Python Developer 5+ years Exp. Questions

1. Question: Write a function to reverse a string.

Answer:

```
def reverse_string(s):
    return s[::-1]

# Example usage
print(reverse_string("Hello"))
```

Output: "olleH"

2. Question: How would you merge two dictionaries in Python?

Answer:

```
def merge_dicts(dict1, dict2):
    merged_dict = {**dict1, **dict2}
    return merged_dict

# Example usage
dict1 = {'a': 1, 'b': 2}
dict2 = {'b': 3, 'c': 4}
print(merge_dicts(dict1, dict2))
```

Output: {'a': 1, 'b': 3, 'c': 4}

3. Question: Write a function to find the longest common prefix string amongst an array of strings.

Answer:

```
def longest_common_prefix(strs):
    if not strs:
        return ""

    shortest_str = min(strs, key=len)

    for i, char in enumerate(shortest_str):
        for other in strs:
            if other[i] != char:
                return shortest_str[:i]

    return shortest_str

# Example usage
strs = ["flower", "flow", "flight"]
print(longest_common_prefix(strs))
```

Output: "fl"

4. Question: Explain the difference between deepcopy and shallow copy in Python.

Answer:

A shallow copy creates a new object but inserts references into it to the objects found in the original. In contrast, a deep copy creates a new object and recursively copies all objects found in the original.

Example:

```
import copy

original_list = [[1, 2, 3], [4, 5, 6]]

shallow_copy = copy.copy(original_list)
deep_copy = copy.deepcopy(original_list)

original_list[0][0] = 'X'
```

```
print(shallow_copy)  # Output: [['X', 2, 3], [4, 5, 6]]
print(deep_copy)     # Output: [[1, 2, 3], [4, 5, 6]]
```

Output:

[['X', 2, 3], [4, 5, 6]]

[[1, 2, 3], [4, 5, 6]]

5. Question: Write a generator function to yield the Fibonacci sequence.

Answer:

```
def fibonacci_sequence(n):
    a, b = 0, 1
    for _ in range(n):
        yield a
        a, b = b, a + b

# Example usage
for num in fibonacci_sequence(10):
    print(num)
```

6. Question: Write a function to find the second largest number in a list.

Answer:

```
def second_largest(numbers):
    first, second = float('-inf'), float('-inf')
    for num in numbers:
        if num > first:
            first, second = num, first
        elif first > num > second:
            second = num
    return second

# Example usage
numbers = [10, 20, 4, 45, 99]
```

```
print(second_largest(numbers))
```
Output: 45

7. Question: Write a function to check if a string is a valid palindrome.

Answer:

```
def is_palindrome(s):
    s = ''.join(c.lower() for c in s if c.isalnum())
    return s == s[::-1]
# Example usage
print(is_palindrome("A man, a plan, a canal: Panama"))
# Output: True
print(is_palindrome("race a car"))
# Output: False
```

8. Question: Explain the concept of Python decorators and provide an example.

Answer:

A decorator in Python is a function that modifies the behavior of another function. Decorators allow us to wrap another function to extend its behavior without permanently modifying it.

Example:

```
def my_decorator(func):
    def wrapper():
        print("Something is happening before the function is called.")
        func()
        print("Something is happening after the function is called.")
    return wrapper

@my_decorator
def say_hello():
```

```
    print("Hello!")

# Example usage
say_hello()
```

Output:
Something is happening before the function is called.
Hello!
Something is happening after the function is called.

9. Question: Write a function to flatten a nested list.

Answer:

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

# Example usage
nested_list = [1, [2, [3, 4], 5], 6, [7, 8]]
print(flatten_list(nested_list))
```

Output: [1, 2, 3, 4, 5, 6, 7, 8]

10. Question: Explain the GIL (Global Interpreter Lock) in Python.

Answer:

The Global Interpreter Lock (GIL) is a mutex that protects access to Python objects, preventing multiple threads from executing Python bytecodes at once. This lock is necessary because Python's memory management is not thread-safe. While it simplifies the implementation of CPython and avoids issues with concurrent memory management, it also means that Python threads are not truly concurrent, which can be a bottleneck in CPU-bound and multi-threaded code.