1. OOP1: Classes, Objects (5 min)
   We could go about this either with constructor functions or prototypes. Constructor functions could be used to initialize objects and it's honestly up to the coder to make sure that everything is done in a uniform manner. Using prototypes is more efficient however because it allows methods to be shared among all instances, rather than being duplicated for each object.

2. OOP2: Classes, Objects, Encapsulation (5 min)
   Allowing direct access to member variables can weaken encapsulation–you get a weaker layer of control since there's no prevention from certain code from accessing certain functions that it shouldn't access and it's kinda up to the developer to make sure that doesn't happen. This is done because Python's design philosophy emphasizes simplicity and "consenting adults"—the idea that developers should have the freedom to use attributes as they see fit without unnecessary restrictions.

3. OOP3: Interfaces and Types (5 min)
   Only pointers, because it's impossible to create an object of type shape but it is possible to create objects that are subtypes of shapes, like diff types of shapes like circles or squares.

4. OOP4: Classes, Getters, Setters (5 min)
   The key difference is that the Java Protected keyword allows the members of the class to be accessed by any other class in that same package, even if unrelated by inheritance. The advantages is that it is convenient to access certain variables in the same package but could weaken encapsulation.

5. OOP5: Classes, Interfaces in Dynamically-typed Languages (5 min)
   In dynamic languages, interfaces are primarily guidelines rather than strict enforcement. Interfaces in dynamically-typed languages define a set of methods or properties that a class or object should implement. This serves as a guide for developers, ensuring consistency without compiler checks.

6. OOP6: Subclass Inheritance, Interface Inheritance (5 min)
   If you had something within multiple (mostly) unrelated subtypes without a common ancestor, this is when you'd want to use interface inheritance. Let's say you had an interface called savable that could be used for files, video games, etc. Obviously they don't have a common ancestor but they share the interface–savable–and you can reuse code in this aspect. You would want to use subclass inheritance when classes are directly related with one another.

7. OOP7: Interface Inheritance, Supertypes and Subtypes (8 min)
    a. Class A: no supertypes
       Class B: supertype A
       Class C: supertype A
       Class D: superclass B, C, A
       Class E: superclass C, A
       Class F: superclass D, B, C, A
       Class G: superclass B, A

    b. D, F, and G
    c. No, because class A might be an object of let’s say class B instead of C. This would run for weakly typed languages but it may fail at runtime