Q1. Is an assignment operator like += only for show? Is it possible that it would lead to faster results at the runtime?

A1. The **+=** operator is not just for show, it can actually lead to faster results at runtime in some cases. The reason for this is that **+=** can modify the object in place, whereas concatenation with the **+** operator creates a new object each time. For example, if we have a large list and we want to append a value to it multiple times, it is more efficient to use **list.append()** or **+=** than to use concatenation with the **+** operator.

However, in some cases, concatenation with the **+** operator may be more efficient, especially if we are concatenating small strings. The best approach depends on the specific use case and the size and type of data we are working with. It is recommended to benchmark different approaches to determine which one is the most efficient for our specific use case.

Q2. What is the smallest number of statements you'd have to write in most programming languages to replace the Python expression a, b = a + b, a?

A2. In most programming languages, we would need three statements to replace the Python expression a, b = a + b, a:

1. Create a temporary variable to hold the value of a before the assignment: **temp = a**.
2. Update the value of a by adding b: **a = a + b**.
3. Update the value of b to be the original value of a (stored in temp): **b = temp**.

Q3. In Python, what is the most effective way to set a list of 100 integers to 0?

A3. The most effective way to set a list of 100 integers to 0 is to use the following code:

my\_list = [0] \* 100

This will create a list of 100 integers, with each element initialized to 0. This is a concise and efficient way to initialize a list with a large number of elements to a specific value. The **\*** operator is used to repeat the value **0** a total of 100 times to create the list.

Q4. What is the most effective way to initialise a list of 99 integers that repeats the sequence 1, 2, 3? If necessary, show step-by-step instructions on how to accomplish this.

A4. The way to initialize a list of 99 integers repeating the sequence 1, 2, 3 is to use a list comprehension:

my\_list = [i % 3 + 1 for i in range(99)]

This creates a list of 99 integers, where each integer is the result of the modulo operation on the index of the list (i.e., **i**) with the value 3, plus 1.

Q5. If you're using IDLE to run a Python application, explain how to print a multidimensional list as efficiently?

A5. To print a multidimensional list in IDLE, we can use a nested loop to iterate through the elements of each sub-list and print them one by one. Here is an example:

my\_list = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

for row in my\_list:

for item in row:

print(item, end=' ')

print() # print a newline after each row

In this example, the outer loop iterates through each sub-list in **my\_list**, and the inner loop iterates through each item in the current sub-list. The **end=' '** argument passed to the **print** function tells it to print a space after each item, rather than the default newline character. The **print()** statement at the end of each outer loop iteration prints a newline character to start a new line for the next row of the matrix.

This approach is efficient and easy to read, and can be used to print multidimensional lists of any size.

Q6. Is it possible to use list comprehension with a string? If so, how can you go about doing it?

A6. Yes, list comprehension can be used with a string in Python. Here's an example:

string = "Hello World!"

list\_of\_chars = [char for char in string]

print(list\_of\_chars)

This code will output a list of characters in the string:

['H', 'e', 'l', 'l', 'o', ' ', 'W', 'o', 'r', 'l', 'd', '!']

Q7. From the command line, how do you get support with a user-written Python programme? Is this possible from inside IDLE?

A7. From the command line, we can get support with a user-written Python program by using the **--help** option or passing in the **-h** flag. For example, if our Python program is called **myprogram.py**, we can type **python myprogram.py --help** to get a list of command-line options and arguments that our program accepts.

In IDLE, we can also get help on a specific function or module by using the built-in **help()** function. To use **help()**, simply type **help(function\_name)** or **help(module\_name)** in the IDLE shell, replacing **function\_name** or **module\_name** with the name of the function or module we want help on.

We can also use the IDLE debugger to step through our Python code line-by-line and see the values of variables at each step, which can be helpful for debugging. To access the debugger in IDLE, go to the "Debug" menu and select "Debugger" or press the F5 key.

Q8. Functions are said to be “first-class objects” in Python but not in most other languages, such as C++ or Java. What can you do in Python with a function (callable object) that you can't do in C or C++?

A8. In Python, functions are first-class objects, which means that they can be treated as any other object, such as integers or strings. This provides several benefits, including the ability to pass functions as arguments to other functions, return functions from functions, and store functions in data structures such as lists or dictionaries.

This functionality enables a more dynamic programming style in Python that is not possible in C or C++. For example, in Python, we can define a function that takes another function as an argument and applies it to a list of values, which is known as higher-order functions. This makes it easier to write generic and reusable code that can work with different functions and data types.

In C and C++, functions are not first-class objects, so we cannot pass them as arguments or return them from functions in the same way that we can in Python. This limits the flexibility and expressiveness of the code that can be written in these languages.

Q9. How do you distinguish between a wrapper, a wrapped feature, and a decorator?

A9. In Python, a wrapper is a function that takes another function as an argument, adds some functionality to it, and returns a new function that incorporates the original function. The new function can be called in the same way as the original function but with additional functionality.

A wrapped feature is the original function or class that is being wrapped or decorated. The wrapped feature is passed as an argument to the wrapper function, which returns the new decorated feature.

A decorator is a wrapper function that takes a function or a class as input and returns a new function or class with additional functionality. Decorators are used to modify the behavior of functions or classes without changing their source code. Decorators are usually written using the "@" symbol and can be applied to any function or class.

In short, a wrapper is a function that adds functionality to another function, a wrapped feature is the original function that is being wrapped or decorated, and a decorator is a wrapper function that returns a new function or class with additional functionality.

Q10. If a function is a generator function, what does it return?

A10. If a function is a generator function, it returns a generator object. The generator object can be iterated over to generate a sequence of values. When a generator function is called, it returns a generator object without executing the body of the function. The body of the function is executed only when the generator object's **\_\_next\_\_()** method is called, and it generates the next value in the sequence. The **yield** keyword is used in a generator function to yield a value, which is returned by the generator object's **\_\_next\_\_()** method. When a **yield** statement is executed, the generator function's state is saved, and execution is paused until the next value is requested by the caller.

Q11. What is the one improvement that must be made to a function in order for it to become a generator function in the Python language?

A11.   
In order to turn a regular function into a generator function in Python, the **yield** keyword must be used instead of the **return** keyword. The **yield** keyword causes the function to return a generator object which can be used to produce a series of values, one at a time, through subsequent calls to the generator's **\_\_next\_\_()** method. When a generator function is called, it does not execute the code inside the function immediately, but rather it returns a generator object that can be used to iterate over the series of values produced by the function. Each time the **yield** keyword is encountered, the function pauses and saves its state so that it can resume execution from that point on the next call.

Q12. Identify at least one benefit of generators.

A12. One of the main benefits of generators is that they allow us to generate a sequence of values without having to create and store all of the values in memory at once. This means that generators are more memory-efficient than other approaches that might generate an entire sequence upfront. Generators also allow us to generate values on-the-fly, as needed, which can be useful when working with large data sets or when we don't need all of the values in a sequence at once. Additionally, generators can make code more readable and easier to maintain, since they allow us to separate the generation of a sequence from the logic that processes the sequence.