1. DATA4: Scope, Lifetime, Shadowing
    a. The scope for res is local to the function boop(), since it is created inside boop() it's only accessible within that function. However the lifetime of boop() is similarly restricted though it could have a longer lifetime if the object referred inside, num_boops, is referenced after boop() is done being called.

       The scope for the dictionary that res is bound to is initially bound to res within boop but can be accessed outside if assigned elsewhere. The lifetime of the dictionary persists as long as there's a reference to it. In this case it's just printed and then discarded, so that means after the program is over the dictionary dies.
    b. Because x gets destroyed once we are out of the curly brackets, meaning that when n tries to point to x, it doesn't exist anymore-thus throwing an error.
    c. It could be due to the stack memory, that when x ends, the memory used for x is technically available for reuse but still contains 42 bc it isn't immediately cleared, but this is incredibly risky to rely on.
    d. This employs lexical scoping because we see in the context of the program as a whole (ignoring lines 3-6), we see that x still maintains the same values when it gets altered and printed. However this also employs shadowing (similar to Rust), where lines 3-6 can declare a variable of the same name that can shadow the place of the more wide variable x and have it temporarily be the value of 1 before leaving the shadow and letting the wider x take over for the rest of the program.
    e. This is actually still lexical despite the fact that x is accessed outside of boop(), but it is function-level lexical so it is okay. We know it is for sure lexical because dynamic would've had x been altered to 2 when it got printed but it didn't. And also we get undefined behavior when we leave the scope of the functions to print out x again.
2. DATA5: Smart Pointers
    a. It should be int* since we need a pointer to an integer to ensure that each my_shared_ptr instance points to the same reference counter
    b.

```
my_shared_ptr(int * ptr) {
    this->ptr = ptr;
    refCount = new int(1);
}
```

    c.

```
my_shared_ptr(const my_shared_ptr & other) {
    ptr = other.ptr;
    refCount = other.refCount;
    (*refCount)++;
}
```

d.

```
~my_shared_ptr() {
    if (refCount) {
        (*refCount)--;
        if (*refCount == 0) {
            delete ptr;
            delete refCount;
        }
    }
}
```

3. DATA6: Garbage Collection
    a. Languages that do not use garbage collections and instead use destructors to free objects, close networks, delete files, etc. are more efficient since we can guarantee that we have open memory with good code that will optimize performance which is something that we are going to need in outer space dodging asteroids like Michael Jukeson. Garbage collection is slow and run the risk of not being cleaned on time, leading to models that don't properly utilize a computer's RAM and memory.
    b. I do not agree, since if we use mark and sweep we run the risk of having memory fragmentation where it could become slow or borderline impossible to find free chunks of memory to allocate new objects which in this high-action environment we are going to need all the performance we can get.
    c. Mark and compact would be the better choice. Since we get variable sized coordinate numbers, we run the risk of needing larger chunks of memory for the larger variables and we cannot run the risk of memory fragmentation if we are to need larger chunks.
    d. In C++, the destructor ~RoomView ensures that the Socket object is cleaned up when the RoomView instance goes out of scope. However in Go if sockets aren't explicitly cleaned up they can remain open even if RoomView goes out of scope. This means that if RoomView instances are frequently made and closed it would lead to a resource leak where we could run out of sockets easily.
4. DATA7: Binding Semantics
   It is sorta similar to C++ but instead the addresses are bound to the variables themselves and when we assign them to each other we assign the addresses instead of the actual values. The == may actually unwrap the pointers for the actual values when it comes to the comparison though, hence why put s1 == s2 resolves as true.
5. DATA8: Binding Semantics/Parameter Passing

a. Pass by reference, since we are affecting the variable values of x and y out of the scope of f(). It cannot be pass by pointer since we assign x and y to new values in f()
b. Pass by value, since if we pass by value it doesn't affect x and y on the outside.
c. It would still return 5 since the object's state (the object's variable x or x.x) is modified