This documentation describes a series of Python scripts (`demo_1.py`, `demo_2.py`, `demo_3.py`) that demonstrate the use of the `py_trees` library for building behaviour trees. These examples showcase fundamental concepts like actions, conditions, composite nodes (sequences), and decorators (inverters), along with handling long-running behaviours.

1. Concepts

The core concept demonstrated here is a **Behaviour Tree (BT)**, a hierarchical state machine used for controlling autonomous agents (like robots or AI characters). Behaviour trees offer a modular and reactive way to design complex behaviours.

Key concepts from the `py_trees` library utilized in these examples include:

* **Behaviour**: The fundamental building block of a behaviour tree. Every node in the tree is a `Behaviour`. It has a defined lifecycle (`setup`, `initialise`, `update`, `terminate`) and returns a `Status` after execution.
* **Status**: An enumeration representing the outcome of a behaviour's `update` method.
* `SUCCESS`: The behaviour completed its task successfully.
* `FAILURE`: The behaviour failed to complete its task.
* `RUNNING`: The behaviour is currently executing and requires more time (or ticks) to complete.
* `INVALID`: An initial or uninitialised state.
* **Composites**: Nodes that have children and control their execution flow.
* **Sequence**: A composite node that executes its children one by one in order. It returns `RUNNING` if a child returns `RUNNING`, `FAILURE` if any child returns `FAILURE`, and `SUCCESS` only if all children successfully complete. The `memory=True` flag indicates that it remembers the last running child and resumes from there on the next tick.
* **Decorators**: Nodes that have a single child and modify its status or control its execution in some way.
* **Inverter**: A decorator that inverts the `SUCCESS` and `FAILURE` status of its child. If the child returns `SUCCESS`, the Inverter returns `FAILURE`, and vice-versa. If the child returns `RUNNING`, the Inverter also returns `RUNNING`.
* **Behaviour Lifecycle**:
* `setup(**kwargs)`: Called once when the behaviour tree is initialised, typically for one-time resource allocation.
* `initialise()`: Called every time the behaviour transitions from a non-`RUNNING` state (e.g., `INVALID`, `SUCCESS`, `FAILURE`) to potentially `RUNNING` (i.e., when it's first ticked or re-entered). Used for resetting state specific to a single execution.
* `update()`: The main logic of the behaviour. It's called repeatedly as long as the behaviour is active (either `RUNNING` or being evaluated). It must return a `Status`.
* `terminate(new_status)`: Called when the behaviour stops running (either by completing, failing, or being interrupted). It receives the `new_status` that caused the termination.

2. Description

The provided `demo_*.py` files collectively illustrate the construction and execution of simple behaviour trees using `py_trees`. The overarching purpose is to simulate a basic robotic task: checking battery, opening a gripper, approaching an object, and closing the gripper.

* **`demo_1.py` (Basic Execution)**:
This script introduces the fundamental structure of a behaviour tree. It defines generic `Action` and `Condition` behaviours that always return `Status.SUCCESS` after a simulated delay (`sleep(1)`). A

`Sequence` composite node orchestrates these behaviours. The tree is "ticked" only once, demonstrating a single pass through the sequence where all behaviours succeed.

* **`demo_2.py` (Long-Running Actions)**:
This script extends `demo_1.py` by introducing `Action` behaviours that can take multiple "ticks" to complete. The `Action` class is modified with `max_attempt_count` and `attempt_count`. The `update()` method decrements `attempt_count` and returns `Status.RUNNING` until the count reaches zero, at which point it returns `Status.SUCCESS`. This simulates actions that require continuous effort over time. The `make_bt()` function configures different `Action` nodes with varying `max_attempt_count` values. The main execution loop ticks the tree multiple times, demonstrating how the `Sequence` composite handles `RUNNING` children and resumes their execution on subsequent ticks.

* **`demo_3.py` (Decorators and Failure Handling)**:
This script demonstrates the use of a decorator, specifically `py_trees.decorators.Inverter`. In this version, the `Action` behaviours' `update()` method is modified to always return `Status.FAILURE`. To counteract this and allow the `Sequence` to potentially progress, the `make_bt()` function wraps each `Action` node with an `Inverter`. This means that when an `Action` returns `FAILURE`, its parent `Inverter` will return `SUCCESS`, allowing the `Sequence` to proceed to the next child. The tree is ticked once, illustrating how decorators can modify the flow of control based on child status.

3. Structure

Each `demo_*.py` file follows a consistent structure:

1. **Imports**: Necessary components from `time` and `py_trees` are imported.
2. **`Action` Class Definition**: A custom behaviour class inheriting from `py_trees.behaviour.Behaviour`.
* It implements the `__init__`, `setup`, `initialise`, `update`, and `terminate` methods, logging its state transitions.
* `demo_2.py` adds `max_attempt_count` and `attempt_count` attributes.
* `demo_1.py` and `demo_3.py` actions complete in one tick (SUCCESS/FAILURE), while `demo_2.py` actions can take multiple ticks (RUNNING).
3. **`Condition` Class Definition**: Another custom behaviour class inheriting from `py_trees.behaviour.Behaviour`.
* Similar lifecycle methods as `Action`, primarily used for checking states.
* In all demos, `Condition.update()` always returns `Status.SUCCESS`.
4. **`make_bt()` Function**: A factory function responsible for constructing and returning the root of the behaviour tree.
* It instantiates a `py_trees.composites.Sequence` as the root, configured with `memory=True`.
* It then instantiates `Condition` and `Action` (and `Inverter` in `demo_3.py`) behaviours.
* These behaviours are added as children to the root `Sequence`.
5. **Main Execution Block (`if __name__ == "__main__":`)**:
* Sets the `py_trees` logging level to `DEBUG` to observe detailed behaviour tree execution.
* Calls `make_bt()` to create the behaviour tree instance.
* "Ticks" the tree using `tree.tick_once()`.
* `demo_1.py` and `demo_3.py` tick the tree once.
* `demo_2.py` ticks the tree in a loop to demonstrate long-running behaviours.

The overall architecture is a shallow behaviour tree with a single `Sequence` root and a few direct children, some of which might be wrapped in decorators. This structure is simple enough to highlight the core `py_trees` concepts without unnecessary complexity.

4. Key Components

Classes

* **`Action(py_trees.behaviour.Behaviour)`**:
* **Purpose**: Represents a task or action that the agent performs.
* **`__init__(self, name: str, max_attempt_count: int = 1)`**: Constructor. Initializes the behaviour with a given `name`. In `demo_2.py`, `max_attempt_count` is added to control how many ticks an action runs before succeeding, with `attempt_count` tracking current attempts.
* **`setup(self, **kwargs)`**: Logs the setup phase.
* **`initialise(self)`**: Logs the initialization phase. In `demo_2.py`, it resets `attempt_count` to `max_attempt_count`.
* **`update(self)`**: The main logic.
* `demo_1.py`: Sleeps for 1 second, then returns `Status.SUCCESS`.
* `demo_2.py`: Decrements `attempt_count`. Sleeps for 1 second. Returns `Status.SUCCESS` if `attempt_count` is 0, otherwise `Status.RUNNING`.
* `demo_3.py`: Sleeps for 1 second, then returns `Status.FAILURE`.
* **`terminate(self, new_status: py_trees.common.Status)`**: Logs the termination phase and the `new_status`.

* **`Condition(py_trees.behaviour.Behaviour)`**:
* **Purpose**: Represents a check or a pre-condition that must be met for a sequence of actions to proceed.
* **`__init__(self, name: str)`**: Constructor. Initializes the behaviour with a given `name`.
* **`setup(self, **kwargs)`**: Logs the setup phase.
* **`initialise(self)`**: Logs the initialization phase.
* **`update(self)`**: The main logic. Sleeps for 1 second, then consistently returns `Status.SUCCESS` in all demos.
* **`terminate(self, new_status: py_trees.common.Status)`**: Logs the termination phase and the `new_status`.

Functions

* **`make_bt() -> py_trees.behaviour.Behaviour`**:
* **Purpose**: Constructs and returns the root node of the behaviour tree.
* **Details**:
* Creates a `py_trees.composites.Sequence` named "sequence" with `memory=True`.
* Instantiates `Condition` and `Action` nodes: `check_battery`, `open_gripper`, `approach_object`, `close_gripper`.
* In `demo_2.py`, `Action` nodes are initialized with specific `max_attempt_count` values.
* In `demo_3.py`, `Action` nodes are wrapped in `py_trees.decorators.Inverter` instances to reverse their outcome.
* Adds these behaviours (or their `Inverter` wrappers) as children to the `Sequence` root.
* Returns the configured `Sequence` node.

`py_trees` Library Components

* **`py_trees.behaviour.Behaviour`**: The abstract base class for all behaviours.
* **`py_trees.common.Status`**: An enum providing `SUCCESS`, `FAILURE`, `RUNNING`, `INVALID` states.
* **`py_trees.composites.Sequence`**: A composite node that executes children sequentially. `memory=True` ensures it remembers its state across ticks.
* **`py_trees.decorators.Inverter`**: A decorator that inverts the `SUCCESS`/`FAILURE` status of its child (used in `demo_3.py`).
* **`py_trees.logging as log_tree`**: A module for configuring and outputting logging messages from the behaviour tree. `log_tree.level = log_tree.Level.DEBUG` enables detailed logging.