

Decorator

★ first-class functions allow us to treat functions like any other objects. ★

- Reviewing closures:

→ Closure notes

```
def outer_function(msg):  
    message = msg
```

```
    def inner_function():  
        print(message)
```

```
    return inner_function
```

```
hi_func = outer_function('Hi')
```

```
bye_func = outer_function('Bye')
```

```
hi_func()  
bye_func()
```

Returned: **Hi**
Bye

'closure' when

Closures allow us to

each variable
is equal to our
inner function

→ Pass the
messages here

Decorator

★ A Decorator is a function that takes another function as an argument, add some kind of functionality and returns another function. All of these without altering the source code of the original function that you passed in.

Example

```
def decorator_function(original_function):
```

```
    def wrapper_function():
```

```
        return original_function()  
    return wrapper_function
```

```
def display():
```

```
    print('Display Function ran')
```

accept function as an
argument

→ which is
waiting to be executed
decorator function that we passed in, that's
what a decorator does. ★

```
decorated_display = decorator_function(display)
```

```
decorated_display()
```

Returned --> **Display Function ran**

→ passing in display function !

now this is
equal to
our wrapper-function

returns wrapper function, waiting to be executed. And

executes wrapper-function

when then executes our
display function

★ Why? adds functionality ★

```
def decorator_function(original_function):  
    def wrapper_function():  
        print('Wrapper executed this before {}'.format(original_function.__name__))  
        return original_function()  
    return wrapper_function
```

↖ Add to wrapper-function

```
@decorator_function  
def display():  
    print('Display Function ran')
```

↖ Placeholder

display()

Return --> Wrapper executed this before display
Display Function ran

Anytime that you
run this now, it will have new
functionality added on!

@decorator_function and display = decorator_function(display) are exactly the same thing

↳ decorator

Example:

```
def decorator_function(original_function):  
    def wrapper_function():  
        print('Wrapper executed this before {}'.format(original_function.__name__))  
        return original_function()  
    return wrapper_function
```

```
@decorator_function
```

```
def display():  
    print('Display Function ran')
```

```
def display_info(name, age):  
    print('display_info ran with arguments ({}, {})'.format(name, age))
```

```
display_info('John', 25)
```

```
#display()
```

Result --> display_info ran with arguments (John, 25)

↳ just returning display_info

Decorate both ?

Since we're passing arguments in now?

```
def decorator_function(original_function):  
    def wrapper_function(*args, **kwargs):  
        print('Wrapper executed this before {}'.format(original_function.__name__))  
        return original_function(*args, **kwargs)  
    return wrapper_function
```

@decorator_function

```
def display():  
    print('Display Function ran')
```

Always use to accept any number
keyword arguments for our function

@decorator_function

```
def display_info(name, age):  
    print('display_info ran with arguments ({}, {})'.format(name, age))
```

```
display_info('John', 25)
```

```
display()
```

Works with both

Results -->
Wrapper executed this before display_info
display_info ran with arguments (John, 25)
Wrapper executed this before display
Display Function ran

Using the Class as the Decorator instead of the function:

```
class decorator_class(object):  
    def __init__(self, original_function):  
        self.original_function = original_function  
  
    def __call__(self,*args, **kwargs ):  
        print('call method executed this before {}'.format(self.original_function.__name__))  
        return self.original_function(*args, **kwargs)
```

@decorator_class

```
def display():  
    print('Display Function ran')
```

@decorator_class

```
def display_info(name, age):  
    print('display_info ran with arguments ({}, {})'.format(name, age))
```

```
display_info('John', 25)
```

```
display()
```

Returned --> call method executed this before display_info
display_info ran with arguments (John, 25)
call method executed this before display
Display Function ran

Practical Examples for Decorators

Setting up log file that
✓ matches our original
function.

```
def my_logger(orig_func):  
    import logging  
    logging.basicConfig(filename = '{}.log'.format(orig_func.__name__), level = logging.INFO)  
  
    def wrapper(*args, **kwargs):  
        logging.info('Ran with args: {}, and kwargs: {}'.format(args, kwargs))  
        return orig_func(*args, **kwargs)  
    return wrapper
```

runs
this

run original-function, then return that result
return wrapper function, allow us to run with added functionality.

Decorator → @my_logger
def display_info(name, age):

```
    print('display_info ran with arguments {}, {}'.format(name, age))  
  
display_info('John', 25)
```

Returned --> display_info ran with arguments (John, 25)

Created this Log -->



★ Logging, Keeping track # times
a function has been ran. and
what arguments were passed
★

If you open up the display_info.log

INFO:root:Ran with args: ('John', 25), and kwargs: {}

Then if:

```
display_info('Devin', 69)
```

```
INFO:root:Ran with args: ('John', 25), and kwargs: {}  
INFO:root:Ran with args: ('Devin', 69), and kwargs: {}
```

passed in new arguments

Note: We can reuse @my_logger decorator on any new function.

Time it look function to run

Example

```
def my_timer(orig_func):  
    import time
```

```
    def wrapper(*args, **kwargs):
```

```
        t1 = time.time()
```

```
        result = orig_func(*args, **kwargs)
```

```
        t2 = time.time() - t1
```

```
        print('{} ran in: {} sec'.format(orig_func.__name__, t2))
```

```
        return result
```

```
    return wrapper
```

```
import time
```

```
@my_timer
```

```
def display_info(name, age):
```

```
    time.sleep(1)
```

```
    print('display_info ran with arguments ({}, {})'.format(name, age))
```

```
display_info('Devin', 69)
```

Results -- > display_info ran with arguments (Devin, 69)

display_info ran in: 1.0038840770721436 sec

beginning
time
→
how long it
took to run
that function
→

yet
↓
not returning results b/c
we want the end time.

to let one sec to run *

[22:00 min
in video]

Two decorators together

```
def my_logger(orig_func):
    import logging
    logging.basicConfig(filename = '{}.log'.format(orig_func.__name__), level = logging.INFO)

    def wrapper(*args, **kwargs):
        logging.info( 'Ran with args: {}, and kwargs: {}'.format(args, kwargs))

        return orig_func(*args, **kwargs)

    return wrapper

def my_timer(orig_func):
    import time

    def wrapper(*args, **kwargs):
        t1 = time.time()
        result = orig_func(*args, **kwargs)
        t2 = time.time() - t1
        print('{} ran in: {} sec'.format(orig_func.__name__, t2))
        return result

    return wrapper
```

import time

@my_logger
@my_timer

} stacking these decorators

```
def display_info(name, age):
    time.sleep(1)
    print('display_info ran with arguments {}, {}'.format(name, age))
```

display_info('Devin', 69)

Results --> display_info ran with arguments (Devin, 69)
display_info ran in: 1.0028259754180908 sec



△ Created wrapper.log

```
INFO:root:Ran with args: ('Devin', 69), and kwargs: {}
```

Something is

display_info = my_logger(my_timer(display_info))

* 25:26 + Video *

now Equal to wrapper function that
was returned by the timer

```
def my_logger(orig_func):  
    import logging  
    logging.basicConfig(filename = '{}.log'.format(orig_func.__name__), level= logging.INFO)  
  
    def wrapper(*args, **kwargs):  
        logging.info( 'Ran with args: {}, and kwargs: {}'.format(args, kwargs))  
  
        return orig_func(*args, **kwargs)  
  
    return wrapper
```

```
def my_timer(orig_func):  
    import time  
  
    def wrapper(*args, **kwargs):  
        t1 = time.time()  
        result = orig_func(*args, **kwargs)  
        t2 = time.time() - t1  
        print('{} ran in: {} sec'.format(orig_func.__name__, t2))  
        return result  
  
    return wrapper  
  
import time  
  
# @my_logger  
# @my_timer  
def display_info(name, age):  
    time.sleep(1)  
    print('display_info ran with arguments ({}, {})'.format(name, age))  
  
display_info = my_timer(display_info)  
  
print(display_info.__name__)  
Return -- > wrapper
```

```
display_info = my_logger(my_timer(display_info))
```

All of this is equal to that wrapper function

* that's why it created wrapper.py instead of display_info.py like
we expected *


```
from functools import wraps
```

```
def my_logger(orig_func):  
    import logging  
    logging.basicConfig(filename='{}.log'.format(orig_func.__name__), level=logging.INFO)
```

```
    @wraps(orig_func)  
    def wrapper(*args, **kwargs):  
        logging.info(  
            'Ran with args: {}, and kwargs: {}'.format(args, kwargs))  
        return orig_func(*args, **kwargs)
```

```
    return wrapper
```

```
def my_timer(orig_func):  
    import time
```

```
    @wraps(orig_func)  
    def wrapper(*args, **kwargs):  
        t1 = time.time()  
        result = orig_func(*args, **kwargs)  
        t2 = time.time() - t1  
        print('{} ran in: {} sec'.format(orig_func.__name__, t2))  
        return result
```

```
    return wrapper
```

```
import time
```

```
# @my_logger
```

```
# @my_timer
```

```
def display_info(name, age):  
    time.sleep(1)  
    print('display_info ran with arguments ({}, {})'.format(name, age))
```

```
display_info = my_timer(display_info)
```

```
print(display_info.__name__)
```

```
Returned --> display_info
```

```
from functools import wraps
```

```
def my_logger(orig_func):  
    import logging  
    logging.basicConfig(filename='{}.log'.format(orig_func.__name__), level=logging.INFO)
```

```
    @wraps(orig_func)  
    def wrapper(*args, **kwargs):  
        logging.info(  
            'Ran with args: {}, and kwargs: {}'.format(args, kwargs))  
        return orig_func(*args, **kwargs)
```

```
    return wrapper
```

```
def my_timer(orig_func):  
    import time
```

```
    @wraps(orig_func)  
    def wrapper(*args, **kwargs):  
        t1 = time.time()  
        result = orig_func(*args, **kwargs)  
        t2 = time.time() - t1  
        print('{} ran in: {} sec'.format(orig_func.__name__, t2))  
        return result
```

```
    return wrapper
```

```
import time
```

```
@my_logger  
@my_timer
```

```
def display_info(name, age):  
    time.sleep(1)  
    print('display_info ran with arguments {}, {}'.format(name, age))
```

```
display_info('Tommy', 45)
```

Results --> display_info ran with arguments (Tommy, 45)
display_info ran in: 1.0032918453216553 sec



logged correct arguments

```
INFO:root:Ran with args: ('Tommy', 45), and kwargs: {}
```