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

**Answer:**  
The += operator is not just for show. It performs **in-place modification** in many cases, particularly for mutable types (like lists). This can lead to better performance than creating a new object and reassigning it, especially when working with large datasets.

For example:

lst = [1, 2, 3]

lst += [4, 5]

This modifies lst in place, while the equivalent code:

lst = lst + [4, 5]

creates a new list, which may take longer.

So, in scenarios where you’re modifying a mutable object, += can be faster at runtime because it avoids creating new objects and reassigning them.

**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?**

**Answer:**  
In Python, the expression a, b = a + b, a is a **simultaneous assignment** that swaps the values of a and b while also updating a to be the sum of the old values.

In **most other languages** (like C, C++, or Java), you’d need to write at least **3 statements** to accomplish this. Here’s how you’d do it:

temp = a; // Store value of a in temp

a = a + b; // Set a to a + b

b = temp; // Set b to the old value of a

Thus, you'd need 3 statements in most languages to achieve the same result.

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

**Answer:**  
The most effective way to create a list of 100 integers set to 0 is using the multiplication operator:

lst = [0] \* 100

This approach is **fast** and **concise**, initializing the list with 100 zeros in one operation.

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

**Answer:**  
The most effective way to initialize a list of 99 integers repeating the sequence [1, 2, 3] is to use list multiplication in combination with the sequence:

lst = [1, 2, 3] \* 33 # This creates a list with 99 elements (3 \* 33 = 99)

**Step-by-step:**

1. Create the sequence [1, 2, 3].
2. Multiply the sequence by 33 to repeat it 33 times, resulting in 99 elements.

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

**Answer:**  
In IDLE, you can simply print a multidimensional list (like a list of lists) by calling print() on it. Python’s default string representation for lists will format it in a readable manner:

matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

print(matrix)

This will print the entire multidimensional list in a single call. If you need to print it in a more structured way (one row per line), you can loop through the inner lists:

for row in matrix:

print(row)

This approach is efficient and easy to use within IDLE.

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

**Answer:**  
Yes, it is possible to use list comprehension with a string. Since a string is iterable, you can loop through each character in the string and apply a condition or transformation.

Example: Create a list of all uppercase letters from a string:

s = "Hello World!"

uppercase\_letters = [char for char in s if char.isupper()]

print(uppercase\_letters) # Output: ['H', 'W']

Here, the list comprehension loops through each character in the string and collects the uppercase characters.

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

**Answer:**  
To get support from the command line, you can use the built-in **help()** function in Python, which provides documentation for Python's standard library. For user-written code, you can use **docstrings** to document your program and then use help() to view it.

In the command line:

1. Enter Python by typing python or python3.
2. Use help() to access general help, or help(function\_name) to get specific help on your function.

From inside IDLE:

* You can also call help() directly from the IDLE shell or use print() to display any debug information.

**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++?**

**Answer:**  
In Python, functions are first-class objects, meaning they can be:

1. **Assigned to variables**:
2. def greet():
3. return "Hello"
4. hello = greet # Assigning function to variable
5. print(hello()) # Output: Hello
6. **Passed as arguments to other functions**:
7. def execute\_function(func):
8. return func()
9. print(execute\_function(greet)) # Output: Hello
10. **Returned from other functions**:
11. def outer\_function():
12. def inner\_function():
13. return "Inside"
14. return inner\_function
15. func = outer\_function() # Returns inner\_function
16. print(func()) # Output: Inside

In languages like C++ or Java, functions cannot be treated as first-class objects and are usually tied to specific classes or structures.

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

**Answer:**

* **Wrapper**: A function or object that **encapsulates** another function, providing additional behavior before or after calling the original function.
* **Wrapped feature**: The original function or feature that is being wrapped or encapsulated by another function.
* **Decorator**: A specific type of wrapper in Python that is used to **modify** or **extend** the behavior of a function or method without changing its code. Decorators are typically applied using the @decorator\_name syntax.

Example of a decorator:

def my\_decorator(func):

def wrapper():

print("Before function")

func()

print("After function")

return wrapper

@my\_decorator

def greet():

print("Hello!")

greet() # Output: Before function, Hello!, After function

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

**Answer:**  
A generator function **does not return** a value in the traditional sense. Instead, it **yields** a sequence of values using the yield keyword. When called, a generator function returns a **generator object**, which can be iterated over to produce values one at a time.

def my\_generator():

yield 1

yield 2

yield 3

gen = my\_generator() # This returns a generator object

for value in gen:

print(value) # Output: 1, 2, 3

**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?**

**Answer:**  
To convert a regular function into a generator function, you must replace **return** with **yield**. The yield keyword allows the function to return a value while preserving its state, making it capable of resuming from where it left off.

**Q12. Identify at least one benefit of generators.**

**Answer:**  
**Memory efficiency** is a key benefit of generators. Since generators produce values one at a time and do not store the entire sequence in memory, they are especially useful for working with large datasets or streams of data.

Example:

def large\_range():

for i in range(1000000):

yield i # Only one value is in memory at a time

gen = large\_range() # Generates values on demand, saving memory