**PART 1: Answer the following short questions.**

1. Why do modern programming languages not include the *goto* unconditional branch command? When and why do these articles say there are some situations where "*goto"* is justified? (2 points)

I think modern programming languages ditch the goto command because it makes code a mess—real "spaghetti code" territory. Back when I was looking at older languages like Fortran or BASIC, goto let me jump anywhere, which sounded cool until I realized it turned debugging into a nightmare. In 1968, Dijkstra's "Go To Statement Considered Harmful" hit me hard—it showed how goto wrecks structured programming. I prefer loops and conditionals in languages like Python or Java; they keep my control flow clean and predictable.

But when I checked out those articles you linked, I saw some exceptions. The Kerneltrap piece on Linux kernel coding got me thinking—sometimes I'd use goto for error handling in low-level stuff. Say I'm allocating memory and locks; if something fails, I could goto a cleanup label instead of nesting a bunch of ifs. It's cleaner that way. The XML.com article backed me up too—when I'm escaping nested loops or tweaking performance in a parser, goto can cut through the clutter. So, I avoid it unless I'm in a tight spot where it actually simplifies things.

2. Explain the key design issues for counter-controlled loop statements with a short example. (1 point)

Counter-controlled loops (e.g., for loops) repeat a block of code a set number of times, driven by a counter variable. Designing them involves three key issues:

1. Initialization: Where does the counter start? It needs a clear, predictable starting value.
2. Termination Condition: When does the loop stop? This must be well-defined to avoid infinite loops or off-by-one errors.
3. Increment/Decrement: How does the counter change each iteration? The step size must be consistent and appropriate.

Poor design in any of these can lead to bugs. For example, consider this Python snippet:

Example

```
for i in range(1, 5, 2):  # Start at 1, stop before 5, step by 2
   print(i)
# Output: 1, 3
```

Here, i starts at 1 (initialization), stops before 5 (termination), and increments by 2 (step). If the step were accidentally set to 0, you'd get an infinite loop—a classic design flaw. Balancing these elements ensures the loop behaves as intended.

3. In C++, for a large size data, why pass-by-reference parameter passing is preferred over pass-by-value? What is the most well-known drawback of pass-by-reference? How can it be mitigated? (1 point)

Pass-by-reference is preferred for large data structures in C++ because it avoids copying overhead, which improves performance.

Drawback: Pass-by-reference allows the called function to modify the caller's variable, potentially leading to unintended side effects.

Mitigation: Use const references to prevent modification:

Example,

```
void printData(const std::vector<int>& data) {
   for (int i : data) std::cout << i << " ";
}
```

4. Define variadic arguments and explain how they are implemented in Python. Describe the two types of variadic arguments with examples (1 point)

Variadic arguments let a function accept a variable number of arguments, giving flexibility to the caller. In Python, they're implemented using special syntax in function definitions, packing arguments into tuples or dictionaries.

There are two types:

1. Positional Variadic Arguments (*args): Collects extra positional arguments into a tuple. Use * before a parameter name.

Example:

```
def sum_all(*numbers):
  return sum(numbers)
print(sum_all(1, 2, 3, 4))  # Output: 10
print(sum_all(5))         # Output: 5
```

Here, *numbers grabs all positional inputs as a tuple (1, 2, 3, 4) or (5).

2. Keyword Variadic Arguments (**kwargs): Collects extra keyword arguments into a dictionary. Use ** before a parameter name.

Example:

```
def print_info(**details):
  for key, value in details.items():
    print(f"{key}: {value}")
print_info(name="Alex", age=25)
# Output:
# name: Alex
# age: 25
```

**details packs name="Alex" and age=25 into a dict {"name": "Alex", "age": 25}.

5. Explain what coroutines are and why they are referred as "quasi-concurrency". How can you implement coroutines in Python? (2 points)

- Refer to Chapter 9, Subprograms. pp 407 – 410

Coroutines are subprograms that allow suspension and resumption at specific points using yield or await. Unlike functions, coroutines maintain state between executions.

They are called "quasi-concurrent" because they allow asynchronous behavior (like waiting on I/O) without real parallelism, thus simulating concurrency in a single-threaded environment.

Implementing Coroutines in Python:

1. Using generators with yield:

```
def counter():
  i = 0
  while True:
    yield i
    i += 1
c = counter()
print(next(c))  # 0
print(next(c))  # 1
```

2. Using async / await (Python's async coroutines):

```
import asyncio

async def greet():
  await asyncio.sleep(1)
  print("Hello!")

asyncio.run(greet())
```

6. List the four pillars of OOP and explain each using examples from your HW 5 implementation in your chosen programming language. (3 points)

The four pillars of OOP are Encapsulation, Inheritance, Polymorphism, and Abstraction. Below, I'll explain each with Swift code inspired by a hypothetical HW 5 Library system.

1. Encapsulation
   Definition: Bundling data with methods that operate on that data, restricting direct access.
   Example: In the Book class, properties like title, author, and price are encapsulated. Access is controlled through initializers and methods, not exposed directly for modification.

2. Abstraction
   Definition: Hiding complex implementation details and showing only essential features.
   Example: The Sellable protocol abstracts the concept of sellable items by requiring only name and price, regardless of whether the item is a Video, Audio, or Book.

3. Inheritance
   Definition: Allowing a class to inherit properties and methods from another class.
   Example: EBook and PaperBook inherit from the Book class, gaining access to shared properties like title and methods like printInfo().

4. Polymorphism
   Definition: Allowing different classes to be treated as instances of the same superclass or protocol.
   Example: A list of Sellable items can contain Audio, Video, EBook, and PaperBook objects, and calling price on each behaves appropriately based on the actual type.