In [18]:
```
#Decorator to add cheese
def add_cheese(func):
    def wrapper(burger_type):
        result=func(burger_type)
        result+="with cheese"
        return result
    return wrapper
#Decorator to add spices
def add_spices(func):
     def wrapper(burger_type):
        result=func(burger_type)
        result+=" and spices"
        return result
     return wrapper


#base burger function  with decorator
@add_spices
@add_cheese
def create_burger(burger_type):
    return f"{burger_type}"
#call the function
burger=create_burger("Chicken Burger");
print("Ordered:",burger)

# def add cheese(func)--<func=create burger
#def wrapper(burger_type):-->intercept call to the create burger
#result=func(burger_type)-->call the original fn
#result+="with cheese"-->add chhese to the burger
#return the result
```

```
Ordered: Chicken Burgerwith cheese and spices
```

In [24]:
```
def my_decorator(func):
    def wrapper(*args,**kwargs):
        print("Function being called with:",args,kwargs)
        return func(*args,**kwargs)
    return wrapper
@my_decorator
def add(x,y):
    return x+y
print(add(5,3))
```

```
Function being called with: (5, 3) {}
8
```

In [30]:
```
def authenticate(func):
    def wrapper(user):
        if user.get("Logged_in"):
            return func(user)
        else:
            return "Access Denied"
    return wrapper
@authenticate
def show_dashboard(user):
```

```
    return f"Welcome{user['name']}"
user={"name":"Arman","Logged_in":False}
print(show_dashboard(user))
```

```
Access Denied
```

In [41]:
```
#converting method to read only property
#self is representing instance of class and  is the first parameter of every instan
#@Property --->converting method to read only property
class Circle:
    def __init__(self,radius):
        self._radius=radius
    @property
    def area(self):
        return 3.14*self._radius**2

c = Circle(5)
print(c.area)
```

```
78.5
```

In [47]:
```
class Logger:
     def __init__(self,func):#,it calls Logger.__init(self,func)
         self.func=func#greet(storing original greet function in logger -->Logger(g
     def __call__(self,*args,**kwargs):#Calling Logger(greet)--calling instance of
         print(f"Calling{self.func.__name__}")
         return self.func(*args,**kwargs)
@Logger
def greet(name):
    print(f"Hello{name}")
greet("Ananya") ##not calling function greet directly ,calling instance of class
   #logger is decorator
#logger is a class not function
#so when greet=Logger(greet),it calls Logger.__init(self,func)
#self.func=func-->greet(storing original greet function in logger )
#Calling Logger(greet)
#not calling function greet directly ,calling instance of class
```

```
Callinggreet
HelloAnanya
```

In [49]:
```
def greet(name):
    print(f"Hello{name}")
greet("Ananya")
#calling function greet
```

```
HelloAnanya
```

In [53]:
```
import time

class Timer:
    def __init__(self, func):
        self.func = func

    def __call__(self, *args, **kwargs):
        start = time.time()
        result = self.func(*args, **kwargs)
        end = time.time()
```

```
        print(f"{self.func.__name__} took {end - start:.4f} seconds")
        return result

@Timer
def slow_function():
    time.sleep(5)
    print("Finished slow task")

slow_function()
```

```
Finished slow task
slow_function took 5.0011 seconds
```

In [ ]: