Python Assignment Questions: Functions, Arguments, Lambdas, Closures & Decorators

Basic Function Definition and Calling (1-10)

1. Write a function `greet(name)` that returns "Hello, [name]!" and call it with your name.
2. Create a function `area_rectangle(length, width)` that calculates and returns the area of a rectangle.
3. Write a function `is_even(num)` that returns `True` if a number is even and `False` otherwise.
4. Define a function `maximum_of_three(a, b, c)` that returns the largest of three numbers.
5. Create a function `convert_temperature(temp, unit)` that converts temperature between Celsius and Fahrenheit. If `unit` is 'C', convert from Celsius to Fahrenheit; if 'F', convert from Fahrenheit to Celsius.
6. Write a function `calculate_bmi(weight, height)` that calculates Body Mass Index (weight in kg, height in meters).
7. Create a function `count_vowels(text)` that returns the number of vowels in a string.
8. Define a function `reverse_string(text)` that returns the reverse of an input string.
9. Write a function `fibonacci(n)` that returns the nth Fibonacci number (starting with 0 and 1).
10. Create a function `is_palindrome(text)` that checks if a string is a palindrome (reads the same forwards and backwards).

Function Arguments (11-20)

11. Modify the `greet` function to include a default greeting: `greet(name, greeting="Hello")` that returns "[greeting], [name]!"
12. Create a function `calculate_total(prices, tax_rate=0.1)` that calculates the total cost including tax for a list of prices.
13. Write a function `create_profile(name, age, city="Unknown", **additional_info)` that returns a dictionary with all the provided information.
14. Define a function `print_args_kwargs(*args, **kwargs)` that prints all positional and keyword arguments it receives.
15. Create a function `filter_positive_even(numbers)` that returns a list of positive even numbers from the input list.
16. Write a function `combine_lists(*lists)` that takes any number of lists and returns a single combined list.
17. Define a function `calculate_discount(price, discount_percent=10, max_discount=None)` that applies a discount but doesn't exceed max_discount if provided.

18. Create a function `format_name(first, last, middle="")` that formats a full name, handling the case when middle name is not provided.
19. Write a function `search_by_attributes(items, /, **attributes)` that finds all items that match all the given attributes. Use the `/` syntax to require positional-only for the first parameter.
20. Define a function `run_command(command, *, timeout=30, capture_output=True)` that simulates running a command with keyword-only arguments for options.

Lambda Functions (21-25)

21. Use a lambda function to create a sorter for a list of tuples based on the second element.
22. Write a lambda function that checks if a number is in a specific range (e.g., between 10 and 20).
23. Create a dictionary of mathematical operations where each value is a lambda function (add, subtract, multiply, divide).
24. Use `filter()` and a lambda function to extract all strings that start with a vowel from a list of strings.
25. Implement a simple calculator program that uses lambda functions to perform operations based on user input.

Closures and Function Factories (26-30)

26. Create a counter function that returns a function which increments and returns a counter variable each time it's called.
27. Write a function `create_multiplier(factor)` that returns a function that multiplies its argument by `factor`.
28. Implement a function `create_power_function(exponent)` that returns a function which raises its argument to the given exponent.
29. Create a function `create_greeting(greeting)` that returns a function that greets a person with the specified greeting.
30. Write a function `create_sequence_generator(sequence_type)` that returns different sequence generators (fibonacci, arithmetic, geometric) based on the input.

Basic Decorators (31-38)

31. Create a decorator `@timer` that measures and prints the execution time of a function.
32. Implement a decorator `@debug` that prints the function name, arguments, and return value when a function is called.
33. Write a decorator `@retry(n)` that retries a function up to n times if it raises an exception.
34. Create a decorator `@memoize` that caches the results of a function call based on its arguments.
35. Implement a decorator `@validate_types` that checks if the arguments passed to a function match the type hints in the function signature.

36. Write a decorator `@rate_limit(per_second)` that limits how many times a function can be called per second.
37. Create a decorator `@singleton` that ensures a class has only one instance.
38. Implement a decorator `@deprecated` that issues a warning when a deprecated function is used.

Advanced Decorators and Class Decorators (39-45)

39. Create stacked decorators for logging, timing, and memoizing a recursive Fibonacci function.
40. Implement a decorator that registers functions in a central registry, which can be used to build a plugin system.
41. Write a decorator `@validate_json` that validates JSON input for a function that processes JSON data.
42. Create a decorator that transforms the return value of a function (e.g., converting a dictionary to a custom object).
43. Implement a class decorator that adds serialization methods (`to_json`, `to_xml`) to the decorated class.
44. Write a decorator that converts a synchronous function to return a `Future` object (a simple implementation, not using actual threading).
45. Create a decorator that makes a function's arguments and return values conform to a specific contract (e.g., non-negative numbers, strings of a certain length).