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

Ans: No, it serves a specific purpose in programming, which is to modify the value of a variable by adding another value to it.

For example, if we have a variable x with a value of 5, and we want to add 3 to it, we could write x = x + 3. However, using the += operator allows we to write this more concisely as x += 3. This not only saves typing time, but can also make the code more readable and easier to understand.

In terms of runtime performance, using the += operator can actually be faster than using the equivalent x = x + 3 expression. This is because the += operator is a compound operator, which means that it combines an arithmetic operation (addition in this case) with an assignment operation. This can result in more efficient code generation and execution by the compiler or interpreter, which can lead to faster results at runtime.

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?

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

Ex:

temp = a + b

b = a

a = temp

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

Ans:

my\_list = [0 for \_ in range(100)]

or

my\_list = [0] \* 100

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

Ans: my\_list = [1, 2, 3] \* 33 + [1, 2, 3][:99 % 3]

Here's a step-by-step breakdown of how this works:

a. [1, 2, 3] \* 33 creates a list of 99 integers, repeating the sequence 1, 2, 3, 33 times.

b. [1, 2, 3][:99 % 3] takes the first 99 % 3 = 0 elements of the list [1, 2, 3], which is an empty list in this case. This ensures that the resulting list has exactly 99 elements.

c. The + operator concatenates the two lists together, giving you a list of 99 integers that repeats the sequence 1, 2, 3.

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

Ans: To print a multidimensional list efficiently in IDLE, you can use a nested loop to iterate over the rows and columns of the list and print each element. Here's an example:

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

# Print each element of the list on a separate line

for row in my\_list:

for col in row:

print(col, end=' ')

print()

This code iterates over each row of the list, then over each element in each row, and prints each element followed by a space. The end parameter in the print() function is set to a space to ensure that each element is printed on the same line. After printing all the elements in a row, the code prints a newline character to move on to the next row.

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

Ans: Yes, it is possible to use list comprehension with a string in Python. List comprehension is a concise way of creating a new list by applying an operation to each element of an existing list or iterable, and strings are iterable in Python. Here's an example of how you can use list comprehension with a string:

my\_string = "hello world"

# Create a list of the ASCII codes for each character in the string

ascii\_list = [ord(char) for char in my\_string]

# Create a list of the lowercase characters in the string

lowercase\_list = [char for char in my\_string if char.islower()]

print(ascii\_list) # Output: [104, 101, 108, 108, 111, 32, 119, 111, 114, 108, 100]

print(lowercase\_list) # Output: ['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?

Ans: From the command line, you can get support with a user-written Python program by running the program with the -h or --help command-line argument. This will display a brief usage summary and any command-line options that are available for the program.

For example, if you have a Python program called my\_program.py, you can get help by running the following command:

python my\_program.py -h

Alternatively, you can run the program with the --help argument:

python my\_program.py --help

This will display a brief summary of the program's usage and options.

In IDLE, you can get help with a user-written Python program by opening the program in the editor window and using the built-in help functions. For example, you can use the help() function to get information on a specific module or function that is used in the program, or use the dir() function to get a list of available functions and variables.

To use the help() function, simply call it with the name of the module or function you want to get help on. For example:

import my\_module

help(my\_module.my\_function)

This will display the help documentation for the my\_function function in the my\_module module.

To use the dir() function, simply call it with no arguments:

import my\_module

dir(my\_module)

This will display a list of all available functions and variables in the my\_module module.

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++?

Ans:

a. Functions can be assigned to variables

b. Functions can be passed as arguments to other functions

c. Functions can be returned by other functions

d. Functions can be defined inside other functions

e. Functions can have default arguments and variable-length argument lists

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

Ans: A wrapper provides a simplified or modified interface to an existing code or feature, a wrapped feature is the original code or feature being modified, and a decorator is a special type of wrapper that adds new behavior to an existing code or feature.

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

Ans: A generator function in Python returns a generator object, which is an iterator that can be used to generate a sequence of values on the fly, without storing them all in memory at once. When a generator function is called, it returns a generator object without actually executing the body of the function. The generator object can then be iterated over using a for loop or by calling the next() function repeatedly.

Generator functions are defined using the yield keyword instead of return, which allows them to generate a series of values over time, rather than returning a single value and exiting. Each time the yield keyword is encountered in the generator function, the function's execution is suspended and the value is returned to the caller. The next time the generator is iterated, execution resumes from where it left off, and the next value is generated.

For example, consider the following generator function that generates the Fibonacci sequence:

def fibonacci():

a, b = 0, 1

while True:

yield a

a, b = b, a + b

When this function is called, it returns a generator object that can be iterated over to generate the Fibonacci sequence:

fib = fibonacci()

next(fib) #0

next(fib) #1

next(fib) #1

next(fib) #2

....

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?

Ans: To turn a regular function into a generator function in Python, the function's return statements should be replaced with yield statements. This is the main difference between a regular function and a generator function.

A yield statement in a function causes the function to become a generator that can be used to produce a sequence of values, one at a time, on demand. When a yield statement is executed, the current value is returned to the caller, but the state of the function is saved so that execution can be resumed later. Each time the generator is iterated, execution resumes from where it left off, and the next value is generated.

Here's an example of a simple generator function in Python that generates a sequence of numbers:

def number\_generator(n):

for i in range(n):

yield i

# Using the generator

gen = number\_generator(5)

for num in gen:

print(num)

In this example, the number\_generator() function is defined as a generator function by using the yield keyword instead of return. The function generates a sequence of numbers from 0 to n-1, and each time the yield statement is executed, the next number is returned to the caller.

Q12. Identify at least one benefit of generators.

Ans: One benefit of generators in Python is that they can be used to generate and process large sequences of values without having to store all the values in memory at once. This can be more memory-efficient than creating a list or other data structure that holds all the values at once. Also generators also be faster than creating and processing large lists of values, since they generate and process values on the fly, rather than all at once.

Generators produce values on demand, one at a time, and only generate the next value when it is needed. This can be especially useful when working with large datasets or infinite sequences of values, where it may not be practical or even possible to store all the values in memory at once.

Generators can also be used to create complex sequences of values that are difficult or impossible to create using other methods. For example, generators can be used to generate sequences of values that depend on the previous values generated, or that are based on complex mathematical or statistical calculations.