# 🧙 Master Python Decorators with Magical Mini-Quests 🪄

**Unlock the power of Python decorators** by solving these real-world inspired coding challenges. From boosting performance to debugging like a pro — let's wrap functions like a Python wizard! 🧠✨

---

## ▶ 1. ⏱ Time Tracker Spell: Measure Execution Time

### 🧩 Challenge

Create a decorator `@timer` that measures and prints how long a function takes to run. Perfect for performance tuning! 🧪

📌 *Use Case:* Benchmarking slow functions, loops, or data processing tasks.

---

### ☑ Code

```python
import time

def timer(func):
    def wrapper(*args, **kwargs):
        start = time.time()                         # Record start time
        result = func(*args, **kwargs)              # Run the actual function
        end = time.time()                           # Record end time
        print(f"{func.__name__} ran in {end - start:.4f} seconds")
        return result                               # Return original result
    return wrapper

@timer
def example_function(n):
    time.sleep(n)                                   # Simulates a time-consuming task

example_function(2)
```

---

### 🧠 Explanation:

- `@timer` is a **decorator** that wraps any function and records:
  - When it **starts**
  - When it **ends**
  - The **difference** gives us execution time.
- `time.sleep(n)` simulates delay (like data processing or API calls).
- The decorator is **reusable** for any function — small or large.

---

### 🔍 Output:

```
example_function ran in 2.0021 seconds
```

---

---

## ▶ 2. 🐞 Debug Mirror Charm: Trace Function Calls

### 🧩 Challenge

Create a decorator @debug that prints the function name and all the arguments passed every time it's called.

📌 *Use Case:* Debugging complex logic, watching function flow, or auditing arguments.

---

### ✅ Code

```
def debug(func):
    def wrapper(*args, **kwargs):
        args_value = ', '.join(str(arg) for arg in args)              #
Convert positional args to string
        kwargs_value = ', '.join(f"{k}={v}" for k, v in kwargs.items())     #
Convert keyword args to string
        print(f"🔔 Calling: {func.__name__} with args [{args_value}] and kwargs
[{kwargs_value}]")
        return func(*args, **kwargs)                                  # Run
actual function
    return wrapper

@debug
def hello():
    print("Hello 🧙")

@debug
def greet(name, greeting="Hello"):
    print(f"{greeting}, {name}!")

hello()
greet("Chai", greeting="Hanji")
```

---

### 🧠 Explanation:

- The @debug decorator wraps any function and logs:
    - Function name (func.__name__)
    - Arguments (*args) and keyword arguments (**kwargs)

- Helps track **what's being passed** to your function every time.

---

### 🔍 Output:

```
📣 Calling: hello with args [] and kwargs []
Hello 🧙
📣 Calling: greet with args [Chai] and kwargs [greeting=Hanji]
Hanji, Chai!
```

---

---

## ▶ 3. 🧠 Memory Potion: Cache Function Results

### 🧩 Challenge

Build a decorator `@cache` that **stores return values** of a function so repeated calls with the same inputs return the result instantly!

📌 *Use Case:* Recursive computations (Fibonacci, factorial), expensive calculations, or API responses.

---

### ✅ Code

```python
import time

def cache(func):
    cache_value = {}                        # Dictionary to store cached results

    def wrapper(*args):
        if args in cache_value:
            print(f"⚡ Returning cached result for {args}")
            return cache_value[args]        # Return cached result if available

        print(f"🧪 Computing new result for {args}")
        result = func(*args)                # Call actual function
        cache_value[args] = result          # Save result to cache
        return result
    return wrapper

@cache
def long_running_function(a, b):
    time.sleep(4)                           # Simulate slow calculation
    return a + b

print(long_running_function(2, 3))          # First call - slow
print(long_running_function(2, 3))          # Second call - cached
print(long_running_function(4, 3))          # New call - slow again
```

---

### 🧠 Explanation:

- `cache_value` is a **dictionary** that stores input `args` as keys and output as values.
- When the function is called:
  - If the input exists → Return from cache ⚡
  - Else → Compute, then store 🧪
- This saves time when dealing with **repeated calls**.

---

### 🔍 Output:

```
🧪 Computing new result for (2, 3)
5
⚡ Returning cached result for (2, 3)
5
🧪 Computing new result for (4, 3)
7
```

---

---

## 📚 Summary Table

| Decorator | Purpose | Emoji | Use Case |
|---|---|---|---|
| `@timer` | Measure execution time | ⏱️ | Performance tuning, benchmarking |
| `@debug` | Log function calls and arguments | 🐞 | Debugging, tracing |
| `@cache` | Store and reuse results | 🧠 | Expensive calls, recursion |

---