invocation with a pre/post console message

```
def simpleDecorator(func) :

    # Define an inner function, which wraps (decorates) the target function.
    def innerFunc() :
        print("Start of simpleDecorator()")
        func()
        print("End of simpleDecorator()")

    # Return the inner function.
    return innerFunc
```

decorators1.py

Applying a Decorator Function Manually

- In order to understand how decorators work, let's first of all see how to apply a decorator function manually:

```
# Some function that we want to decorate.
def myfunc1() :
    print("Hi from myfunc1()")

# Client code.
pointerToInnerFunc = simpleDecorator(myfunc1)    # Line 1
pointerToInnerFunc()                             # Line 2
```

decorators1.py

- Line 1 calls the decorator function manually, passing a pointer to the target function as an argument
  - This statement returns a pointer to the inner function

- Line 2 calls the inner function
  - This invokes the target function, with the desired decoration

Applying a Decorator Function Properly

- The previous slide showed how to call a decorator function manually, to wrap a target function

- Now let's see how to apply a decorator function properly, i.e. using the `@decorator` syntax

```
# Some function, which we now decorate explicitly.
@simpleDecorator
def myfunc1() :
    print("Hi from myfunc1()")

# Client code.
myfunc1()
```

decorators2.py

- Note the client code just calls `myfunc1()` directly
  - Python intervenes, thanks to the `@simpleDecorator` decorator, and converts the code to the equivalent of the previous slide

2. Additional Decorator Techniques

- Decorating a function that takes arguments
- Decorating a function that returns a result

Decorating a Function that Takes Arguments

- Consider the following function, which takes arguments
  - Note that we've decorated the function

```
@parameterAwareDecorator
def myfunc1(firstName, lastName, nationality) :
    print("Hi %s %s, your nationality is %s" % (firstName, lastName, nationality))
```

- This is how to define the decorator function
  - The inner function receives variadic args and passes to target func

```
def parameterAwareDecorator(func) :

   def innerFunc(*args, **kwargs) :
        print("Start of parameterAwareDecorator()")
        func(*args, **kwargs)
        print("End of parameterAwareDecorator()")

   return innerFunc
```

- Client code:

```
myfunc1("Ola", "Nordmann", "Norsk")
myfunc1(nationality="Cymraeg", lastName="Olsen", firstName="Jayne")
```

decorators3.py

Decorating a Function that Returns a Result

- Consider the following function, which returns a result

```
@returnAwareDecorator
def myfunc1(firstName, lastName, nationality) :
    return "Hi %s %s, your nationality is %s" % (firstName, lastName, nationality)
```

- This is how to define the decorator function
  - The inner function returns the result of the target function

```
def returnAwareDecorator(func) :

    def innerFunc(*args, **kwargs) :
        print("Start of returnAwareDecorator()")
        returnValueFromFunc = func(*args, **kwargs)
        print("End of returnAwareDecorator()")
        return returnValueFromFunc

    return innerFunc
```

- Client code:

```
res1 = myfunc1("Kari", "Nordmann", "Norsk")
res2 = myfunc1(nationality="Cymraeg", lastName="Olsen", firstName="Andy")
```

3. Parameterized Decorators

- Overview
- Defining a parameterized decorator
- Applying a parameterized decorator manually
- Applying a parameterized decorator properly

Overview

- Decorators can take parameters, to make them flexible

- E.g. imagine a flexible decorator that displays custom pre/post messages around a target function call
  - You might apply the decorator as follows
  - The decorator takes parameters specifying the pre/post messages

```
@parameterizedDecorator("HELLO", "GOODBYE")
def myfunc1(firstName, lastName, nationality) :
    return "Hi %s %s, your nationality is %s" % (firstName, lastName, nationality)
```

Defining a Parameterized Decorator

- Here's how to define a parameterized decorator

```
def parameterizedDecorator(prefix, suffix):

    # Define inner function, which just wraps a function.
    def innerFunc1(func):

        # Define inner-inner function, which decorates/calls target function.
        def innerFunc2(*args, **kwargs):
            print(prefix)
            returnValueFromFunc = func(*args, **kwargs)
            print(suffix)
            return returnValueFromFunc

        # Return innerFunc2, i.e. the inner-inner function.
        return innerFunc2

    # Return innerFunc1, i.e. the inner function.
    return innerFunc1
```

1
2
3

- We have a layering of functions, to handle all the args:
  1. Arguments to the decorator itself
  2. The target function to be invoked
  3. Arguments to pass in to the target function

Applying a Parameterized Decorator Manually

- In order to understand how parameterized decorators work, let's first see how to apply the decorator manually:

```
# Some function, which we don't decorate explicitly here.
def myfunc1(firstName, lastName, nationality) :
    return "Hi %s %s, your nationality is %s" % (firstName, lastName, nationality)

# Client code
pointerToInnerFunc1 = parameterizedDecorator("HELLO", "GOODBYE")  # Line 1
pointerToInnerFunc2 = pointerToInnerFunc1(myfunc1)                # Line 2
res = pointerToInnerFunc2("Per", "Nordmann", "Norsk")             # Line 3
```

decorators5.py

- Line 1 calls the decorator manually, passing args into it
  - This statement returns a pointer to `innerFunc1`
- Line 2 calls `innerFunc1`, passing target function into it
  - This just return a pointer to `innerFunc2`
- Line 3 calls `innerFunc2`, passing args for the target func
  - This invokes the target func, with the desired decoration

Applying a Parameterized Decorator Properly

- The previous slide showed how to call a parameterized decorator function manually, to wrap a target function

- Now let's see how to apply a parameterized decorator function properly, i.e. using the `@decorator` syntax

```
# Some function, which we now decorate explicitly.
@parameterizedDecorator("HELLO", "GOODBYE")
def myfunc1(firstName, lastName, nationality) :
    return "Hi %s %s, your nationality is %s" % (firstName, lastName, nationality)

# Client code.
res1 = myfunc1("Kari", "Nordmann", "Norsk")
```

decorators6.py

- Note the client code just calls `myfunc1()` directly
  - Python intervenes, thanks to the `@parameterizedDecorator`, and cascades through the necessary sequence of function calls and argument-passing

Any Questions?