# COL1000: Introduction to Programming

## Manual Caching, Decorators

Subodh Sharma | Lec 24 | Oct 14

# Memoization

- Using in-built caches
  - **From functools import lru_cache**

```
@lru_cache(maxsize=None)
def fib_cached(n: int) -> int:
    if n < 2:
        return n
    return fib_cached(n-1) + fib_cached(n-2)

print(fib_cached(100))
```

# Manual Memoization

- Maintain a separate dictionary (_cache)
  - **Update the dictionary and check of the result is cached in it**

```
_cache = {0: 0, 1: 1}
def fib(n: int) -> int:
    if n < 0:
        raise ValueError("n must be non-negative")
    if n in _cache:
        return _cache[n]
    _cache[n] = fib(n - 1) + fib(n - 2)
    return _cache[n]
```

# Decorators — Instance of Higher Order Functions

- Decorators are Python Callables that takes another function, **wraps it,** and **returns a new callable**
  - One could add new behaviour to the wrapped function without modifying it
- **What is their use?**
  - **For supporting cross-cutting concerns beyond the functionality of a function**
    - **Such as logging, timing, caching, validation, access control ….**

# Decorators — An example definition

- func_new = decorator (func);

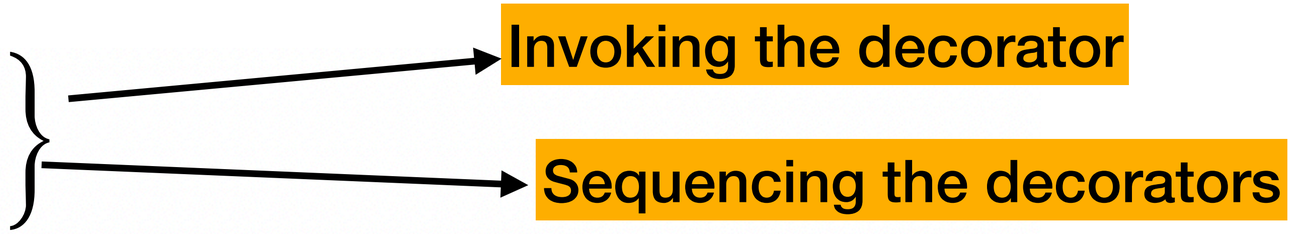
```
import time
import functools

def timed(func):
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        t0 = time.perf_counter()

        out= func(*args, **kwargs)

        dt = time.perf_counter() - t0
        print(f"[timed] {func.__name__} took {dt:.6f}s")
        return out
    return wrapper
```

preserve name, doc, annotations

Pre code

Post code

# Decorators — An example usage




```
@lru_cache(maxsize=64)
@trace
@timed
_cache = {0: 0, 1: 1}
def fib(n: int) -> int:
    if n < 0:
        raise ValueError("n must be non-negative")
    if n in _cache:
        return _cache[n]
    _cache[n] = fib(n - 1) + fib(n - 2)
    return _cache[n]

fib(100)
print(fib.cache_info())
```