Python in Plain English

Member-only story

7 Useful Python Decorators Every Developer Should Know

Subhajit Bhar Follow 4 min read · Jul 2, 2025

By Subhajit, visualised with GPT-4o.

If you've ever asked yourself, ***What is a Python decorator?***, you're not alone. Decorators are one of Python's most powerful and misunderstood features.

In this article, we'll break down:

- What Python decorators are (with analogies)
- How to use them effectively
- 7 powerful decorators (built-in and external)
- Practical examples including @wraps, @retry, @cache, and more
- Common pitfalls and memory considerations

Whether you're writing code or automating tasks, these tools will save your time, tidy up your code, and make you look like a Python expert.

What Is a Python Decorator?

A **Python decorator** is a function that modifies the behaviour of another function , without changing its actual code.

Think of it like wrapping a gift box: the contents are the same, but the presentation (and maybe how it's handled) changes. In Python, you **wrap** functions using the @decorator_name syntax.

For example:

```
@my_decorator
def say_hello():
  print("Hello")
```

This is equivalent to:

```
def say_hello():
  print("Hello")

say_hello = my_decorator(say_hello)
```

> *Decorators are a clean way to add features like logging, caching, retries, or permission checks without modifying the original logic.*

1. @wraps: Preserve the Metadata

When writing decorators, always use **@wraps** from the functools module. It preserves the metadata (such as **__name__** and **__doc__**) of the original function.

```python
from functools import wraps

def my_decorator(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        print("Before call")
        result = func(*args, **kwargs)
        print("After call")
        return result
    return wrapper
```

📌 **Why it matters:** Without @wraps, your decorated function might lose its identity, which can break tools like debuggers, doc generators, and more.

2. @retry: Automatic Retry on Failure

If you're calling unreliable APIs or systems that are prone to failure, retries are crucial.

Install it with:

```
pip install retry
```

Usage:

```
from retry import retry

@retry(tries=3, delay=2)
def fetch_data():
  print("Trying…")
  raise Exception("Fail")


fetch_data()
```

📌 **Use case:** network requests, unstable I/O operations, flaky databases.

This is an excellent example of a **Python decorator with arguments** — the tries and delay values customise the behaviour.

3. @execution_time: Measure How Long a Function Takes

You can build this yourself! Here's an example of how to create a decorator that measures execution time.

```
import time
from functools import wraps

def execution_time(func):
  @wraps(func)
  def wrapper(*args, **kwargs):
    start = time.time()
    result = func(*args, **kwargs)
    duration = time.time() - start
    print(f"{func.__name__} took {duration:.2f}s")
    return result

  return wrapper

## Usage

@execution_time
def slow_function():
  time.sleep(1)

slow_function()
```

📌 **Why it's useful:** performance bottlenecks hide in plain sight — this helps catch them early.

4. @cache: Speed Up Expensive Calls

Python's built-in @cache or @lru_cache from functools stores function results to avoid redundant calculations.

```
from functools import cache

@cache
def fib(n):
  if n < 2:
    return n
  return fib(n-1) + fib(n-2)

# Or with a size limit:

from functools import lru_cache

@lru_cache(maxsize=128)
def compute_something(x):
  return expensive_operation(x)
```

> ⚠️ *Memory Warning: Caching can consume significant memory resources. Constantly monitor cache size in long-running applications.*

📌 This is a classic **Python decorator example** that improves speed without changing your logic.

5. @log: Custom Logging Decorator

Let's build a basic logger:

```
from functools import wraps

def log(func):
  @wraps(func)
  def wrapper(*args, **kwargs):
    print(f"Calling {func.__name__} with args={args} kwargs={kwargs}")
    result = func(*args, **kwargs)
    print(f"{func.__name__} returned {result}")
    return result
  return wrapper

#Usage:
@log
def add(a, b):
  return a + b
```

📌 Useful for debugging, tracing calls, or understanding user inputs.

6. @deprecated: Mark Old Code

When refactoring codebases, marking old functions as deprecated is polite and helpful.

Install:

```
pip install Deprecated
```

Usage:

```
from deprecated import deprecated

@deprecated(reason="Use new_function() instead")
def old_function():
  print("This is old.")
```

📌 **Why it matters:** This adds a runtime warning, making deprecation visible to developers.

7. @contextmanager : Build Clean Resource Managers

Ever used with open(...)? You can build your context managers using this decorator:

```
from contextlib import contextmanager

@contextmanager
def open_file(path, mode):
  f = open(path, mode)
  try:
    yield f
  finally:
    f.close()

#Usage:
with open_file('file.txt', 'w') as f:
  f.write('hello')
```

📌 **Cleaner than try-finally**, and makes your APIs safer to use.

Final Thoughts

So, what is a Python decorator? **It's your secret weapon for writing DRY, expressive, and powerful code.**

Whether you're using a built-in like @wraps, or a 3rd-party one like @retry, the right decorator can:

- Speed up performance.
- Add logging or retry logic.
- Warn users about deprecated features.
- Manage resources safely.
- Keep your code elegant and reusable.

Want to go deeper into **Python decorators with arguments** or explore async-compatible versions? Let me know in the comments!

> ***Enjoy posts like this?*** *Here's where to find more:*
>
> ***@JitWrites on Medium*** *— real-world ML, project breakdowns, and lessons from freelancing*
>
> ***@BuildWithJit on X*** *— daily updates as I build a life with Python, ML & freelancing*
>
> *Curious about my story? Read my About Me*

Thanks for reading.

— Jit

Thank you for being a part of the community

Before you go:

- Be sure to **clap** and **follow** the writer 👏
- Follow us: **X** | **LinkedIn** | **YouTube** | **Newsletter** | **Podcast** | **Twitch**

- Start your own free AI-powered blog on Differ 🚀
- Join our content creators community on Discord 🧑‍💻
- For more content, visit plainenglish.io + stackademic.com

Python Programming | Python Decorators | Python Best Practices | Python

Published in Python in Plain English

77K followers · Last published 6 hours ago

Follow

New Python content every day. Follow to join our 3.5M+ monthly readers.

Written by Subhajit Bhar

51 followers · 44 following

Follow

Freelance Data Scientist with Physics background. Writing about Statistics, Python and Machine Learning concepts in plain English.

Responses (2)

Write a response

What are your thoughts?

Shubham Kumar
Jul 2

That's a pretty good article. I am not a python programmer but technically I understood what you are trying to explain. I hope this would be helpful for Python devs. Thanks for sharing it. Keep it up!

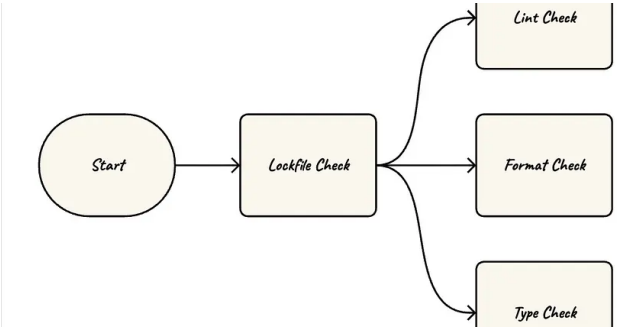
1 Reply

Pawel Jastrzebski
Jul 11

Decorators look intimidating but they're not difficult to understand and are ultimately super useful!

Reply

More from Subhajit Bhar and Python in Plain English

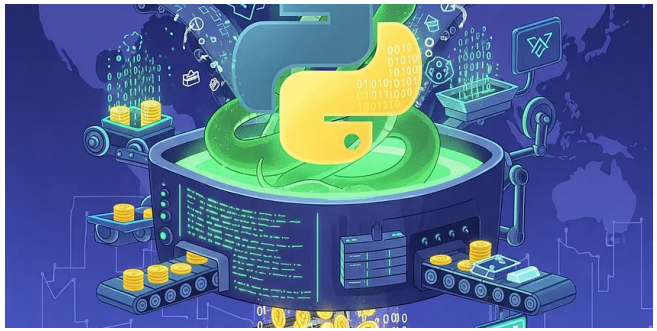


PY In Python in Plain English by Subhajit Bhar

How to Set Up a Python Code Quality CI Pipeline

You can create a Python Code Quality CI pipeline using uv, Ruff, and ty within 5...

Oct 22 138 2

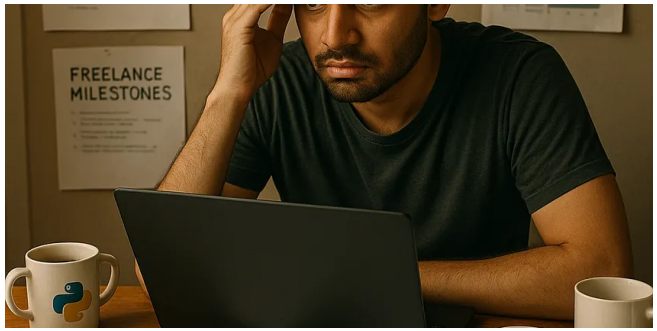


PY In Python in Plain English by Suleman Safdar

The Python Tool I Built in a Weekend That Now Pays My Rent

How I turned a tiny automation script into a paid product using libraries, clean OOP, and ...

Aug 11 4K 76



In Python in Plain English by Arslan Qutab

I Thought My Python Code Was Clean Then a Senior Developer...

The One Review That Changed How I Write Code Forever (Yes, Even After 4 Years of...

Jul 27 383 12

Subhajit Bhar

My First $10K on Upwork: What I Wish I Knew as a Beginner Pytho...

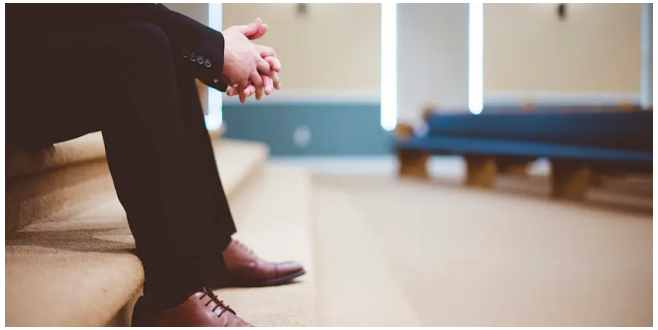
This post is for anyone starting as a freelancer, especially if you feel stuck, slow, ...

Jun 28 100 4

See all from Subhajit Bhar

See all from Python in Plain English

Recommended from Medium

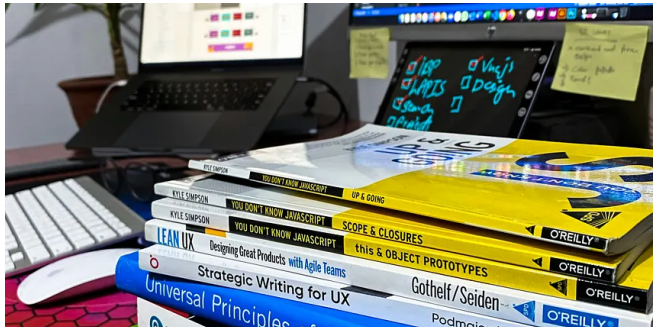


In Python in Plain English by Mayuresh K

Async Programming in Python: Part 1—The Fundamentals

From synchronous to asynchronous: A developer's practical guide to getting starte...

Jun 25 208 6



Brian Jenney

I Used to Suck at Coding Interviews. Then I Quadrupled My...

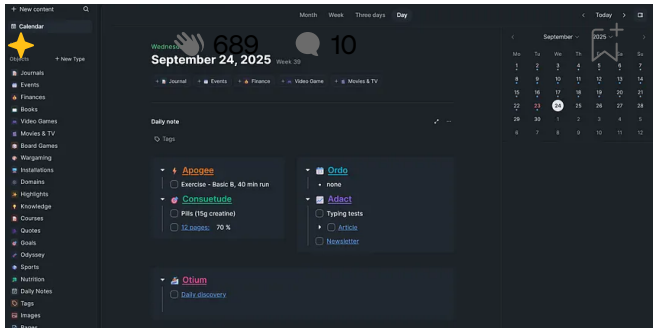
Five years ago, during an interview for a senior dev role, I had a panic attack.

Jul 19 1.1K 18

In The Pythonworld by Aashish Kumar

10 Python Concepts That Took Me Years to Understand — Until I Saw...

Sometimes one example can do more than a hundred explanations. Here are the ones tha...



Tosny

7 Websites I Visit Every Day in 2025

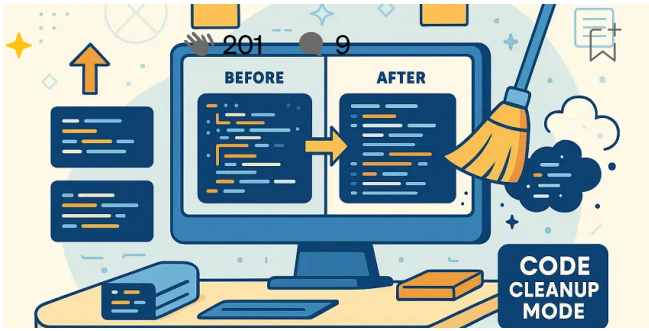
If there is one thing I am addicted to, besides coffee, it is the internet.

Sep 23 6.8K 247

Matías Salinas

I Made My Python Code 10x Faster Using These Little-Known Tricks...

When I published the first part of this article, I honestly didn't expect the impact it would...



In Top Python Libraries by Abdur Rahman

5 Python Refactoring Techniques That Instantly Cleaned Up My...

So you don't Google "how to clean up my code."

Jun 27 773 7

See more recommendations