# decorators

February 9, 2024

[31]:
```
# Functions can be used just like any other object in Python
```

[32]:
```
# Functions can be passed as argument:

def greeter():
    return "Hello, "

def greet(name: str, greeter):
    return greeter() + name

greet("Simon", greeter=greeter)
```

[32]:
```
'Hello, Simon'
```

[33]:
```
# Functions can be defined inside of other functions:

def say_hello(name):

    def shout(txt):
        return txt.upper()

    return f"Hello, {shout(name)}"

say_hello("Simon")
```

[33]:
```
'Hello, SIMON'
```

[34]:
```
# Functions can be returned:

def repeat_yourself():
    def phrase(n: int):
        return n * "Hello "
    return phrase

returned_function = repeat_yourself()
returned_function(3)
```

[34]:
```
'Hello Hello Hello '
```

[35]:
```
# Decorators are functions that accept a function,
# create a new function which does something before/ after the passed in␣
 ↪function is called
# and return the newly created function.

# Example:
def simple_decorator(func):
    def wrapper():
        print("Do something before calling the function.")
        func()
        print("Do something after calling the function.")
    return wrapper

# Let's define a function to be wrapped:
def hi():
    print("Hi!")

# And now let's decorate it:

hi_decorated = simple_decorator(hi)
hi_decorated()

# Python adds some syntactic sugar to sweaten the deal so instead we could have␣
 ↪done:
@simple_decorator
def bye():
    print("Bye")

bye()
```

```
Do something before calling the function.
Hi!
Do something after calling the function.
Do something before calling the function.
Bye
Do something after calling the function.
```

[36]:
```
# There are two important things to consider:
# - first: functions may take arguments, so our wrapper needs to be able to␣
 ↪handle it.
# - second: functions may return something, so our wrapper needs to be able to␣
 ↪handle that as weel.
```

[37]:
```
# To demonstrate the first issue:
def simple_decorator(func):
    def wrapper():
        print("Do something before calling the function.")
        func()
        print("Do something after calling the function.")
    return wrapper

@simple_decorator
def hello(name):
    print(f"Hello, {name}")

# Calling the decorated function will cause an error as the wrapper is passed␣
 ↪in an argument,
# but he doesn't accept any!
# Calling a decorated function basically means, under the hood, calling the␣
 ↪wrapper function!
hello("Simon")
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[37], line 16
     11     print(f"Hello, {name}")
     13 # Calling the decorated function will cause an error as the wrapper is␣
 ↪passed in an argument,
     14 # but he doesn't accept any!
     15 # Calling a decorated function basically means, under the hood, calling␣
 ↪the wrapper function!
---> 16 hello("Simon")

TypeError: simple_decorator.<locals>.wrapper() takes 0 positional arguments but␣
 ↪1 was given
```

[38]:
```
# Let's fix issue one:

def simple_decorator(func):
    def wrapper(*args, **kwargs):
        print("Do something before calling the function.")
        func(*args, **kwargs)
        print("Do something after calling the function.")
    return wrapper

@simple_decorator
def hello(name):
    print(f"Hello, {name}")

hello("Simon")
```

```
Do something before calling the function.
Hello, Simon
Do something after calling the function.
```

[39]:
```python
# With that in mind let's consider the second issue.

def simple_decorator(func):
    def wrapper(*args, **kwargs):
        print("Do something before calling the function.")
        func(*args, **kwargs)
        print("Do something after calling the function.")
    return wrapper

@simple_decorator
def add_one(n):
    return n + 1

y = add_one(5)
print(y)

# Ooops, our decorated function returns None!
# Remember, we are in essence calling the wrapper when calling a decorated␣
 ↪function.
# And our wrapper doesn't return anything!
```

```
Do something before calling the function.
Do something after calling the function.
None
```

[40]:
```python
# To fix that:
def simple_decorator(func):
    def wrapper(*args, **kwargs):
        print("Do something before calling the function.")
        value = func(*args, **kwargs)
        print("Do something after calling the function.")
        return value
    return wrapper

@simple_decorator
def add_one(n):
    return n + 1

y = add_one(5)
print(y)
```

```
Do something before calling the function.
Do something after calling the function.
6
```

[41]:
```python
# One final note: as we are basically calling the wrapper, our decorated␣
 ↪function
# looses its sense of who it is.

def simple_decorator(func):
    def wrapper(*args, **kwargs):
        print("Do something before calling the function.")
        value = func(*args, **kwargs)
        print("Do something after calling the function.")
        return value
    return wrapper

@simple_decorator
def add_one(n):
    return n + 1

print(add_one.__name__)
print(add_one)

# Our function thinks it's called wrapper!
```

```
wrapper
<function simple_decorator.<locals>.wrapper at 0x70e3a442d080>
```

[42]:
```python
# To fix the identity crisis we can decorate our wrapper. Yeah, I know.

import functools


def simple_decorator(func):
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        print("Do something before calling the function.")
        value = func(*args, **kwargs)
        print("Do something after calling the function.")
        return value
    return wrapper

@simple_decorator
def add_one(n):
    return n + 1

print(add_one.__name__)
print(add_one)

# Identity crisis averted!
```

```
add_one
<function add_one at 0x70e3a442fb00>
```

[43]:
```
# Here is a boilerplate template for constructing decorators
# which adds everything discussed together!

def boilerplate(func):
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        # do something before
        value = func(*args, **kwargs)
        # do something after
        return value

    return wrapper
```

[ ]:
```

```