Python Tasks and Interview Concepts

1. List Comprehension: Creates a new list by applying an expression to each item in an iterable.

```
squares = [x * x for x in range(10)]
# print(squares) # Output: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

2. Dictionary Comprehension: Creates a dictionary from an iterable with key:value mapping.

```
even_squares = {x: x*x for x in range(10) if x % 2 == 0}
# print(even_squares) # Output: {0: 0, 2: 4, 4: 16, 6: 36, 8: 64}
```

3. Set Comprehension: Quickly finds unique elements from an iterable.

```
unique_letters = {char.upper() for word in ["apple", "banana"] for char in word}
# print(unique_letters) # Output: {'P', 'L', 'E', 'B', 'A', 'N'}
```

4. String Reversal: Reversing a string using slicing (the most Pythonic way).

```
my_string = "hello world"
reversed_string = my_string[::-1]
# print(reversed_string) # Output: dlrow olleh
```

5. Palindrome Check: Checking if a string is the same forward and backward.

```
is_palindrome = "madam" == "madam"[::-1]
# print(is_palindrome) # Output: True
```

6. Function with Type Hints: Defining a function with default arguments and explicit types.

```
def calculate_area(length: float, width: float = 1.0) -> float:
    """Calculates the area of a rectangle."""
    return length * width
```

7. Lambda Function: A small anonymous function used for quick, simple operations.

```
add_ten = lambda x: x + 10
# print(add_ten(5)) # Output: 15
```

8. Map Function: Applies a function (often a lambda) to all items in an input list.

```
numbers = [1, 2, 3, 4]
doubled_numbers = list(map(lambda x: x * 2, numbers))
# print(doubled_numbers) # Output: [2, 4, 6, 8]
```

9. Filter Function: Constructs a list from elements for which a function returns true.

```
even_numbers = list(filter(lambda x: x % 2 == 0, numbers))
# print(even_numbers) # Output: [2, 4]
```

10. Find Most Frequent Element: Using collections.Counter for frequency analysis.

```
from collections import Counter
data = [1, 2, 2, 3, 4, 2, 1]
most_common_element = Counter(data).most_common(1)[0][0]
# print(most_common_element) # Output: 2
```

11. Find Unique Elements: Using the set type conversion for fast uniqueness check.

```
unique_list = list(set(data))
```

```
# print(unique_list) # Output: [1, 2, 3, 4] (Order may vary)
```

12. OOP Class Definition: Defining a class with an initializer (__init__) and a method.

```
class Car:
    def __init__(self, make, model):
        self.make = make
        self.model = model

    def get_full_name(self):
        return f"{self.make} {self.model}"
```

13. OOP Object Instantiation: Creating an instance of the class.

```
my_car = Car("Toyota", "Camry")
# print(my_car.get_full_name()) # Output: Toyota Camry
```

14. Exception Handling: Using try-except-finally for robust code execution.

```
def safe_divide(a, b):
    try:
        result = a / b
    except ZeroDivisionError:
        print("Error: Cannot divide by zero.")
        result = None
    except Exception as e:
        print(f"An unexpected error occurred: {e}")
        result = None
    finally:
        print("Execution finished.")
    return result
```

15. Pandas DataFrame Creation: Creating a simple DataFrame (data structure in Pandas).

```
import pandas as pd
data_frame = pd.DataFrame({
  'Name': ['Alice', 'Bob', 'Charlie'],
  'Age': [25, 30, 35],
  'City': ['NY', 'LA', 'NY']
})
```

16. Pandas Column Selection: Accessing a single column (Series) by name.

```
ages = data_frame['Age']
```

17. Pandas Filtering Rows: Selecting rows based on a boolean condition.

```
ny_residents = data_frame[data_frame['City'] == 'NY']
# print(ny_residents)
```

18. Pandas Grouping and Aggregation: Calculating the mean of 'Age' per 'City'.

```
age_by_city = data_frame.groupby('City')['Age'].mean()
# print(age_by_city)
```

19. Merging Dictionaries (Python 3.9+): Combining dictionaries using the union operator (|).

```
dict_a = {'a': 1, 'b': 2}
dict_b = {'c': 3, 'd': 4}
merged_dict = dict_a | dict_b
# print(merged_dict) # Output: {'a': 1, 'b': 2, 'c': 3, 'd': 4}
```

20. Generator Function: Using 'yield' instead of 'return' to create an iterator for memory efficiency.

```
def count_up_to(max_val):
    i = 0
    while i <= max_val:
        yield i
        i += 1
# for num in count_up_to(3):
#     print(num) # Prints 0, 1, 2, 3 lazily
```

21. F-Strings: Formatted String Literals (the modern way to embed expressions in strings).

```
name = "Gemini"
age = 1
greeting = f"Hello, my name is {name} and I am {age} year{'s' if age != 1 else ''} old."
# print(greeting) # Output: Hello, my name is Gemini and I am 1 year old.
```

22. Sorting with Custom Key: Sorting a list of tuples/objects using a lambda function as the key.

```
items = [('apple', 3), ('banana', 1), ('cherry', 2)]
sorted_by_count = sorted(items, key=lambda x: x[1])
# print(sorted_by_count) # Output: [('banana', 1), ('cherry', 2), ('apple', 3)]
```

23. Using Enumerate: Iterating through an iterable while keeping track of the index.

```
colors = ['red', 'green', 'blue']
# for index, color in enumerate(colors):
```

```
#     print(f"Item {index}: {color}")
```

24. Dictionary get() Method: Safely accessing dictionary values with a default fallback.

```
user_data = {'name': 'Alice', 'role': 'Admin'}
phone = user_data.get('phone', 'N/A')
# print(phone) # Output: N/A (prevents KeyError)
```

25. Tuple Unpacking: Assigning elements of a tuple (or list) to multiple variables in one line.

```
coordinates = (10, 20, 30)
x, y, z = coordinates
# print(x, y, z) # Output: 10 20 30
```

26. Simple Function Decorator: A function that takes another function and extends its behavior.

```
def simple_decorator(func):
    def wrapper():
        print("Before function execution.")
        func()
        print("After function execution.")
    return wrapper

@simple_decorator
def say_hello():
    print("Hello!")
# say_hello() # Outputs: Before..., Hello!, After...
```

27. Static Method in OOP: A method that belongs to the class but doesn't need access to instance data.

```
class Utility:
    @staticmethod
    def is_valid_age(age):
        return 0 < age < 150
# print(Utility.is_valid_age(30)) # Output: True
```

28. Closure (Nested Function): A nested function that remembers and accesses variables from its enclosing scope.

```
def outer_function(msg):
    # 'msg' is the free variable
    def inner_function():
        print(msg) # The inner function "closes over" the 'msg' variable
    return inner_function

greet = outer_function("Hi there!")
# greet() # Output: Hi there!
```