

# Les decorators

---

## C'est quoi un decorator ?

Imaginez que vous avez une fonction qui fait quelque chose de précis. Un decorator, c'est un moyen d'ajouter des comportements autour de cette fonction, avant qu'elle s'exécute, après, ou même à la place, sans toucher à son code.

Concrètement, un decorator prend votre fonction en entrée, la "enveloppe" dans une autre fonction qui fait des choses en plus, puis vous rend quelque chose de callable. D'où le mot "décoration" : vous habillez votre fonction sans la modifier.

Les decorators servent à plein de choses dans la vraie vie :

- vérifier les arguments avant l'exécution
- modifier ce que la fonction reçoit ou retourne
- mesurer le temps d'exécution
- logger des événements
- synchroniser des threads
- mettre en cache des résultats
- gérer les droits d'accès

## La forme de base

Voici comment fonctionne un decorator dans sa version la plus simple. On passe notre fonction à une autre fonction, et on récupère ce qu'elle retourne :

```
def simple_hello():  
    print("Hello from simple function!")  
  
def simple_decorator(function):  
    print('We are about to call "{}"'.format(function.__name__))  
    return function  
  
decorated = simple_decorator(simple_hello)  
decorated()
```

En pratique, vous n'allez jamais écrire ça comme ça. Python vous offre le symbole `@` pour faire exactement la même chose de façon bien plus lisible :

```
def simple_decorator(function):  
    print('We are about to call "{}"'.format(function.__name__))  
    return function  
  
@simple_decorator  
def simple_hello():
```

```
print("Hello from simple function!")

simple_hello()
```

Résultat :

```
We are about to call "simple_hello"
Hello from simple function!
```

`function.__name__` est un attribut que Python attache automatiquement à toute fonction. Il contient son nom sous forme de chaîne. Très utile dans les decorators pour afficher des messages de log sans coder le nom en dur.

Le `@simple_decorator` au-dessus de la fonction, c'est du "sucre syntaxique" : Python le transforme automatiquement en `simple_hello = simple_decorator(simple_hello)` au moment où il interprète le fichier. C'est exactement pareil, mais beaucoup plus propre à lire.

## Rendre un decorator universel

Le problème avec notre decorator précédent : si la fonction décorée attend des arguments, ça plante. Pour qu'un decorator fonctionne avec n'importe quelle fonction, quelle que soit sa signature, on utilise `*args` et `**kwargs`.

Rappel : `*args` capture tous les arguments positionnels dans un tuple, `**kwargs` capture tous les arguments nommés dans un dictionnaire. En les utilisant dans le wrapper, on garantit que peu importe comment la fonction décorée est appelée, les arguments lui seront transmis intacts.

```
def simple_decorator(own_function):
    def internal_wrapper(*args, **kwargs):
        print(f'"{own_function.__name__}" was called with the following arguments')
        print(f'\t{args}\n\t{kwargs}\n')
        own_function(*args, **kwargs)
        print('Decorator is still operating')
    return internal_wrapper

@simple_decorator
def combiner(*args, **kwargs):
    print("\tHello from the decorated function; received arguments:", args, kwargs)

combiner('a', 'b', exec='yes')
```

On voit ici une structure importante : le decorator retourne une fonction interne (`internal_wrapper`) qui elle-même appelle la fonction d'origine. C'est le schéma classique du decorator en Python.

## Exemple concret : mesurer le temps d'exécution

Un cas d'usage très courant, c'est de savoir combien de temps met une fonction à tourner. Plutôt que d'ajouter du code de mesure dans chaque fonction, on crée un decorator une seule fois et on l'applique partout :

```
import time

def timing_decorator(func):
    def wrapper(*args, **kwargs):
        start_time = time.time()
        result = func(*args, **kwargs)
        end_time = time.time()
        execution_time = end_time - start_time
        print(f"La fonction {func.__name__} a pris {execution_time} secondes pour s'exécuter.")
    return wrapper

@timing_decorator
def add(a, b):
    time.sleep(1) # Simuler une opération prenant du temps
    return a + b

@timing_decorator
def multiply(a, b):
    time.sleep(2) # Simuler une opération prenant du temps
    return a * b

result_add = add(5, 3)
result_multiply = multiply(4, 6)
```

Les deux fonctions font des choses différentes, mais elles partagent le même mécanisme de mesure, sans dupliquer une seule ligne.

## Gérer les droits d'accès

Un autre cas très parlant : vérifier si un utilisateur est authentifié avant d'autoriser l'exécution d'une fonction sensible. C'est exactement ce que font les frameworks web comme Flask ou Django sous le capot.

```
# Simulons une classe d'utilisateur pour cet exemple
class User:
    def __init__(self, username, authenticated=False):
        self.username = username
        self.authenticated = authenticated

# Décorateur pour vérifier l'authentification de l'utilisateur
def authentication_required(func):
    def wrapper(user, *args, **kwargs):
        if user.authenticated:
```

```

        return func(user, *args, **kwargs)
    else:
        raise PermissionError("L'utilisateur n'est pas authentifié.
Accès refusé.")
    return wrapper

@authentication_required
def sensitive_operation(user):
    return f"Opération sensible effectuée pour l'utilisateur
{user.username}."

if __name__ == "__main__":
    # Création d'un utilisateur non authentifié
    guest_user = User(username="invité")

    try:
        result = sensitive_operation(guest_user)
    except PermissionError as e:
        result = str(e)

    print(result)

    # Authentification de l'utilisateur
    authenticated_user = User(username="utilisateur_authentifié",
authenticated=True)

    result = sensitive_operation(authenticated_user)
    print(result)

```

## Les decorators avec paramètres

Jusqu'ici nos decorators ne reçoivent pas d'arguments propres. Mais on peut aller plus loin : un decorator qui accepte ses propres paramètres. Il suffit d'ajouter une couche supplémentaire de fonction.

```

def warehouse_decorator(material):
    def wrapper(our_function):
        def internal_wrapper(*args):
            print(f'- Wrapping items from {our_function.__name__} with
{material}')
            our_function(*args)
            print()
        return internal_wrapper
    return wrapper

@warehouse_decorator('kraft')
def pack_books(*args):
    print("We'll pack books:", args)

@warehouse_decorator('foil')
def pack_toys(*args):
    print("We'll pack toys:", args)

```

```
@warehouse_decorator('cardboard')
def pack_fruits(*args):
    print("We'll pack fruits:", args)

pack_books('Alice in Wonderland', 'Winnie the Pooh')
pack_toys('doll', 'car')
pack_fruits('plum', 'pear')
```

Ce qui se passe quand Python rencontre `@warehouse_decorator('kraft')` :

1. `warehouse_decorator('kraft')` est appelé et retourne `wrapper`
2. `wrapper` reçoit `pack_books` comme argument
3. `wrapper` retourne `internal_wrapper`, qui devient la nouvelle version de `pack_books`

L'avantage est immédiat : on n'a pas touché aux fonctions `pack_*`, et pourtant chacune a un comportement différent selon le matériau passé.

Un exemple plus proche du monde réel, gérer des niveaux d'autorisation différents selon le rôle de l'utilisateur :

```
# Simulons une classe d'utilisateur avec des rôles pour cet exemple
class User:
    def __init__(self, username, roles=None):
        self.username = username
        self.roles = roles or []

# Décorateur avec des paramètres pour vérifier le rôle de l'utilisateur
def authorization_required(required_roles):
    def decorator(func):
        def wrapper(user, *args, **kwargs):
            if any(role in user.roles for role in required_roles):
                return func(user, *args, **kwargs)
            else:
                raise PermissionError(f"L'utilisateur {user.username} n'a pas les rôles requis pour accéder à cette fonction.")
        return wrapper
    return decorator

@authorization_required(["admin", "moderator"])
def admin_operation(user):
    return f"Opération d'administration effectuée pour l'utilisateur {user.username}."

@authorization_required(["user"])
def user_operation(user):
    return f"Opération utilisateur effectuée pour l'utilisateur {user.username}."

if __name__ == "__main__":
    regular_user = User(username="utilisateur_ordinaire", roles=["user"])
```

```

try:
    result = admin_operation(regular_user)
except PermissionError as e:
    result = str(e)

print(result)

admin_user = User(username="admin_utilisateur", roles=["admin"])

result = admin_operation(admin_user)
print(result)

try:
    result = user_operation(admin_user)
except PermissionError as e:
    result = str(e)

print(result)

```

## Empiler plusieurs decorators (decorator stacking)

Python vous permet d'appliquer plusieurs decorators sur la même fonction. L'ordre dans lequel vous les écrivez a de l'importance : le decorator le plus proche de la fonction s'exécute en premier.

```

@outer_decorator
@inner_decorator
def function():
    pass

```

Sans le @, ça s'écrirait :

```
function = outer_decorator(inner_decorator(function))
```

L'ordre d'exécution : `outer_decorator` appelle `inner_decorator`, qui lui appelle `function`. Quand `function` termine, `inner_decorator` reprend la main, puis c'est au tour de `outer_decorator` de finir son travail.

Voici un exemple avec deux decorators distincts qu'on cumule :

```

def big_container(collective_material):
    def wrapper(our_function):
        def internal_wrapper(*args):
            our_function(*args)
            print('- The whole order would be packed with',
collective_material)
        print()

```

```

        return internal_wrapper
    return wrapper

def warehouse_decorator(material):
    def wrapper(our_function):
        def internal_wrapper(*args):
            our_function(*args)
            print(f'- Wrapping items from {our_function.__name__} with {material}')
        return internal_wrapper
    return wrapper

```

```

@big_container('plain cardboard')
@warehouse_decorator('bubble foil')
def pack_books(*args):
    print("We'll pack books:", args)

pack_books('Alice in Wonderland', 'Winnie the Pooh')

```

We'll pack books: ('Alice in Wonderland', 'Winnie the Pooh')

- Wrapping items from pack\_books with bubble foil
- The whole order would be packed with plain cardboard

```

@big_container('colourful cardboard')
@warehouse_decorator('foil')
def pack_toys(*args):
    print("We'll pack toys:", args)

pack_toys('doll', 'car')

```

We'll pack toys: ('doll', 'car')

- Wrapping items from pack\_toys with foil
- The whole order would be packed with colourful cardboard

```

@big_container('strong cardboard')
@warehouse_decorator('cardboard')
def pack_fruits(*args):
    print("We'll pack fruits:", args)

pack_fruits('plum', 'pear')

```

Un exemple qui combine stacking et contrôle d'accès, ici on vérifie à la fois l'authentification et un seuil de staking :



```

class User:
    def __init__(self, username, staked_amount=0, authenticated=False):
        self.username = username
        self.staked_amount = staked_amount
        self.authenticated = authenticated

    def staking_required(minimum_stake):
        def decorator(func):
            def wrapper(user, *args, **kwargs):
                if user.staked_amount >= minimum_stake:
                    return func(user, *args, **kwargs)
                else:
                    raise PermissionError(f"L'utilisateur {user.username} doit bloquer au moins {minimum_stake} ressources pour accéder à cette fonction.")
            return wrapper
        return decorator

    def authentication_required(func):
        def wrapper(user, *args, **kwargs):
            if user.authenticated:
                return func(user, *args, **kwargs)
            else:
                raise PermissionError("L'utilisateur n'est pas authentifié. Accès refusé.")
        return wrapper

    @staking_required(minimum_stake=100)
    @authentication_required
    def secure_operation(user):
        return f"Opération sécurisée effectuée pour l'utilisateur {user.username}."

if __name__ == "__main__":
    regular_user = User(username="utilisateur_ordinaire", staked_amount=50)

    try:
        result = secure_operation(regular_user)
    except PermissionError as e:
        result = str(e)

    print(result)

    authenticated_staking_user =
    User(username="utilisateur_authentifié_avec_staking", staked_amount=150,
    authenticated=True)

    result = secure_operation(authenticated_staking_user)
    print(result)

```

## Les decorators sous forme de classe

Un decorator n'est pas forcément une fonction. En Python, une classe peut jouer exactement le même rôle, à condition d'implémenter la méthode spéciale `__call__`. Cette méthode est appelée automatiquement lorsqu'on utilise l'objet comme si c'était une fonction.

```
class SimpleDecorator:
    def __init__(self, own_function):
        self.func = own_function

    def __call__(self, *args, **kwargs):
        print(f'"{self.func.__name__}" was called with the following arguments')
        print(f'\t{args}\n\t{kwargs}\n')
        self.func(*args, **kwargs)
        print('Decorator is still operating')

@SimpleDecorator
def combiner(*args, **kwargs):
    print("\tHello from the decorated function; received arguments:", args,
kwargs)

combiner('a', 'b', exec='yes')
```

L'intérêt par rapport aux decorators fonctionnels : vous disposez de toute la puissance des classes, héritage, méthodes utilitaires, état interne. C'est particulièrement utile quand le decorator a besoin de maintenir des données entre les appels.

Exemple : enregistrer les statistiques d'exécution dans une base de données SQLite à chaque appel de la fonction décorée.

```
import time
import sqlite3

class ExecutionStatsDecorator:
    def __init__(self, func):
        self.func = func
        self.conn = sqlite3.connect("execution_stats.db")
        self.create_table()

    def create_table(self):
        cursor = self.conn.cursor()
        cursor.execute('''
            CREATE TABLE IF NOT EXISTS execution_stats (
                function_name TEXT,
                execution_time REAL,
                execution_date TEXT
            )
        ''')
        self.conn.commit()

    def __call__(self, *args, **kwargs):
```

```

        start_time = time.time()
        result = self.func(*args, **kwargs)
        end_time = time.time()
        execution_time = end_time - start_time
        function_name = self.func.__name__
        execution_date = time.strftime('%Y-%m-%d %H:%M:%S')

        self.save_to_database(function_name, execution_time,
                               execution_date)

        print(f"La fonction {function_name} a pris {execution_time}
secondes pour s'exécuter et les statistiques ont été enregistrées.")
        return result

    def save_to_database(self, function_name, execution_time,
                        execution_date):
        cursor = self.conn.cursor()
        cursor.execute('''
            INSERT INTO execution_stats (function_name, execution_time,
            execution_date)
            VALUES (?, ?, ?)
            ''', (function_name, execution_time, execution_date))
        self.conn.commit()

    def __del__(self):
        self.conn.close()

@ExecutionStatsDecorator
def example_function():
    time.sleep(2)
    print("Fonction exécutée.")

example_function()

```

La connexion à la base de données est ouverte une seule fois dans `__init__` et fermée proprement dans `__del__`. Avec un decorator fonctionnel classique, ce genre de gestion d'état serait bien plus compliqué à mettre en place.

## Classe decorator avec paramètres

Même chose qu'avec les fonctions : une classe decorator peut recevoir des arguments. Dans ce cas, `__init__` reçoit les paramètres, et `__call__` reçoit la fonction à décorer.

```

class WarehouseDecorator:
    def __init__(self, material):
        self.material = material

    def __call__(self, own_function):
        def internal_wrapper(*args, **kwargs):
            print(f'- Wrapping items from {own_function.__name__} with
{self.material}')

```

```

        own_function(*args, **kwargs)
        print()
    return internal_wrapper

@WarehouseDecorator('kraft')
def pack_books(*args):
    print("We'll pack books:", args)

@WarehouseDecorator('foil')
def pack_toys(*args):
    print("We'll pack toys:", args)

@WarehouseDecorator('cardboard')
def pack_fruits(*args):
    print("We'll pack fruits:", args)

pack_books('Alice in Wonderland', 'Winnie the Pooh')
pack_toys('doll', 'car')
pack_fruits('plum', 'pear')

```

Avantage par rapport à la version fonctionnelle : on évite les fonctions imbriquées à trois niveaux. La classe aplatit cette structure et la rend plus lisible.

### Classe decorator avec timeout

Pour finir, un exemple qui combine paramètres et gestion de thread : on veut pouvoir annuler une fonction si elle dépasse un temps limite.

```

import time
import threading

class TimeoutDecorator:
    def __init__(self, timeout_seconds=5):
        self.timeout_seconds = timeout_seconds

    def __call__(self, func):
        def wrapper(*args, **kwargs):
            result = None
            exception = None

            def target():
                nonlocal result, exception
                try:
                    result = func(*args, **kwargs)
                except Exception as e:
                    exception = e

            thread = threading.Thread(target=target)
            thread.start()
            thread.join(timeout=self.timeout_seconds)

```

```

        if thread.is_alive():
            thread.join() # Attendre que le thread se termine
naturellement
            raise TimeoutError(f"L'exécution de la fonction
{func.__name__} a dépassé la limite de temps de {self.timeout_seconds}
secondes.")

        if exception:
            raise exception

        return result

    return wrapper

@TimeoutDecorator(timeout_seconds=3)
def example_function():
    time.sleep(5)
    print("Fonction exécutée avec succès après une attente de 5 secondes.")

try:
    example_function()
except TimeoutError as e:
    print(e)

```

La fonction tourne dans un thread séparé. Si elle ne termine pas dans les `timeout_seconds` impartis, on lève une `TimeoutError`. C'est un pattern utile pour tout appel réseau ou opération longue dont vous voulez maîtriser la durée.

Quelques points sur le code :

- `threading.Thread(target=target)` : crée un thread qui va exécuter la fonction `target` quand on le démarre.
- `thread.start()` : lance l'exécution du thread.
- `thread.join(timeout=self.timeout_seconds)` : bloque le code appelant jusqu'à ce que le thread termine — ou jusqu'à ce que le délai `timeout` soit écoulé, selon ce qui arrive en premier. Si le délai expire avant la fin, `thread.join()` rend la main sans erreur, et on peut ensuite vérifier avec `thread.is_alive()` si le thread tourne encore.
- `nonlocal result, exception` : dans une fonction imbriquée, `nonlocal` permet de modifier des variables définies dans la fonction parente (ici `wrapper`). Sans ça, l'assignation `result = ...` dans `target()` créerait une variable locale à `target`, et `wrapper` ne la verrait jamais.

## argparse : construire de vrais outils en ligne de commande

---

### Pourquoi argparse ?

Jusqu'ici, quand vous voulez passer des valeurs à un script Python, vous les codez en dur ou vous utilisez `input()`. Ça marche pour un exercice, mais dans la vraie vie, un outil en ligne de commande doit

pouvoir recevoir des arguments directement à l'exécution :

```
python scanner.py --host 192.168.1.1 --port 80
python backup.py --source /home/data --destination /mnt/backup --verbose
```

C'est exactement ce que fait **argparse** : il parse les arguments passés à votre script et vous les rend sous forme d'objet Python, avec validation, valeurs par défaut et messages d'aide générés automatiquement.

**argparse** fait partie de la bibliothèque standard Python, rien à installer.

## La base : un script avec des arguments

```
import argparse

parser = argparse.ArgumentParser(description="Un outil de salutation")
parser.add_argument("nom", help="Le nom de la personne à saluer")
parser.add_argument("--majuscule", action="store_true", help="Afficher le message en majuscules")

args = parser.parse_args()

message = f"Bonjour, {args.nom} !"
if args.majuscule:
    message = message.upper()

print(message)
```

```
$ python salut.py Alice
Bonjour, Alice !

$ python salut.py Alice --majuscule
BONJOUR, ALICE !
```

Deux types d'arguments ici :

- **Positionnel** (**nom**) : obligatoire, passé sans tiret, dans l'ordre.
- **Optionnel** (**--majuscule**) : facultatif, préfixé par **--**. Avec **action="store\_true"**, il fonctionne comme un booléen : présent = **True**, absent = **False**.

## Arguments avec types et valeurs par défaut

Par défaut, **argparse** traite tout comme des chaînes de caractères. Vous pouvez spécifier le type attendu :

```
import argparse

parser = argparse.ArgumentParser(description="Scanner de ports")
parser.add_argument("host", help="Adresse IP ou hostname cible")
parser.add_argument("-p", "--port", type=int, default=80, help="Port à scanner (défaut: 80)")
parser.add_argument("-t", "--timeout", type=float, default=1.0, help="Timeout en secondes (défaut: 1.0)")

args = parser.parse_args()

print(f"Scan de {args.host} sur le port {args.port} (timeout: {args.timeout}s)")
```

```
$ python scanner.py 192.168.1.1
Scan de 192.168.1.1 sur le port 80 (timeout: 1.0s)

$ python scanner.py 192.168.1.1 -p 443 -t 2.5
Scan de 192.168.1.1 sur le port 443 (timeout: 2.5s)
```

Notez le `-p` : c'est un alias court pour `--port`. Convention classique dans les outils CLI.

## L'aide automatique

Un des gros avantages d'argparse : le `--help` est généré tout seul à partir de ce que vous avez déclaré.

```
$ python scanner.py --help
usage: scanner.py [-h] [-p PORT] [-t TIMEOUT] host

Scanner de ports

positional arguments:
  host                  Adresse IP ou hostname cible

options:
  -h, --help            show this help message and exit
  -p PORT, --port PORT  Port à scanner (défaut: 80)
  -t TIMEOUT, --timeout TIMEOUT
                        Timeout en secondes (défaut: 1.0)
```

Vous n'avez rien codé pour ça. C'est `argparse` qui construit ce message à partir des `add_argument()` et de la `description` du parser.

## Restreindre les valeurs possibles avec choices

Parfois un argument ne peut prendre que certaines valeurs. Plutôt que de valider ça vous-même après le parsing, `argparse` peut le faire pour vous :

```
import argparse

parser = argparse.ArgumentParser(description="Analyseur réseau")
parser.add_argument("protocole", choices=["tcp", "udp", "icmp"],
                    help="Protocole à analyser")
parser.add_argument("-v", "--verbosity", type=int, choices=[0, 1, 2],
                    default=0, help="Niveau de verbosité")

args = parser.parse_args()
print(f"Analyse {args.protocole} (verbosité: {args.verbosity})")
```

```
$ python analyse.py tcp -v 2
Analyse tcp (verbosité: 2)

$ python analyse.py http
usage: analyse.py [-h] [-v {0,1,2}] {tcp,udp,icmp}
analyse.py: error: argument protocole: invalid choice: 'http' (choose from
'tcp', 'udp', 'icmp')
```

Si l'utilisateur passe une valeur hors de `choices`, `argparse` refuse et affiche un message d'erreur clair.

## Accepter plusieurs valeurs avec nargs

Pour un argument qui accepte plusieurs valeurs :

```
import argparse

parser = argparse.ArgumentParser(description="Scanner multi-ports")
parser.add_argument("host", help="Adresse IP cible")
parser.add_argument("-p", "--ports", nargs="+", type=int, required=True,
                    help="Liste de ports à scanner")

args = parser.parse_args()

for port in args.ports:
    print(f"Scan de {args.host}:{port}")
```

```
$ python scanner.py 192.168.1.1 -p 22 80 443
Scan de 192.168.1.1:22
Scan de 192.168.1.1:80
Scan de 192.168.1.1:443
```



Les valeurs possibles de `nargs` :

- `"+"` : une ou plusieurs valeurs (erreur si aucune)
- `"*"` : zéro ou plusieurs valeurs
- `"?"` : zéro ou une valeur
- un entier (ex: `3`) : exactement ce nombre de valeurs

## Les sous-commandes avec `add_subparsers`

Quand votre outil grossit, vous pouvez le découper en sous-commandes, comme `git commit`, `git push`, `git log` :

```
import argparse

parser = argparse.ArgumentParser(description="Outil réseau")
subparsers = parser.add_subparsers(dest="commande", help="Commande à exécuter")

# Sous-commande "scan"
scan_parser = subparsers.add_parser("scan", help="Scanner un host")
scan_parser.add_argument("host", help="Adresse IP cible")
scan_parser.add_argument("-p", "--ports", nargs="+", type=int, default=[80, 443])

# Sous-commande "ping"
ping_parser = subparsers.add_parser("ping", help="Pinger un host")
ping_parser.add_argument("host", help="Adresse IP cible")
ping_parser.add_argument("-c", "--count", type=int, default=4, help="Nombre de pings")

args = parser.parse_args()

if args.commande == "scan":
    print(f"Scan de {args.host} sur les ports {args.ports}")
elif args.commande == "ping":
    print(f"Ping de {args.host} ({args.count} fois)")
else:
    parser.print_help()
```

```
$ python netool.py scan 192.168.1.1 -p 22 80
Scan de 192.168.1.1 sur les ports [22, 80]
```

```
$ python netool.py ping 10.0.0.1 -c 10
Ping de 10.0.0.1 (10 fois)
```

```
$ python netool.py --help
usage: netool.py [-h] {scan,ping} ...
```

Outil réseau

```
positional arguments:
  {scan,ping}  Commande à exécuter
    scan      Scanner un host
    ping      Pinger un host
```

Chaque sous-commande a ses propres arguments, indépendants les uns des autres.

## Combiner argparse et decorators

argparse et les decorators se combinent bien ensemble. Vous pouvez créer un decorator qui ajoute automatiquement du logging ou du timing à n'importe quelle commande CLI :

```
import argparse
import time

def timed_command(func):
    def wrapper(args):
        if args.verbose:
            print(f"[DEBUG] Lancement de la commande...")
            start = time.time()
            result = func(args)
            elapsed = time.time() - start
            print(f"[DEBUG] Terminé en {elapsed:.2f}s")
            return result
        return func(args)
    return wrapper

parser = argparse.ArgumentParser(description="Outil réseau")
parser.add_argument("host", help="Adresse IP cible")
parser.add_argument("-p", "--port", type=int, default=80)
parser.add_argument("-v", "--verbose", action="store_true", help="Mode
verbeux")

@timed_command
def scan(args):
    print(f"Scan de {args.host}:{args.port}")
    time.sleep(0.5)

args = parser.parse_args()
scan(args)
```

```
$ python scanner.py 192.168.1.1 -p 443 -v
[DEBUG] Lancement de la commande...
Scan de 192.168.1.1:443
[DEBUG] Terminé en 0.50s
```

Le decorator `timed_command` ne sait rien du scan, et le scan ne sait rien du timing. Chacun fait son travail, c'est exactement le principe qu'on a vu plus haut.