Basic Transition Commands in Manim

1. Fade in and fade out.
    - Gradual appearance or disappearance of an object.
    - Example:
      ```
      self.play(FadeIn(mobject))
      self.play(FadeOut(mobject))
      ```

2. Create
    - Simulates drawing or constructing an object from nothing.
    - Example:
      ```
      self.play(Create(mobject))
      ```

3. Uncreate
    - The reverse of create, making the object disappear in a reverse-creation manner.
    - Example:
      ```
      self.play(Uncreate(mobject))
      ```

4. Transform
    - Smoothly morphs one object into another.
    - Example:
      ```
      self.play(Transform(mobject1, mobject2))
      ```

5. ReplacementTrandform.
    - Similar to Transform, but optimized for replacing one object with another.
    - Example:
      ```
      self.play(ReplacementTransform(mobject1, mobject2))
      ```

6. ApplyMethod
    - Applies a transformation method to a mobject.
    - Example:
      ```
      self.play(ApplyMethod(mobject.shift, UP))
      ```

7. MoveToTarget.
    - Moves a mobject to its predefined “target” position.
    - Example:
      ```
      mobject.generate_target()
      mobject.target.shift(RIGHT)
      ```

```
self.play(MoveToTarget(mobject))
```

8. FadeTransform.
   - Combines fading out of one mobject and fading in another.
   - Example:
     ```
     self.play(FadeTransform(mobject1, mobject2))
     ```

9. Grow From Center.
   - Animates the object growing outward from its center.
   - Example:
     ```
     self.play(GrowFromCenter(mobject))
     ```

10. Shrink Center
    - Animates the object shrinking into its center.
    - Example:
      ```
      self.play(ShrinkToCenter(mobject))
      ```

11. Rotate
    - Rotates a mobject by a specified angle.
    - Example:
      ```
      self.play(Rotate(mobject, angle=PI/4))
      ```

12. Scale.
    - Scales the size of a mobject.
    - Example:
      ```
      self.play(mobject.animate.scale(2))
      ```

13. Spinnin From Nothing
    - Spins and fades in the object.

14. Write.
    - Simulates writing or drawing text stroke-by-stroke.
    - Example:
      ```
      self.play(Write(text))
      ```

15. Draw Border Then Fill
    - Animates drawing border of an object, followed by filling it.
    - Example:
      ```
      self.play(DrawBorderThenFill(mobject))
      ```

16. Lagged Start.
    - Staggers the animations of multiple objects for a cascading effect.
    - Example:
      self.play(LaggedStart(*[FadeIn(obj) for obj in group]))

Manim transitions can often be combined using AnimationGroup or by sequentially chaining self.play() calls.

- AnimationGroup: Plays multiple animations simultaneously.
    - self.play(AnimationGroup(FadeIn(obj1), Write(obj2), lag_ratio=0.5))
- Chaining Animations:
    - self.play(FadeIn(obj1))
    - self.play(Transform(obj1, obj2))
    -

¿What is a class in Object-Oriented programming (OOP)?

A class in Object-Oriented Programming (OOP) is a blueprint or template for creating objects. It defines a data structure that contains:

- Attributes: Characteristics or properties of the objects (variables).
- Methods: Actions or behaviors that the objects can perform (functions).

Classes provide a way to group related data and functions together, making code more modular, reusable, and easier to manage.

Key Characteristics of a Class:

- Encapsulation: Bundles data (attributes) and methods (functions) together.
- Inheritance: Allows a class to inherit attributes and methods from another class.
- Polymorphism: Allows objects to be treated as instances of their parent class while maintaining their unique behaviors.
- Abstraction: Hides complex implementation details and exposes only the relevant features.

When Should a Class Be Used?

A class should be used when you need to model real-world entities, behaviors, or concepts that have related properties and actions. It is particularly useful in the following scenarios:

1. When Organizing Related Data and Behaviors

- Use classes to group related attributes (data) and methods (functions).
- Example: Modeling a User with attributes like name, email, and methods like login and logout.

2. When Reusability is Needed

- Classes allow you to reuse code by creating multiple objects with the same blueprint.
- Example: A Product class can be reused for different types of products with slight variations.

3. For Abstraction and Encapsulation

- Hide implementation details and expose only the necessary functionality.
- Example: A BankAccount class may abstract methods for deposit and withdraw, while hiding how transactions are processed internally.

4. When Using Inheritance

- Classes can share behavior and properties through inheritance.
- Example: A Vehicle class can have subclasses like Car and Bike that inherit common attributes (speed, color) and methods (start, stop).

5. For Polymorphism

- Define a general interface for multiple types of objects.
- Example: A Shape class with a draw method can have subclasses like Circle and Square that implement draw differently.

6. To Improve Code Modularity

- Breaking down a program into smaller, self-contained classes makes the codebase easier to maintain and extend.