Python Coding Interview Questions

E1: String Formatter

Create a Python program that consists of two functions: `format_string` and `print_formatted`. The `format_string` function should take a string and a boolean flag, capitalizing every letter if the flag is True, or making every letter lowercase if the flag is False. The `print_formatted` function should take a list of strings and a boolean flag, call `format_string` on each string with the given flag, and print each formatted string on a new line.

Examples:

1. **Input:**

   ```
   format_string("Hello World", True)
   ```

   Output:

   ```
   "HELLO WORLD"
   ```

   Explanation: The flag is True, so all letters are capitalized.

2. **Input:**

   ```
   format_string("Hello World", False)
   ```

   Output:

   ```
   "hello world"
   ```

   Explanation: The flag is False, so all letters are made lowercase.

3. **Input:**

   ```
   format_string("123!@#$%^&*()_+", True)
   ```

   Output:

   ```
   "123!@#$%^&*()_+"
   ```

   Explanation: Non-alphabetic characters remain unchanged.

4. **Input:**

   ```
   strings = ["Hello", "World", "Python"]
   print_formatted(strings, True)
   ```

   Output:

   ```
   HELLO
   WORLD
   PYTHON
   ```

   Explanation: Each string in the list is capitalized and printed on a new line.

5. **Input:**

   ```
   strings = ["Hello", "World", "Python"]
   print_formatted(strings, False)
   ```

   Output:

   ```
   hello
   world
   python
   ```

   Explanation: Each string in the list is made lowercase and printed on a new line.

6. **Input:**

   ```
   strings = ["", "ABC", "123"]
   print_formatted(strings, True)
   ```

   Output:

   ```

   ABC
   123
   ```

   Explanation: Empty strings result in blank lines, while other strings are capitalized.

7. **Input:**

   ```
   strings = []
   print_formatted(strings, True)
   ```

   Output: (no output) **Explanation:** An empty list results in no output.

Note: For the `print_formatted` function, the actual output will be printed to the console. The test should verify that the correct strings are printed in the correct format.

E2: Temperature Converter

Create a Python program with three functions: `celsius_to_fahrenheit`, `fahrenheit_to_celsius`, and `convert_temperatures`. The first two functions should convert temperatures between Celsius and Fahrenheit, rounding results to 2 decimal places. The `convert_temperatures` function should take a list of temperatures and a conversion direction ('CtoF' or 'FtoC'), apply the appropriate conversion to each temperature, and return a new list of converted values.

Examples:

1. **Input:**

   ```
   celsius_to_fahrenheit(0)
   ```

Output:

```
32.00
```

Explanation: 0°C is equivalent to 32°F.

2. **Input:**

   ```
   fahrenheit_to_celsius(32)
   ```

   Output:

   ```
   0.00
   ```

   Explanation: 32°F is equivalent to 0°C.

3. **Input:**

   ```
   celsius_to_fahrenheit(-40)
   ```

   Output:

   ```
   -40.00
   ```

   Explanation: -40°C is the point where Celsius and Fahrenheit scales intersect.

4. **Input:**

   ```
   fahrenheit_to_celsius(98.6)
   ```

   Output:

   ```
   37.00
   ```

   Explanation: 98.6°F (normal body temperature) is equivalent to 37°C.

5. **Input:**

   ```
   temperatures = [0, 100, -40, 37]
   convert_temperatures(temperatures, 'CtoF')
   ```

   Output:

   ```
   [32.00, 212.00, -40.00, 98.60]
   ```

   Explanation: Each Celsius temperature in the list is converted to Fahrenheit.

6. **Input:**

   ```
   temperatures = [32, 212, -40, 98.6]
   convert_temperatures(temperatures, 'FtoC')
   ```

   Output:

   ```
   [0.00, 100.00, -40.00, 37.00]
   ```

Explanation: Each Fahrenheit temperature in the list is converted to Celsius.

7. **Input:**

   ```
   temperatures = []
   convert_temperatures(temperatures, 'FtoC')
   ```

   Output:

   ```
   []
   ```

   Explanation: An empty input list results in an empty output list.

Note: All temperature values in the output should be rounded to 2 decimal places.

E3: Fibonacci Sequence

Write a function that returns the nth number in the Fibonacci sequence. The Fibonacci sequence is defined as follows: the first two numbers are 0 and 1, and each subsequent number is the sum of the two preceding ones.

Examples:

1. **Input:**

   ```
   fibonacci(0)
   ```

   Output:

   ```
   0
   ```

   Explanation: The 0th number in the Fibonacci sequence is 0.

2. **Input:**

   ```
   fibonacci(1)
   ```

   Output:

   ```
   1
   ```

   Explanation: The 1st number in the Fibonacci sequence is 1.

3. **Input:**

   ```
   fibonacci(2)
   ```

   Output:

   ```
   1
   ```

   Explanation: The 2nd number in the Fibonacci sequence is 1 $(0 + 1 = 1)$.

4. **Input:**

```
fibonacci(5)
```

Output:

```
5
```

Explanation: The 5th number in the Fibonacci sequence is 5 (0, 1, 1, 2, 3, 5).

5. **Input:**

   ```
   fibonacci(10)
   ```

   Output:

   ```
   55
   ```

   Explanation: The 10th number in the Fibonacci sequence is 55 (0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55).

Note: The function should handle inputs of n >= 0. For large values of n, consider the potential for integer overflow and optimize your solution accordingly.

E4: Prime Number Check

Write a function that checks whether a given number is prime. A prime number is a natural number greater than 1 that is only divisible by 1 and itself.

Examples:

1. **Input:**

   ```
   is_prime(2)
   ```

   Output:

   ```
   True
   ```

   Explanation: 2 is the smallest prime number.

2. **Input:**

   ```
   is_prime(3)
   ```

   Output:

   ```
   True
   ```

   Explanation: 3 is a prime number as it's only divisible by 1 and 3.

3. **Input:**

   ```
   is_prime(4)
   ```

   Output:

   ```
   False
   ```

Explanation: 4 is not a prime number as it's divisible by 2.

4. **Input:**

   ```
   is_prime(29)
   ```

   Output:

   ```
   True
   ```

   Explanation: 29 is a prime number.

5. **Input:**

   ```
   is_prime(100)
   ```

   Output:

   ```
   False
   ```

   Explanation: 100 is not a prime number as it's divisible by 2, 4, 5, 10, 20, 25, 50.

Note: The function should handle inputs of n > 1. For large values of n, consider optimizing your solution for efficiency.

M1: Custom Sort Function

Write a Python program that includes a function `custom_sort` which sorts a list of tuples based on a given index of the tuple. The function should take a list of tuples and an integer representing the index to sort by. Additionally, create a function `sort_and_print` that takes the same arguments, calls `custom_sort`, and then prints each tuple on a new line. Include error handling to manage cases where the index is out of bounds for some tuples.

Test cases for `custom_sort` function:

- **Input:** Sorting by first index

  ```
  data = [(3, 'a'), (1, 'b'), (2, 'c')]
  custom_sort(data, 0)
  ```

  Expected Output:

  ```
  [(1, 'b'), (2, 'c'), (3, 'a')]
  ```

- **Input:** Sorting by last index

  ```
  data = [(1, 'c'), (2, 'a'), (3, 'b')]
  custom_sort(data, -1)
  ```

  Expected Output:

  ```
  [(2, 'a'), (3, 'b'), (1, 'c')]
  ```

- **Input:** Handling out-of-bounds index

```
data = [(1, 'a'), (2, 'b')]
custom_sort(data, 2)
```

Expected Output: Should raise an IndexError

- **Input:** Sorting empty list

  ```
  data = []
  custom_sort(data, 0)
  ```

  Expected Output:

  ```
  []
  ```

- **Input:** Sorting identical tuples

  ```
  data = [(1, 'a'), (1, 'a'), (1, 'a')]
  custom_sort(data, 0)
  ```

  Expected Output:

  ```
  [(1, 'a'), (1, 'a'), (1, 'a')]
  ```

Test cases for `sort_and_print` function:

- **Input:**

  ```
  data = [(3, 'x'), (1, 'y'), (2, 'z')]
  sort_and_print(data, 0)
  ```

  Expected Output:

  ```
  (1, 'y')
  (2, 'z')
  (3, 'x')
  ```

- **Input:**

  ```
  data = [(1, 'c'), (2, 'a'), (3, 'b')]
  sort_and_print(data, -1)
  ```

  Expected Output:

  ```
  (2, 'a')
  (3, 'b')
  (1, 'c')
  ```

- **Input:**

  ```
  data = [(1, 'a'), (2, 'b')]
  sort_and_print(data, 2)
  ```

  Expected Output: Should print an error message about index out of bounds

- **Input:**

```
data = []
sort_and_print(data, 0)
```

Expected Output: (no output, as the list is empty)

Note: For the `sort_and_print` function, the actual output will be printed to the console. The test should verify that the correct tuples are printed in the correct order.

M2: Log Parser

Write a Python program that parses a log file and summarizes the number of different log levels (e.g., INFO, ERROR, WARNING) present in the file. The program should include a function `parse_log` that takes the path to a log file and returns a dictionary with log levels as keys and their counts as values. Additionally, create a function `print_log_summary` that takes the dictionary returned by `parse_log` and prints a summary in the format: "LEVEL: count". The program should handle cases where the log file might not exist or be accessible.

Test cases for `parse_log` function:

- **Input:** Log file with mixed log levels

  ```
  # Assume log.txt contains:
  # INFO: System started
  # WARNING: Low memory
  # ERROR: Connection failed
  # INFO: Task completed
  parse_log('log.txt')
  ```

  Expected Output:

  ```
  {'INFO': 2, 'WARNING': 1, 'ERROR': 1}
  ```

- **Input:** Log file with unusual log levels

  ```
  # Assume unusual_log.txt contains:
  # DEBUG: Debugging information
  # CRITICAL: System crash
  # INFO: Normal operation
  # UNKNOWN: Unrecognized message
  parse_log('unusual_log.txt')
  ```

  Expected Output:

  ```
  {'DEBUG': 1, 'CRITICAL': 1, 'INFO': 1, 'UNKNOWN': 1}
  ```

- **Input:** Non-existent log file

  ```
  parse_log('non_existent.txt')
  ```

Expected Output: Should raise a FileNotFoundError

- **Input:** Empty log file

  ```
  # Assume empty_log.txt is an empty file
  parse_log('empty_log.txt')
  ```

  Expected Output:

  ```
  {}
  ```

Test cases for `print_log_summary` function:

- **Input:**

  ```
  log_summary = {'INFO': 3, 'WARNING': 2, 'ERROR': 1}
  print_log_summary(log_summary)
  ```

  Expected Output:

  ```
  INFO: 3
  WARNING: 2
  ERROR: 1
  ```

- **Input:**

  ```
  log_summary = {}
  print_log_summary(log_summary)
  ```

  Expected Output: (no output, as the dictionary is empty)

Note: For the `print_log_summary` function, the actual output will be printed to the console. The test should verify that the correct log levels and counts are printed in the correct format.

H1: Multi-threaded File Downloader

Write a Python program that downloads multiple files concurrently from given URLs. The program should include a function `download_file` that takes a URL and a destination path and downloads the file to the path. Then, create a function `download_files_concurrently` that takes a list of URLs and destination paths, and uses multi-threading to download all the files at the same time. Include error handling for cases such as invalid URLs or network issues, and ensure that the program can handle a large number of files without crashing.

Test cases:

1. **Single file download: Input:**

   ```
   url = "https://example.com/file.txt"
   destination = "/path/to/download/file.txt"
   download_file(url, destination)
   ```

Expected Output: File should be downloaded to the specified destination

2. **Multiple file downloads: Input:**

   ```
   urls = ["https://example.com/file1.txt", "https://example.com/file2.txt", "https://exam
   destinations = ["/path/to/download/file1.txt", "/path/to/download/file2.txt", "/path/to
   download_files_concurrently(urls, destinations)
   ```

   Expected Output: All files should be downloaded to their respective destinations

3. **Invalid URL: Input:**

   ```
   url = "https://invalid-url.com/nonexistent.txt"
   destination = "/path/to/download/nonexistent.txt"
   download_file(url, destination)
   ```

   Expected Output: Should raise an appropriate exception (e.g., URLError)

4. **Insufficient permissions: Input:**

   ```
   url = "https://example.com/file.txt"
   destination = "/root/restricted_folder/file.txt"  # Assuming no write permission
   download_file(url, destination)
   ```

   Expected Output: Should raise a Perm

issionError

5. **Mixed valid and invalid URLs: Input:**

   ```
   urls = ["https://example.com/file1.txt", "https://invalid-url.com/nonexistent.txt", "ht
   destinations = ["/path/to/download/file1.txt", "/path/to/download/nonexistent.txt", "/p
   download_files_concurrently(urls, destinations)
   ```

   Expected Output: Valid files should be downloaded, while invalid URLs should be handled gracefully (e.g., logging errors without crashing the program)

Note: These test cases assume the existence of `download_file` and `download_files_concurrently` functions. The actual implementation should handle exceptions, manage threads, and ensure proper file I/O operations.