Decorators

1. Getting started with decorators
2. Additional decorator techniques
3. Parameterized 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 to decorate functions and classes using the @decorator syntax

The decorator enhances the function/class with extra capabilities

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

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

This section 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 (1/2)

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

```
1   def simpleDecorator(func) :
2
3       # Define an inner function, which wraps (decorates
4       def innerFunc() :
5           print("Start of simpleDecorator()")
6           func()
7           print("End of simpleDecorator()")
8
9       # Return the inner function.
10      return innerFunc
```

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:

```
1   # Some function that we want to decorate.
2   def myfunc1() :
3       print("Hi from myfunc1()")
4
5   # Client code.
6   pointerToInnerFunc = simpleDecorator(myfunc1)
7   pointerToInnerFunc()
```

Line 6 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 7 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()
```

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

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(f"Hi {firstName} {lastName}, your nationality is {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")
```

Decorating a function that returns a result

Consider the following function, which returns a result

```
@returnAwareDecorator
def myfunc1(firstName, lastName, nationality) :
    return f"Hi {firstName} {lastName}, your nationality is {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:

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

```
1   @parameterizedDecorator("HELLO", "GOODBYE")
2   def myfunc1(firstName, lastName, nationality) :
3       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 functi
    def innerFunc1(func) :

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

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

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

We have a layering of functions, to handle all the args:

- Arguments to the decorator itself
- The target function to be invoked
- 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:

```
1   # Some function, which we don't decorate explicitly he
2   def myfunc1(firstName, lastName, nationality) :
3       return "Hi %s %s, your nationality is %s" % (first
4
5   # Client code
6   pointerToInnerFunc1 = parameterizedDecorator("HELLO",
7   pointerToInnerFunc2 = pointerToInnerFunc1(myfunc1)
8   res = pointerToInnerFunc2("Per", "Nordmann", "Norsk")
```

Line 6 calls the decorator manually, passing args into it

- This statement returns a pointer to innerFunc1

Line 7 calls innerFunc1, passing target function into it

- This just return a pointer to innerFunc2

Line 8 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")
```

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?