Question 38: How would you sort a list of dictionaries by a specific key?

You can sort a list of dictionaries using the `sort()` method or the built-in `sorted()` function. The `key` parameter accepts a function that extracts the comparison key from each dictionary.

```
test_list = [
    {"name": "erico"},
    {"name": "antonio"}
]
key = "name"

def sort_list_by_dict_key(test_list: list[dict], key):
    test_list.sort(key=lambda x: x[key])
    return test_list

print(sort_list_by_dict_key(test_list, key))
```

Question 40: How would you implement a retry mechanism for a function that might fail?

A retry mechanism is used to attempt a function multiple times before failing. It's useful when dealing with unreliable operations like network requests. You can implement it using a decorator that wraps the target function.

```
import time

def retry_decorator(max_retries, delay):
    def decorator(func):
        def wrapper(*args, **kwargs):
            for attempt in range(max_retries):
                try:
                    return func(*args, **kwargs)
                except Exception as err:
                    if attempt == max_retries - 1:
                        raise err
                    time.sleep(delay)
        return wrapper
    return decorator

@retry_decorator(max_retries=3, delay=1)
def fail_function():
    return 3 / 0

print(fail_function())
```

Question 41: How would you implement a simple rate limiter for a function in Python?

A rate limiter restricts how often a function can be called. This can be implemented using a decorator that tracks the time between calls and enforces a wait if needed.

```
import time

def rate_limiter(calls_per_second):
    interval = 1 / calls_per_second
```

```
    last_call_time = [0]
    def decorator(func):
        def wrapper(*args, **kwargs):
            elapsed_time = time.time() - last_call_time[0]
            if elapsed_time < interval:
                print("Rate limit exceeded")
                time.sleep(interval - elapsed_time)
            last_call_time[0] = time.time()
            return func(*args, **kwargs)
        return wrapper
    return decorator
```

Question 42: How would you flatten a nested list in Python?

Flattening a nested list involves converting a list that contains other lists into a single-level list. This can be done recursively by checking each item and extending the result if it is a list.

```
test_list = [
    1,
    [1, 2, [1, 2]],
    [1, 2, 3]
]

def flatten_list(test_list):
    flattened_list = []
    for item in test_list:
        if isinstance(item, list):
            flattened_list.extend(flatten_list(item))
        else:
            flattened_list.append(item)
    return flattened_list

print(flatten_list(test_list))
```