Python Interview Study Guide - Day Two

22. Purpose of __call__ Method:

The '__call__' method allows an instance of a class to be called like a regular function.

It can be useful for scenarios where you want an object to have a callable behavior while maintaining state.

Example:

```
class Adder:
    def __init__(self, value):
        self.value = value

    def __call__(self, x):
        return self.value + x

add_five = Adder(5)
print(add_five(10))  # Output: 15
print(add_five(20))  # Output: 25
```

23. Purpose of __slots__:

The '__slots__' attribute limits the attributes an object can have and prevents the creation of a dynamic __dict__.

This saves memory, especially in large numbers of instances.

Example:

```
class Person:
    __slots__ = ['name', 'age']
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

24. Difference between iter() and next():

- iter(): Converts an iterable (like a list) into an iterator.
- next(): Retrieves the next item from an iterator.

Example:

```
my_list = [1, 2, 3]
my_iterator = iter(my_list)
print(next(my_iterator))  # 1
print(next(my_iterator))  # 2
print(next(my_iterator))  # 3

try:
    print(next(my_iterator))
except StopIteration:
    print("End of iteration")
```

31. threading vs multiprocessing:
- threading: Runs multiple threads on the same core (due to GIL). Good for I/O-bound tasks.
- multiprocessing: Runs separate processes on different cores. Suitable for CPU-bound tasks.

Example:

```
import threading
import multiprocessing

def print_numbers():
    for i in range(5):
        print(i)

# Threading
t = threading.Thread(target=print_numbers)
t.start()

# Multiprocessing
p = multiprocessing.Process(target=print_numbers)
```

```
p.start()
```

34. Merging Two Dictionaries:

You can merge using dictionary unpacking:

```
dict_1 = {"a": 1, "b": 2}
dict_2 = {"b": 3, "c": 4}
merged_dict = {**dict_1, **dict_2}
print(merged_dict)  # Output: {'a': 1, 'b': 3, 'c': 4}
```

35. Removing Duplicates While Preserving Order:

You can use a set to track seen elements while iterating through the list.

```
def unique_list(test_list):
    seen = set()
    result = []
    for n in test_list:
        if n not in seen:
            seen.add(n)
            result.append(n)
    return result

print(unique_list([1, 1, 4, 3, 5, 6, 7]))
```

37. Finding Intersection of Two Lists:

Using set lookup for efficiency:

```
def find_intersection(list_1, list_2):
    set_2 = set(list_2)
    return [item for item in list_1 if item in set_2]

list_1 = [1, 2, 3, 4]
list_2 = [3, 4, 5, 6]
```

```
print(find_intersection(list_1, list_2))  # Output: [3, 4]
```