1. hw_args_kwargs.py

```
python
CopyEdit
def sum_numbers(*args):
    my_sum = 0
    for x in args:
        my_sum += x
    return my_sum

print(sum_numbers(1, 2, 3, 4))
```

- `def sum_numbers(*args):`
  Defines a function `sum_numbers` that can accept any number of positional arguments. The `*args` syntax means "pack all extra positional arguments into a tuple called `args`."
- `my_sum = 0`
  Initializes a variable to hold the running total.
- `for x in args:`
  Loop through each argument passed in.
- `my_sum += x`
  Add each argument to `my_sum`.
- `return my_sum`
  After the loop, return the total.
- `print(sum_numbers(1, 2, 3, 4))`
  Calls the function with four numbers and prints the result (`10`).

```
python
CopyEdit
@print_args
def combine_values(*args, **kwargs):
    pass  # כתוב כאן את הפתרון

combine_values(2, 4, 6, name="Tom", job="Dev")
```

- `@print_args`
  A *decorator* that wraps the function `combine_values`, likely printing its arguments before execution.
- `def combine_values(*args, **kwargs):`
  Another function that accepts any number of positional (`*args`) and keyword (`**kwargs`) arguments. `**kwargs` packs named arguments into a dictionary.
- `pass`
  Placeholder: you're supposed to implement functionality here.
- The final line calls `combine_values`, passing in three numbers and two named values. The decorator will print them—your job is to handle them inside the function (e.g., maybe combine or return a summary).

2. main.py

```
python
CopyEdit
import threading

print('current name:', threading.current_thread().name)
```

- `import threading`
  Imports Python's built-in threading module for working with threads.
- `threading.current_thread().name`
  Fetches the name of the thread currently running (by default "MainThread").
- It then prints: `current name: MainThread`.

3. thread1.py

```
python
CopyEdit
import threading

def print_numbers():
```

```
    for i in range(5000):
        print(f"Number {i}", end=' ')

def print_letters():
    for letter in "ABCDE":
        print(f"Letter {letter}")

# Creating threads
thread1 = threading.Thread(target=print_numbers)
thread2 = threading.Thread(target=print_letters)

# Starting threads
thread1.start()
thread2.start()

# Waiting for both to finish
# thread1.join()
# thread2.join()

for i in range(10):
    print('[[===main====', i**3, '=====]] ')

print("Both threads have finished execution!")
```

- Defines two functions: one prints numbers 0–4999; the other prints letters A–E.
- Creates two threads tied to each function.
- `start()` launches them concurrently.
- The `join()` lines are commented out, so the main program doesn't wait for threads to finish.
- Meanwhile, the main thread prints cube numbers of 0–9.
- Finally prints a completion message—though threads may still be running.

4. thread2.py

```
python
CopyEdit
```

```
import threading
import time

def print_numbers(n):
    for i in range(n):
        print(f"Thread {threading.current_thread().name}: {i}")
        time.sleep(0.5)

thread1 = threading.Thread(target=print_numbers, args=(5,))
thread2 = threading.Thread(target=print_numbers, args=(3,))

thread1.start()
thread2.start()
thread1.join()
thread2.join()

print("All threads completed!")
```

- Function `print_numbers` prints current thread's name and numbers up to n, sleeping half a second between prints.
- Two threads are created with different n values (5 and 3).
- `join()` ensures the main program waits until both threads finish.
- At the end, prints "All threads completed!".

5. thread3.py

```
python
CopyEdit
import threading
from concurrent.futures import ThreadPoolExecutor
import time

def task(n):
    time.sleep(1)
    print(threading.current_thread().name, f"working on {n}")
```

```
with ThreadPoolExecutor(max_workers=3) as executor:
    numbers = [1, 2, 3, 4, 5]
    executor.map(task, numbers)

print("All tasks completed!")
```

- Using a thread pool that allows up to 3 threads to run tasks in parallel.
- Defines `task(n)` that sleeps one second then prints which thread is working on which number.
- `executor.map(task, numbers)` schedules all numbers to be processed.
- Once all tasks are done, prints completion message.

6. thread4.py (Multiprocessing Example)

```
python
CopyEdit
import time
import multiprocessing

# Shared global for demonstration
x = 1

def count_range(start, end, name, return_dict=None):
    counter = 0
    global x
    x = x + 1
    for i in range(start, end):
        counter += 1
    print(f"{name} finished.\nFinal count: {counter}")
    if return_dict is not None:
        return_dict[name] = counter

# Functions to compare single vs. parallel processes...
```

- Uses `multiprocessing` to create separate processes (not threads).

- `count_range()` counts numbers, updates shared dictionary if given.
- Measures and compares execution time for counting half a billion numbers in one process vs two processes.
- Prints performance improvement, and explains why multiprocessing can outperform threading in CPU-bound tasks (GIL bypass).

7. thread5.py (Threading CPU test)

```
python
CopyEdit
import threading
import time

def count_range(start, end, name):
    counter = 0
    for i in range(start, end):
        counter += 1
    print(f"{name} finished.\nFinal count: {counter}")

# Similar single- vs two-thread comparisons...
```

- Similar to `thread4`, but using threads not processes.
- Measures time and finds that threads do not speed up CPU-bound work in Python due to the Global Interpreter Lock (GIL).

8. thread6.py (Using Locks)

```
python
CopyEdit
import threading
import time

counter = 0
lock = threading.Lock()
```

```
def increment():
    global counter
    with lock:
        for _ in range(10):
            print(threading.current_thread().name, counter)
            counter += 1

threads = []
for _ in range(5):
    t = threading.Thread(target=increment)
    threads.append(t)
    t.start()

for t in threads:
    t.join()

print(f"Final counter value: {counter}")
```

- Shows shared variable `counter` updated by multiple threads.
- A Lock ensures that only one thread can enter the `with lock:` block at a time, avoiding race conditions.
- Each of 5 threads increments `counter` 10 times safely.

9. thread7.py (Web Scraping with Threads)

```
python
CopyEdit
import threading
import requests
from bs4 import BeautifulSoup
from urllib.parse import urljoin

urls = [ ... list of five websites ... ]

def fetch_content(url):
```

```
    try:
        response = requests.get(url, timeout=5)
        response.raise_for_status()
    except requests.RequestException as e:
        print(f"Error fetching {url}: {e}")
        return

    soup = BeautifulSoup(response.text, 'html.parser')
    title = soup.title.string.strip() if soup.title else "No title
found"
    meta_desc = soup.find("meta", attrs={"name": "description"})
    description = meta_desc["content"].strip() if meta_desc else "No
description found"
    links = [urljoin(url, a["href"]) for a in soup.find_all("a",
href=True)][:5]

    print(f"\n {url}")
    print(f"Title: {title}")
    print(f"Description: {description}")
    print("Top 5 Links:")
    for link in links:
        print(f" - {link}")

threads = []
for url in urls:
    t = threading.Thread(target=fetch_content, args=(url,))
    threads.append(t)
    t.start()

for t in threads:
    t.join()

print("\n✅ Finished scraping all websites!")
```

- Imports modules for HTTP requests and parsing HTML.
- `fetch_content` downloads each URL, checks for errors, parses title, meta description, and top 5 links.
- Creates a thread per URL to scrape concurrently.

- Waits for them all to finish, then prints a final message.

🧠 Final Summary

- **`*args`** and **`**kwargs`** let you pass flexible numbers of arguments into functions.
- **`threading.Thread`** lets you run tasks concurrently within the same process.
- **`.start()`** begins a thread; **`.join()`** waits for it to finish.
- Python's **GIL** means CPU-heavy work doesn't speed up with threads—use **multiprocessing** instead.
- Use **Locks** to prevent race conditions when accessing shared data.
- Threads can be used effectively for IO-bound tasks (like web scraping).