AnnotatedWrapper

1. Class Description

Goals:

- Provide a base class that allows objects to manage annotations (e.g., tags, comments, metadata) easily and efficiently.

How:

- Using a dictionary (e.g., std::unordered_map or Python's dict) for storing annotations with key-value pairs.

Restrictions:

- Clear separation between the annotated object and its annotations

Important Features:

- Allow adding, removing, and updating annotations without impacting the annotated object.
- Provide consistent and predictable access to annotations

High-Level Functionality:

- Allow derived classes to easily attach, retrieve, and manage annotations.
- Support dynamic annotations with minimal overhead while maintaining flexibility for different data types.

---

2. Similar Classes in the Standard Library

- **collections.UserDict**: For creating a custom dictionary-like interface for managing annotations.
- **functools.wraps**: To ensure that wrapping functionality in derived classes preserves key properties like docstrings and function signatures.
- **dataclasses**: Provides mechanisms for associating metadata with fields, which could be useful in managing annotations.
- **logging.LoggerAdapter**: An example of a wrapper class that adds context-specific functionality to existing objects.

---

3. Key Functions

- **add_annotation(key: str, value: Any) -> None**
  Add a new annotation with the specified key and value.
- **get_annotation(key: str) -> Any**
  Retrieve the annotation associated with the given key.
- **remove_annotation(key: str) -> None**
  Remove an annotation by its key.
- **list_annotations() -> Dict[str, Any]**
  Return a dictionary of all annotations currently associated with the object.
- **clear_annotations() -> None**
  Remove all annotations from the object.

---

4. Error Conditions

Below are error conditions that the class will address, categorized by their source:

1. **Programmer Errors**:
   - Adding an annotation with an invalid key (e.g., None or an unsupported type).
   - Attempting to access or remove an annotation that does not exist.
2. **User Errors**:
   - Supplying invalid input to public methods, such as an invalid annotation format.

---

5. Expected Challenges

- Designing the class to remain lightweight and efficient while still being flexible for a variety of use cases.
- Ensuring compatibility with derived classes, especially when annotations need to integrate seamlessly with custom data.

---

6. Dependencies on Other Projects

- None?