

---

# Function Caching in Python – Colorful Notes

**Function caching** = remembering function results for repeated inputs to avoid recalculation.

---

## Why use caching?

- ✨ Improves **speed** for repeated calls.
  - ✨ Saves **time** for expensive computations.
  - ✨ Useful for **recursive functions**, **API calls**, **database queries**.
- 

## Normal Function vs Cached Function

Feature	Normal Function 🐢	Cached Function 🌳
Calculation	Runs every time	Runs only for new input
Speed	Slower if repeated	Faster for repeated calls
Memory	Minimal	Uses memory to store results
Example	Recomputes Fibonacci	Returns stored Fibonacci

### Normal function

```
def add(a, b):  
    print("Calculating...")  
    return a + b  
  
print(add(2, 3)) # Calculating... 5  
print(add(2, 3)) # Calculating... 5
```

### Cached function

```
from functools import lru_cache  
  
@lru_cache()  
def add(a, b):  
    print("Calculating...")  
    return a + b
```

```
print(add(2, 3)) # Calculating... 5
print(add(2, 3)) # 5 (from cache 🦜)
```

🦜 Cached function **skips recalculation** if it already has the answer.



## Using `functools.lru_cache`

```
from functools import lru_cache

@lru_cache(maxsize=128) # Stores last 128 results
def fibonacci(n):
    if n < 2:
        return n
    return fibonacci(n-1) + fibonacci(n-2)

print(fibonacci(10)) # 55
```



### Notes:

- `maxsize` = number of results to store ( `None` = unlimited)
- First call → calculates normally 🐢
- Repeated call → instantly returns cached result ⚡



## Manual caching with dictionary

```
cache = {}

def fibonacci(n):
    if n in cache:
        return cache[n] # 🦜 Return cached result
    if n < 2:
        result = n
    else:
        result = fibonacci(n-1) + fibonacci(n-2)
    cache[n] = result # 😊 Store in cache
    return result

print(fibonacci(10)) # 55
```

```
nums={}
def add_num(a,b):
    if (a,b) in nums:
        return nums[(a,b)]
    print("calculating...")
    nums[(a,b)]=a+b
    return a+b
print(add_num(2,3))
print(add_num(2,3))
print(add_num(7,3))
print(nums)
```

(a,b) tuple as key



Full control over caching.



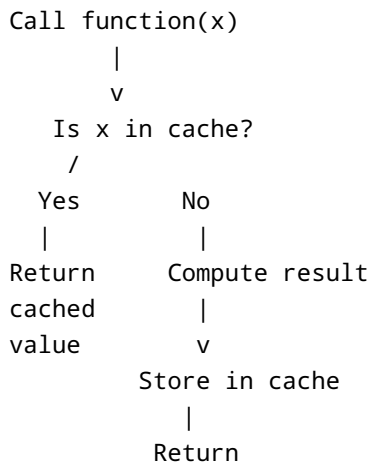
More code compared to `lru_cache`.



## How caching works (flow)

1. Call function with input `x` 🌸
2. **Check cache:**
3. 🦉 If result exists → return cached value
4. 🦉 If not → compute → store in cache → return

**Flow diagram:**



## When to use caching

- 🌸 Recursive functions → Fibonacci, factorial, DP problems
- 🌸 Heavy computations that repeat
- 🌸 Repeated API/database calls

## Summary

- 🦉 Normal function → always calculates.
- 🦉 Cached function → remembers results → **faster on repeated inputs**.
- `functools.lru_cache` = **easy & powerful**.
- Manual dictionary caching = **flexible but longer code**.