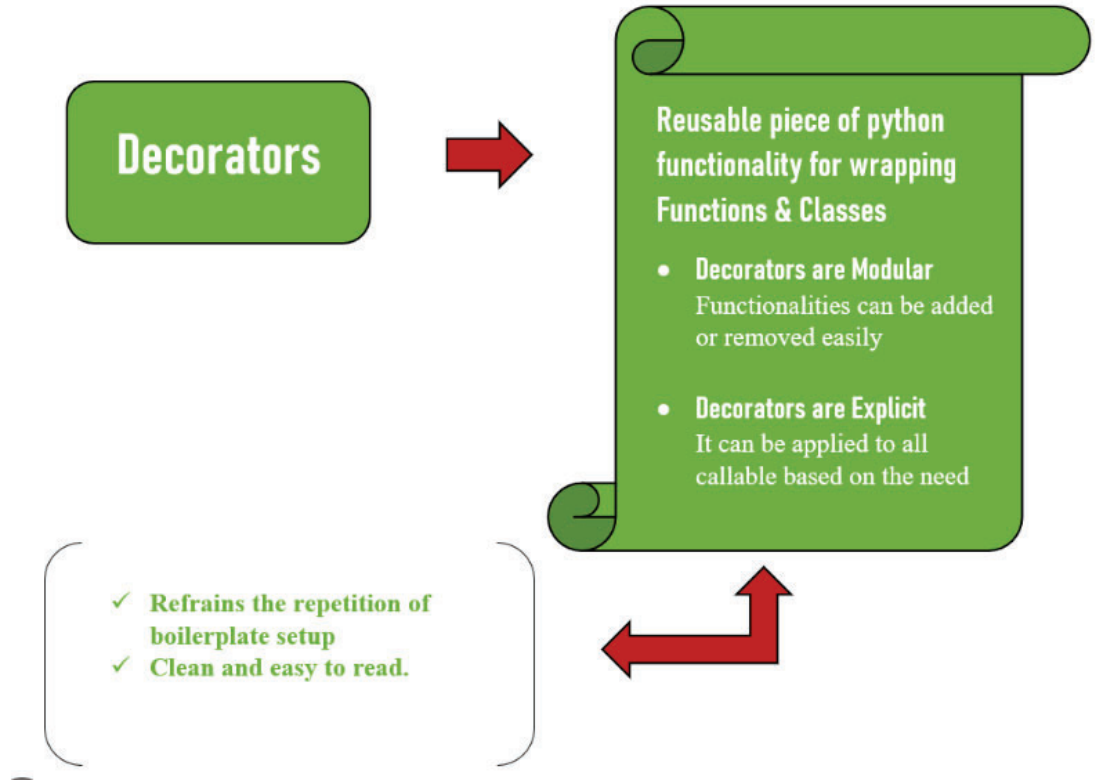


Let's go over it quickly.

Decorators provide a super flexible way to modify or enhance the behavior of functions or methods without altering their code. At its core, a decorator is just a higher-order function, a function that takes one or more functions as arguments and returns a new function.

The fundamental theory behind decorators is the ability of Python functions to be first-class citizens, which means they can be passed around and used as arguments.

```
def decorator_function(func):
    def wrapper():
        print("Something is happening before the function is called.")
        func()
        print("Something is happening after the function is called.")
    return wrapper

@decorator_function
def say_hello():
    print("Hello!")

say_hello()
```

When `say_hello()` is called, the output will be:

```
Something is happening before the function is called.
Hello!
Something is happening after the function is called.
```

**👉 The `@decorator_function` syntax is a more readable way of expressing `say_hello = decorator_function(say_hello)`. It allows us to "wrap" the behavior of the `say_hello` function with additional code from `decorator_function`, without modifying `say_hello` directly.**

---

## 👉 Common Decorators in Real-life Python Projects

1. **Logging**: It's quite common to have a decorator that logs the metadata or actual data about the function execution. This helps in debugging and monitoring.

```
import logging
logging.basicConfig(level=logging.INFO)

def log_decorator(func):
    def wrapper(*args, **kwargs):
        logging.info(f"Running '{func.__name__}' with arguments {args} and keyword
arguments {kwargs}")
        return func(*args, **kwargs)
    return wrapper

@log_decorator
def add(x, y):
    return x + y

add(5, 3)
# INFO:root:Running 'add' with arguments (5, 3) and keyword arguments {}
```

👉 **Value Add**: Automatically logs every time the `add` function is called, without having to add logging logic inside the `add` function itself.

---

## 2. Timing:

Timing decorators measure the time a function takes to execute, which is helpful for performance monitoring.

```
import time

def timing_decorator(func):
    def wrapper(*args, **kwargs):
        start_time = time.time()
        result = func(*args, **kwargs)
        end_time = time.time()
        print(f"'{func.__name__}' took {end_time - start_time} seconds to execute")
        return result
    return wrapper

@timing_decorator
def long_running_task():
    time.sleep(2)
```

```
long_running_task()
#'long_running_task' took 2.003575325012207 seconds to execute
```

👉 **Value Add**: Without altering the `long_running_task` function, we can measure its execution time.

---

## 3. Authentication/Authorization:

In web frameworks like Flask or Django, decorators can be used to restrict access to certain views based on user roles or authentication states.

```
# Assuming a hypothetical web framework
current_user = {"is_authenticated": True, "role": "admin"}

def admin_required(func):
    def wrapper(*args, **kwargs):
        if not current_user["is_authenticated"] or current_user["role"] != "admin":
            raise PermissionError("Admin privileges are required.")
        return func(*args, **kwargs)
    return wrapper

@admin_required
def view_admin_dashboard():
    print("Admin dashboard viewed!")

view_admin_dashboard()
```

👉 **Value Add**: The decorator ensures that the `view_admin_dashboard` function can only be accessed by authenticated users with an admin role.

---

## 4. 👉 Retry Mechanism:

In scenarios where a function might fail due to temporary issues (e.g., a network problem), a decorator can be used to retry the function a specified number of times before finally failing.

```
import time

def retry_decorator(max_retries=3, delay=1):
    def decorator(func):
        def wrapper(*args, **kwargs):
            for _ in range(max_retries):
                try:
                    return func(*args, **kwargs)
                except Exception as e:
                    print(f"Failure. Retrying in {delay} seconds...")
                    time.sleep(delay)
            raise e
        return wrapper
    return decorator

@retry_decorator(max_retries=5, delay=2)
def unreliable_function():
    if time.time() % 2:
        raise Exception("Failed due to some reason!")
    print("Function executed successfully!")
```

```
unreliable_function()
```

👉 **Value Add**: The decorator provides a mechanism to handle temporary failures, making the system more robust and resilient.

---

## 5. 👉 Caching/Memoization:

For functions that are computationally expensive, a decorator can be used to cache results, ensuring that subsequent calls with the same arguments return quickly.

```
def memoize(func):
    cache = {}

    def wrapper(*args):
        if args in cache:
            return cache[args]
        else:
            result = func(*args)
            cache[args] = result
            return result
    return wrapper

@memoize
def expensive_computation(x, y):
    time.sleep(5)
    return x + y

expensive_computation(1, 2)  # Takes 5 seconds
expensive_computation(1, 2)  # Returns instantly due to caching
```

👉 **Value Add**: The decorator reduces the number of expensive computations by caching results, leading to faster execution for repeated calls with the same arguments.

---

Examples of decorators being used for rate limiting - A simple but very useful example

In this example, we'll limit a function to be called no more than a certain number of times per second.

Let's break down the code:

1. The `rate_limited` function is a decorator factory. It takes as input the maximum number of times a function can be called per second (`max_per_second`). It calculates the minimum interval between calls (`min_interval`) from this input.
2. The `decorate` function is the actual decorator. It decorates/wraps the function with some rate limiting behavior.
3. Inside `decorate`, we define `rate_limited_function`. This function is what will be called instead of the original function. It measures the time elapsed since the last call, and if not enough time has passed, it waits the remaining time. It then calls the function and stores the current time.
4. The `wraps(func)` decorator is used to make the `rate_limited_function` look like `func` from the outside. It takes the name, module, and docstring from `func`.
5. When we use `@rate_limited(2)`, we're telling Python to limit this function to be called no more than twice per second. This is useful when interacting with external services that may have usage limits, or to prevent your own services from being overloaded.

Remember that this is a simple example and does not handle multiple threads or processes calling the function concurrently. A production-grade rate-limiter might use a more advanced algorithm, keep its state in a shared, thread-safe data structure, or even use a distributed system when multiple processes or machines need to be rate-limited together.

```
import time
from functools import wraps

def rate_limited(max_per_second):
    min_interval = 1.0 / float(max_per_second)

    def decorate(func):
        last_time_called = [0.0]

        @wraps(func)
        def rate_limited_function(*args, **kwargs):
            elapsed = time.perf_counter() - last_time_called[0]
            left_to_wait = min_interval - elapsed

            if left_to_wait > 0:
                time.sleep(left_to_wait)

            ret = func(*args, **kwargs)
            last_time_called[0] = time.perf_counter()
            return ret

        return rate_limited_function
    return decorate

@rate_limited(2)  # 2 per second at most
def print_hello():
    print(f"Hello, World! Time: {time.perf_counter()}")

# Test the rate limit
for _ in range(10):
    print_hello()
```

---

## Example of a more exhaustive rate limiter decorator that uses Python's `threading` library to ensure thread-safety.

This version uses a `threading.Lock` to prevent race conditions. We'll use the token bucket algorithm for rate limiting, which allows for bursty calls up to a certain limit, and refills the bucket with tokens at a constant rate.

1. We define a `RateLimiter` class that will act as our decorator. It holds a number of "tokens" (representing the number of times the function can be called), the maximum rate at which the tokens refill, and the last time a function call was attempted.
2. The lock (`self.lock = threading.Lock()`) is used to ensure that only one thread at a time can execute the code within the `with self.lock:` block, which ensures thread-safety.
3. In the `wrapper` function, we calculate the number of tokens that have been refilled since the last function call (`self.tokens += elapsed * self.max_rate`). We limit the number of tokens to `max_rate`, simulating a "bucket" that fills up to a certain point.

4. If there are not enough tokens to call the function (i.e., `self.tokens < 1`), we wait until enough tokens have been refilled.
5. Once we've ensured that there are enough tokens, we "spend" a token (`self.tokens -= 1`) and call the function.

This code is thread-safe, but does not handle multiple processes. A more complex version might use multiprocessing primitives, or an external service like Redis, to handle rate limiting across multiple processes and machines.

However, implementing such an advanced and reliable rate-limiter from scratch can be quite challenging and error-prone. There are many excellent libraries available, like ratelimit, redislimiter, Flask-Limiter (for Flask applications), etc. which can do this job for you in a robust and efficient manner.

---

When you run this code, you should see "Hello, World!" printed roughly two times per second, even though there are multiple threads trying to call print_hello() at once. The rate limiter ensures that, on average, no more than two calls are made per second.

Remember that the timing may not be perfectly even due to the nature of thread scheduling and other system operations. Also, sleeping a thread does not guarantee that the thread will resume exactly after the sleep time has passed. The actual delay may be longer due to other running threads and system factors.

```
import time
import threading
from functools import wraps

class RateLimiter:
    def __init__(self, max_rate):
        self.tokens = max_rate
        self.max_rate = max_rate
        self.last = time.perf_counter()
        self.lock = threading.Lock()

    def __call__(self, func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            with self.lock:
                now = time.perf_counter()
                elapsed = now - self.last
                self.last = now
                self.tokens += elapsed * self.max_rate
                if self.tokens > self.max_rate:
                    self.tokens = self.max_rate
                if self.tokens < 1:
                    left_to_wait = (1 - self.tokens) / self.max_rate
                    time.sleep(left_to_wait)
                self.tokens -= 1
            return func(*args, **kwargs)
        return wrapper

@RateLimiter(2)  # 2 per second at most
def print_hello():
    print(f"Hello, World! Time: {time.perf_counter()}")

# Now, let's test this
threads = []
```

```
for _ in range(10):
    thread = threading.Thread(target=print_hello)
    threads.append(thread)
    thread.start()

for thread in threads:
    thread.join()
```

In conclusion, decorators leverage Python's first-class function capabilities to add layers of logic before and/or after function execution. They provide an elegant mechanism to augment functions without modifying their core logic, making them a valuable tool in the Python programmer's toolkit.

---

## Now, Let's see an example WITH and then WITHOUT the "Decorators in Python"

### Without Using Decorators

📌 Imagine you're developing a web application and you want to measure the execution time of certain functions to ensure performance. Here's an example of how you might approach this without using decorators:

```
import time

def expensive_function():
    time.sleep(2)
    print("Function executed!")

def another_expensive_function():
    time.sleep(3)
    print("Another function executed!")

# Monitoring the execution time of expensive_function
start_time = time.time()
expensive_function()
end_time = time.time()
print(f"expensive_function took {end_time - start_time} seconds to execute.")

# Monitoring the execution time of another_expensive_function
start_time = time.time()
another_expensive_function()
end_time = time.time()
print(f"another_expensive_function took {end_time - start_time} seconds to execute.")
```

📌 Issues with the above code:

- There's a lot of repetition in how we measure the time.
- Every time we want to measure a new function, we need to add more timing code.
- It's easy to make mistakes, especially if there are many functions to monitor.

## Using Decorators to Refactor

📌 Now, let's introduce a decorator to handle the timing aspect, which will result in cleaner and more maintainable code.

```
import time

def time_it(func):
    def wrapper(*args, **kwargs):
        start_time = time.time()
        result = func(*args, **kwargs)
        end_time = time.time()
        print(f"{func.__name__} took {end_time - start_time} seconds to execute.")
        return result
    return wrapper

@time_it
def expensive_function():
    time.sleep(2)
    print("Function executed!")

@time_it
def another_expensive_function():
    time.sleep(3)
    print("Another function executed!")

expensive_function()
another_expensive_function()
```

📌 Advantages with the refactored code:

- The code is cleaner and more organized.
- We've removed the repetitive timing code and placed it in a single location, the `time_it` decorator.
- It's now very easy to measure the execution time of any new function; just prepend it with `@time_it`.

📌 By employing decorators, you've essentially abstracted away the repetitive task and made the codebase more maintainable and cleaner. This is one of the numerous uses of decorators in Python, and as you delve deeper into Python development, especially web frameworks like Flask or Django, you'll find decorators being employed for various purposes such as route handling, user authentication, and more.

---

## Now, Let's see another example WITH and then WITHOUT the "Decorators in Python"

Let's look at an example where decorators can be instrumental in simplifying code related to access control for an API endpoint.

## Without Using Decorators

📌 Imagine you have an API endpoint, and you want to ensure that the requester has the correct authorization token before they can access certain endpoints.

```
def authenticate(token):
    # Mock authentication mechanism
    return token == "VALID_TOKEN"

def get_user_data(token, user_id):
    if not authenticate(token):
        return "Unauthorized Access", 403

    # Suppose this function fetches user data from a database
    data = {"id": user_id, "name": "John Doe", "email": "john.doe@example.com"}
    return data, 200

def update_user_data(token, user_id, new_data):
    if not authenticate(token):
        return "Unauthorized Access", 403

    # Here, we'd update user data in a database
    # For simplicity, let's assume it's successful
    return "User data updated successfully", 200
```

📌 Issues with the above code:

- Repetition: The authentication check is repeated in each function where it's required.
- Scalability: For every new endpoint, you'd have to remember to add the authentication check, increasing the chances of errors.
- Maintenance: If the authentication method changes, you'd have to update it in multiple places.

## Using Decorators to Refactor

📌 We can use a decorator to handle the authentication, ensuring a DRY (Don't Repeat Yourself) approach.

```
def requires_authentication(func):
    def wrapper(token, *args, **kwargs):
        if not authenticate(token):
            return "Unauthorized Access", 403
        return func(token, *args, **kwargs)
    return wrapper

def authenticate(token):
    # Mock authentication mechanism
    return token == "VALID_TOKEN"

@requires_authentication
def get_user_data(token, user_id):
    # Suppose this function fetches user data from a database
    data = {"id": user_id, "name": "John Doe", "email": "john.doe@example.com"}
    return data, 200

@requires_authentication
def update_user_data(token, user_id, new_data):
```

```
    # Here, we'd update user data in a database
    # For simplicity, let's assume it's successful
    return "User data updated successfully", 200
```

📌 Advantages with the refactored code:

- Centralized Authentication: The authentication logic is abstracted and placed in a single location.
- Cleaner Function Signatures: Each function's primary task remains clear, without being cluttered with authentication logic.
- Enhanced Maintainability: If the authentication method needs to be changed, it can be done in one place, benefiting maintainability and reducing potential bugs.

📌 This pattern, with slight modifications, is prevalent in web frameworks like Flask and Django, where decorators handle route definitions, authentication, caching, and more, keeping the core application logic concise and clean.

---

All the above were examples of general user-defined decorators.

But Python has many built-in decorators (like `@property`, `@staticmethod`, `@classmethod`)👇👇

---

## A) `@property`

The `@property` decorator allows a method in a class to be accessed like an attribute rather than as a method. It's a way of defining getters for instance attributes in an object-oriented way.

Example:

```
class Circle:
    def __init__(self, radius):
        self._radius = radius

    @property
    def radius(self):
        return self._radius

    @property
    def diameter(self):
        return self._radius * 2

c = Circle(5)
print(c.radius)    # 5
print(c.diameter)  # 10
```

👉 Another Example - `@property` and `@<property>.setter`

The `@property` decorator, along with its sibling `@<property>.setter`, is used in scenarios where you need to perform some extra processing or validation when getting or setting an attribute.

```
class Employee:
    def __init__(self, first_name, last_name):
        self._first_name = first_name
        self._last_name = last_name

    @property
```

```
    def email(self):
        return f"{self._first_name}.{self._last_name}@company.com"

    @property
    def full_name(self):
        return f"{self._first_name} {self._last_name}"

    @full_name.setter
    def full_name(self, name):
        self._first_name, self._last_name = name.split(' ')

emp = Employee('John', 'Doe')
print(emp.email)       # John.Doe@company.com
print(emp.full_name)   # John Doe
emp.full_name = 'Jane Doe'
print(emp.email)       # Jane.Doe@company.com
```

In this example, the `full_name` property is defined to return a formatted string. The `email` property is defined based on the first and last names. When we set `full_name`, it automatically updates the first and last names, as well as the email.

---

## Let's see an example WITH and then WITHOUT the "@property` decorator"

📌 Consider the following code for a `Circle` class which does not utilize the `@property` decorator:

```
class Circle:
    def __init__(self, radius):
        self.radius = radius
        self.diameter = 2 * radius

circle = Circle(5)
print(circle.diameter)  # Expected output: 10

# Later in the code, if someone updates the radius
circle.radius = 10
print(circle.diameter)  # Expected output: 10, but 20 would be the accurate diameter
for radius 10
```

📌 The problem in the above code is that the `diameter` attribute does not get updated when the `radius` is modified. This can lead to incorrect calculations or erroneous data.

📌 Now, let's refactor the `Circle` class by using the `@property` decorator and address the issue:

```
class Circle:
    def __init__(self, radius):
        self._radius = radius

    @property
    def radius(self):
        return self._radius

    @radius.setter
    def radius(self, value):
        if value < 0:
```

```
            raise ValueError("Radius cannot be negative.")
        self._radius = value

    @property
    def diameter(self):
        return 2 * self._radius

circle = Circle(5)
print(circle.diameter)  # Output: 10

# If we update the radius, the diameter property will also reflect the change.
circle.radius = 10
print(circle.diameter)  # Output: 20
```

📌 Here are the improvements in the refactored code:

1. The `_radius` attribute is made private (by convention) to prevent direct modifications.
2. The `@property` decorator allows us to create a "getter" method for the radius without needing to call it as a method. This makes accessing object properties more intuitive.
3. The `@radius.setter` decorator creates a setter method for the radius. This allows us to perform validation checks or any other operations when setting the value of the radius.
4. The diameter is now a property, computed based on the current radius. This ensures it is always consistent with the radius, avoiding the inconsistency issue we had earlier.

📌 Using the `@property` decorator, we have added a layer of abstraction to manage how attributes are accessed and modified, ensuring consistency and allowing for additional validation or computation when necessary.

---

## Let's see another example WITH and then WITHOUT the "@property` decorator"

Let's consider a scenario that might be more complex and relevant to large codebases.

📌 **Scenario**: Imagine you're working on an e-commerce platform's backend. Products in the platform have a `base_price` and potential `discount`. The `final_price` of a product should be computed based on these two factors. Additionally, it's crucial that the final price doesn't drop below a certain minimum threshold, no matter how big the discount is.

---

📌 **Without `@property` decorator**:

```
class Product:
    def __init__(self, name, base_price, discount=0):
        self.name = name
        self.base_price = base_price
        self.discount = discount
        self.final_price = self.base_price - self.discount

item = Product("Laptop", 1000, 50)
print(item.final_price)  # 950

# Now, if the discount changes, the final price doesn't reflect that change
item.discount = 100
print(item.final_price)  # Still 950, but we'd expect 900
```

📌 The issue is similar to the previous example: changes in related attributes (`base_price` or `discount`) don't automatically update the dependent attribute (`final_price`).

---

📌 **Refactoring with `@property` decorator**:

```
class Product:
    MIN_PRICE = 100  # Minimum threshold price

    def __init__(self, name, base_price, discount=0):
        self.name = name
        self._base_price = base_price
        self._discount = discount

    @property
    def base_price(self):
        return self._base_price

    @base_price.setter
    def base_price(self, value):
        if value < 0:
            raise ValueError("Base price cannot be negative.")
        self._base_price = value

    @property
    def discount(self):
        return self._discount

    @discount.setter
    def discount(self, value):
        if not (0 <= value <= self._base_price):
            raise ValueError(f"Discount cannot be negative or greater than base
price.")
        self._discount = value

    @property
    def final_price(self):
        calculated_final = self._base_price - self._discount
        return max(self.MIN_PRICE, calculated_final)  # Ensure it doesn't go below
minimum threshold

item = Product("Laptop", 1000, 50)
print(item.final_price)  # 950

# Change the discount, and the final price automatically updates
item.discount = 100
print(item.final_price)  # 900

# Set a huge discount, but the final price will not drop below the threshold
item.discount = 950
print(item.final_price)  # 100, the minimum threshold
```

📌 **Explanation**:

1. The `base_price` and `discount` attributes have been made private to prevent direct unauthorized modifications.