Solutions — Type Hints, Decorators, Concurrency

◈ Easy (10)

1. Annotate a variable `age` as an `int` and assign it the value `25`.

```
age: int = 25
print(age)
```

Explanation: Simple variable annotation. Type hints do not change runtime behavior but are used by tools like linters and `mypy`.

---

2. Write a function `greet(name: str) -> str` that returns `"Hello, <name>"`.

```
def greet(name: str) -> str:
    return f"Hello, {name}"

print(greet("Alice"))
```

Explanation: Parameter and return type annotated.

---

3. Create a function `add(a: int, b: int) -> int` and test it with integers.

```
def add(a: int, b: int) -> int:
    return a + b

print(add(3, 4))  # 7
```

---

4. Write a function `is_even(n: int) -> bool` that returns `True` if a number is even.

```
def is_even(n: int) -> bool:
    return (n % 2) == 0

print(is_even(10))  # True
print(is_even(7))   # False
```

---

5. Annotate a list of integers named `scores` using `List[int]`.

```
from typing import List

scores: List[int] = [95, 82, 74]
print(scores)
```

Explanation: `typing.List` used for static annotation (Python 3.9+ can also use `list[int]`).

---

6. Write a simple decorator **`@announce`** that prints **`"Starting function..."`** before the wrapped function executes.

```
from functools import wraps

def announce(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        print("Starting function...")
        return func(*args, **kwargs)
    return wrapper

@announce
def say_hi():
    print("Hi!")

say_hi()
```

Output

```
Starting function...
Hi!
```

---

7. Use **`@staticmethod`** in a class **`MathUtils`** for a method **`square(x)`** that returns **`x**2`**.

```
class MathUtils:
    @staticmethod
    def square(x):
        return x * x

print(MathUtils.square(5))  # 25
```

Explanation: `@staticmethod` doesn't receive `self` or `cls`.

---

8. Use **`@classmethod`** in a class **`Person`** that creates an instance from a birth year.

```
from datetime import date

class Person:
    def __init__(self, name: str, birth_year: int):
        self.name = name
```

```
        self.birth_year = birth_year

    @classmethod
    def from_age(cls, name: str, age: int):
        current_year = date.today().year
        birth_year = current_year - age
        return cls(name, birth_year)

p = Person.from_age("Bob", 30)
print(p.name, p.birth_year)
```

Explanation: `@classmethod` gets `cls` and can return a new instance.

---

9. Print the difference between CPU-bound and I/O-bound tasks in your own words.

```
CPU-bound tasks spend most time doing computations on the CPU (e.g., number crunching).
I/O-bound tasks spend most time waiting for input/output (e.g., network, disk).
```

---

10. Start a thread with **`threading.Thread(target=...)`** that prints **`"Hello from thread!"`**.

```
import threading

def worker():
    print("Hello from thread!")

t = threading.Thread(target=worker)
t.start()
t.join()
```

Explanation: Basic thread start and join to wait for completion.

◈ Medium (10)

11. **`find_user(users: Dict[int, str], user_id: int) -> Optional[str]`**.

```
from typing import Dict, Optional

def find_user(users: Dict[int, str], user_id: int) -> Optional[str]:
    return users.get(user_id)

users = {1: "Alice", 2: "Bob"}
```

```
print(find_user(users, 1))  # Alice
print(find_user(users, 3))  # None
```

---

12. `concat(items: List[str]) -> str` that joins a list of strings into one.

```
from typing import List

def concat(items: List[str]) -> str:
    return "".join(items)

print(concat(["a", "b", "c"]))  # "abc"
```

---

13. Add type hints to `calculate_area(width, height)`.

```
def calculate_area(width: float, height: float) -> float:
    return width * height

print(calculate_area(3.5, 2))  # 7.0
```

Explanation: Using `float` helps express that halves/decimals are allowed.

---

14. Decorator `@timeit` that measures how long a function takes to run.

```
import time
from functools import wraps

def timeit(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        start = time.perf_counter()
        result = func(*args, **kwargs)
        end = time.perf_counter()
        print(f"{func.__name__} took {end - start:.6f} seconds")
        return result
    return wrapper

@timeit
def slow(n):
    total = 0
    for i in range(n):
        total += i
    return total

slow(100000)
```

15. Decorator `@repeat(n)` that runs a function `n` times.

```python
from functools import wraps
from typing import Callable

def repeat(n: int):
    def decorator(func: Callable):
        @wraps(func)
        def wrapper(*args, **kwargs):
            results = []
            for _ in range(n):
                results.append(func(*args, **kwargs))
            return results
        return wrapper
    return decorator

@repeat(3)
def say(msg):
    print(msg)
    return "done"

print(say("Hi"))
```

Explanation: Parameterized decorator — returns a decorator that returns a wrapper.

---

16. Decorator `@logger` that logs function name and args before execution.

```python
from functools import wraps

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

@logger
def multiply(a, b):
    return a * b

print(multiply(3, 4))
```

---

17. Use two decorators on the same function (e.g., `@logger` and `@timeit`) and check order.

```
@logger
@timeit
def compute(n):
    s = 0
    for i in range(n):
        s += i
    return s

compute(100000)
```

Explanation: Decorators stack top-down: `@timeit` wraps `compute`, then `@logger` wraps the result. So logging happens outside timing in this stacking (logger logs call of the timeit-wrapped function). If you swap order, behavior/printed order changes.

---

18. Start 3 threads, each printing `"Thread <n> working..."`.

```
import threading

def worker(n):
    print(f"Thread {n} working...")

threads = []
for i in range(1, 4):
    t = threading.Thread(target=worker, args=(i,))
    threads.append(t)
    t.start()

for t in threads:
    t.join()
```

---

19. Use `ThreadPoolExecutor` to square numbers 1–5.

```
from concurrent.futures import ThreadPoolExecutor, as_completed

def square(x):
    return x * x

with ThreadPoolExecutor(max_workers=3) as executor:
    futures = [executor.submit(square, i) for i in range(1, 6)]
    for future in as_completed(futures):
        print(future.result())
```

Explanation: Executor manages pool; `as_completed` yields completed futures.

---

20. Explain GIL (2–3 sentences).

`The Global Interpreter Lock (GIL) is a mutex in CPython that allows only one thread to execu`

◈ Hard / Tricky (5)

21. Decorator `@retry(n)` that retries a function up to `n` times if it raises an exception.

```python
from functools import wraps
import time
from typing import Callable

def retry(n: int, delay: float = 0.0):
    def decorator(func: Callable):
        @wraps(func)
        def wrapper(*args, **kwargs):
            last_exc = None
            for attempt in range(1, n + 1):
                try:
                    return func(*args, **kwargs)
                except Exception as e:
                    last_exc = e
                    print(f"Attempt {attempt} failed: {e}")
                    if delay:
                        time.sleep(delay)
            # after retries, re-raise the last exception
            raise last_exc
        return wrapper
    return decorator

counter = {"calls": 0}

@retry(3, delay=0.1)
def flaky():
    counter["calls"] += 1
    if counter["calls"] < 3:
        raise ValueError("Temporary failure")
    return "Success"

print(flaky())  # Should succeed on 3rd try
```

Explanation: Parameterized decorator stores attempts and optionally sleeps between retries.

---

22. Combine `@logger` and `@retry(3)` on a function and test interaction.

```
@logger
@retry(3, delay=0.05)
def risky(x):
    if x < 0:
        raise ValueError("x must be non-negative")
    return x * 2

print(risky(2))  # logger prints call; retry not needed
try:
    print(risky(-1))  # will log and retry, then raise
except Exception as e:
    print("Final exception:", e)
```

Explanation: Order matters. Here `retry` wraps `risky` first, then `logger` logs the calls to the retry-wrapper. If you want to log each attempt inside `retry`, add logging inside the retry loop instead.

---

23. Use `ThreadPoolExecutor` to read multiple files concurrently (using `open()` & `read()`).

```
from concurrent.futures import ThreadPoolExecutor, as_completed
from pathlib import Path
from typing import List

def read_file(path: str) -> str:
    with open(path, "r", encoding="utf-8") as f:
        return f.read()

def read_files_concurrently(paths: List[str]):
    with ThreadPoolExecutor(max_workers=4) as executor:
        futures = {executor.submit(read_file, p): p for p in paths}
        results = {}
        for future in as_completed(futures):
            p = futures[future]
            try:
                results[p] = future.result()
            except Exception as exc:
                results[p] = f"Error: {exc}"
    return results
```

```
# Example use:
# Create some small text files on disk first, then call:
# results = read_files_concurrently(["f1.txt", "f2.txt", "f3.txt"])
# for path, content in results.items():
#     print(path, len(content))
```

Explanation: ThreadPoolExecutor is good for I/O-bound tasks like disk reads.

---

24. Demonstrate how type hints can catch a bug using `mypy`.

```
# file: example_typing.py
from typing import List

def total_length(items: List[str]) -> int:
    # Suppose a bug: we accidentally try to add integers in list
    total = 0
    for item in items:
        total += len(item)
    return total

# If someone passes a list with an int, mypy would flag the call site:
# total_length(["a", "bb"])        # ok
# total_length(["a", 2])           # mypy error: List item 1 has incompatible type "int"; ex
```

How to check with mypy (run in terminal):

```
mypy example_typing.py
```

Explanation: `mypy` checks mismatched types. At runtime Python would raise `TypeError` when `len(2)` is attempted; static checking catches it earlier.

---

25. Show how the GIL prevents true parallelism in CPU-bound tasks by comparing threading with multiprocessing.

> Note: This is a demonstration snippet. It compares elapsed time for CPU-bound work using threads vs processes. On CPython you will see `Thread` version is not much faster than sequential, while `Process` version can be faster on multiple cores.

```
import time
from concurrent.futures import ThreadPoolExecutor, ProcessPoolExecutor

def cpu_task(n):
    # CPU-bound: compute sum of squares
    total = 0
    for i in range(1, n):
        total += i * i
```

```
    return total

N = 3000000  # adjust for your machine

def time_executor(executor_class, workers):
    start = time.perf_counter()
    with executor_class(max_workers=workers) as ex:
        futures = [ex.submit(cpu_task, N) for _ in range(workers)]
        results = [f.result() for f in futures]
    end = time.perf_counter()
    return end - start

print("ThreadPoolExecutor time (workers=4):", time_executor(ThreadPoolExecutor, 4))
print("ProcessPoolExecutor time (workers=4):", time_executor(ProcessPoolExecutor, 4))
```

Expected observation: On CPython the thread-based run will not scale well because of the GIL — process-based run uses separate interpreters and gains parallel CPU usage.

◈ Use-case / Practical (5)

26. Create a decorator `@cache` that caches results of a function (e.g., Fibonacci).

```
from functools import wraps
from typing import Callable, Dict, Tuple, Any

def cache(func: Callable):
    memo: Dict[Tuple[Any, ...], Any] = {}
    @wraps(func)
    def wrapper(*args):
        if args in memo:
            return memo[args]
        res = func(*args)
        memo[args] = res
        return res
    return wrapper

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

print(fib(30))  # fast due to caching
```

Explanation: Simple memoization using argument tuple as cache key.

---

27. Threaded program that simulates downloading 5 files (use `time.sleep()` + print).

```
import time
import threading
import random

def download_sim(idx):
    duration = random.uniform(0.5, 1.5)
    print(f"Start download {idx}, will take {duration:.2f}s")
    time.sleep(duration)
    print(f"Finished download {idx}")

threads = []
for i in range(1, 6):
    t = threading.Thread(target=download_sim, args=(i,))
    t.start()
    threads.append(t)

for t in threads:
    t.join()
print("All downloads complete")
```

Explanation: Simulates concurrency for I/O-bound tasks.

---

28. `ThreadPoolExecutor` to process lines in a text file concurrently (count words per line).

```
from concurrent.futures import ThreadPoolExecutor, as_completed

def count_words_line(line: str) -> int:
    return len(line.split())

def process_file_concurrently(path: str):
    with open(path, "r", encoding="utf-8") as f:
        lines = f.readlines()

    with ThreadPoolExecutor(max_workers=4) as executor:
        futures = {executor.submit(count_words_line, line): idx for idx, line in enumerate(l
        counts = {}
        for future in as_completed(futures):
            idx = futures[future]
            counts[idx] = future.result()
```

```
    # return counts in original order
    return [counts[i] for i in sorted(counts)]

# Example:
# word_counts = process_file_concurrently("bigfile.txt")
# print(word_counts[:10])
```

Explanation: Each line processed in thread pool; good for I/O or light CPU per-line.

---

29. Annotate `parse_log_line(line: str) -> Dict[str, str]`.

```
from typing import Dict

def parse_log_line(line: str) -> Dict[str, str]:
    # Dummy example: parse "LEVEL: message"
    parts = line.strip().split(":", 1)
    if len(parts) == 2:
        level, msg = parts
        return {"level": level.strip(), "message": msg.strip()}
    return {"level": "UNKNOWN", "message": line.strip()}

print(parse_log_line("ERROR: Disk full"))
```

Explanation: Annotated signature clarifies inputs/outputs; real parser would use regex.

---

30. Build a program that uses type hints, a logging decorator, and threads for 3 simulated tasks.

```
from typing import Callable
from functools import wraps
import threading
import time

def log(func: Callable):
    @wraps(func)
    def wrapper(*args, **kwargs):
        print(f"[LOG] Starting {func.__name__} with args={args}, kwargs={kwargs}")
        result = func(*args, **kwargs)
        print(f"[LOG] Finished {func.__name__}")
        return result
    return wrapper

@log
```

```python
def task(name: str, duration: float) -> None:
    print(f"{name} running for {duration}s")
    time.sleep(duration)
    print(f"{name} done")

# Launch 3 tasks concurrently
threads = [
    threading.Thread(target=task, args=("Task-A", 1.0)),
    threading.Thread(target=task, args=("Task-B", 1.5)),
    threading.Thread(target=task, args=("Task-C", 0.8)),
]

for t in threads:
    t.start()
for t in threads:
    t.join()

print("All tasks finished")
```

Explanation: Uses type hints, `@log` decorator, and `threading` to run tasks concurrently.

Additional Notes & Tips

- Type Hints: Use `typing` for Python <3.9 (e.g., `List`, `Dict`, `Optional`). From Python 3.9 onward you can use native generics (`list[int]`, `dict[str, int]`) if you prefer.
- `mypy`: To perform static checking, install `mypy` (`pip install mypy`) and run `mypy your_file.py`. It is optional but very helpful for catching type mismatch bugs before running code.
- Decorators: Always use `functools.wraps` so wrapped function metadata (name, docstring) is preserved.
- Concurrency: Use `threading`/`ThreadPoolExecutor` for I/O-bound tasks. For CPU-bound tasks prefer `multiprocessing`/`ProcessPoolExecutor`.
- Safety: When using threads and shared mutable data, synchronize access using `threading.Lock()` where needed.
