Below is a structured guide for your Python classes, organized with nested inheritance for clarity and readability. Each class includes the methods you should create, their purpose, and a comprehensive explanation of why they are that type (class, instance, static, or classmethod). I've followed your request to nest related classes (e.g., `Transformer` → `TeamTransformer`, etc.) and provided five+ simple sentences explaining the type choices for both classes and methods.

---

1. `AppConfig`

Type: `@dataclass`
Purpose: Holds configuration data like paths, seeds, and formats.
Why a dataclass?
1. It's only used to store data, not to perform actions.
2. A dataclass auto-generates methods like `__init__` and `__repr__`, saving you from writing repetitive code.
3. If it were a regular class, you'd have to manually define an initializer just for data storage.
4. If it were an abstract base class (ABC), it would imply subclasses with behavior, which isn't needed here.
5. A dataclass clearly signals "this is a simple data container" to anyone reading the code.

Methods:
- `load_from_file`
- **Type:** `@classmethod`
- **What it does:** Loads configuration from a file and returns an `AppConfig` instance.
- **Why we need it:** Provides an easy way to create an `AppConfig` from a file without manually constructing it.
- **Why a classmethod?**
1. It works on the class itself to create an instance, not on an existing object.
2. If it were an instance method, you'd need an object first, which doesn't make sense for loading.
3. If it were static, it wouldn't have access to the class type to return `AppConfig`.
4. A classmethod ensures the method is tied to the class and can be overridden if needed.
5. It's a standard pattern for alternate constructors, making the code intuitive.

---

2. `FileManager`

Type: Regular class
Purpose: Manages file input/output operations like reading text or saving JSON/YAML files.

Why a regular class?
1. It groups related file operations into one object that can hold state, like a base directory.
2. If it were all static methods, you couldn't change configurations (e.g., root path) at runtime.
3. If it were an ABC, it would suggest multiple file manager types, but we only need one now.
4. If it were a dataclass, it couldn't include the behavior we need for file handling.
5. A regular class allows future extensions (e.g., cloud storage) without changing its core design.

Methods:
- `load_text`
- **Type:** Instance method
- **What it does:** Reads and returns the content of a text file.
- **Why we need it:** Centralizes file reading so it's reusable across the app.
- **Why an instance method?**
1. It can use object state, like a base directory, to find files.
2. If it were static, you'd have to pass the base path every time, which is clunky.
3. If it were a classmethod, it wouldn't belong to an instance's configuration.
4. An instance method keeps the logic tied to a specific `FileManager` object.
5. It's more flexible for future changes, like adding caching or logging.

- `save_json`
  - **Type:** Instance method
  - **What it does:** Saves data as a JSON file.
  - **Why we need it:** Ensures consistent JSON file writing across the app.
  - **Why an instance method?**
    1. It can use instance settings, like a default output directory.
    2. If it were static, you'd repeat parameters like encoding on every call.
    3. If it were a classmethod, it wouldn't leverage instance-specific state.
    4. An instance method keeps the behavior encapsulated with the object.
    5. It's easier to test or mock with an instance than a static function.
- `update_yaml`
  - **Type:** Instance method

- **What it does:** Updates or creates a YAML file with given data.
- **Why we need it:** Handles YAML operations, useful for configs or metadata.
- **Why an instance method?**
  1. It can reuse instance state, like file paths or formatting rules.
  2. If it were static, you'd pass all config details every time, making it tedious.
  3. If it were a classmethod, it wouldn't tie to an instance's settings.
  4. An instance method keeps related file logic together cleanly.
  5. It allows future enhancements, like logging updates, without redesign.

---

3. `VersionManager`

Type: Regular class
Purpose: Tracks and retrieves file versions for reproducibility.
Why a regular class?
1. It needs to maintain state, like where versions are stored, across method calls.
2. If it were a dataclass, it couldn't include version-handling logic.
3. If it were static-only, you couldn't configure version policies per object.
4. If it were an ABC, it would force subclassing for a single implementation.
5. A regular class supports evolving version strategies without breaking the interface.

Methods:
- `record_version`
- **Type:** Instance method
- **What it does:** Records a file's version, possibly by copying or logging it.
- **Why we need it:** Tracks changes to files for data pipeline reliability.
- **Why an instance method?**
1. It uses instance state, like the version storage location.
2. If it were static, you'd pass the storage path every time, which is repetitive.
3. If it were a classmethod, it wouldn't belong to a specific manager's setup.
4. An instance method keeps versioning tied to the object's context.
5. It's more adaptable for adding features like version naming rules.

- `get_latest_version`
  - **Type:** Instance method

- **What it does:** Retrieves the latest version of a file.
- **Why we need it:** Ensures the system uses the most current file data.
- **Why an instance method?**
  1. It relies on instance state, like the version directory.
  2. If it were static, you'd need to specify the directory each call.
  3. If it were a classmethod, it wouldn't use per-object settings.
  4. An instance method keeps the logic consistent with the object.
  5. It's easier to extend, like adding version filtering, later.

---

4. `UniqueIdGenerator`

Type: Regular class
Purpose: Generates unique IDs for teams and players.
Why a regular class?
1. It organizes ID generation methods under one logical unit.
2. If it were a dataclass, it couldn't include generation logic.
3. If it were an ABC, it would imply multiple ID generator types, which we don't need.
4. A regular class allows static methods while keeping the option for future instance state (e.g., a seed).
5. It keeps related utilities together without forcing unnecessary instantiation.

Methods:
- `generate_team_id`
- **Type:** `@staticmethod`
- **What it does:** Creates a unique ID for a team based on its name.
- **Why we need it:** Ensures teams have consistent, unique identifiers.
- **Why a static method?**
1. It's a pure function that doesn't need instance state—just the name.
2. If it were an instance method, you'd instantiate an object for no reason.
3. If it were a classmethod, it would suggest class-level overrides, which isn't needed.
4. A static method signals "this is stateless and reusable."
5. It's simple to call directly on the class, improving readability.

- `generate_player_id`
  - **Type:** `@staticmethod`
  - **What it does:** Creates a unique ID for a player based on their

name.

- **Why we need it:** Ensures players have unique identifiers across the system.

- **Why a static method?**
  1. It doesn't rely on object state, just the input name.
  2. If it were an instance method, it would imply state changes, which it doesn't have.
  3. If it were a classmethod, it wouldn't gain anything over static.
  4. A static method keeps it lightweight and independent.
  5. It's a utility function that doesn't need an object context.

---

5. `Transformer`

Type: `typing.Protocol`
Purpose: Defines a standard interface for transformation classes.
Why a Protocol?
1. It only requires a `transform` method, without forcing inheritance.
2. If it were an ABC, every transformer would need to subclass it, which is heavier than needed.
3. If it were a regular class, you might inherit unwanted attributes or methods.
4. A Protocol supports structural subtyping—any class with `transform` works, even third-party ones.
5. It's a lightweight way to enforce a contract without extra boilerplate.

Methods:
- `transform`
- **Type:** Instance method (required by Protocol)
- **What it does:** Transforms input data into a new format (defined by subclasses).
- **Why we need it:** Standardizes how data is processed across different transformers.
- **Why an instance method?**
1. It allows transformers to use their own settings or state during transformation.
2. If it were static, you couldn't store config like a staging directory.
3. If it were a classmethod, it wouldn't belong to an instance's context.
4. An instance method supports per-object customization.
5. It's the natural choice for a method tied to a specific transformer's logic.

5a. `TeamTransformer`

Type: Concrete class implementing `Transformer`
Purpose: Transforms team-related data into a standard format.
Why a regular class?
1. It provides a concrete implementation of the `Transformer` interface.
2. If it were a dataclass, it couldn't include transformation logic.
3. If it were an ABC, it wouldn't be usable on its own.
4. A regular class lets it hold settings (e.g., ID generator) and behavior together.
5. It's a specific transformer type, not an abstract concept.

Methods:
- `transform`
- **Type:** Instance method
- **What it does:** Converts team data into a consistent structure.
- **Why we need it:** Ensures team data is uniform for processing or storage.
- **Why an instance method?**
1. It can use instance-specific settings, like an ID generator.
2. If it were static, you'd pass config every time, which is awkward.
3. If it were a classmethod, it wouldn't tie to the object's state.
4. An instance method keeps the logic bundled with the transformer.
5. It's flexible for adding team-specific rules later.

5b. `PlayerTransformer`

Type: Concrete class implementing `Transformer`
Purpose: Transforms player-related data into a standard format.
Why a regular class?
1. It implements `Transformer` with player-specific logic.
2. If it were a dataclass, it couldn't perform transformations.
3. If it were an ABC, it wouldn't be a usable class.
4. A regular class combines state and behavior for player data.
5. It's a concrete tool, not a generic or abstract one.

Methods:
- `transform`
- **Type:** Instance method
- **What it does:** Standardizes player data.
- **Why we need it:** Keeps player data consistent across the system.
- **Why an instance method?**
1. It can leverage instance state, like player ID rules.
2. If it were static, you'd repeat parameters on every call.
3. If it were a classmethod, it wouldn't use object-specific settings.
4. An instance method ties the logic to the transformer instance.
5. It supports future tweaks, like adding validation.

5c. `EventTransformer`

Type: Concrete class implementing `Transformer`
Purpose: Transforms game event data into a structured format.
Why a regular class?
1. It gives a specific implementation of `Transformer` for events.
2. If it were a dataclass, it couldn't handle transformation logic.
3. If it were an ABC, it wouldn't work without subclassing.
4. A regular class encapsulates event-specific behavior and state.
5. It's a practical, standalone transformer type.

Methods:
- `transform`
- **Type:** Instance method
- **What it does:** Structures event data for analysis or storage.
- **Why we need it:** Makes game events consistent and usable.
- **Why an instance method?**
1. It can use instance settings, like event formatting rules.
2. If it were static, you'd pass config repeatedly.
3. If it were a classmethod, it wouldn't belong to the instance.
4. An instance method keeps event logic with the object.
5. It's extensible for event-specific processing later.

5d. `LineFilterTransformer`

Type: Concrete class implementing `Transformer`
Purpose: Filters and transforms specific lines of data.
Why a regular class?
1. It provides a concrete `Transformer` for line filtering.
2. If it were a dataclass, it couldn't include filtering logic.
3. If it were an ABC, it wouldn't be directly usable.
4. A regular class holds both the filtering rules and behavior.
5. It's a specific, functional transformer, not an abstract idea.

Methods:
- `transform`
- **Type:** Instance method
- **What it does:** Filters out irrelevant data lines and transforms the rest.
- **Why we need it:** Cleans data to improve downstream processing.
- **Why an instance method?**
1. It can store filtering criteria as instance state.
2. If it were static, you'd pass filters every time, which is inefficient.
3. If it were a classmethod, it wouldn't use instance context.
4. An instance method keeps filtering tied to the object.
5. It's adaptable for adding new filter rules later.

---

6. `Normalizer`

Type: Regular class
Purpose: Manages a pipeline of transformation steps to normalize raw text.
Why a regular class?
1. It holds a dynamic list of transformers that changes at runtime.
2. If it were a dataclass, it couldn't orchestrate transformation steps.
3. If it were an ABC, you'd lose the concrete pipeline logic.
4. A regular class lets you build and reuse normalization pipelines.
5. It's a coordinator, not just a data holder or abstract type.

Methods:
- `add_step`
- **Type:** Instance method
- **What it does:** Adds a transformer to the normalization pipeline.
- **Why we need it:** Lets you dynamically build the transformation sequence.
- **Why an instance method?**
1. It modifies the instance's list of steps.
2. If it were static, you couldn't maintain a pipeline per object.
3. If it were a classmethod, it wouldn't belong to a specific normalizer.
4. An instance method keeps the pipeline tied to the object.
5. It's flexible for adding steps in different orders.

- `normalize`
  - **Type:** Instance method
  - **What it does:** Applies all transformation steps to raw text and returns a `NormalizedLog`.
  - **Why we need it:** Executes the full normalization process in one call.
  - **Why an instance method?**
    1. It uses the instance's list of transformers.
    2. If it were static, you'd pass the pipeline every time, which is clumsy.
    3. If it were a classmethod, it wouldn't use instance state.
    4. An instance method keeps the logic with the normalizer object.
    5. It's easier to test or tweak with an instance-based approach.

---

7. `NormalizedLog`

Type: `@dataclass`
Purpose: Stores the result of normalization (tokens, events, metadata).
Why a dataclass?
1. It's just a container for data, with no behavior needed.
2. A dataclass auto-generates useful methods like `__init__` and `__eq__`.
3. If it were a regular class, you'd write unnecessary constructors.
4. If it were an ABC, it would imply subclasses, which isn't the intent.
5. A dataclass keeps it simple and focused on holding results.

Fields (no methods):
- `tokens` – List of processed text tokens.
- `events` – List of event records from the log.
- `metadata` – Dictionary of additional data.

---

8. `Exporter`

Type: `abc.ABC`
Purpose: Defines a common interface for exporting normalized logs.
Why an abstract base class?
1. It ensures all exporters implement the `export` method.
2. If it were a Protocol, you couldn't add shared helper methods later.
3. If it were a regular class, users might instantiate it directly, which isn't usable.
4. An ABC prevents misuse by requiring concrete subclasses.
5. It signals "this is a blueprint for exporters" clearly.

Methods:
- `export`
- **Type:** `@abc.abstractmethod` (instance method)
- **What it does:** Exports a `NormalizedLog` to a destination (defined by subclasses).
- **Why we need it:** Provides a consistent export API across formats.
- **Why an instance method?**
1. It allows subclasses to use instance state, like output settings.
2. If it were static, you couldn't configure exporters individually.
3. If it were a classmethod, it wouldn't belong to a specific exporter instance.
4. An instance method supports per-exporter customization.
5. It's the standard for ABCs to enforce behavior in subclasses.

8a. `JsonExporter`

Type: Concrete subclass of `Exporter`
Purpose: Exports a `NormalizedLog` as a JSON file.
Why a regular class?
1. It provides a specific implementation of `Exporter`.

2. If it were a dataclass, it couldn't include export logic.
3. If it were an ABC, it wouldn't be usable on its own.
4. A regular class encapsulates JSON-specific behavior.
5. It's a concrete tool for a specific export format.

Methods:
- `export`
- **Type:** Instance method
- **What it does:** Saves the log as a JSON file to the given destination.
- **Why we need it:** JSON is widely used for data sharing.
- **Why an instance method?**
1. It can use instance settings, like a file path prefix.
2. If it were static, you'd pass config every call, which is redundant.
3. If it were a classmethod, it wouldn't tie to the instance.
4. An instance method keeps JSON logic with the object.
5. It's flexible for adding JSON-specific options later.

8b. `SqliteExporter`

Type: Concrete subclass of `Exporter`
Purpose: Exports a `NormalizedLog` to a SQLite database.
Why a regular class?
1. It implements `Exporter` for SQLite specifically.
2. If it were a dataclass, it couldn't handle database logic.
3. If it were an ABC, it wouldn't work without further subclassing.
4. A regular class combines SQLite behavior and state.
5. It's a practical exporter for structured data storage.

Methods:
- `export`
- **Type:** Instance method
- **What it does:** Saves the log to a SQLite database at the destination.
- **Why we need it:** Enables querying and storing data efficiently.
- **Why an instance method?**
1. It can store database connection settings in the instance.
2. If it were static, you'd repeat connection details every time.
3. If it were a classmethod, it wouldn't use instance state.
4. An instance method keeps SQLite logic encapsulated.
5. It supports future tweaks, like table schemas.

8c. `CsvExporter`

Type: Concrete subclass of `Exporter`
Purpose: Exports a `NormalizedLog` as CSV files.
Why a regular class?
1. It gives a concrete `Exporter` implementation for CSV.
2. If it were a dataclass, it couldn't export anything.

3. If it were an ABC, it wouldn't be usable directly.
4. A regular class handles CSV-specific logic cleanly.
5. It's a specific exporter for a common tabular format.

Methods:
- `export`
- **Type:** Instance method
- **What it does:** Saves the log as CSV files to the destination.
- **Why we need it:** CSV works with many data tools.
- **Why an instance method?**
1. It can use instance config, like CSV delimiters.
2. If it were static, you'd pass options every call.
3. If it were a classmethod, it wouldn't belong to the instance.
4. An instance method keeps CSV logic with the object.
5. It's extensible for adding CSV-specific features.

8d. `ExcelExporter`

Type: Concrete subclass of `Exporter`
Purpose: Exports a `NormalizedLog` to an Excel file.
Why a regular class?
1. It implements `Exporter` for Excel specifically.
2. If it were a dataclass, it couldn't include export logic.
3. If it were an ABC, it wouldn't be functional alone.
4. A regular class encapsulates Excel-specific behavior.
5. It's a concrete exporter for a popular analysis format.

Methods:
- `export`
- **Type:** Instance method
- **What it does:** Saves the log to an Excel file at the destination.
- **Why we need it:** Excel is great for reporting and analysis.
- **Why an instance method?**
1. It can store instance settings, like sheet names.
2. If it were static, you'd repeat config every time.
3. If it were a classmethod, it wouldn't use instance state.
4. An instance method keeps Excel logic with the object.
5. It's adaptable for Excel-specific tweaks later.

---

9. `GameParser`

Type: Regular class
Purpose: Parses a normalized log and exports it in specified formats.
Why a regular class?
1. It maintains a registry of exporters that can change at runtime.
2. If it were static, you couldn't swap exporter sets easily.

3. If it were an ABC, you'd lose the default registry logic.
4. A regular class lets you manage parsing and exporting together.
5. It follows the Registry Pattern for flexible format handling.

Methods:
- `register_exporter`
- **Type:** Instance method
- **What it does:** Adds an exporter to the parser's registry under a name.
- **Why we need it:** Allows dynamic support for multiple export formats.
- **Why an instance method?**
1. It modifies the instance's exporter dictionary.
2. If it were static, you couldn't maintain a per-object registry.
3. If it were a classmethod, it wouldn't tie to a specific parser.
4. An instance method keeps the registry with the object.
5. It's flexible for registering different exporters per parser.

- `parse`
  - **Type:** Instance method
  - **What it does:** Processes the log and exports it using specified formats.
  - **Why we need it:** Centralizes parsing and exporting in one step.
  - **Why an instance method?**
    1. It uses the instance's registered exporters.
    2. If it were static, you'd pass the registry every time.
    3. If it were a classmethod, it wouldn't use instance state.
    4. An instance method keeps parsing tied to the parser object.
    5. It's easier to extend or debug with an instance.

---

10. `GameFlowManager`

Type: Regular class (Facade/Orchestrator)
Purpose: Runs the entire process from loading to exporting.
Why a regular class?
1. It simplifies the end-to-end flow into one object.
2. If it were static, you'd scatter pipeline logic across your code.
3. If it were an ABC, it would imply multiple pipeline types, which we don't need.
4. A regular class hides complexity behind a single method call.

5. It's a Facade Pattern, making the system easy to use.

Methods:
- `run`
- **Type:** Instance method
- **What it does:** Orchestrates loading, normalizing, and exporting from a source path.
- **Why we need it:** Provides a simple way to execute the full pipeline.
- **Why an instance method?**
1. It can use instance state, like a `FileManager` or `Normalizer`.
2. If it were static, you'd pass all dependencies every call.
3. If it were a classmethod, it wouldn't belong to a specific manager.
4. An instance method keeps the flow tied to the object.
5. It's more testable and extensible with an instance.

This structure uses nesting (e.g., `Transformer` → `TeamTransformer`) for clarity and lists all required methods with their types and justifications. Each class and method is designed to balance flexibility, readability, and single responsibility. Let me know if you need further refinements!