

# The Decorator Pattern

Welcome back to CS 2100!

Prof. Rasika Bhalerao

# Function Decorators

We have already seen some decorators:

- `@property` (to create a property named after the method)
- `*.setter` (to give that property a "setter" method)
- `@classmethod` (for `setUpClass(cls)` and `tearDownClass(cls)` to run once before / after unit testing)

We will find out how these decorators were implemented / how to create our own decorators

# Functions are objects that can be mutated

We already knew functions are objects:

```
from typing import Callable

def double(num: int) -> int:
    return num * 2

def triple(num: int) -> int:
    return num * 3

functions: list[Callable[[int], int]] = [double, triple]

def apply_functions(num: int) -> list[int]:
    return [func(num) for func in functions]

print(apply_functions(5)) # [10, 15]
```

A function decorator is a way to mutate a function to modify what it does.

# Function decorators mutate functions

Example: `@time_calls` modifies a given function to print the time it took to run it:

```
@time_calls
def multiply_list(nums: list[int], multiplier: int = 1) -> list[int]:
    return [num * multiplier for num in nums]

print(multiply_list([i for i in range(5)], multiplier = 3))
```

Because we modified `multiply_list()` to do what `@time_calls` tells it to do, this is the printed output:

```
Executed multiply_list with ([0, 1, 2, 3, 4],) and {'multiplier': 3} in 7.152557373046875e-06ms
[0, 3, 6, 9, 12]
```

How we created that `@time_calls` decorator:

```
def time_calls(func: Callable[..., Any]) -> Callable[..., Any]:  
    def wrapper(*args: Any, **kwargs: Any) -> Any:  
        now = time.time()  
        return_value = func(*args, **kwargs)  
        print(f'Executed {func.__name__} with {args} and {kwargs} in {time.time() - now}ms')  
        return return_value  
    return wrapper
```

`time_calls()` takes a function as its argument and returns a function.

The returned function is a modified version of the argument function.

That modification: in addition to running the function as usual, it prints the number of milliseconds that it took to run it.

## Poll: What does the `@mystery` decorator do?

```
T = TypeVar('T')

def mystery(func: Callable[[float], T]) -> Callable[[float], T]:
    def wrapper(x: float) -> T:
        if x < 0:
            raise ValueError("Argument must be positive")
        return func(x)
    return wrapper

@mystery
def square_root(x: float) -> Any:
    return x ** 0.5

print(square_root(9))
```

1. Modifies the function to print the number of milliseconds it took to run it
2. Modifies the function such that it raises a `ValueError` if the argument is negative
3. Modifies the function to run it twice
4. Nothing

# Caching / memoization: a common use of decorators

Memoization / caching: temporarily storing values that are recalculated often (so they only need to be calculated once).

```
def fibonacci(n: int) -> int:  
    if n < 2:  
        return n  
    return fibonacci(n-1) + fibonacci(n-2)
```

In order to calculate the 5th Fibonacci number, we need to calculate the 4th and 3rd ones.  
And to calculate the 4th number, we need to calculate the 3rd and 2nd ones.  
And to calculate the 3rd number, we need to calculate the 2nd and 1st ones...

There are a lot of recalculated numbers here. Let's memoize.

Let's store each Fibonacci number *the first time it is calculated*, so that each subsequent time, we can just look it up

```
R = TypeVar('R')

def memoize(func: Callable[..., R]) -> Callable[..., R]:
    cache: dict[tuple[Any, ...], R] = {}
    def wrapper(*args: Any) -> R:
        if args in cache:
            return cache[args]
        result = func(*args)
        cache[args] = result
        return result
    return wrapper

@memoize
def fibonacci(n: int) -> int:
    if n < 2:
        return n
    return fibonacci(n-1) + fibonacci(n-2)

print(fibonacci(10))
```

## Poll: Can we re-use the `@memoize` decorator to perform memoization for other functions in the same file?

1. Yes
2. Yes, but that other function would share the "cache" of stored values, which are specific to Fibonacci
3. No, it would not run

Easiest way to test this: create a duplicate method called `fibonacci2()`, decorate it, call it after the regular `fibonacci()`, and step through it in the debugger.

# Preserving function metadata

Functions have metadata such as their name and docstring:

```
def double(num: int) -> int:  
    """Doubles a number."""  
    return num * 2  
  
print(double.__name__) # double  
print(double.__doc__) # Doubles a number.
```

Its metadata is overwritten by the wrapper's metadata:

```
def my_decorator(  
    func: Callable[..., Any]  
) -> Callable[..., Any]:  
  
    def wrapper(  
        *args: Any, **kwargs: Any  
    ) -> Any:  
        print("Before calling function")  
        result = func(*args, **kwargs)  
        print("After calling function")  
        return result  
    return wrapper  
  
@my_decorator  
def double(num: int) -> int:  
    """Doubles a number."""  
    return num * 2  
  
print(double.__name__) # wrapper  
print(double.__doc__) # None
```

## @wraps decorator copies metadata from passed function to wrapper function

```
from functools import wraps

def my_decorator(func: Callable[..., Any]) -> Callable[..., Any]:
    @wraps(func) # This copies func's metadata to wrapper
    def wrapper(*args: Any, **kwargs: Any) -> Any:
        print("Before calling function")
        result = func(*args, **kwargs)
        print("After calling function")
        return result
    return wrapper

@my_decorator
def double(num: int) -> int:
    """Doubles a number."""
    return num * 2

print(double.__name__) # double
print(double.__doc__) # Doubles a number.
```

## Poll: Why do we use `wraps` from the `functools` package?

1. To copy the metadata from the original (un-decorated) function to the wrapper function that decorates it
2. To copy the metadata from the decorating wrapper function to the original (un-decorated) function
3. To make it so we can use a function as a decorator with the `@` symbol
4. `wraps` is like a type hint: it doesn't do anything and isn't enforced in Python by default

## Class decorators

*Function* decorators are popular in Python.

*Class* decorators are more popular in other languages including Java.

A *class* decorator is also a decorator that uses the @ symbol. The only difference is that it decorates a class instead of a function.

`@add_logging` modifies a class's `__init__()` to add a `print()` statement for logging.

```
class HasInit(Protocol):
    # to ensure we only use `@add_logging` on classes that have `__init__()`
    def __init__(self, *args: Any, **kwargs: Any) -> None: ...

C = TypeVar('C', bound=HasInit)

def add_logging(cls: Type[C]) -> Type[C]:
    original_init = cls.__init__

    @wraps(original_init)
    def new_init(self: Any, *args: Any, **kwargs: Any) -> None:
        print(f"Creating instance of {cls.__name__}")
        original_init(self, *args, **kwargs)

    cls.__init__ = new_init # type: ignore[misc]
    return cls

@add_logging
class User:
    def __init__(self, name: str) -> None:
        self.name = name

user = User('Mini')
```

## **Poll:**

- 1. What is your main takeaway from today?**
- 2. What would you like to revisit next time?**