py_trees Documentation

Release 0.6.9

Daniel Stonier

Jan 10, 2021

Guide

CHAPTER 1

Background

1.1 Introduction

Note: Behaviour trees are a decision making engine often used in the gaming industry.

Others include hierarchical finite state machines, task networks, scripting engines all of which have various pros and cons. Behaviour trees sit somewhere in the middle of these allowing you a good blend of purposeful planning towards goals with enough reactivity to shift in the presence of important events. They are also wonderfully simple to compose.

There's much information already covering behaviour trees. Rather than regurgitating it here, dig through some of these first. A good starter is AI GameDev - Behaviour Trees (free signup and login) which puts behaviour trees in context alongside other techniques. A simpler read is Patrick Goebel's Behaviour Trees For Robotics. Other readings are listed at the bottom of this page.

Some standout features of behaviour trees that makes them very attractive:

- **Ticking** - the ability to *tick* allows for work between executions without multi-threading
- **Priority Handling** - switching mechansims that allow higher priority interruptions is very natural
- **Simplicity** - very few core components, making it easy for designers to work with it
- **Dynamic** - change the graph on the fly, between ticks or from parent behaviours themselves

1.2 Motivation

The driving use case for this package was to implement a higher level decision making layer in robotics, i.e. scenarios with some overlap into the control layer. Behaviour trees turned out to be a much more apt fit to handle the many concurrent processes in a robot after attempts with finite state machines became entangled in wiring complexity as the problem grew in scope.

Note: There are very few open behaviour tree implementations.

Most of these have either not progressed significantly (e.g. Owyl), or are accessible only in some niche, e.g. Behaviour Designer, which is a frontend to the trees in the unity framework. Does this mean people do not use them? It is more probable that most behaviour tree implementations happen within the closed doors of gaming/robot companies.

Youtube - Second Generation of Behaviour Trees is an enlightening video about behaviour trees and the developments of the last ten years from an industry expert. It also walks you through a simple c++ implementation. His advice? If you can't find one that fits, roll your own. It is relatively simple and this way you can flexibly cater for your own needs.

1.3 Design

The requirements for the previously discussed robotics use case match that of the more general:

Note: Rapid development of medium scale decision engines that don't need to be real time reactive.

Developers should expect to be able to get up to speed and write their own trees with enough power and flexibility to adapt the library to their needs. Robotics is a good fit. The decision making layer typically does not grow too large (~ hundreds of behaviours) and does not need to handle the reactive decision making that is usually directly incorporated into the controller subsystems. On the other hand, it is not scoped to enable an NPC gaming engine with hundreds of characters and thousands of beahaviours for each character.

This implementation uses all the whizbang tricks (generators, decorators) that python has to offer. Some design constraints that have been assumed to enable a practical, easy to use framework:

- No interaction or sharing of data between tree instances
- No parallelisation of tree execution
- Only one behaviour initialising or executing at a time

Hint: A c++ version is feasible and may come forth if there's a need..

1.4 Readings

- AI GameDev - Behaviour Trees - from a gaming expert, good big picture view
- Youtube - Second Generation of Behaviour Trees - from a gaming expert, in depth c++ walkthrough (on github).
- Behaviour trees for robotics - by pirobot, a clear intro on its usefulness for robots.
- A Curious Course on Coroutines and Concurrency - generators and coroutines in python.
- Behaviour Trees in Robotics and AI - a rather verbose, but chock full with examples and comparisons with other approaches.

CHAPTER 2

Behaviours

A *Behaviour* is the smallest element in a behaviour tree, i.e. it is the *leaf*. Behaviours are usually representative of either a check (am I hungry?), or an action (buy some chocolate cookies).

2.1 Skeleton

Behaviours in py_trees are created by subclassing the *Behaviour* class. A skeleton with informative comments is shown below.

```
# doc/examples/skeleton_behaviour.py

import py_trees
import random


class Foo(py_trees.Behaviour):
    def __init__(self, name):
        """
        Minimal one-time initialisation. A good rule of thumb is
        to only include the initialisation relevant for being able
        to insert this behaviour in a tree for offline rendering to
        dot graphs.

        Other one-time initialisation requirements should be met via
        the setup() method.
        """
        super(Foo, self).__init__(name)

    def setup(self, timeout):
        """
        When is this called?
          This function should be either manually called by your program
          or indirectly called by a parent behaviour when it's own setup
```

(continues on next page)

(continued from previous page)

```
          method has been called.

          If you have vital initialisation here, a useful design pattern
          is to put a guard in your initialise() function to barf the
          first time your behaviour is ticked if setup has not been
          called/succeeded.

        What to do here?
          Delayed one-time initialisation that would otherwise interfere
          with offline rendering of this behaviour in a tree to dot graph.
          Good examples include:
          - Hardware or driver initialisation
          - Middleware initialisation (e.g. ROS pubs/subs/services)
        """
        self.logger.debug("  %s [Foo::setup()]" % self.name)

    def initialise(self):
        """
        When is this called?
          The first time your behaviour is ticked and anytime the
          status is not RUNNING thereafter.

        What to do here?
          Any initialisation you need before putting your behaviour
          to work.
        """
        self.logger.debug("  %s [Foo::initialise()]" % self.name)

    def update(self):
        """
        When is this called?
          Every time your behaviour is ticked.

        What to do here?
          - Triggering, checking, monitoring. Anything...but do not block!
          - Set a feedback message
          - return a py_trees.Status.[RUNNING, SUCCESS, FAILURE]
        """
        self.logger.debug("  %s [Foo::update()]" % self.name)
        ready_to_make_a_decision = random.choice([True, False])
        decision = random.choice([True, False])
        if not ready_to_make_a_decision:
            return py_trees.Status.RUNNING
        elif decision:
            self.feedback_message = "We are not bar!"
            return py_trees.Status.SUCCESS
        else:
            self.feedback_message = "Uh oh"
            return py_trees.Status.FAILURE

    def terminate(self, new_status):
        """
        When is this called?
           Whenever your behaviour switches to a non-running state.
            - SUCCESS || FAILURE : your behaviour's work cycle has finished
            - INVALID : a higher priority branch has interrupted, or shutting down
        """
```

(continues on next page)

(continued from previous page)

```
82          self.logger.debug("  %s [Foo::terminate().terminate()][%s->%s]" % (self.name,␣
    ↪self.status, new_status))
```

2.2 Lifecycle

Getting a feel for how this works in action can be seen by running the *py-trees-demo-behaviour-lifecycle* program (click the link for more detail and access to the sources):

Important points to focus on:

- The `initialise()` method kicks in only when the behaviour is not already running
- The parent `tick()` method is responsible for determining when to call `initialise()`, `stop()` and `terminate()` methods.
- The parent `tick()` method always calls update()
- The `update()` method is responsible for deciding the behaviour *Status*.

2.3 Initialisation

With no less than three methods used for initialisation, it can be difficult to identify where your initialisation code needs to lurk.

Note: `__init__` should instantiate the behaviour sufficiently for offline dot graph generation

---

Later we'll see how we can render trees of behaviours in dot graphs. For now, it is sufficient to understand that you need to keep this minimal enough so that you can generate dot graphs for your trees from something like a CI server (e.g. Jenkins). This is a very useful thing to be able to do.

- No hardware connections that may not be there, e.g. usb lidars
- No middleware connections to other software that may not be there, e.g. ROS pubs/subs/services
- No need to fire up other needlessly heavy resources, e.g. heavy threads in the background

---

Note: `setup` handles all other one-time initialisations of resources that are required for execution

---

Essentially, all the things that the constructor doesn't handle - hardware connections, middleware and other heavy resources.

Note: `initialise` configures and resets the behaviour ready for (repeated) execution

---

Initialisation here is about getting things ready for immediate execution of a task. Some examples:

- Initialising/resetting/clearing variables
- Starting timers
- Just-in-time discovery and establishment of middleware connections

- Sending a goal to start a controller running elsewhere on the system
- ...

2.4 Status

The most important part of a behaviour is the determination of the behaviour's status in the `update()` method. The status gets used to affect which direction of travel is subsequently pursued through the remainder of a behaviour tree. We haven't gotten to trees yet, but it is this which drives the decision making in a behaviour tree.

class py_trees.common.**Status**
An enumerator representing the status of a behaviour

FAILURE = 'FAILURE'
Behaviour check has failed, or execution of its action finished with a failed result.

INVALID = 'INVALID'
Behaviour is uninitialised and inactive, i.e. this is the status before first entry, and after a higher priority switch has occurred.

RUNNING = 'RUNNING'
Behaviour is in the middle of executing some action, result still pending.

SUCCESS = 'SUCCESS'
Behaviour check has passed, or execution of its action has finished with a successful result.

The `update()` method must return one of `RUNNING`. `SUCCESS` or `FAILURE`. A status of `INVALID` is the initial default and ordinarily automatically set by other mechansims (e.g. when a higher priority behaviour cancels the currently selected one).

2.5 Feedback Message

```
1     def initialise(self):
2         """
3         Reset a counter variable.
4         """
```

A behaviour has a naturally built in feedback message that can be cleared in the `initialise()` or `terminate()` methods and updated in the `update()` method.

Tip: Alter a feedback message when **significant events** occur.

The feedback message is designed to assist in notifying humans when a significant event happens or for deciding when to log the state of a tree. If you notify or log every tick, then you end up with alot of noise sorting through an abundance of data in which nothing much is happening to find the one point where something significant occurred that led to surprising or catostrophic behaviour.

Setting the feedback message is usually important when something significant happens in the `RUNNING` state or to provide information associated with the result (e.g. failure reason).

Example - a behaviour responsible for planning motions of a character is in the `RUNNING` state for a long period of time. Avoid updating it with a feedback message at every tick with updated plan details. Instead, update the message whenever a significant change occurs - e.g. when the previous plan is re-planned or pre-empted.

2.6 Loggers

These are used throughout the demo programs. They are not intended to be for anything heavier than debugging simple examples. This kind of logging tends to get rather heavy and requires alot of filtering to find the points of change that you are interested in (see comments about the feedback messages above).

2.7 Complex Example

The *py-trees-demo-action-behaviour* program demonstrates a more complicated behaviour that illustrates a few concepts discussed above, but not present in the very simple lifecycle *Counter* behaviour.

- Mocks an external process and connects to it in the `setup` method
- Kickstarts new goals with the external process in the `initialise` method
- Monitors the ongoing goal status in the `update` method
- Determines `RUNNING`/`SUCCESS` pending feedback from the external process

---

Note: A behaviour's `update()` method never blocks, at most it just monitors the progress and holds up any decision making required by a tree that is ticking the behaviour by setting it's status to `RUNNING`. At the risk of being confusing, this is what is generally referred to as a *blocking* behaviour.

2.8 Meta Behaviours

Attention: This module is the least likely to remain stable in this package. It has only received cursory attention so far and a more thoughtful design for handling behaviour 'hats' might be needful at some point in the future.

Meta behaviours are created by utilising various programming techniques pulled from a magic bag of tricks. Some of these minimise the effort to generate a new behaviour while others provide mechanisms that greatly expand your library of usable behaviours without having to increase the number of explicit behaviours contained therein. The latter is achieved by providing a means for behaviours to wear different 'hats' via python decorators.

Each function or decorator listed below includes its own example code demonstrating its use.

Factories

- *py_trees.meta.create_behaviour_from_function()*
- *py_trees.meta.create_imposter()*

Decorators (Hats)

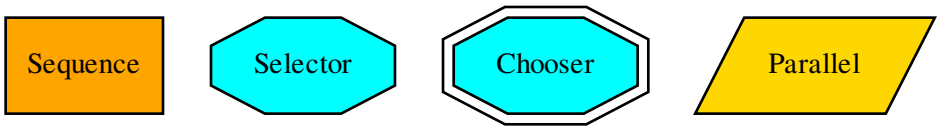
- *py_trees.meta.condition()*

- *py_trees.meta.inverter()*
- *py_trees.meta.failure_is_running()*
- *py_trees.meta.failure_is_success()*
- py_trees.meta.oneshot()
- *py_trees.meta.running_is_failure()*
- *py_trees.meta.running_is_success()*
- *py_trees.meta.success_is_failure()*
- *py_trees.meta.success_is_running()*
- *py_trees.meta.timeout()*

CHAPTER 3

Composites

Composites are the **factories** and **decision makers** of a behaviour tree. They are responsible for shaping the branches.



Tip: You should never need to subclass or create new composites.

Most patterns can be achieved with a combination of the above. Adding to this set exponentially increases the complexity and subsequently making it more difficult to design, introspect, visualise and debug the trees. Always try to find the combination you need to achieve your result before contemplating adding to this set. Actually, scratch that. . . just don't contemplate it!

Composite behaviours typically manage children and apply some logic to the way they execute and return a result, but generally don't do anything themselves. Perform the checks or actions you need to do in the non-composite behaviours.

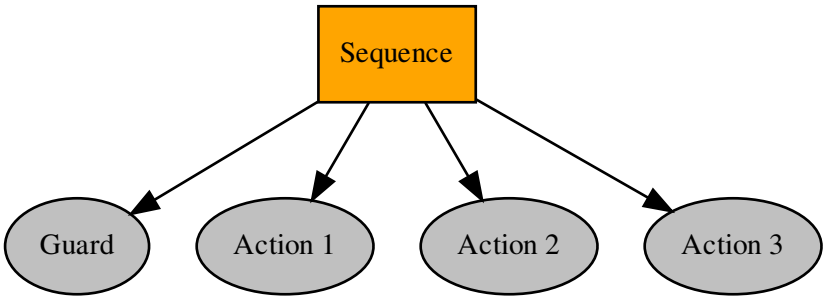
- `Sequence`: execute children sequentially
- `Selector`: select a path through the tree, interruptible by higher priorities
- `Chooser`: like a selector, but commits to a path once started until it finishes
- `Parallel`: manage children concurrently

The subsections below introduce each composite briefly. For a full listing of each composite's methods, visit the *py_trees.composites* module api documentation.

Tip: First time through, make sure to follow the link through to relevant demo programs.

3.1 Sequence

class py_trees.composites.**Sequence**(*name='Sequence'*, *children=None*, **args*, ***kwargs*)
Sequences are the factory lines of Behaviour Trees



A sequence will progressively tick over each of its children so long as each child returns *SUCCESS*. If any child returns *FAILURE* or *RUNNING* the sequence will halt and the parent will adopt the result of this child. If it reaches the last child, it returns with that result regardless.

Note: The sequence halts once it sees a child is RUNNING and then returns the result. *It does not get stuck in the running behaviour*.

See also:

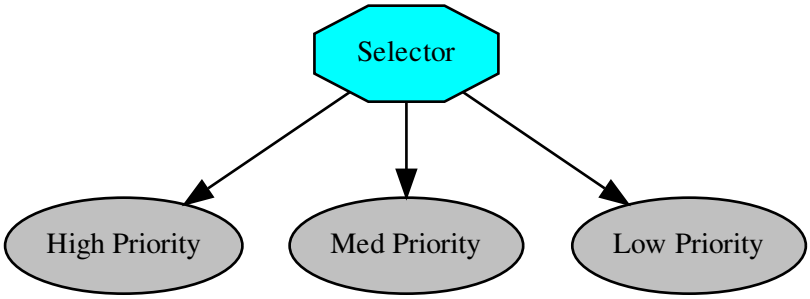
The *py-trees-demo-sequence* program demos a simple sequence in action.

Parameters

- **name** (str) – the composite behaviour name
- **children** ([*Behaviour*]) – list of children to add
- ***args** – variable length argument list
- ****kwargs** – arbitrary keyword arguments

3.2 Selector

class py_trees.composites.**Selector**(*name='Selector'*, *children=None*, **args*, ***kwargs*)
Selectors are the Decision Makers



A selector executes each of its child behaviours in turn until one of them succeeds (at which point it itself returns *RUNNING* or *SUCCESS*, or it runs out of children at which point it itself returns *FAILURE*. We usually refer to selecting children as a means of *choosing between priorities*. Each child and its subtree represent a decreasingly lower priority path.

Note: Switching from a low -> high priority branch causes a *stop(INVALID)* signal to be sent to the previously executing low priority branch. This signal will percolate down that child's own subtree. Behaviours should make sure that they catch this and *destruct* appropriately.

Make sure you do your appropriate cleanup in the `terminate()` methods! e.g. cancelling a running goal, or restoring a context.

See also:

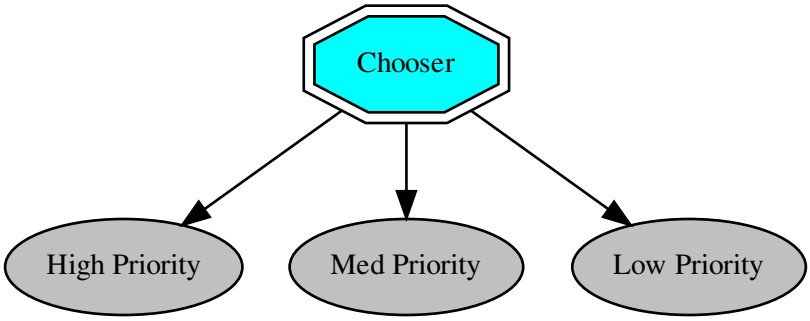
The *py-trees-demo-selector* program demos higher priority switching under a selector.

Parameters

- **name** (`str`) – the composite behaviour name
- **children** ([*Behaviour*]) – list of children to add
- ***args** – variable length argument list
- ****kwargs** – arbitrary keyword arguments

3.3 Chooser

class py_trees.composites.**Chooser**(*name='Chooser'*, *children=None*, **args*, ***kwargs*)
Choosers are Selectors with Commitment



A variant of the selector class. Once a child is selected, it cannot be interrupted by higher priority siblings. As soon as the chosen child itself has finished it frees the chooser for an alternative selection. i.e. priorities only come into effect if the chooser wasn't running in the previous tick.

Note: This is the only composite in py_trees that is not a core composite in most behaviour tree implementations. Nonetheless, this is useful in fields like robotics, where you have to ensure that your manipulator doesn't drop it's payload mid-motion as soon as a higher interrupt arrives. Use this composite sparingly and only if you can't find another way to easily create an elegant tree composition for your task.

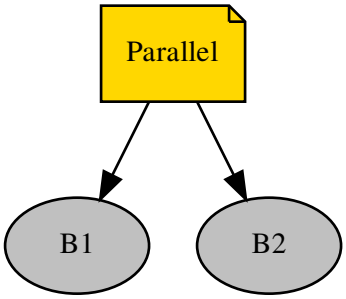
Parameters

- **name** (str) – the composite behaviour name
- **children** ([*Behaviour*]) – list of children to add
- ***args** – variable length argument list
- ****kwargs** – arbitrary keyword arguments

3.4 Parallel

class py_trees.composites.**Parallel**(*name='Parallel'*, *policy=<ParallelPolicy.SUCCESS_ON_ALL: 'SUCCESS_ON_ALL'>*, *children=None*, **args*, ***kwargs*)

Parallels enable a kind of concurrency



Ticks every child every time the parallel is run (a poor man's form of paralellism).

- Parallels will return *FAILURE* if any child returns *FAILURE*
- Parallels with policy *SUCCESS_ON_ONE* return *SUCCESS* if **at least one** child returns *SUCCESS* and others are *RUNNING*.
- Parallels with policy *SUCCESS_ON_ALL* only returns *SUCCESS* if **all** children return *SUCCESS*

See also:

The *py-trees-demo-context-switching* program demos a parallel used to assist in a context switching scenario.

Parameters

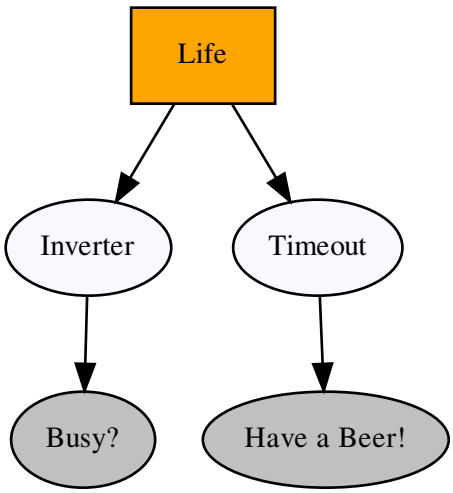
- **name** (str) – the composite behaviour name
- **policy** (*ParallelPolicy*) – policy to use for deciding success or otherwise
- **children** ([*Behaviour*]) – list of children to add
- ***args** – variable length argument list
- ****kwargs** – arbitrary keyword arguments

CHAPTER 4

Decorators

Decorators are behaviours that manage a single child and provide common modifications to their underlying child behaviour (e.g. inverting the result). i.e. they provide a means for behaviours to wear different 'hats' depending on their context without a behaviour tree.

An example:




```
#!/usr/bin/env python

import py_trees.decorators
import py_trees.display

if __name__ == '__main__':

    root = py_trees.composites.Sequence(name="Life")
    timeout = py_trees.decorators.Timeout(
        name="Timeout",
        child=py_trees.behaviours.Success(name="Have a Beer!")
    )
    failure_is_success = py_trees.decorators.Inverter(
        name="Inverter",
        child=py_trees.behaviours.Success(name="Busy?")
        )
    root.add_children([failure_is_success, timeout])
    py_trees.display.render_dot_tree(root)
```

Decorators (Hats)

Decorators with very specific functionality:

- *py_trees.decorators.Condition()*
- *py_trees.decorators.Inverter()*
- *py_trees.decorators.OneShot()*
- py_trees.decorators.TimeOut()

And the X is Y family:

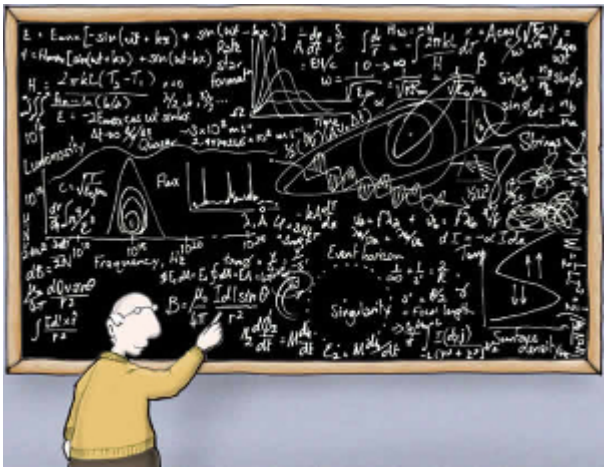
- *py_trees.decorators.FailureIsRunning()*
- *py_trees.decorators.FailureIsSuccess()*

- *py_trees.decorators.RunningIsFailure()*
- *py_trees.decorators.RunningIsSuccess()*
- *py_trees.decorators.SuccessIsFailure()*
- *py_trees.decorators.SuccessIsRunning()*

CHAPTER 5

Blackboards

Blackboards are not a necessary component, but are a fairly standard feature in most behaviour tree implementations. See, for example, the design notes for blackboards in Unreal Engine.



Implementations however, tend to vary quite a bit depending on the needs of the framework using them. Some of the usual considerations include scope and sharing of blackboards across multiple tree instances.

For this package, we've decided to keep blackboards extremely simple to fit with the same 'rapid development for small scale systems' principles that this library is designed for.

- No sharing between tree instances
- No locking for reading/writing
- Global scope, i.e. any behaviour can access any variable
- No external communications (e.g. to a database)

class py_trees.blackboard.**Blackboard**
Borg style key-value store for sharing amongst behaviours.

Examples

You can instantiate the blackboard from anywhere in your program. Even disconnected calls will get access to the same data store. For example:

```
def check_foo():
    blackboard = Blackboard()
    assert(blackboard.foo, "bar")

if __name__ == '__main__':
    blackboard = Blackboard()
    blackboard.foo = "bar"
    check_foo()
```

If the key value you are interested in is only known at runtime, then you can set/get from the blackboard without the convenient variable style access:

```
blackboard = Blackboard()
result = blackboard.set("foo", "bar")
foo = blackboard.get("foo")
```

The blackboard can also be converted and printed (with highlighting) as a string. This is useful for logging and debugging.

```
print(Blackboard())
```

Warning: Be careful of key collisions. This implementation leaves this management up to the user.

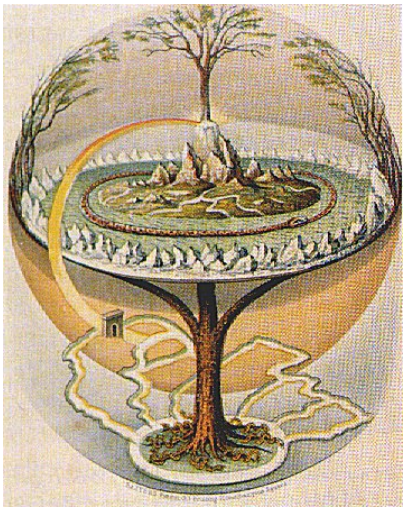
See also:

The *py-trees-demo-blackboard* program demos use of the blackboard along with a couple of the blackboard behaviours.

CHAPTER 6

Trees

While a graph of connected behaviours and composites form a tree in their own right (i.e. it can be initialised and ticked), it is usually convenient to wrap your tree in another class to take care of alot of the housework and provide some extra bells and whistles that make your tree flourish.



This package provides a default reference implementation that is directly usable, but can also be easily used as inspiration for your own tree custodians.

6.1 The Behaviour Tree

class py_trees.trees.**BehaviourTree**(*root*)

Grow, water, prune your behaviour tree with this, the default reference implementation. It features a few enhancements to provide richer logging, introspection and dynamic management of the tree itself:

- Pre and post tick handlers to execute code automatically before and after a tick

- Visitor access to the parts of the tree that were traversed in a tick
- Subtree pruning and insertion operations
- Continuous tick-tock support

See also:

The *py-trees-demo-tree-stewardship* program demonstrates the above features.

Parameters **root** (*Behaviour*) – root node of the tree

Variables

- **count** (int) – number of times the tree has been ticked.
- **root** (*Behaviour*) – root node of the tree
- ***visitors*** ([*visitors*]) – entities that visit traversed parts of the tree when it ticks
- **pre_tick_handlers** ([func]) – functions that run before the entire tree is ticked
- **post_tick_handlers** ([func]) – functions that run after the entire tree is ticked

Raises AssertionError – if incoming root variable is not the correct type

6.2 Skeleton

The most basic feature of the behaviour tree is it's automatic tick-tock. You can *tick_tock()* for a specific number of iterations, or indefinitely and use the *interrupt()* method to stop it.

```
#!/usr/bin/env python

import py_trees

if __name__ == '__main__':

    root = py_trees.composites.Selector("Selector")
    high = py_trees.behaviours.Success(name="High Priority")
    med = py_trees.behaviours.Success(name="Med Priority")
    low = py_trees.behaviours.Success(name="Low Priority")
    root.add_children([high, med, low])

    behaviour_tree = py_trees.trees.BehaviourTree(root)
    behaviour_tree.setup(15)
    try:
        behaviour_tree.tick_tock(
            sleep_ms=500,
            number_of_iterations=py_trees.trees.CONTINUOUS_TICK_TOCK,
            pre_tick_handler=None,
            post_tick_handler=None
        )
    except KeyboardInterrupt:
        behaviour_tree.interrupt()
```

or create your own loop and tick at your own leisure with the *tick()* method.

6.3 Pre/Post Tick Handlers

Pre and post tick handlers can be used to perform some activity on or with the tree immediately before and after ticking. This is mostly useful with the continuous `tick_tock()` mechanism.

This is useful for a variety of purposes:

- logging
- doing introspection on the tree to make reports
- extracting data from the blackboard
- triggering on external conditions to modify the tree (e.g. new plan arrived)

This can be done of course, without locking since the tree won't be ticking while these handlers run. This does however, mean that your handlers should be light. They will be consuming time outside the regular tick period.

The *py-trees-demo-tree-stewardship* program demonstrates a very simple pre-tick handler that just prints a line to stdout notifying the user of the current run. The relevant code:

Listing 1: pre-tick-handler-function

```

def pre_tick_handler(behaviour_tree):
    """
    This prints a banner and will run immediately before every tick of the tree.

    Args:
        behaviour_tree (:class:`~py_trees.trees.BehaviourTree`): the tree custodian

    """
    print("\n--------- Run %s ---------\n" % behaviour_tree.count)
```

Listing 2: pre-tick-handler-adding

```
    # Rendering
    ####################
```

6.4 Visitors

Visitors are entities that can be passed to a tree implementation (e.g. `BehaviourTree`) and used to either visit each and every behaviour in the tree, or visit behaviours as the tree is traversed in an executing tick. At each behaviour, the visitor runs its own method on the behaviour to do as it wishes - logging, introspecting, etc.

Warning: Visitors should not modify the behaviours they visit.

The *py-trees-demo-tree-stewardship* program demonstrates the two reference visitor implementations:

- `DebugVisitor` prints debug logging messages to stdout and
- `SnapshotVisitor` collects runtime data to be used by visualisations

Adding visitors to a tree:

```
behaviour_tree = py_trees.trees.BehaviourTree(root)
behaviour_tree.visitors.append(py_trees.visitors.DebugVisitor())
snapshot_visitor = py_trees.visitors.SnapshotVisitor()
behaviour_tree.visitors.append(snapshot_visitor)
```

These visitors are automatically run inside the tree's *tick* method. The former immediately logs to screen, the latter collects information which is then used to display an ascii tree:

```
behaviour_tree.tick()
ascii_tree = py_trees.display.ascii_tree(
    behaviour_tree.root,
    snapshot_information=snapshot_visitor)
)
print(ascii_tree)
```

CHAPTER 7

Visualisation

Behaviour trees are significantly easier to design, monitor and debug with visualisations. Py Trees does provide minimal assistance to render trees to various simple output formats. Currently this includes dot graphs, strings or stdout.

7.1 Ascii Trees

You can get a very simple ascii representation of the tree on stdout with *print_ascii_tree()*:

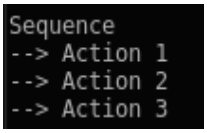
py_trees.display.**print_ascii_tree**(*root*, *indent=0*, *show_status=False*)
Print the ASCII representation of an entire behaviour tree.

Parameters

- **root** (*Behaviour*) – the root of the tree, or subtree you want to show
- **indent** (int) – the number of characters to indent the tree
- **show_status** (bool) – additionally show feedback message and status of every element

Examples

Render a simple tree in ascii format to stdout.




```
root = py_trees.composites.Sequence("Sequence")
for action in ["Action 1", "Action 2", "Action 3"]:
    b = py_trees.behaviours.Count(
            name=action,
```

(continues on next page)

(continued from previous page)

```
            fail_until=0,
            running_until=1,
            success_until=10)
    root.add_child(b)
py_trees.display.print_ascii_tree(root)
```

Tip: To additionally display status and feedbback message from every behaviour in the tree, simply set the `show_status` flag to True.

7.2 Ascii Trees (Runtime)

When a tree is ticking, it is important to be able to catch the status and feedback message from each behaviour that has been traversed. You can do this by using the *SnapshotVisitor* in conjunction with the *ascii_tree()* function:

py_trees.display.**ascii_tree**(*tree*, *indent=0*, *snapshot_information=None*)

Build an ascii tree representation as a string for redirecting to elsewhere other than stdout. This can be the entire tree, or a recorded snapshot of the tree (i.e. just the part that was traversed).

Parameters

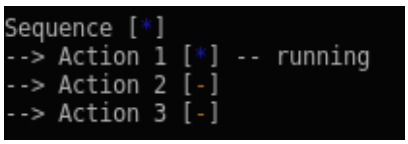
- **tree** (*Behaviour*) – the root of the tree, or subtree you want to show
- **indent** (int) – the number of characters to indent the tree
- **snapshot_information** (*visitors*) – a visitor that recorded information about a traversed tree (e.g. *SnapshotVisitor*)
- **snapshot_information** – a visitor that recorded information about a traversed tree (e.g. *SnapshotVisitor*)

Returns an ascii tree (i.e. in string form)

Return type str

Examples

Use the *SnapshotVisitor* and *BehaviourTree* to generate snapshot information at each tick and feed that to a post tick handler that will print the traversed ascii tree complete with status and feedback messages.




```
def post_tick_handler(snapshot_visitor, behaviour_tree):
    print(py_trees.display.ascii_tree(behaviour_tree.root,
          snapshot_information=snapshot_visitor))

root = py_trees.composites.Sequence("Sequence")
for action in ["Action 1", "Action 2", "Action 3"]:
    b = py_trees.behaviours.Count(
            name=action,
```

(continues on next page)

(continued from previous page)

```
            fail_until=0,
            running_until=1,
            success_until=10)
    root.add_child(b)
behaviour_tree = py_trees.trees.BehaviourTree(root)
snapshot_visitor = py_trees.visitors.SnapshotVisitor()
behaviour_tree.add_post_tick_handler(
    functools.partial(post_tick_handler,
                      snapshot_visitor))
behaviour_tree.visitors.append(snapshot_visitor)
```

7.3 Render to File (Dot/SVG/PNG)

API

You can render trees into dot/png/svg files simply by calling the *render_dot_tree()* function.

Should you wish to capture the dot graph result directly (as a dot graph object), use the *generate_pydot_graph()* method.

Command Line Utility

You can also render any exposed method in your python packages that creates a tree and returns the root of the tree from the command line using the *py-trees-render* program.

Blackboxes and Visibility Levels

There is also an experimental feature that allows you to flag behaviours as blackboxes with multiple levels of granularity. This is purely for the purposes of showing different levels of detail in rendered dot graphs. A fullly rendered dot graph with hundreds of behaviours is not of much use when wanting to visualise the big picture.

The *py-trees-demo-dot-graphs* program serves as a self-contained example of this feature.

CHAPTER 8

Surviving the Crazy Hospital

Your behaviour trees are misbehaving or your subtree designs seem overly obtuse? This page can help you stay focused on what is important. . . staying out of the padded room.

Note: Many of these guidelines we've evolved from trial and error and are almost entirely driven by a need to avoid a burgeoning complexity (aka *flying spaghetti monster*). Feel free to experiment and provide us with your insights here as well!

8.1 Behaviours

- Keep the constructor minimal so you can instantiate the behaviour for offline rendering
- Put hardware or other runtime specific initialisation in `setup()`
- Update `feedback_message` for *significant events* only so you don't end up with too much noise
- The `update()` method must be light and non-blocking so a tree can keep ticking over

- Keep the scope of a single behaviour tight and focused, deploy larger concepts as subtrees

8.2 Composites

- Avoid creating new composites, this increases the decision complexity by an order of magnitude
- Don't subclass merely to auto-populate it, build a `create_<xyz>_subtree()` library instead

8.3 Trees

- Make sure your pre/post tick handlers and visitors are all very light.
- A good tick-tock rate for higher level decision making is around 500ms.

CHAPTER 9

Terminology

blocking A behaviour is sometimes referred to as a 'blocking' behaviour. Technically, the execution of a behaviour should be non-blocking (i.e. the tick part), however when it's progress from 'RUNNING' to 'FAILURE/SUCCESS' takes more than one tick, we say that the behaviour itself is blocking. In short, *blocking == RUNNING*.

fsm

flying spaghetti monster Whilst a serious religous entity in his own right (see pastafarianism), it's also very easy to imagine your code become a spiritual flying spaghetti monster if left unchecked:

```
  _  _(o)_(o)_  _
._\`:_ F S M _:' \_,
    / (`---'\ `-.
 ,-`  _)    (_,
```

tick

ticks

ticking A key feature of behaviours and their trees is in the way they *tick*. A tick is merely an execution slice, similar to calling a function once, or executing a loop in a control program once.

When a **behaviour** ticks, it is executing a small, non-blocking chunk of code that checks a variable or triggers/monitors/returns the result of an external action.

When a **behaviour tree** ticks, it traverses the behaviours (starting at the root of the tree), ticking each behaviour, catching its result and then using that result to make decisions on the direction the tree traversal will take. This is the decision part of the tree. Once the traversal ends back at the root, the tick is over.

Once a tick is done..you can stop for breath! In this space you can pause to avoid eating the cpu, send some statistics out to a monitoring program, manipulate the underlying blackboard (data), ... At no point does the traversal of the tree get mired in execution - it's just in and out and then stop for a coffee. This is absolutely awesome - without this it would be a concurrent mess of locks and threads.

Always keep in mind that your behaviours' executions must be light. There is no parallelising here and your tick time needs to remain small. The tree should be solely about decision making, not doing any actual blocking

work. Any blocking work should be happening somewhere else with a behaviour simply in charge of starting/monitoring and catching the result of that work.

Add an image of a ticking tree here.

CHAPTER 10

FAQ

Tip: For hints and guidelines, you might also like to browse *Surviving the Crazy Hospital*.

Will there be a c++ implementation?

Certainly feasible and if there's a need. If such a things should come to pass though, the c++ implementation should compliment this one. That is, it should focus on decision making for systems with low latency and reactive requirements. It would use triggers to tick the tree instead of tick-tock and a few other tricks that have evolved in the gaming industry over the last few years. Having a c++ implementation for use in the control layer of a robotics system would be a driving use case.

CHAPTER 11

Demos

11.1 py-trees-demo-action-behaviour

Demonstrates the characteristics of a typical 'action' behaviour.

- Mocks an external process and connects to it in the setup() method
- Kickstarts new goals with the external process in the initialise() method
- Monitors the ongoing goal status in the update() method
- Determines RUNNING/SUCCESS pending feedback from the external process

```
usage: py-trees-demo-action-behaviour [-h]
```

class py_trees.demos.action.**Action**(*name='Action'*)

Bases: *py_trees.behaviour.Behaviour*

Connects to a subprocess to initiate a goal, and monitors the progress of that goal at each tick until the goal is completed, at which time the behaviour itself returns with success or failure (depending on success or failure of the goal itself).

This is typical of a behaviour that is connected to an external process responsible for driving hardware, conducting a plan, or a long running processing pipeline (e.g. planning/vision).

Key point - this behaviour itself should not be doing any work!

__init__(*name='Action'*)

Default construction.

initialise()

Reset a counter variable.

setup(*unused_timeout=15*)

No delayed initialisation required for this example.

terminate(*new_status*)
Nothing to clean up in this example.

update()
Increment the counter and decide upon a new status result for the behaviour.

py_trees.demos.action.**main**()
Entry point for the demo script.

py_trees.demos.action.**planning**(*pipe_connection*)
Emulates an external process which might accept long running planning jobs.

Listing 1: py_trees/demos/action.py

```
#!/usr/bin/env python
#
# License: BSD
#   https://raw.githubusercontent.com/stonier/py_trees/devel/LICENSE
#
##############################################################################
# Documentation
##############################################################################

"""
.. argparse::
   :module: py_trees.demos.action
   :func: command_line_argument_parser
   :prog: py-trees-demo-action-behaviour

.. image:: images/action.gif
"""

##############################################################################
# Imports
##############################################################################

import argparse
import atexit
import multiprocessing
import py_trees
import time

import py_trees.console as console

##############################################################################
# Classes
##############################################################################


def description():
    content = "Demonstrates the characteristics of a typical 'action' behaviour.\n"
    content += "\n"
    content += "* Mocks an external process and connects to it in the setup() method\n"
    content += "* Kicksstarts new goals with the external process in the initialise() method\n"
    content += "* Monitors the ongoing goal status in the update() method\n"
    content += "* Determines RUNNING/SUCCESS pending feedback from the external process\n"
```

(continues on next page)

(continued from previous page)

```
43
44      if py_trees.console.has_colours:
45          banner_line = console.green + "*" * 79 + "\n" + console.reset
46          s = "\n"
47          s += banner_line
48          s += console.bold_white + "Action Behaviour".center(79) + "\n" + console.reset
49          s += banner_line
50          s += "\n"
51          s += content
52          s += "\n"
53          s += banner_line
54      else:
55          s = content
56      return s
57
58
59  def epilog():
60      if py_trees.console.has_colours:
61          return console.cyan + "And his noodly appendage reached forth to tickle the␣
    ↪blessed...\n" + console.reset
62      else:
63          return None
64
65
66  def command_line_argument_parser():
67      return argparse.ArgumentParser(description=description(),
68                                     epilog=epilog(),
69                                     formatter_class=argparse.
    ↪RawDescriptionHelpFormatter,
70                                     )
71
72
73  def planning(pipe_connection):
74      """
75      Emulates an external process which might accept long running planning jobs.
76      """
77      idle = True
78      percentage_complete = 0
79      try:
80          while(True):
81              if pipe_connection.poll():
82                  pipe_connection.recv()
83                  percentage_complete = 0
84                  idle = False
85              if not idle:
86                  percentage_complete += 10
87                  pipe_connection.send([percentage_complete])
88                  if percentage_complete == 100:
89                      idle = True
90              time.sleep(0.5)
91      except KeyboardInterrupt:
92          pass
93
94
95  class Action(py_trees.behaviour.Behaviour):
96      """
97      Connects to a subprocess to initiate a goal, and monitors the progress
```

(continues on next page)

(continued from previous page)

```
    of that goal at each tick until the goal is completed, at which time
    the behaviour itself returns with success or failure (depending on
    success or failure of the goal itself).

    This is typical of a behaviour that is connected to an external process
    responsible for driving hardware, conducting a plan, or a long running
    processing pipeline (e.g. planning/vision).

    Key point - this behaviour itself should not be doing any work!
    """
    def __init__(self, name="Action"):
        """
        Default construction.
        """
        super(Action, self).__init__(name)
        self.logger.debug("%s.__init__()" % (self.__class__.__name__))

    def setup(self, unused_timeout=15):
        """
        No delayed initialisation required for this example.
        """
        self.logger.debug("%s.setup()->connections to an external process" % (self.__
↪class__.__name__))
        self.parent_connection, self.child_connection = multiprocessing.Pipe()
        self.planning = multiprocessing.Process(target=planning, args=(self.child_
↪connection,))
        atexit.register(self.planning.terminate)
        self.planning.start()
        return True

    def initialise(self):
        """
        Reset a counter variable.
        """
        self.logger.debug("%s.initialise()->sending new goal" % (self.__class__.__
↪name__))
        self.parent_connection.send(['new goal'])
        self.percentage_completion = 0

    def update(self):
        """
        Increment the counter and decide upon a new status result for the behaviour.
        """
        new_status = py_trees.Status.RUNNING
        if self.parent_connection.poll():
            self.percentage_completion = self.parent_connection.recv().pop()
            if self.percentage_completion == 100:
                new_status = py_trees.Status.SUCCESS
        if new_status == py_trees.Status.SUCCESS:
            self.feedback_message = "Processing finished"
            self.logger.debug("%s.update()[%s->%s][%s]" % (self.__class__.__name__,␣
↪self.status, new_status, self.feedback_message))
        else:
            self.feedback_message = "{0}%".format(self.percentage_completion)
            self.logger.debug("%s.update()[%s][%s]" % (self.__class__.__name__, self.
↪status, self.feedback_message))
        return new_status
```

(continues on next page)

(continued from previous page)

```
150
151     def terminate(self, new_status):
152         """
153         Nothing to clean up in this example.
154         """
155         self.logger.debug("%s.terminate()[%s->%s]" % (self.__class__.__name__, self.
    ↪status, new_status))
156
157
158 ##############################################################################
159 # Main
160 ##############################################################################
161
162 def main():
163     """
164     Entry point for the demo script.
165     """
166     command_line_argument_parser().parse_args()
167
168     print(description())
169
170     py_trees.logging.level = py_trees.logging.Level.DEBUG
171
172     action = Action()
173     action.setup()
174     try:
175         for unused_i in range(0, 12):
176             action.tick_once()
177             time.sleep(0.5)
178         print("\n")
179     except KeyboardInterrupt:
180         pass
```

11.2 py-trees-demo-behaviour-lifecycle

Demonstrates a typical day in the life of a behaviour.

This behaviour will count from 1 to 3 and then reset and repeat. As it does so, it logs and displays the methods as they are called - construction, setup, initialisation, ticking and termination.

```
usage: py-trees-demo-behaviour-lifecycle [-h]
```

class py_trees.demos.lifecycle.**Counter**(*name='Counter'*)

Bases: *py_trees.behaviour.Behaviour*

Simple counting behaviour that facilitates the demonstration of a behaviour in the demo behaviours lifecycle program.

- Increments a counter from zero at each tick
- Finishes with success if the counter reaches three
- Resets the counter in the initialise() method.

__init__(*name='Counter'*)
Default construction.

initialise()
Reset a counter variable.

setup(*unused_timeout=15*)
No delayed initialisation required for this example.

terminate(*new_status*)
Nothing to clean up in this example.

update()
Increment the counter and decide upon a new status result for the behaviour.

py_trees.demos.lifecycle.**main**()
Entry point for the demo script.

Listing 2: py_trees/demos/lifecycle.py

```
#!/usr/bin/env python
#
# License: BSD
#   https://raw.githubusercontent.com/stonier/py_trees/devel/LICENSE
#
##############################################################################
# Documentation
##############################################################################

"""
.. argparse::
   :module: py_trees.demos.lifecycle
   :func: command_line_argument_parser
   :prog: py-trees-demo-behaviour-lifecycle

.. image:: images/lifecycle.gif
"""

##############################################################################
# Imports
##############################################################################

import argparse
import py_trees
import time

import py_trees.console as console

##############################################################################
# Classes
##############################################################################


def description():
    content = "Demonstrates a typical day in the life of a behaviour.\n\n"
    content += "This behaviour will count from 1 to 3 and then reset and repeat. As␣
↪it does\n"
    content += "so, it logs and displays the methods as they are called -␣
↪construction, setup,\n"
```

(continues on next page)

(continued from previous page)

```
    content += "initialisation, ticking and termination.\n"
    if py_trees.console.has_colours:
        banner_line = console.green + "*" * 79 + "\n" + console.reset
        s = "\n"
        s += banner_line
        s += console.bold_white + "Behaviour Lifecycle".center(79) + "\n" + console.reset
        s += banner_line
        s += "\n"
        s += content
        s += "\n"
        s += banner_line
    else:
        s = content
    return s


def epilog():
    if py_trees.console.has_colours:
        return console.cyan + "And his noodly appendage reached forth to tickle the blessed...\n" + console.reset
    else:
        return None


def command_line_argument_parser():
    return argparse.ArgumentParser(description=description(),
                                   epilog=epilog(),
                                   formatter_class=argparse.RawDescriptionHelpFormatter,
                                   )


class Counter(py_trees.behaviour.Behaviour):
    """
    Simple counting behaviour that facilitates the demonstration of a behaviour in
    the demo behaviours lifecycle program.

    * Increments a counter from zero at each tick
    * Finishes with success if the counter reaches three
    * Resets the counter in the initialise() method.
    """
    def __init__(self, name="Counter"):
        """
        Default construction.
        """
        super(Counter, self).__init__(name)
        self.logger.debug("%s.__init__()" % (self.__class__.__name__))

    def setup(self, unused_timeout=15):
        """
        No delayed initialisation required for this example.
        """
        self.logger.debug("%s.setup()" % (self.__class__.__name__))
        return True

    def initialise(self):
```

(continues on next page)

(continued from previous page)

```
        """
        Reset a counter variable.
        """
        self.logger.debug("%s.initialise()" % (self.__class__.__name__))
        self.counter = 0

    def update(self):
        """
        Increment the counter and decide upon a new status result for the behaviour.
        """
        self.counter += 1
        new_status = py_trees.Status.SUCCESS if self.counter == 3 else py_trees.
↪Status.RUNNING
        if new_status == py_trees.Status.SUCCESS:
            self.feedback_message = "counting...{0} - phew, thats enough for today".
↪format(self.counter)
        else:
            self.feedback_message = "still counting"
        self.logger.debug("%s.update()[%s->%s][%s]" % (self.__class__.__name__, self.
↪status, new_status, self.feedback_message))
        return new_status

    def terminate(self, new_status):
        """
        Nothing to clean up in this example.
        """
        self.logger.debug("%s.terminate()[%s->%s]" % (self.__class__.__name__, self.
↪status, new_status))


##############################################################################
# Main
##############################################################################

def main():
    """
    Entry point for the demo script.
    """
    command_line_argument_parser().parse_args()

    print(description())

    py_trees.logging.level = py_trees.logging.Level.DEBUG

    counter = Counter()
    counter.setup()
    try:
        for unused_i in range(0, 7):
            counter.tick_once()
            time.sleep(0.5)
        print("\n")
    except KeyboardInterrupt:
        print("")
        pass
```

11.3 py-trees-demo-blackboard

Demonstrates usage of the blackboard and related behaviours.

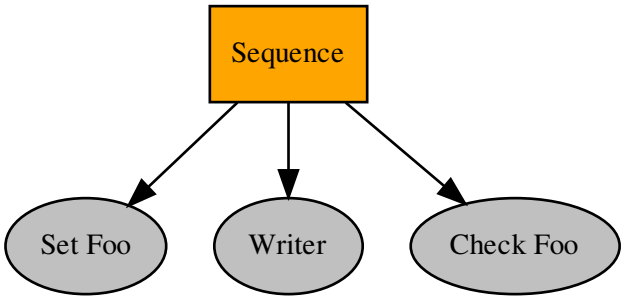
A sequence is populated with a default set blackboard variable behaviour, a custom write to blackboard behaviour that writes a more complicated structure, and finally a default check blackboard variable beheaviour that looks for the first variable.

```
usage: py-trees-demo-blackboard [-h] [-r]
```

11.3.1 Named Arguments

-r, --render render dot tree to file

Default: False



class py_trees.demos.blackboard.**BlackboardWriter**(*name='Writer'*)

Bases: *py_trees.behaviour.Behaviour*

Custom writer that submits a more complicated variable to the blackboard.

__init__(*name='Writer'*)

Initialize self. See help(type(self)) for accurate signature.

update()

Write a dictionary to the blackboard and return SUCCESS.

py_trees.demos.blackboard.**main**()

Entry point for the demo script.

Listing 3: py_trees/demos/blackboard.py

```
#!/usr/bin/env python
#
# License: BSD
#   https://raw.githubusercontent.com/stonier/py_trees/devel/LICENSE
#
##############################################################################
```

(continues on next page)

(continued from previous page)

```
# Documentation
##############################################################################

"""
.. argparse::
   :module: py_trees.demos.blackboard
   :func: command_line_argument_parser
   :prog: py-trees-demo-blackboard

.. graphviz:: dot/demo-blackboard.dot

.. image:: images/blackboard.gif
"""

##############################################################################
# Imports
##############################################################################

import argparse
import py_trees
import sys

import py_trees.console as console

##############################################################################
# Classes
##############################################################################


def description():
    content = "Demonstrates usage of the blackboard and related behaviours.\n"
    content += "\n"
    content += "A sequence is populated with a default set blackboard variable\n"
    content += "behaviour, a custom write to blackboard behaviour that writes\n"
    content += "a more complicated structure, and finally a default check\n"
    content += "blackboard variable beheaviour that looks for the first variable.\n"

    if py_trees.console.has_colours:
        banner_line = console.green + "*" * 79 + "\n" + console.reset
        s = "\n"
        s += banner_line
        s += console.bold_white + "Blackboard".center(79) + "\n" + console.reset
        s += banner_line
        s += "\n"
        s += content
        s += "\n"
        s += banner_line
    else:
        s = content
    return s


def epilog():
    if py_trees.console.has_colours:
        return console.cyan + "And his noodly appendage reached forth to tickle the␣
↪blessed...\n" + console.reset
    else:
```

(continues on next page)

(continued from previous page)

```
        return None


def command_line_argument_parser():
    parser = argparse.ArgumentParser(description=description(),
                                     epilog=epilog(),
                                     formatter_class=argparse.
↪RawDescriptionHelpFormatter,
                                     )
    parser.add_argument('-r', '--render', action='store_true', help='render dot tree␣
↪to file')
    return parser


class BlackboardWriter(py_trees.behaviour.Behaviour):
    """
    Custom writer that submits a more complicated variable to the blackboard.
    """
    def __init__(self, name="Writer"):
        super(BlackboardWriter, self).__init__(name)
        self.logger.debug("%s.__init__()" % (self.__class__.__name__))
        self.blackboard = py_trees.blackboard.Blackboard()

    def update(self):
        """
        Write a dictionary to the blackboard and return :data:`~py_trees.Status.
↪SUCCESS`.
        """
        self.logger.debug("%s.update()" % (self.__class__.__name__))
        self.blackboard.spaghetti = {"type": "Gnocchi", "quantity": 2}
        return py_trees.Status.SUCCESS


def create_tree():
    root = py_trees.composites.Sequence("Sequence")
    set_blackboard_variable = py_trees.blackboard.SetBlackboardVariable(name="Set Foo
↪", variable_name="foo", variable_value="bar")
    write_blackboard_variable = BlackboardWriter(name="Writer")
    check_blackboard_variable = py_trees.blackboard.CheckBlackboardVariable(name=
↪"Check Foo", variable_name="foo", expected_value="bar")
    root.add_children([set_blackboard_variable, write_blackboard_variable, check_
↪blackboard_variable])
    return root


##############################################################################
# Main
##############################################################################

def main():
    """
    Entry point for the demo script.
    """
    args = command_line_argument_parser().parse_args()
    print(description())
    py_trees.logging.level = py_trees.logging.Level.DEBUG

```

(continues on next page)

(continued from previous page)

```
    tree = create_tree()

    ####################
    # Rendering
    ####################
    if args.render:
        py_trees.display.render_dot_tree(tree)
        sys.exit()

    ####################
    # Execute
    ####################
    tree.setup(timeout=15)
    print("\n--------- Tick 0 ---------\n")
    tree.tick_once()
    print("\n")
    py_trees.display.print_ascii_tree(tree, show_status=True)
    print("\n")
    print(py_trees.blackboard.Blackboard())
```

11.4 py-trees-demo-context-switching

Demonstrates context switching with parallels and sequences.

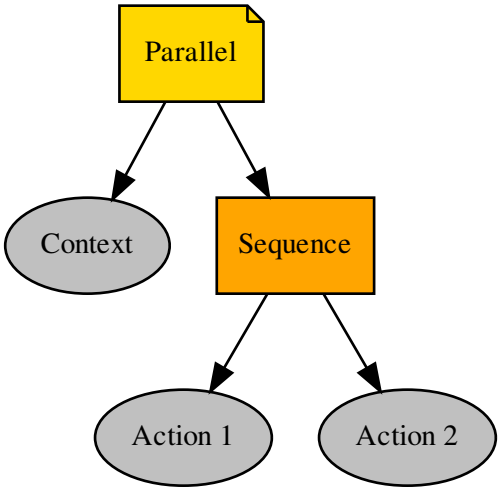
A context switching behaviour is run in parallel with a work sequence. Switching the context occurs in the initialise() and terminate() methods of the context switching behaviour. Note that whether the sequence results in failure or success, the context switch behaviour will always call the terminate() method to restore the context. It will also call terminate() to restore the context in the event of a higher priority parent cancelling this parallel subtree.

```
usage: py-trees-demo-context-switching [-h] [-r]
```

11.4.1 Named Arguments

-r, --render render dot tree to file

Default: False



class py_trees.demos.context_switching.**ContextSwitch**(*name='ContextSwitch'*)
Bases: *py_trees.behaviour.Behaviour*

An example of a context switching class that sets (in `initialise()`) and restores a context (in `terminate()`). Use in parallel with a sequence/subtree that does the work while in this context.

> **Attention:** Simply setting a pair of behaviours (set and reset context) on either end of a sequence will not suffice for context switching. In the case that one of the work behaviours in the sequence fails, the final reset context switch will never trigger.

__init__(*name='ContextSwitch'*)
Initialize self. See help(type(self)) for accurate signature.

initialise()
Backup and set a new context.

terminate(*new_status*)
Restore the context with the previously backed up context.

update()
Just returns RUNNING while it waits for other activities to finish.

py_trees.demos.context_switching.**main**()
Entry point for the demo script.

Listing 4: py_trees/demos/contex_switching.py

```
#!/usr/bin/env python
#
# License: BSD
#   https://raw.githubusercontent.com/stonier/py_trees/devel/LICENSE
```

(continues on next page)

(continued from previous page)

```
#
##############################################################################
# Documentation
##############################################################################

"""
.. argparse::
   :module: py_trees.demos.context_switching
   :func: command_line_argument_parser
   :prog: py-trees-demo-context-switching

.. graphviz:: dot/demo-context_switching.dot

.. image:: images/context_switching.gif
"""

##############################################################################
# Imports
##############################################################################

import argparse
import py_trees
import sys
import time

import py_trees.console as console

##############################################################################
# Classes
##############################################################################


def description():
    content = "Demonstrates context switching with parallels and sequences.\n"
    content += "\n"
    content += "A context switching behaviour is run in parallel with a work sequence.
↪\n"
    content += "Switching the context occurs in the initialise() and terminate()␣
↪methods\n"
    content += "of the context switching behaviour. Note that whether the sequence␣
↪results\n"
    content += "in failure or success, the context switch behaviour will always call␣
↪the\n"
    content += "terminate() method to restore the context. It will also call␣
↪terminate()\n"
    content += "to restore the context in the event of a higher priority parent␣
↪cancelling\n"
    content += "this parallel subtree.\n"
    if py_trees.console.has_colours:
        banner_line = console.green + "*" * 79 + "\n" + console.reset
        s = "\n"
        s += banner_line
        s += console.bold_white + "Context Switching".center(79) + "\n" + console.
↪reset
        s += banner_line
        s += "\n"
        s += content
```

(continues on next page)

(continued from previous page)

```
        s += "\n"
        s += banner_line
    else:
        s = content
    return s


def epilog():
    if py_trees.console.has_colours:
        return console.cyan + "And his noodly appendage reached forth to tickle the blessed...\n" + console.reset
    else:
        return None


def command_line_argument_parser():
    parser = argparse.ArgumentParser(description=description(),
                                     epilog=epilog(),
                                     formatter_class=argparse.RawDescriptionHelpFormatter,
                                     )
    parser.add_argument('-r', '--render', action='store_true', help='render dot tree to file')
    return parser


class ContextSwitch(py_trees.behaviour.Behaviour):
    """
    An example of a context switching class that sets (in ``initialise()``)
    and restores a context (in ``terminate()``). Use in parallel with a
    sequence/subtree that does the work while in this context.

    .. attention:: Simply setting a pair of behaviours (set and reset context) on
        either end of a sequence will not suffice for context switching. In the case
        that one of the work behaviours in the sequence fails, the final reset context
        switch will never trigger.

    """
    def __init__(self, name="ContextSwitch"):
        super(ContextSwitch, self).__init__(name)
        self.feedback_message = "old context"

    def initialise(self):
        """
        Backup and set a new context.
        """
        self.logger.debug("%s.initialise()[switch context]" % (self.__class__.__name__))
        self.feedback_message = "new context"

    def update(self):
        """
        Just returns RUNNING while it waits for other activities to finish.
        """
        self.logger.debug("%s.update()[RUNNING][%s]" % (self.__class__.__name__, self.feedback_message))
        return py_trees.Status.RUNNING
```

(continues on next page)

(continued from previous page)

```
107
108     def terminate(self, new_status):
109         """
110         Restore the context with the previously backed up context.
111         """
112         self.logger.debug("%s.terminate()[%s->%s][restore context]" % (self.__class__.
    ↪__name__, self.status, new_status))
113         self.feedback_message = "old context"
114
115
116 def create_tree():
117     root = py_trees.composites.Parallel(name="Parallel", policy=py_trees.common.
    ↪ParallelPolicy.SUCCESS_ON_ONE)
118     context_switch = ContextSwitch(name="Context")
119     sequence = py_trees.composites.Sequence(name="Sequence")
120     for job in ["Action 1", "Action 2"]:
121         success_after_two = py_trees.behaviours.Count(name=job,
122                                                       fail_until=0,
123                                                       running_until=2,
124                                                       success_until=10)
125         sequence.add_child(success_after_two)
126     root.add_child(context_switch)
127     root.add_child(sequence)
128     return root
129
130
131 ##############################################################################
132 # Main
133 ##############################################################################
134
135 def main():
136     """
137     Entry point for the demo script.
138     """
139     args = command_line_argument_parser().parse_args()
140     print(description())
141     py_trees.logging.level = py_trees.logging.Level.DEBUG
142
143     tree = create_tree()
144
145     ####################
146     # Rendering
147     ####################
148     if args.render:
149         py_trees.display.render_dot_tree(tree)
150         sys.exit()
151
152     ####################
153     # Execute
154     ####################
155     tree.setup(timeout=15)
156     for i in range(1, 6):
157         try:
158             print("\n--------- Tick {0} ---------\n".format(i))
159             tree.tick_once()
160             print("\n")
161             py_trees.display.print_ascii_tree(tree, show_status=True)
```

(continues on next page)

(continued from previous page)

```
            time.sleep(1.0)
        except KeyboardInterrupt:
            break
    print("\n")
```

11.5 py-trees-demo-dot-graphs

Renders a dot graph for a simple tree, with blackboxes.

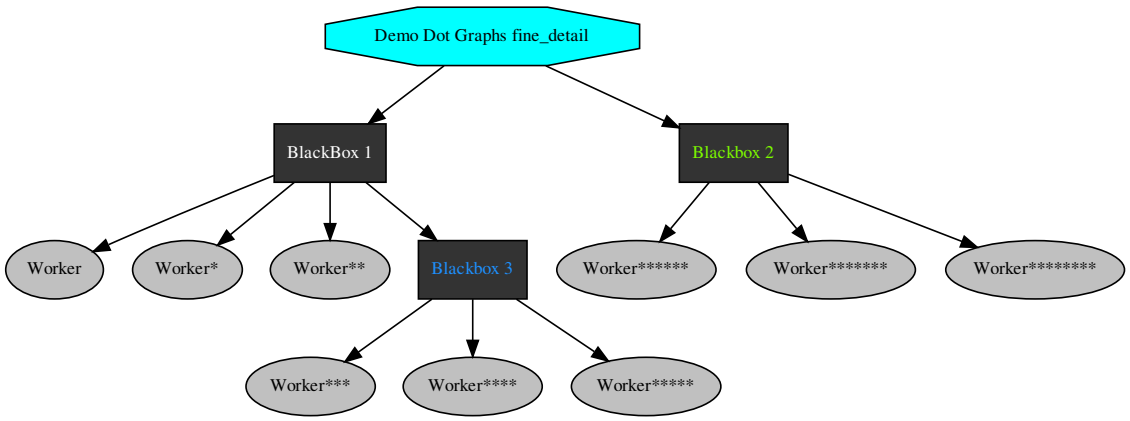
```
usage: py-trees-demo-dot-graphs [-h]
                                [-l {all,fine_detail,detail,component,big_picture}]
```

11.5.1 Named Arguments

-l, --level Possible choices: all, fine_detail, detail, component, big_picture

visibility level

Default: "fine_detail"



py_trees.demos.dot_graphs.**main**()

Entry point for the demo script.

Listing 5: py_trees/demos/dot_graphs.py

```
#!/usr/bin/env python
#
# License: BSD
#   https://raw.githubusercontent.com/stonier/py_trees/devel/LICENSE
#
##############################################################################
# Documentation
##############################################################################
```

(continues on next page)

(continued from previous page)

```
9
10  """
11  .. argparse::
12     :module: py_trees.demos.dot_graphs
13     :func: command_line_argument_parser
14     :prog: py-trees-demo-dot-graphs
15
16  .. graphviz:: dot/demo-dot-graphs.dot
17
18  """
19
20  ##############################################################################
21  # Imports
22  ##############################################################################
23
24  import argparse
25  import subprocess
26  import py_trees
27
28  import py_trees.console as console
29
30  ##############################################################################
31  # Classes
32  ##############################################################################
33
34
35  def description():
36      name = "py-trees-demo-dot-graphs"
37      content = "Renders a dot graph for a simple tree, with blackboxes.\n"
38      if py_trees.console.has_colours:
39          banner_line = console.green + "*" * 79 + "\n" + console.reset
40          s = "\n"
41          s += banner_line
42          s += console.bold_white + "Dot Graphs".center(79) + "\n" + console.reset
43          s += banner_line
44          s += "\n"
45          s += content
46          s += "\n"
47          s += console.white
48          s += console.bold + "    Generate Full Dot Graph" + console.reset + "\n"
49          s += "\n"
50          s += console.cyan + "        {0}".format(name) + console.reset + "\n"
51          s += "\n"
52          s += console.bold + "    With Varying Visibility Levels" + console.reset + "\
    ↪"
53          s += "\n"
54          s += console.cyan + "        {0}".format(name) + console.yellow + " --
    ↪level=all" + console.reset + "\n"
55          s += console.cyan + "        {0}".format(name) + console.yellow + " --
    ↪level=detail" + console.reset + "\n"
56          s += console.cyan + "        {0}".format(name) + console.yellow + " --
    ↪level=component" + console.reset + "\n"
57          s += console.cyan + "        {0}".format(name) + console.yellow + " --
    ↪level=big_picture" + console.reset + "\n"
58          s += "\n"
59          s += banner_line
60      else:
```

(continues on next page)

(continued from previous page)

```
        s = content
    return s


def epilog():
    if py_trees.console.has_colours:
        return console.cyan + "And his noodly appendage reached forth to tickle the blessed...\n" + console.reset
    else:
        return None


def command_line_argument_parser():
    parser = argparse.ArgumentParser(description=description(),
                                     epilog=epilog(),
                                     formatter_class=argparse.RawDescriptionHelpFormatter,
                                     )
    parser.add_argument('-l', '--level', action='store',
                        default='fine_detail',
                        choices=['all', 'fine_detail', 'detail', 'component', 'big_picture'],
                        help='visibility level')
    return parser


def create_tree(level):
    root = py_trees.composites.Selector("Demo Dot Graphs %s" % level)
    first_blackbox = py_trees.composites.Sequence("BlackBox 1")
    first_blackbox.add_child(py_trees.behaviours.Running("Worker"))
    first_blackbox.add_child(py_trees.behaviours.Running("Worker"))
    first_blackbox.add_child(py_trees.behaviours.Running("Worker"))
    first_blackbox.blackbox_level = py_trees.common.BlackBoxLevel.BIG_PICTURE
    second_blackbox = py_trees.composites.Sequence("Blackbox 2")
    second_blackbox.add_child(py_trees.behaviours.Running("Worker"))
    second_blackbox.add_child(py_trees.behaviours.Running("Worker"))
    second_blackbox.add_child(py_trees.behaviours.Running("Worker"))
    second_blackbox.blackbox_level = py_trees.common.BlackBoxLevel.COMPONENT
    third_blackbox = py_trees.composites.Sequence("Blackbox 3")
    third_blackbox.add_child(py_trees.behaviours.Running("Worker"))
    third_blackbox.add_child(py_trees.behaviours.Running("Worker"))
    third_blackbox.add_child(py_trees.behaviours.Running("Worker"))
    third_blackbox.blackbox_level = py_trees.common.BlackBoxLevel.DETAIL
    root.add_child(first_blackbox)
    root.add_child(second_blackbox)
    first_blackbox.add_child(third_blackbox)
    return root


##############################################################################
# Main
##############################################################################

def main():
    """
    Entry point for the demo script.
    """
```

(continues on next page)

(continued from previous page)

```
    args = command_line_argument_parser().parse_args()
    args.enum_level = py_trees.common.string_to_visibility_level(args.level)
    print(description())
    py_trees.logging.level = py_trees.logging.Level.DEBUG

    root = create_tree(args.level)
    py_trees.display.render_dot_tree(root, args.enum_level)

    if py_trees.utilities.which("xdot"):
        try:
            subprocess.call(["xdot", "demo_dot_graphs_%s.dot" % args.level])
        except KeyboardInterrupt:
            pass
    else:
        print("")
        console.logerror("No xdot viewer found, skipping display [hint: sudo apt-install xdot]")
        print("")
```

11.6 py-trees-demo-selector

Higher priority switching and interruption in the children of a selector.

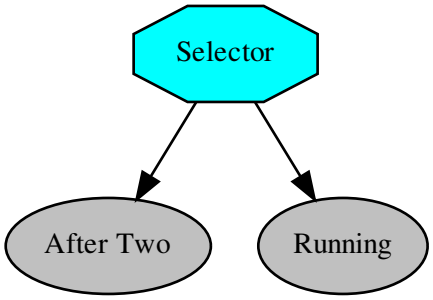
In this example the higher priority child is setup to fail initially, falling back to the continually running second child. On the third tick, the first child succeeds and cancels the hitherto running child.

```
usage: py-trees-demo-selector [-h] [-r]
```

11.6.1 Named Arguments

-r, --render render dot tree to file

Default: False



py_trees.demos.selector.**main**()
Entry point for the demo script.

Listing 6: py_trees/demos/selector.py

```
#!/usr/bin/env python
#
# License: BSD
#   https://raw.githubusercontent.com/stonier/py_trees/devel/LICENSE
#
##############################################################################
# Documentation
##############################################################################

"""
.. argparse::
   :module: py_trees.demos.selector
   :func: command_line_argument_parser
   :prog: py-trees-demo-selector

.. graphviz:: dot/demo-selector.dot

.. image:: images/selector.gif

"""
##############################################################################
# Imports
##############################################################################

import argparse
import py_trees
import sys
import time

import py_trees.console as console

##############################################################################
# Classes
##############################################################################


def description():
    content = "Higher priority switching and interruption in the children of a selector.\n"
    content += "\n"
    content += "In this example the higher priority child is setup to fail initially, \n"
    content += "falling back to the continually running second child. On the third\n"
    content += "tick, the first child succeeds and cancels the hitherto running child. \n"
    if py_trees.console.has_colours:
        banner_line = console.green + "*" * 79 + "\n" + console.reset
        s = "\n"
        s += banner_line
        s += console.bold_white + "Selectors".center(79) + "\n" + console.reset
        s += banner_line
        s += "\n"
        s += content
```

(continues on next page)

(continued from previous page)

```
        s += "\n"
        s += banner_line
    else:
        s = content
    return s


def epilog():
    if py_trees.console.has_colours:
        return console.cyan + "And his noodly appendage reached forth to tickle the blessed...\n" + console.reset
    else:
        return None


def command_line_argument_parser():
    parser = argparse.ArgumentParser(description=description(),
                                     epilog=epilog(),
                                     formatter_class=argparse.RawDescriptionHelpFormatter,
                                     )
    parser.add_argument('-r', '--render', action='store_true', help='render dot tree to file')
    return parser


def create_tree():
    root = py_trees.composites.Selector("Selector")
    success_after_two = py_trees.behaviours.Count(name="After Two",
                                                  fail_until=2,
                                                  running_until=2,
                                                  success_until=10)
    always_running = py_trees.behaviours.Running(name="Running")
    root.add_children([success_after_two, always_running])
    return root


##############################################################################
# Main
##############################################################################

def main():
    """
    Entry point for the demo script.
    """
    args = command_line_argument_parser().parse_args()
    print(description())
    py_trees.logging.level = py_trees.logging.Level.DEBUG

    tree = create_tree()

    ####################
    # Rendering
    ####################
    if args.render:
        py_trees.display.render_dot_tree(tree)
        sys.exit()
```

(continues on next page)

(continued from previous page)

```
    ####################
    # Execute
    ####################
    tree.setup(timeout=15)
    for i in range(1, 4):
        try:
            print("\n--------- Tick {0} ---------\n".format(i))
            tree.tick_once()
            print("\n")
            py_trees.display.print_ascii_tree(tree, show_status=True)
            time.sleep(1.0)
        except KeyboardInterrupt:
            break
    print("\n")
```

11.7 py-trees-demo-sequence

Demonstrates sequences in action.

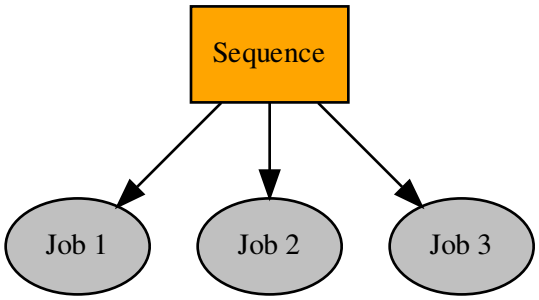
A sequence is populated with 2-tick jobs that are allowed to run through to completion.

```
usage: py-trees-demo-sequence [-h] [-r]
```

11.7.1 Named Arguments

-r, --render render dot tree to file

Default: False



py_trees.demos.sequence.**main**()

Entry point for the demo script.

Listing 7: py_trees/demos/sequence.py

```
#!/usr/bin/env python
#
# License: BSD
#   https://raw.githubusercontent.com/stonier/py_trees/devel/LICENSE
#
##############################################################################
# Documentation
##############################################################################

"""
.. argparse::
   :module: py_trees.demos.sequence
   :func: command_line_argument_parser
   :prog: py-trees-demo-sequence

.. graphviz:: dot/demo-sequence.dot

.. image:: images/sequence.gif
"""

##############################################################################
# Imports
##############################################################################

import argparse
import py_trees
import sys
import time

import py_trees.console as console

##############################################################################
# Classes
##############################################################################


def description():
    content = "Demonstrates sequences in action.\n\n"
    content += "A sequence is populated with 2-tick jobs that are allowed to run through to\n"
    content += "completion.\n"

    if py_trees.console.has_colours:
        banner_line = console.green + "*" * 79 + "\n" + console.reset
        s = "\n"
        s += banner_line
        s += console.bold_white + "Sequences".center(79) + "\n" + console.reset
        s += banner_line
        s += "\n"
        s += content
        s += "\n"
        s += banner_line
    else:
        s = content
    return s
```

(continues on next page)

(continued from previous page)

```
55
56
57  def epilog():
58      if py_trees.console.has_colours:
59          return console.cyan + "And his noodly appendage reached forth to tickle the␣
    ↪blessed...\n" + console.reset
60      else:
61          return None
62
63
64  def command_line_argument_parser():
65      parser = argparse.ArgumentParser(description=description(),
66                                       epilog=epilog(),
67                                       formatter_class=argparse.
    ↪RawDescriptionHelpFormatter,
68                                       )
69      parser.add_argument('-r', '--render', action='store_true', help='render dot tree␣
    ↪to file')
70      return parser
71
72
73  def create_tree():
74      root = py_trees.composites.Sequence("Sequence")
75      for action in ["Action 1", "Action 2", "Action 3"]:
76          success_after_two = py_trees.behaviours.Count(name=action,
77                                                        fail_until=0,
78                                                        running_until=1,
79                                                        success_until=10)
80          root.add_child(success_after_two)
81      return root
82
83
84  ##############################################################################
85  # Main
86  ##############################################################################
87
88  def main():
89      """
90      Entry point for the demo script.
91      """
92      args = command_line_argument_parser().parse_args()
93      print(description())
94      py_trees.logging.level = py_trees.logging.Level.DEBUG
95
96      tree = create_tree()
97
98      ####################
99      # Rendering
100     ####################
101     if args.render:
102         py_trees.display.render_dot_tree(tree)
103         sys.exit()
104
105     ####################
106     # Execute
107     ####################
108     tree.setup(timeout=15)
```

(continues on next page)

(continued from previous page)

```
    for i in range(1, 6):
        try:
            print("\n--------- Tick {0} ---------\n".format(i))
            tree.tick_once()
            print("\n")
            py_trees.display.print_ascii_tree(tree, show_status=True)
            time.sleep(1.0)
        except KeyboardInterrupt:
            break
    print("\n")
```

11.8 py-trees-demo-tree-stewardship

A demonstration of tree stewardship.

A slightly less trivial tree that uses a simple stdout pre-tick handler and both the debug and snapshot visitors for logging and displaying the state of the tree.

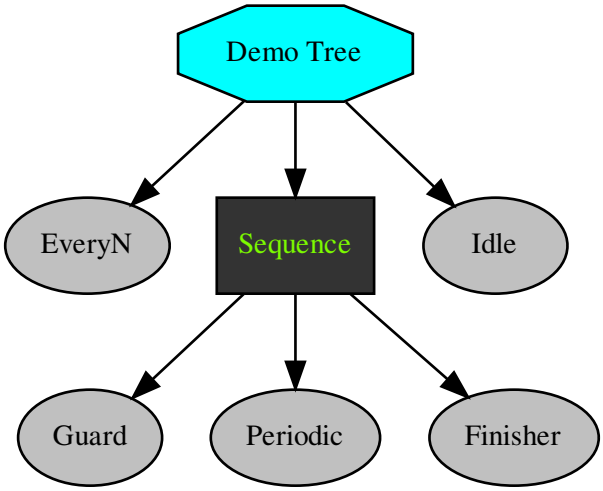
EVENTS

- 3 : sequence switches from running to success
- 4 : selector's first child flicks to success once only
- 8 : the fallback idler kicks in as everything else fails
- 14 : the first child kicks in again, aborting a running sequence behind it

```
usage: py-trees-demo-tree-stewardship [-h] [-r | -i]
```

11.8.1 Named Arguments

-r, --render	render dot tree to file
	Default: False
-i, --interactive	pause and wait for keypress at each tick
	Default: False



py_trees.demos.stewardship.**main**()
Entry point for the demo script.

py_trees.demos.stewardship.**post_tick_handler**(*snapshot_visitor*, *behaviour_tree*)
Prints an ascii tree with the current snapshot status.

py_trees.demos.stewardship.**pre_tick_handler**(*behaviour_tree*)
This prints a banner and will run immediately before every tick of the tree.

Parameters **behaviour_tree** (*BehaviourTree*) – the tree custodian

Listing 8: py_trees/demos/stewardship.py

```
#!/usr/bin/env python
#
# License: BSD
#   https://raw.githubusercontent.com/stonier/py_trees/devel/LICENSE
#
##############################################################################
# Documentation
##############################################################################

"""
.. argparse::
   :module: py_trees.demos.stewardship
   :func: command_line_argument_parser
   :prog: py-trees-demo-tree-stewardship

.. graphviz:: dot/stewardship.dot

.. image:: images/tree_stewardship.gif
"""

```

(continues on next page)

(continued from previous page)

```
##############################################################################
# Imports
##############################################################################

import argparse
import functools
import py_trees
import sys
import time

import py_trees.console as console

##############################################################################
# Classes
##############################################################################


def description(root):
    content = "A demonstration of tree stewardship.\n\n"
    content += "A slightly less trivial tree that uses a simple stdout pre-tick handler\n"
    content += "and both the debug and snapshot visitors for logging and displaying\n"
    content += "the state of the tree.\n"
    content += "\n"
    content += "EVENTS\n"
    content += "\n"
    content += " -  3 : sequence switches from running to success\n"
    content += " -  4 : selector's first child flicks to success once only\n"
    content += " -  8 : the fallback idler kicks in as everything else fails\n"
    content += " - 14 : the first child kicks in again, aborting a running sequence behind it\n"
    content += "\n"
    if py_trees.console.has_colours:
        banner_line = console.green + "*" * 79 + "\n" + console.reset
        s = "\n"
        s += banner_line
        s += console.bold_white + "Trees".center(79) + "\n" + console.reset
        s += banner_line
        s += "\n"
        s += content
        s += "\n"
        s += banner_line
    else:
        s = content
    return s


def epilog():
    if py_trees.console.has_colours:
        return console.cyan + "And his noodly appendage reached forth to tickle the blessed...\n" + console.reset
    else:
        return None


def command_line_argument_parser():
    parser = argparse.ArgumentParser(description=description(create_tree()),
```

(continues on next page)

(continued from previous page)

```
                                    epilog=epilog(),
                                    formatter_class=argparse.
↪RawDescriptionHelpFormatter,
                                    )
    group = parser.add_mutually_exclusive_group()
    group.add_argument('-r', '--render', action='store_true', help='render dot tree␣
↪to file')
    group.add_argument('-i', '--interactive', action='store_true', help='pause and␣
↪wait for keypress at each tick')
    return parser


def pre_tick_handler(behaviour_tree):
    """
    This prints a banner and will run immediately before every tick of the tree.

    Args:
        behaviour_tree (:class:`~py_trees.trees.BehaviourTree`): the tree custodian

    """
    print("\n--------- Run %s ---------\n" % behaviour_tree.count)


def post_tick_handler(snapshot_visitor, behaviour_tree):
    """
    Prints an ascii tree with the current snapshot status.
    """
    print("\n" + py_trees.display.ascii_tree(behaviour_tree.root,
                                             snapshot_information=snapshot_visitor))


def create_tree():
    every_n_success = py_trees.behaviours.SuccessEveryN("EveryN", 5)
    sequence = py_trees.Sequence(name="Sequence")
    guard = py_trees.behaviours.Success("Guard")
    periodic_success = py_trees.behaviours.Periodic("Periodic", 3)
    finisher = py_trees.behaviours.Success("Finisher")
    sequence.add_child(guard)
    sequence.add_child(periodic_success)
    sequence.add_child(finisher)
    sequence.blackbox_level = py_trees.common.BlackBoxLevel.COMPONENT
    idle = py_trees.behaviours.Success("Idle")
    root = py_trees.Selector(name="Demo Tree")
    root.add_child(every_n_success)
    root.add_child(sequence)
    root.add_child(idle)
    return root


##############################################################################
# Main
##############################################################################

def main():
    """
    Entry point for the demo script.
    """
```

(continues on next page)

(continued from previous page)

```
    args = command_line_argument_parser().parse_args()
    py_trees.logging.level = py_trees.logging.Level.DEBUG
    tree = create_tree()
    print(description(tree))

    ####################
    # Rendering
    ####################
    if args.render:
        py_trees.display.render_dot_tree(tree)
        sys.exit()

    ####################
    # Tree Stewardship
    ####################
    behaviour_tree = py_trees.trees.BehaviourTree(tree)
    behaviour_tree.add_pre_tick_handler(pre_tick_handler)
    behaviour_tree.visitors.append(py_trees.visitors.DebugVisitor())
    snapshot_visitor = py_trees.visitors.SnapshotVisitor()
    behaviour_tree.add_post_tick_handler(functools.partial(post_tick_handler,␣
↪snapshot_visitor))
    behaviour_tree.visitors.append(snapshot_visitor)
    behaviour_tree.setup(timeout=15)

    ####################
    # Tick Tock
    ####################
    if args.interactive:
        unused_result = py_trees.console.read_single_keypress()
    while True:
        try:
            behaviour_tree.tick()
            if args.interactive:
                unused_result = py_trees.console.read_single_keypress()
            else:
                time.sleep(0.5)
        except KeyboardInterrupt:
            break
    print("\n")
```

CHAPTER 12

Programs

12.1 py-trees-render

Point this program at a method which creates a root to render to dot/svg/png.

Examples

```
$ py-trees-render py_trees.demos.stewardship.create_tree
$ py-trees-render --name=foo py_trees.demos.stewardship.create_tree
$ py-trees-render --kwargs='{"level":"all"}' py_trees.demos.dot_graphs.create_tree
```

```
usage: py-trees-render [-h]
                       [-l {all,fine_detail,detail,component,big_picture}]
                       [-n NAME] [-k KWARGS]
                       method
```

12.1.1 Positional Arguments

method space separated list of blackboard variables to watch

12.1.2 Named Arguments

-l, --level Possible choices: all, fine_detail, detail, component, big_picture

visibility level

Default: "fine_detail"

-n, --name name to use for the created files (defaults to the root behaviour name)

-k, --kwargs dictionary of keyword arguments to the method

Default: {}

CHAPTER 13

Module API

13.1 py_trees

This is the top-level namespace of the py_trees package.

13.2 py_trees.behaviour

The core behaviour template. All behaviours, standalone and composite, inherit from this class.

class py_trees.behaviour.**Behaviour**(*name=''*, **args*, ***kwargs*)

Bases: object

Defines the basic properties and methods required of a node in a behaviour tree.

Uses all the whizbang tricks from coroutines and generators to do this as optimally as you may in python. When implementing your own behaviour, subclass this class.

Parameters

- **name** (str) – the behaviour name
- ***args** – variable length argument list.
- ****kwargs** – arbitrary keyword arguments.

Variables

- **name** (str) – the behaviour name
- **status** (*Status*) – the behaviour status (*INVALID*, *RUNNING*, *FAILURE*, *SUCCESS*)
- **parent** (*Behaviour*) – a *Composite* instance if nested in a tree, otherwise None
- **children** ([*Behaviour*]) – empty for regular behaviours, populated for composites
- **feedback_message** (str) – a simple message used to notify of significant happenings

- **blackbox_level** (*BlackBoxLevel*) – a helper variable for dot graphs and runtime gui's to collapse/explode entire subtrees dependent upon the blackbox level.

See also:

- *Skeleton Behaviour Template*
- *The Lifecycle Demo*
- *The Action Behaviour Demo*

has_parent_with_instance_type(*instance_type*)
Moves up through this behaviour's parents looking for a behaviour with the same instance type as that specified.

Parameters **instance_type** (str) – instance type of the parent to match

Returns whether a parent was found or not

Return type bool

has_parent_with_name(*name*)
Searches through this behaviour's parents, and their parents, looking for a behaviour with the same name as that specified.

Parameters **name** (str) – name of the parent to match, can be a regular expression

Returns whether a parent was found or not

Return type bool

initialise()

Note: User Customisable Callback

Subclasses may override this method to perform any necessary initialising/clearing/resetting of variables when when preparing to enter this behaviour if it was not previously *RUNNING*. i.e. Expect this to trigger more than once!

iterate(*direct_descendants=False*)
Generator that provides iteration over this behaviour and all its children. To traverse the entire tree:

```
for node in my_behaviour.iterate():
    print("Name: {0}".format(node.name))
```

Parameters **direct_descendants** (bool) – only yield children one step away from this behaviour.

Yields *Behaviour* – one of it's children

setup(*timeout*)
Subclasses may override this method to do any one-time delayed construction that is necessary for runtime. This is best done here rather than in the constructor so that trees can be instantiated on the fly without any severe runtime requirements (e.g. a hardware sensor) on any pc to produce visualisations such as dot graphs.

Note: User Customisable Callback

Parameters **timeout** (float) – time to wait (0.0 is blocking forever)

Returns whether it timed out trying to setup

Return type bool

stop(*new_status=<Status.INVALID: 'INVALID'>*)

Parameters **new_status** (*Status*) – the behaviour is transitioning to this new status

This calls the user defined *terminate()* method and also resets the generator. It will finally set the new status once the user's *terminate()* function has been called.

Warning: Do not use this method, override *terminate()* instead.

terminate(*new_status*)

Note: User Customisable Callback

Subclasses may override this method to clean up. It will be triggered when a behaviour either finishes execution (switching from *RUNNING* to *FAILURE* || *SUCCESS*) or it got interrupted by a higher priority branch (switching to *INVALID*). Remember that the *initialise()* method will handle resetting of variables before re-entry, so this method is about disabling resources until this behaviour's next tick. This could be a indeterminably long time. e.g.

- cancel an external action that got started
- shut down any tempoarary communication handles

Parameters **new_status** (*Status*) – the behaviour is transitioning to this new status

Warning: Do not set *self.status = new_status* here, that is automatically handled by the *stop()* method. Use the argument purely for introspection purposes (e.g. comparing the current state in *self.status* with the state it will transition to in *new_status*.

tick()

This function is a generator that can be used by an iterator on an entire behaviour tree. It handles the logic for deciding when to call the user's *initialise()* and *terminate()* methods as well as making the actual call to the user's *update()* method that determines the behaviour's new status once the tick has finished. Once done, it will then yield itself (generator mechanism) so that it can be used as part of an iterator for the entire tree.

```
for node in my_behaviour.tick():
    print("Do something")
```

Note: This is a generator function, you must use this with *yield*. If you need a direct call, prefer *tick_once()* instead.

Yields *Behaviour* – a reference to itself

tick_once()
A direct means of calling tick on this object without using the generator mechanism.

tip()
Get the *tip* of this behaviour's subtree (if it has one) after it's last tick. This corresponds to the the deepest node that was running before the subtree traversal reversed direction and headed back to this node.

Returns child behaviour, itself or None if its status is *INVALID*

Return type *Behaviour* or None

update()

Note: User Customisable Callback

Returns the behaviour's new status *Status*

Return type *Status*

Subclasses may override this method to perform any logic required to arrive at a decision on the behaviour's new status. It is the primary worker function called on by the *tick()* mechanism.

Tip: This method should be almost instantaneous and non-blocking

visit(*visitor*)
This is functionality that enables external introspection into the behaviour. It gets used by the tree manager classes to collect information as ticking traverses a tree.

Parameters **visitor** (object) – the visiting class, must have a run(*Behaviour*) method.

13.3 py_trees.behaviours

A library of fundamental behaviours for use.

class py_trees.behaviours.**Count**(*name='Count'*, *fail_until=3*, *running_until=5*, *success_until=6*, *reset=True*, **args*, ***kwargs*)
Bases: *py_trees.behaviour.Behaviour*

A counting behaviour that updates its status at each tick depending on the value of the counter. The status will move through the states in order - *FAILURE*, *RUNNING*, *SUCCESS*.

This behaviour is useful for simple testing and demo scenarios.

Parameters

- **name** (str) – name of the behaviour
- **fail_until** (int) – set status to *FAILURE* until the counter reaches this value
- **running_until** (int) – set status to *RUNNING* until the counter reaches this value
- **success_until** (int) – set status to *SUCCESS* until the counter reaches this value
- **reset** (bool) – whenever invalidated (usually by a sequence reinitialising, or higher priority interrupting)

Variables **count** (int) – a simple counter which increments every tick

terminate(*new_status*)

Note: User Customisable Callback

Subclasses may override this method to clean up. It will be triggered when a behaviour either finishes execution (switching from *RUNNING* to *FAILURE* || *SUCCESS*) or it got interrupted by a higher priority branch (switching to *INVALID*). Remember that the *initialise()* method will handle resetting of variables before re-entry, so this method is about disabling resources until this behaviour's next tick. This could be a indeterminably long time. e.g.

- cancel an external action that got started
- shut down any tempoarary communication handles

Parameters **new_status** (*Status*) – the behaviour is transitioning to this new status

Warning: Do not set *self.status = new_status* here, that is automatically handled by the *stop()* method. Use the argument purely for introspection purposes (e.g. comparing the current state in *self.status* with the state it will transition to in *new_status*.

update()

Note: User Customisable Callback

Returns the behaviour's new status *Status*

Return type *Status*

Subclasses may override this method to perform any logic required to arrive at a decision on the behaviour's new status. It is the primary worker function called on by the *tick()* mechanism.

Tip: This method should be almost instantaneous and non-blocking

class py_trees.behaviours.**Failure**(*name=''*, **args*, ***kwargs*)
Bases: *py_trees.behaviour.Behaviour*

class py_trees.behaviours.**Periodic**(*name*, *n*)
Bases: *py_trees.behaviour.Behaviour*

Simply periodically rotates it's status over the *RUNNING*, *SUCCESS*, *FAILURE* states. That is, *RUNNING* for N ticks, *SUCCESS* for N ticks, *FAILURE* for N ticks...

Parameters

- **name** (str) – name of the behaviour
- **n** (int) – period value (in ticks)

Note: It does not reset the count when initialising.

update()

Note: User Customisable Callback

Returns the behaviour's new status *Status*

Return type *Status*

Subclasses may override this method to perform any logic required to arrive at a decision on the behaviour's new status. It is the primary worker function called on by the *tick()* mechanism.

Tip: This method should be almost instantaneous and non-blocking

class py_trees.behaviours.**Running**(*name=''*, **args*, ***kwargs*)
Bases: *py_trees.behaviour.Behaviour*

class py_trees.behaviours.**Success**(*name=''*, **args*, ***kwargs*)
Bases: *py_trees.behaviour.Behaviour*

class py_trees.behaviours.**SuccessEveryN**(*name*, *n*)
Bases: *py_trees.behaviour.Behaviour*

This behaviour updates it's status with *SUCCESS* once every N ticks, *FAILURE* otherwise.

Parameters

- **name** (str) – name of the behaviour
- **n** (int) – trigger success on every n'th tick

Tip: Use with decorators to change the status value as desired, e.g. *py_trees.meta.failure_is_running()*

update()

Note: User Customisable Callback

Returns the behaviour's new status *Status*

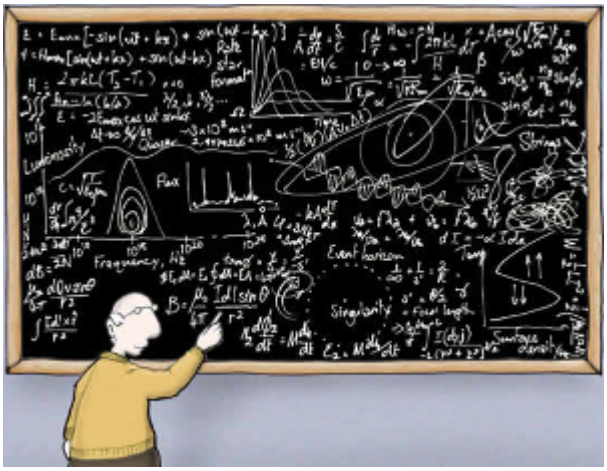
Return type *Status*

Subclasses may override this method to perform any logic required to arrive at a decision on the behaviour's new status. It is the primary worker function called on by the *tick()* mechanism.

Tip: This method should be almost instantaneous and non-blocking

13.4 py_trees.blackboard

Blackboards are not a necessary component, but are a fairly standard feature in most behaviour tree implementations. See, for example, the design notes for blackboards in Unreal Engine.



Implementations however, tend to vary quite a bit depending on the needs of the framework using them. Some of the usual considerations include scope and sharing of blackboards across multiple tree instances.

For this package, we've decided to keep blackboards extremely simple to fit with the same 'rapid development for small scale systems' principles that this library is designed for.

- No sharing between tree instances
- No locking for reading/writing
- Global scope, i.e. any behaviour can access any variable
- No external communications (e.g. to a database)

class py_trees.blackboard.**Blackboard**
Bases: object

Borg style key-value store for sharing amongst behaviours.

Examples

You can instantiate the blackboard from anywhere in your program. Even disconnected calls will get access to the same data store. For example:

```
def check_foo():
    blackboard = Blackboard()
    assert(blackboard.foo, "bar")

if __name__ == '__main__':
    blackboard = Blackboard()
    blackboard.foo = "bar"
    check_foo()
```

If the key value you are interested in is only known at runtime, then you can set/get from the blackboard without the convenient variable style access:

```
blackboard = Blackboard()
result = blackboard.set("foo", "bar")
foo = blackboard.get("foo")
```

The blackboard can also be converted and printed (with highlighting) as a string. This is useful for logging and debugging.

```
print(Blackboard())
```

Warning: Be careful of key collisions. This implementation leaves this management up to the user.

See also:

The *py-trees-demo-blackboard* program demos use of the blackboard along with a couple of the blackboard behaviours.

get(*name*)
For when you only have strings to identify and access the blackboard variables, this provides a convenient accessor.

Parameters **name** (str) – name of the variable to set

set(*name*, *value*, *overwrite=True*)
For when you only have strings to identify and access the blackboard variables, this provides a convenient setter.

Parameters

- **name** (str) – name of the variable to set
- **value** (any) – any variable type
- **overwrite** (bool) – whether to abort if the value is already present

Returns always True unless overwrite was set to False and a variable already exists

Return type bool

class py_trees.blackboard.**CheckBlackboardVariable**(*name*, *variable_name='dummy'*, *expected_value=None*, *comparison_operator=<built-in function eq>*, *clearing_policy=<ClearingPolicy.ON_INITIALISE: 1>*, *debug_feedback_message=False*)

Bases: *py_trees.behaviour.Behaviour*

Check the blackboard to see if it has a specific variable and optionally whether that variable has an expected value. It is a binary behaviour, always updating it's status with either *SUCCESS* or *FAILURE* at each tick.

Parameters

- **name** (str) – name of the behaviour
- **variable_name** (str) – name of the variable to set
- **expected_value** (any) – expected value to find (if *None*, check for existence only)
- **comparison_operator** (func) – one from the python operator module
- **clearing_policy** (any) – when to clear the match result, see *ClearingPolicy*

Tip: If just checking for existence, use the default argument *expected_value=None*.

Tip: There are times when you want to get the expected match once and then save that result thereafter. For example, to flag once a system has reached a subgoal. Use the *NEVER* flag to do this.

initialise()
Clears the internally stored message ready for a new run if `old_data_is_valid` wasn't set.

terminate(*new_status*)
Always discard the matching result if it was invalidated by a parent or higher priority interrupt.

update()
Check for existence, or the appropriate match on the expected value.

Returns *FAILURE* if not matched, *SUCCESS* otherwise.

Return type *Status*

class py_trees.blackboard.**ClearBlackboardVariable**(*name='Clear Blackboard Variable'*, *variable_name='dummy'*)

Bases: `py_trees.meta.Success`

Clear the specified value from the blackboard.

Parameters

- **name** (`str`) – name of the behaviour
- **variable_name** (`str`) – name of the variable to clear

initialise()
Delete the variable from the blackboard.

class py_trees.blackboard.**SetBlackboardVariable**(*name='Set Blackboard Variable'*, *variable_name='dummy'*, *variable_value=None*)

Bases: `py_trees.meta.Success`

Set the specified variable on the blackboard. Usually we set variables from inside other behaviours, but can be convenient to set them from a behaviour of their own sometimes so you don't get blackboard logic mixed up with more atomic behaviours.

Parameters

- **name** (`str`) – name of the behaviour
- **variable_name** (`str`) – name of the variable to set
- **variable_value** (`any`) – value of the variable to set

Todo: overwrite option, leading to possible failure/success logic.

initialise()

Note: User Customisable Callback

Subclasses may override this method to perform any necessary initialising/clearing/resetting of variables when when preparing to enter this behaviour if it was not previously *RUNNING*. i.e. Expect this to trigger more than once!

class py_trees.blackboard.**WaitForBlackboardVariable**(*name*, *variable_name='dummy'*, *expected_value=None*, *comparison_operator=<built-in function eq>*, *clearing_policy=<ClearingPolicy.ON_INITIALISE: 1>*)

Bases: *py_trees.behaviour.Behaviour*

Check the blackboard to see if it has a specific variable and optionally whether that variable has a specific value. Unlike *CheckBlackboardVariable* this class will be in a *RUNNING* state until the variable appears and (optionally) is matched.

Parameters

- **name** (str) – name of the behaviour
- **variable_name** (str) – name of the variable to check
- **expected_value** (any) – expected value to find (if *None*, check for existence only)
- **comparison_operator** (func) – one from the python operator module
- **clearing_policy** (any) – when to clear the match result, see *ClearingPolicy*

Tip: There are times when you want to get the expected match once and then save that result thereafter. For example, to flag once a system has reached a subgoal. Use the *NEVER* flag to do this.

See also:

CheckBlackboardVariable

initialise()
Clears the internally stored message ready for a new run if old_data_is_valid wasn't set.

terminate(*new_status*)
Always discard the matching result if it was invalidated by a parent or higher priority interrupt.

update()
Check for existence, or the appropriate match on the expected value.

Returns *FAILURE* if not matched, *SUCCESS* otherwise.

Return type *Status*

13.5 py_trees.common

Common definitions, methods and variables used by the py_trees library.

class py_trees.common.**BlackBoxLevel**
Bases: enum.IntEnum

Whether a behaviour is a blackbox entity that may be considered collapsible (i.e. everything in its subtree will not be visualised) by visualisation tools.

Blackbox levels are increasingly persistent in visualisations.

Visualisations by default, should always collapse blackboxes that represent *DETAIL*.

BIG_PICTURE = 3
A blackbox that represents a big picture part of the entire tree view.

COMPONENT = 2
A blackbox that encapsulates a subgroup of functionalities as a single group.

DETAIL = 1
A blackbox that encapsulates detailed activity.

NOT_A_BLACKBOX = 4
Not a blackbox, do not ever collapse.

class py_trees.common.**ClearingPolicy**
Bases: enum.IntEnum

Policy rules for behaviours to dictate when data should be cleared/reset. Used by the subscribers module.

NEVER = 3
Never clear the data

ON_INITIALISE = 1
Clear when entering the *initialise()* method.

ON_SUCCESS = 2
Clear when returning *SUCCESS*.

class py_trees.common.**Name**
Bases: enum.Enum

Naming conventions.

AUTO_GENERATED = 'AUTO_GENERATED'
More Foo:py:data:*~py_trees.common.Name.AUTO_GENERATED* leaves it to the behaviour to generate a useful, informative name.

class py_trees.common.**ParallelPolicy**
Bases: enum.Enum

Policy rules for *Parallel* composites.

SUCCESS_ON_ALL = 'SUCCESS_ON_ALL'
SUCCESS only when each and every child returns *SUCCESS*.

SUCCESS_ON_ONE = 'SUCCESS_ON_ONE'
SUCCESS so long as at least one child has *SUCCESS* and the remainder are *RUNNING*

class py_trees.common.**Status**
Bases: enum.Enum

An enumerator representing the status of a behaviour

FAILURE = 'FAILURE'
Behaviour check has failed, or execution of its action finished with a failed result.

INVALID = 'INVALID'
Behaviour is uninitialised and inactive, i.e. this is the status before first entry, and after a higher priority switch has occurred.

RUNNING = 'RUNNING'
Behaviour is in the middle of executing some action, result still pending.

SUCCESS = 'SUCCESS'
Behaviour check has passed, or execution of its action has finished with a successful result.

class py_trees.common.**VisibilityLevel**
Bases: enum.IntEnum

Closely associated with the *BlackBoxLevel* for a behaviour. This sets the visibility level to be used for visualisations.

Visibility levels correspond to reducing levels of visibility in a visualisation.

ALL = 0
Do not collapse any behaviour.

BIG_PICTURE = 3
Collapse any blackbox that isn't marked with *BIG_PICTURE*.

COMPONENT = 2
Collapse blackboxes marked with *COMPONENT* or lower.

DETAIL = 1
Collapse blackboxes marked with *DETAIL* or lower.

common.**string_to_visibility_level**()
Will convert a string to a visibility level. Note that it will quietly return ALL if the string is not matched to any visibility level string identifier.

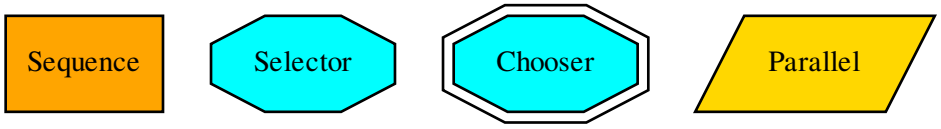
Parameters **level** (*str*) – visibility level as a string

Returns visibility level enum

Return type *VisibilityLevel*

13.6 py_trees.composites

Composites are the **factories** and **decision makers** of a behaviour tree. They are responsible for shaping the branches.



Tip: You should never need to subclass or create new composites.

Most patterns can be achieved with a combination of the above. Adding to this set exponentially increases the complexity and subsequently making it more difficult to design, introspect, visualise and debug the trees. Always try to find the combination you need to achieve your result before contemplating adding to this set. Actually, scratch that...just don't contemplate it!

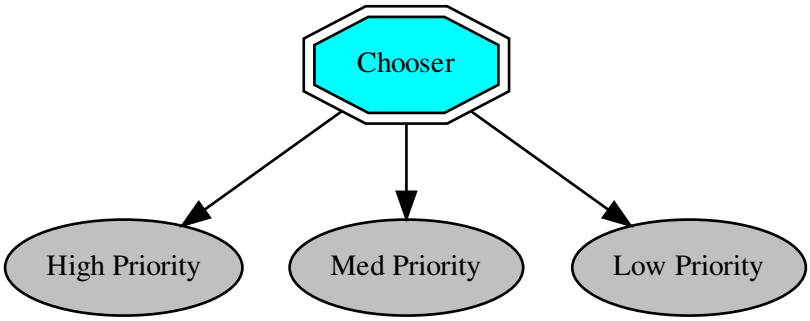
Composite behaviours typically manage children and apply some logic to the way they execute and return a result, but generally don't do anything themselves. Perform the checks or actions you need to do in the non-composite behaviours.

- *Sequence*: execute children sequentially

- *Selector*: select a path through the tree, interruptible by higher priorities
- *Chooser*: like a selector, but commits to a path once started until it finishes
- *Parallel*: manage children concurrently

class py_trees.composites.**Chooser**(*name='Chooser'*, *children=None*, **args*, ***kwargs*)
Bases: *py_trees.composites.Selector*

Choosers are Selectors with Commitment



A variant of the selector class. Once a child is selected, it cannot be interrupted by higher priority siblings. As soon as the chosen child itself has finished it frees the chooser for an alternative selection. i.e. priorities only come into effect if the chooser wasn't running in the previous tick.

Note: This is the only composite in py_trees that is not a core composite in most behaviour tree implementations. Nonetheless, this is useful in fields like robotics, where you have to ensure that your manipulator doesn't drop it's payload mid-motion as soon as a higher interrupt arrives. Use this composite sparingly and only if you can't find another way to easily create an elegant tree composition for your task.

Parameters

- **name** (str) – the composite behaviour name
- **children** ([*Behaviour*]) – list of children to add
- ***args** – variable length argument list
- ****kwargs** – arbitrary keyword arguments

__init__(*name='Chooser'*, *children=None*, **args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

tick()
Run the tick behaviour for this chooser. Note that the status of the tick is (for now) always determined by its children, not by the user customised update function.

Yields *Behaviour* – a reference to itself or one of its children

class py_trees.composites.**Composite**(*name=''*, *children=None*, **args*, ***kwargs*)
Bases: *py_trees.behaviour.Behaviour*

The parent class to all composite behaviours, i.e. those that have children.

Parameters

- **name** (str) – the composite behaviour name
- **children** ([*Behaviour*]) – list of children to add
- ***args** – variable length argument list
- ****kwargs** – arbitrary keyword arguments

__init__(*name=''*, *children=None*, **args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

add_child(*child*)
Adds a child.

Parameters **child** (*Behaviour*) – child to add

Returns unique id of the child

Return type uuid.UUID

add_children(*children*)
Append a list of children to the current list.

Parameters **children** ([*Behaviour*]) – list of children to add

insert_child(*child*, *index*)
Insert child at the specified index. This simply directly calls the python list's `insert` method using the child and index arguments.

Parameters

- **child** (*Behaviour*) – child to insert
- **index** (int) – index to insert it at

Returns unique id of the child

Return type uuid.UUID

prepend_child(*child*)
Prepend the child before all other children.

Parameters **child** (*Behaviour*) – child to insert

Returns unique id of the child

Return type uuid.UUID

remove_all_children()
Remove all children. Makes sure to stop each child if necessary.

remove_child(*child*)
Remove the child behaviour from this composite.

Parameters **child** (*Behaviour*) – child to delete

Returns index of the child that was removed

Return type int

Todo: Error handling for when child is not in this list

---

remove_child_by_id(*child_id*)
Remove the child with the specified id.

Parameters **child_id** (*uuid.UUID*) – unique id of the child

Raises IndexError – if the child was not found

replace_child(*child*, *replacement*)
Replace the child behaviour with another.

Parameters

- **child** (*Behaviour*) – child to delete
- **replacement** (*Behaviour*) – child to insert

setup(*timeout*)
Relays to each child's setup() method in turn.

Parameters **timeout** (float) – time to wait (0.0 is blocking forever)

Returns sucess or failure of the operation

Return type bool

stop(*new_status=<Status.INVALID: 'INVALID'>*)
There is generally two use cases that must be supported here.

1) Whenever the composite has gone to a recognised state (i.e. *FAILURE* or SUCCESS), or 2) when a higher level parent calls on it to truly stop (INVALID).

In only the latter case will children need to be forcibly stopped as well. In the first case, they will have stopped themselves appropriately already.

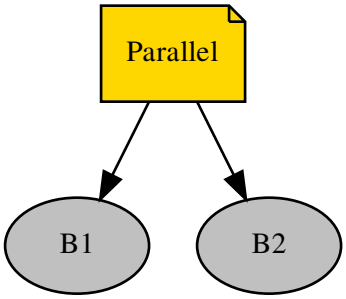
Parameters **new_status** (*Status*) – behaviour will transition to this new status

tip()
Recursive function to extract the last running node of the tree.

Returns class::*~py_trees.behaviour.Behaviour*: the tip function of the current child of this composite or None

class py_trees.composites.**Parallel**(*name='Parallel'*, *policy=<ParallelPolicy.SUCCESS_ON_ALL: 'SUCCESS_ON_ALL'>*, *children=None*, **args*, ***kwargs*)
Bases: *py_trees.composites.Composite*

Parallels enable a kind of concurrency



Ticks every child every time the parallel is run (a poor man's form of paralellism).

- Parallels will return *FAILURE* if any child returns *FAILURE*
- Parallels with policy *SUCCESS_ON_ONE* return *SUCCESS* if **at least one** child returns *SUCCESS* and others are *RUNNING*.
- Parallels with policy *SUCCESS_ON_ALL* only returns *SUCCESS* if **all** children return *SUCCESS*

See also:

The *py-trees-demo-context-switching* program demos a parallel used to assist in a context switching scenario.

Parameters

- **name** (str) – the composite behaviour name
- **policy** (*ParallelPolicy*) – policy to use for deciding success or otherwise
- **children** ([*Behaviour*]) – list of children to add
- ***args** – variable length argument list
- ****kwargs** – arbitrary keyword arguments

__init__(*name='Parallel'*, *policy=<ParallelPolicy.SUCCESS_ON_ALL: 'SUCCESS_ON_ALL'>*, *children=None*, **args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

current_child
Have to check if there's anything actually running first.

Returns the child that is currently running, or None

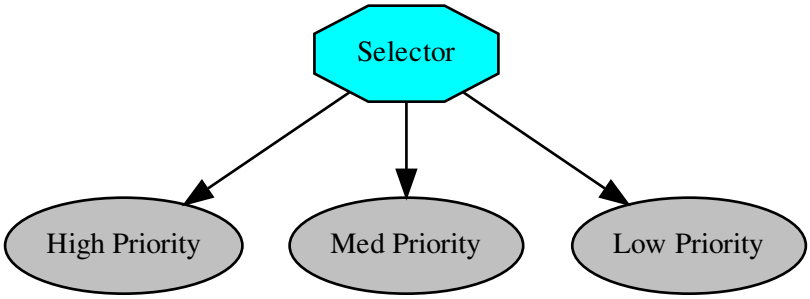
Return type *Behaviour*

tick()
Tick over the children.

Yields *Behaviour* – a reference to itself or one of its children

class py_trees.composites.**Selector**(*name='Selector'*, *children=None*, **args*, ***kwargs*)
Bases: *py_trees.composites.Composite*

Selectors are the Decision Makers



A selector executes each of its child behaviours in turn until one of them succeeds (at which point it itself returns *RUNNING* or *SUCCESS*, or it runs out of children at which point it itself returns *FAILURE*. We usually refer to selecting children as a means of *choosing between priorities*. Each child and its subtree represent a decreasingly lower priority path.

Note: Switching from a low -> high priority branch causes a *stop(INVALID)* signal to be sent to the previously executing low priority branch. This signal will percolate down that child's own subtree. Behaviours should make sure that they catch this and *destruct* appropriately.

Make sure you do your appropriate cleanup in the `terminate()` methods! e.g. cancelling a running goal, or restoring a context.

See also:

The *py-trees-demo-selector* program demos higher priority switching under a selector.

Parameters

- **name** (`str`) – the composite behaviour name
- **children** ([*Behaviour*]) – list of children to add
- ***args** – variable length argument list
- ****kwargs** – arbitrary keyword arguments

__init__(*name='Selector'*, *children=None*, **args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

__repr__()
Simple string representation of the object.

Returns string representation

Return type `str`

stop(*new_status=<Status.INVALID: 'INVALID'>*)
Stopping a selector requires setting the current child to none. Note that it is important to implement this here instead of terminate, so users are free to subclass this easily with their own terminate and not have to remember that they need to call this function manually.

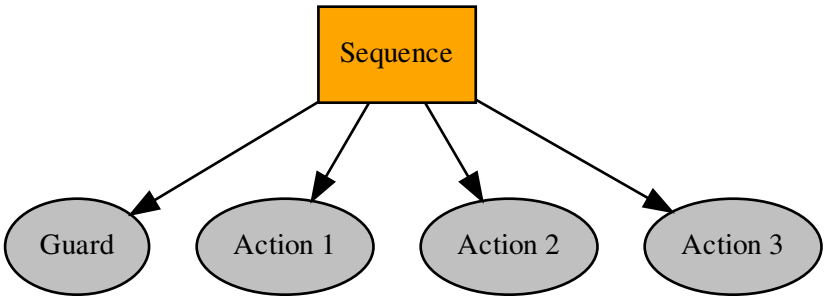
Parameters **new_status** (*Status*) – the composite is transitioning to this new status

tick()
Run the tick behaviour for this selector. Note that the status of the tick is always determined by its children, not by the user customised update function.

Yields *Behaviour* – a reference to itself or one of its children

class py_trees.composites.**Sequence**(*name='Sequence'*, *children=None*, **args*, ***kwargs*)
Bases: *py_trees.composites.Composite*

Sequences are the factory lines of Behaviour Trees



A sequence will progressively tick over each of its children so long as each child returns *SUCCESS*. If any child returns *FAILURE* or *RUNNING* the sequence will halt and the parent will adopt the result of this child. If it reaches the last child, it returns with that result regardless.

Note: The sequence halts once it sees a child is RUNNING and then returns the result. *It does not get stuck in the running behaviour.*

See also:

The *py-trees-demo-sequence* program demos a simple sequence in action.

Parameters

- **name** (str) – the composite behaviour name
- **children** ([*Behaviour*]) – list of children to add
- ***args** – variable length argument list
- ****kwargs** – arbitrary keyword arguments

__init__(*name='Sequence'*, *children=None*, **args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

current_child
Have to check if there's anything actually running first.

Returns the child that is currently running, or None

Return type *Behaviour*

stop(*new_status=<Status.INVALID: 'INVALID'>*)
Stopping a sequence requires taking care of the current index. Note that is important to implement this here intead of terminate, so users are free to subclass this easily with their own terminate and not have to remember that they need to call this function manually.

Parameters **new_status** (*Status*) – the composite is transitioning to this new status

tick()
Tick over the children.

Yields *Behaviour* – a reference to itself or one of its children

13.7 py_trees.console

Simple colour definitions and syntax highlighting for the console.

Colour Definitions

The current list of colour definitions include:

- `Regular`: black, red, green, yellow, blue, magenta, cyan, white,
- `Bold`: bold, bold_black, bold_red, bold_green, bold_yellow, bold_blue, bold_magenta, bold_cyan, bold_white

These colour definitions can be used in the following way:

```
import py_trees.console as console
print(console.cyan + "    Name" + console.reset + ": " + console.yellow + "Dude" +␣
↪console.reset)
```

py_trees.console.**colours = ['', '', '', '', '', '', '', '', '', '', '', '', '', '', '', '',**
List of all available colours.

py_trees.console.**console_has_colours**()
Detects if the console (stdout) has colourising capability.

py_trees.console.**has_colours = False**
Whether the loading program has access to colours or not.

py_trees.console.**logdebug**(*message*)
Prefixes `[DEBUG]` and colours the message green.

Parameters **message** (`str`) – message to log.

py_trees.console.**logerror**(*message*)
Prefixes `[ERROR]` and colours the message red.

Parameters **message** (`str`) – message to log.

py_trees.console.**logfatal**(*message*)
Prefixes `[FATAL]` and colours the message bold red.

Parameters **message** (`str`) – message to log.

py_trees.console.**loginfo**(*message*)
Prefixes `[ INFO]` to the message.

Parameters **message** (`str`) – message to log.

py_trees.console.**logwarn**(*message*)
Prefixes `[ WARN]` and colours the message yellow.

Parameters **message** (str) – message to log.

py_trees.console.**read_single_keypress**()
Waits for a single keypress on stdin.

This is a silly function to call if you need to do it a lot because it has to store stdin's current setup, setup stdin for reading single keystrokes then read the single keystroke then revert stdin back after reading the keystroke.

Returns the character of the key that was pressed

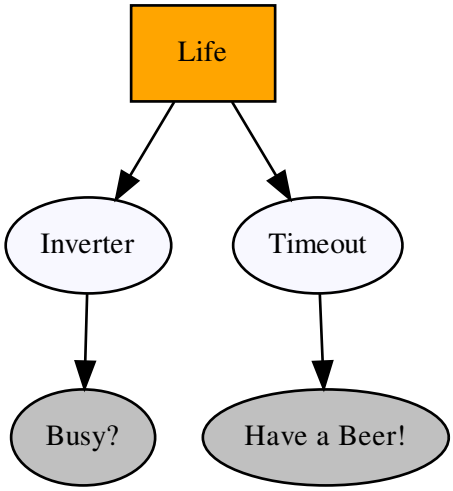
Return type int

Raises KeyboardInterrupt – if CTRL-C was pressed (keycode 0x03)

13.8 py_trees.decorators

Decorators are behaviours that manage a single child and provide common modifications to their underlying child behaviour (e.g. inverting the result). i.e. they provide a means for behaviours to wear different 'hats' depending on their context without a behaviour tree.

An example:




```
#!/usr/bin/env python

import py_trees.decorators
import py_trees.display
```

(continues on next page)

(continued from previous page)

```
5
6  if __name__ == '__main__':
7
8      root = py_trees.composites.Sequence(name="Life")
9      timeout = py_trees.decorators.Timeout(
10         name="Timeout",
11         child=py_trees.behaviours.Success(name="Have a Beer!")
12     )
13     failure_is_success = py_trees.decorators.Inverter(
14         name="Inverter",
15         child=py_trees.behaviours.Success(name="Busy?")
16         )
17     root.add_children([failure_is_success, timeout])
18     py_trees.display.render_dot_tree(root)
```

Decorators (Hats)

Decorators with very specific functionality:

- *py_trees.decorators.Condition()*
- *py_trees.decorators.Inverter()*
- *py_trees.decorators.OneShot()*
- py_trees.decorators.TimeOut()

And the X is Y family:

- *py_trees.decorators.FailureIsRunning()*
- *py_trees.decorators.FailureIsSuccess()*
- *py_trees.decorators.RunningIsFailure()*
- *py_trees.decorators.RunningIsSuccess()*
- *py_trees.decorators.SuccessIsFailure()*
- *py_trees.decorators.SuccessIsRunning()*

class py_trees.decorators.**Condition**(*child*, *name=<Name.AUTO_GENERATED: 'AUTO_GENERATED'>*, *status=<Status.SUCCESS: 'SUCCESS'>*)

Bases: *py_trees.decorators.Decorator*

Encapsulates a behaviour and wait for it's status to flip to the desired state. This behaviour will tick with *RUNNING* while waiting and *SUCCESS* when the flip occurs.

__init__(*child*, *name=<Name.AUTO_GENERATED: 'AUTO_GENERATED'>*, *status=<Status.SUCCESS: 'SUCCESS'>*)

Initialise with child and optional name, status variables. :param child: the child to be decorated :type child: *Behaviour* :param name: the decorator name (can be None) :type name: str :param status: the desired status to watch for :type status: *Status*

update()

SUCCESS if the decorated child has returned the specified status, otherwise *RUNNING*. This decorator will never return *FAILURE* :returns: the behaviour's new status *Status* :rtype: *Status*

class py_trees.decorators.**Decorator**(*child*, *name=<Name.AUTO_GENERATED: 'AUTO_GENERATED'>*)

Bases: *py_trees.behaviour.Behaviour*

A decorator is responsible for handling the lifecycle of a single child beneath

__init__(*child*, *name=<Name.AUTO_GENERATED: 'AUTO_GENERATED'>*)
Common initialisation steps for a decorator - type checks and name construction (if None is given).

Parameters
- **name** (str) – the decorator name (can be None)
- **child** (*Behaviour*) – the child to be decorated

Raises TypeError – if the child is not an instance of *Behaviour*

setup(*timeout*)
Relays to the decorated child's *setup()* method. :param timeout: time to wait (0.0 is blocking forever) :type timeout: float

Raises TypeError – if children's setup methods fail to return a boolean
Returns suceess or failure of the operation
Return type bool

stop(*new_status*)
As with other composites, it checks if the child is running and stops it if that is the case. :param new_status: the behaviour is transitioning to this new status :type new_status: *Status*

tick()
A decorator's tick is exactly the same as a normal proceedings for a Behaviour's tick except that it also ticks the decorated child node.

Yields *Behaviour* – a reference to itself or one of its children

class py_trees.decorators.**FailureIsRunning**(*child*, *name=<Name.AUTO_GENERATED: 'AUTO_GENERATED'>*)
Bases: *py_trees.decorators.Decorator*

Dont stop running.

update()
Return the decorated child's status unless it is *FAILURE* in which case, return *RUNNING*. :returns: the behaviour's new status *Status* :rtype: *Status*

class py_trees.decorators.**FailureIsSuccess**(*child*, *name=<Name.AUTO_GENERATED: 'AUTO_GENERATED'>*)
Bases: *py_trees.decorators.Decorator*

Be positive, always succeed.

update()
Return the decorated child's status unless it is *FAILURE* in which case, return *SUCCESS*. :returns: the behaviour's new status *Status* :rtype: *Status*

class py_trees.decorators.**Inverter**(*child*, *name=<Name.AUTO_GENERATED: 'AUTO_GENERATED'>*)
Bases: *py_trees.decorators.Decorator*

A decorator that inverts the result of a class's update function.

__init__(*child*, *name=<Name.AUTO_GENERATED: 'AUTO_GENERATED'>*)
Init with the decorated child.

Parameters
- **child** (*Behaviour*) – behaviour to time
- **name** (str) – the decorator name

update()
Flip *FAILURE* and *SUCCESS* :returns: the behaviour's new status *Status* :rtype: *Status*

class py_trees.decorators.**OneShot**(*child*, *name=<Name.AUTO_GENERATED: 'AUTO_GENERATED'>*)

Bases: *py_trees.decorators.Decorator*

A decorator that implements the oneshot pattern. This decorator ensures that the underlying child is ticked through to *successful* completion just once and while doing so, will return with the same status as it's child. Thereafter it will return *SUCCESS*.

See also:

`py_trees.idioms.oneshot()`

__init__(*child*, *name=<Name.AUTO_GENERATED: 'AUTO_GENERATED'>*)

Init with the decorated child.

Parameters

- **child** (*Behaviour*) – behaviour to time
- **name** (`str`) – the decorator name

terminate(*new_status*)

If returning *SUCCESS* for the first time, flag it so future ticks will block entry to the child.

tick()

Select between decorator (single child) and behaviour (no children) style ticks depending on whether or not the underlying child has been ticked successfully to completion previously.

update()

Bounce if the child has already successfully completed.

class py_trees.decorators.**RunningIsFailure**(*child*, *name=<Name.AUTO_GENERATED: 'AUTO_GENERATED'>*)

Bases: *py_trees.decorators.Decorator*

Got to be snappy! We want results...yesterday!

update()

Return the decorated child's status unless it is *RUNNING* in which case, return *FAILURE*. :returns: the behaviour's new status *Status* :rtype: *Status*

class py_trees.decorators.**RunningIsSuccess**(*child*, *name=<Name.AUTO_GENERATED: 'AUTO_GENERATED'>*)

Bases: *py_trees.decorators.Decorator*

Don't hang around...

update()

Return the decorated child's status unless it is *RUNNING* in which case, return *SUCCESS*. :returns: the behaviour's new status *Status* :rtype: *Status*

class py_trees.decorators.**SuccessIsFailure**(*child*, *name=<Name.AUTO_GENERATED: 'AUTO_GENERATED'>*)

Bases: *py_trees.decorators.Decorator*

Be depressed, always fail.

update()

Return the decorated child's status unless it is *SUCCESS* in which case, return *FAILURE*. :returns: the behaviour's new status *Status* :rtype: *Status*

class py_trees.decorators.**SuccessIsRunning**(*child*, *name=<Name.AUTO_GENERATED: 'AUTO_GENERATED'>*)

Bases: *py_trees.decorators.Decorator*

It never ends...

update()
Return the decorated child's status unless it is *SUCCESS* in which case, return *RUNNING*. :returns: the behaviour's new status *Status* :rtype: *Status*

class py_trees.decorators.**Timeout**(*child*, *name=<Name.AUTO_GENERATED: 'AUTO_GENERATED'>*, *duration=5.0*)

Bases: *py_trees.decorators.Decorator*

A decorator that applies a timeout pattern to an existing behaviour. If the timeout is reached, the encapsulated behaviour's *stop()* method is called with status *FAILURE* otherwise it will simply directly tick and return with the same status as that of it's encapsulated behaviour.

__init__(*child*, *name=<Name.AUTO_GENERATED: 'AUTO_GENERATED'>*, *duration=5.0*)
Init with the decorated child and a timeout duration.

Parameters
- **child** (*Behaviour*) – behaviour to time
- **name** (str) – the decorator name
- **duration** (float) – timeout length in seconds

initialise()
Reset the feedback message and finish time on behaviour entry.

update()
Terminate the child and return *FAILURE* if the timeout is exceeded.

13.9 py_trees.display

Behaviour trees are significantly easier to design, monitor and debug with visualisations. Py Trees does provide minimal assistance to render trees to various simple output formats. Currently this includes dot graphs, strings or stdout.

py_trees.display.**ascii_bullet**(*node*)
Generate a text bullet for the specified behaviour's type.

Parameters **node** (*Behaviour*) – convert this behaviour's type to text

Returns the text bullet

Return type str)

py_trees.display.**ascii_check_mark**(*status*)
Generate a text check mark for the specified status.

Parameters **status** (*Status*) – convert this status to text

Returns the text check mark

Return type str)

py_trees.display.**ascii_tree**(*tree*, *indent=0*, *snapshot_information=None*)
Build an ascii tree representation as a string for redirecting to elsewhere other than stdout. This can be the entire tree, or a recorded snapshot of the tree (i.e. just the part that was traversed).

Parameters

- **tree** (*Behaviour*) – the root of the tree, or subtree you want to show
- **indent** (int) – the number of characters to indent the tree
- **snapshot_information** (*visitors*) – a visitor that recorded information about a traversed tree (e.g. *SnapshotVisitor*)
- **snapshot_information** – a visitor that recorded information about a traversed tree (e.g. *SnapshotVisitor*)

Returns an ascii tree (i.e. in string form)

Return type str

Examples

Use the *SnapshotVisitor* and *BehaviourTree* to generate snapshot information at each tick and feed that to a post tick handler that will print the traversed ascii tree complete with status and feedback messages.


```
def post_tick_handler(snapshot_visitor, behaviour_tree):
    print(py_trees.display.ascii_tree(behaviour_tree.root,
          snapshot_information=snapshot_visitor))

root = py_trees.composites.Sequence("Sequence")
for action in ["Action 1", "Action 2", "Action 3"]:
    b = py_trees.behaviours.Count(
            name=action,
            fail_until=0,
            running_until=1,
            success_until=10)
    root.add_child(b)
behaviour_tree = py_trees.trees.BehaviourTree(root)
snapshot_visitor = py_trees.visitors.SnapshotVisitor()
behaviour_tree.add_post_tick_handler(
    functools.partial(post_tick_handler,
                      snapshot_visitor))
behaviour_tree.visitors.append(snapshot_visitor)
```

py_trees.display.**generate_pydot_graph**(*root*, *visibility_level*, *collapse_decorators=False*)

Generate the pydot graph - this is usually the first step in rendering the tree to file. See also *render_dot_tree()*.

Parameters

- **root** (*Behaviour*) – the root of a tree, or subtree
- **(** (*visibility_level*) – class‘~py_trees.common.VisibilityLevel‘): collapse subtrees at or under this level
- **collapse_decorators** (bool) – only show the decorator (not the child)

Returns graph

Return type pydot.Dot

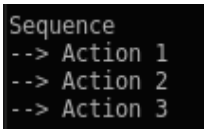
py_trees.display.**print_ascii_tree**(*root*, *indent=0*, *show_status=False*)
Print the ASCII representation of an entire behaviour tree.

Parameters

- **root** (*Behaviour*) – the root of the tree, or subtree you want to show
- **indent** (int) – the number of characters to indent the tree
- **show_status** (bool) – additionally show feedback message and status of every element

Examples

Render a simple tree in ascii format to stdout.




```
root = py_trees.composites.Sequence("Sequence")
for action in ["Action 1", "Action 2", "Action 3"]:
    b = py_trees.behaviours.Count(
            name=action,
            fail_until=0,
            running_until=1,
            success_until=10)
    root.add_child(b)
py_trees.display.print_ascii_tree(root)
```

Tip: To additionally display status and feedbback message from every behaviour in the tree, simply set the show_status flag to True.

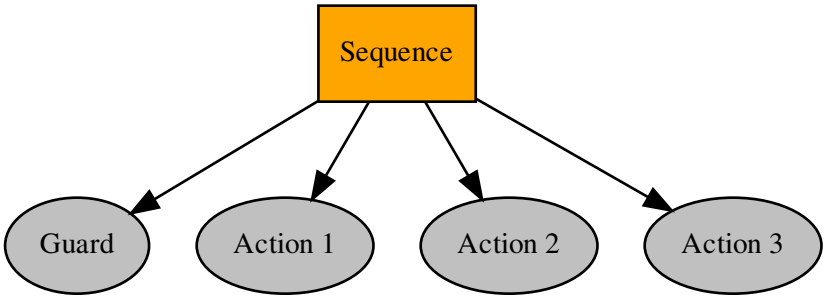
py_trees.display.**render_dot_tree**(*root*, *visibility_level=<VisibilityLevel.DETAIL: 1>*, *collapse_decorators=False*, *name=None*)
Render the dot tree to .dot, .svg, .png. files in the current working directory. These will be named with the root behaviour name.

Parameters

- **root** (*Behaviour*) – the root of a tree, or subtree
- **(** (*visibility_level*) – class‘~py_trees.common.VisibilityLevel‘): collapse subtrees at or under this level
- **collapse_decorators** (bool) – only show the decorator (not the child)
- **name** (str) – name to use for the created files (defaults to the root behaviour name)

Example

Render a simple tree to dot/svg/png file:




```
root = py_trees.composites.Sequence("Sequence")
for job in ["Action 1", "Action 2", "Action 3"]:
    success_after_two = py_trees.behaviours.Count(name=job,
                                                  fail_until=0,
                                                  running_until=1,
                                                  success_until=10)
    root.add_child(success_after_two)
py_trees.display.render_dot_tree(root)
```

Tip: A good practice is to provide a command line argument for optional rendering of a program so users can quickly visualise what tree the program will execute.

py_trees.display.**stringify_dot_tree**(*root*)

Generate dot tree graphs and return a string representation of the dot graph.

Parameters **root** (*Behaviour*) – the root of a tree, or subtree

Returns dot graph as a string

Return type str

13.10 py_trees.meta

Attention: This module is the least likely to remain stable in this package. It has only received cursory attention so far and a more thoughtful design for handling behaviour 'hats' might be needful at some point in the future.

Meta behaviours are created by utilising various programming techniques pulled from a magic bag of tricks. Some of these minimise the effort to generate a new behaviour while others provide mechanisms that greatly expand your library of usable behaviours without having to increase the number of explicit behaviours contained therein. The latter is achieved by providing a means for behaviours to wear different 'hats' via python decorators.

Each function or decorator listed below includes its own example code demonstrating its use.

Factories

- *py_trees.meta.create_behaviour_from_function()*
- *py_trees.meta.create_imposter()*

Decorators (Hats)

- *py_trees.meta.condition()*
- *py_trees.meta.inverter()*
- *py_trees.meta.failure_is_running()*
- *py_trees.meta.failure_is_success()*
- py_trees.meta.oneshot()
- *py_trees.meta.running_is_failure()*
- *py_trees.meta.running_is_success()*
- *py_trees.meta.success_is_failure()*
- *py_trees.meta.success_is_running()*
- *py_trees.meta.timeout()*

py_trees.meta.**condition**(*cls*, *status*)

Encapsulates a behaviour and wait for it's status to flip to the desired state. This behaviour will tick with *RUNNING* while waiting and *SUCCESS* when the flip occurs.

Parameters

- **cls** (*Behaviour*) – an existing behaviour class type
- **status** (*Status*) – the desired status to watch for

Returns the modified behaviour class

Return type *Behaviour*

Examples

```
@condition(py_trees.common.Status.RUNNING)
class HangingAbout(WillStartSoon)
    pass
```

or

```
hanging_about = condition(WillStartSoon, py_trees.common.Status.RUNNING)(name=
↪"Hanging About")
```

py_trees.meta.**create_behaviour_from_function**(*func*)
Create a behaviour from the specified function, dropping it in for the Behaviour *update()* method. Ths function must include the *self* argument and return a `Status` value. It also automatically provides a drop-in for the *terminate()* method that clears the feedback message. Other methods are left untouched.

Parameters **func** (`function`) – a drop-in for the *update()* method

py_trees.meta.**create_imposter**(*cls*)
Creates a new behaviour type impersonating (encapsulating) another behaviour type.

This is primarily used to develop other decorators but can also be useful in itself. It takes care of the handles responsible for making the encapsulation work and leaves you with just the task of replacing the relevant modifications (usually to the *update()* method). The modifications can be made by direct replacement of methods or by inheriting and overriding them. See the examples below.

Parameters **cls** (*Behaviour*) – an existing behaviour class type

Returns the new encapsulated behaviour class

Return type *Behaviour*

Examples

Replacing methods:

```
def _update(self):
    self.original.tick_once()
    if self.original.status == common.Status.FAILURE:
        return common.Status.SUCCESS
    else:
        return self.original.status

FailureIsSuccess = create_imposter(py_trees.behaviours.Failure)
setattr(FailureIsSuccess, "update", _update)
```

Subclassing and overriding:

```
class FailureIsSuccess(create_imposter(py_trees.behaviours.Failure)):

    def __init__(self, *args, **kwargs):
        super(FailureIsSuccess, self).__init__(*args, **kwargs)

    def update(self):
        self.original.tick_once()
        if self.original.status == common.Status.FAILURE:
            return common.Status.SUCCESS
        else:
            return self.original.status
```

py_trees.meta.**failure_is_running**(*cls*)
Dont stop running.

Parameters **cls** (*Behaviour*) – an existing behaviour class type

Returns the modified behaviour class

Return type *Behaviour*

Examples

```
@failure_is_running
class MustGoOnRegardless(ActingLikeAGoon)
    pass
```

or

```
must_go_on_regardless = failure_is_running(ActingLikeAGoon)(name="Goon")
```

py_trees.meta.**failure_is_success**(*cls*)

Be positive, always succeed.

Parameters **cls** (*Behaviour*) – an existing behaviour class type

Returns the modified behaviour class

Return type *Behaviour*

Examples

```
@failure_is_success
class MustGoOnRegardless(ActedLikeAGoon)
    pass
```

or

```
must_go_on_regardless = failure_is_success(ActedLikeAGoon)(name="Goon")
```

py_trees.meta.**inverter**(*cls*)

A decorator that inverts the result of a class's update function.

Parameters **cls** (*Behaviour*) – an existing behaviour class type

Returns the modified behaviour class

Return type *Behaviour*

Examples

```
@inverter
class Failure(Success)
    pass
```

or

```
failure = inverter(Success)("Failure")
```

py_trees.meta.**running_is_failure**(*cls*)

Got to be snappy! We want results...yesterday!

Parameters **cls** (*Behaviour*) – an existing behaviour class type

Returns the modified behaviour class

Return type *Behaviour*

Examples

```
@running_is_failure
class NeedResultsNow(Pontificating)
    pass
```

or

```
need_results_now = running_is_failure(Pontificating)("Greek Philosopher")
```

py_trees.meta.**running_is_success**(*cls*)
Don't hang around...

Parameters **cls** (*Behaviour*) – an existing behaviour class type

Returns the modified behaviour class

Return type *Behaviour*

Examples

```
@running_is_success
class DontHangAround(Pontificating)
    pass
```

or

```
dont_hang_around = running_is_success(Pontificating)("Greek Philosopher")
```

py_trees.meta.**success_is_failure**(*cls*)
Be depressed, always fail.

Parameters **cls** (*Behaviour*) – an existing behaviour class type

Returns the modified behaviour class

Return type *Behaviour*

Examples

```
@success_is_failure
class TheEndIsNigh(ActingLikeAGoon)
    pass
```

or

```
the_end_is_nigh = success_is_failure(ActingLikeAGoon)(name="Goon")
```

py_trees.meta.**success_is_running**(*cls*)
It never ends...

Parameters **cls** (*Behaviour*) – an existing behaviour class type

Returns the modified behaviour class

Return type *Behaviour*

Examples

```
@success_is_running
class TheEndIsSillNotNigh(ActingLikeAGoon)
    pass
```

or

```
the_end_is_still_not_nigh = success_is_running(ActingLikeAGoon)(name="Goon")
```

py_trees.meta.**timeout**(*cls*, *duration*)

A decorator that applies a timeout pattern to an existing behaviour. If the timeout is reached, the encapsulated behaviour's *stop()* method is called with status *FAILURE* otherwise it will simply directly tick and return with the same status as that of it's encapsulated behaviour.

Parameters

- **cls** (*Behaviour*) – an existing behaviour class type
- **duration** (float) – timeout length in seconds

Returns the modified behaviour class with timeout

Return type *Behaviour*

Examples

```
@py_trees.meta.timeout(10)
class WorkBehaviour(py_trees.behaviour.Behaviour)
```

or

```
work_with_timeout = py_trees.meta.timeout(WorkBehaviour, 10.0)(name="Work")
```

13.11 py_trees.timers

Time related behaviours.

class py_trees.timers.**Timer**(*name='Timer'*, *duration=5.0*)

Bases: *py_trees.behaviour.Behaviour*

Simple timer class that is *RUNNING* until the timer runs out, at which point it is *SUCCESS*. This can be used in a wide variety of situations - pause, duration, timeout depending on how it is wired into the tree (e.g. pause in a sequence, duration/timeout in a parallel).

The timer gets reset either upon entry (*initialise()*) if it hasn't already been set and gets cleared when it either runs out, or the behaviour is interrupted by a higher priority or parent cancelling it.

Parameters

- **name** (str) – name of the behaviour
- **duration** (int) – length of time to run (in seconds)

Note: This succeeds the first time the behaviour is ticked **after** the expected finishing time.

Tip: Use the *running_is_failure()* decorator if you need *FAILURE* until the timer finishes.

__init__ (*name='Timer'*, *duration=5.0*)
Initialize self. See help(type(self)) for accurate signature.

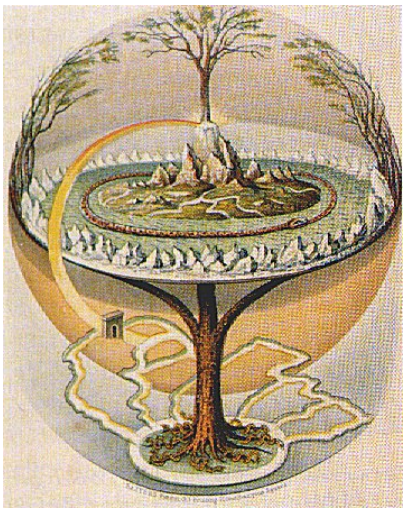
initialise()
Store the expected finishing time.

terminate(*new_status*)
Clear the expected finishing time.

update()
Check current time against the expected finishing time. If it is in excess, flip to *SUCCESS*.

13.12 py_trees.trees

While a graph of connected behaviours and composites form a tree in their own right (i.e. it can be initialised and ticked), it is usually convenient to wrap your tree in another class to take care of alot of the housework and provide some extra bells and whistles that make your tree flourish.



This package provides a default reference implementation that is directly usable, but can also be easily used as inspiration for your own tree custodians.

class py_trees.trees.**BehaviourTree**(*root*)
Bases: object

Grow, water, prune your behaviour tree with this, the default reference implementation. It features a few enhancements to provide richer logging, introspection and dynamic management of the tree itself:

- Pre and post tick handlers to execute code automatically before and after a tick
- Visitor access to the parts of the tree that were traversed in a tick
- Subtree pruning and insertion operations
- Continuous tick-tock support

See also:

The *py-trees-demo-tree-stewardship* program demonstrates the above features.

Parameters **root** (*Behaviour*) – root node of the tree

Variables

- **count** (int) – number of times the tree has been ticked.
- **root** (*Behaviour*) – root node of the tree
- ***visitors*** ([*visitors*]) – entities that visit traversed parts of the tree when it ticks
- **pre_tick_handlers** ([func]) – functions that run before the entire tree is ticked
- **post_tick_handlers** ([func]) – functions that run after the entire tree is ticked

Raises AssertionError – if incoming root variable is not the correct type

add_post_tick_handler(*handler*)

Add a function to execute after the tree has ticked. The function must have a single argument of type *BehaviourTree*.

Some ideas that are often used:

- logging
- modifications on the tree itself (e.g. closing down a plan)
- sending data to visualisation tools
- introspect the state of the tree to make and send reports

Parameters **handler** (func) – function

add_pre_tick_handler(*handler*)

Add a function to execute before the tree is ticked. The function must have a single argument of type *BehaviourTree*.

Some ideas that are often used:

- logging (to file or stdout)
- modifications on the tree itself (e.g. starting a new plan)

Parameters **handler** (func) – function

destroy()

Destroy the tree by stopping the root node.

insert_subtree(*child*, *unique_id*, *index*)

Insert a subtree as a child of the specified parent. If the parent is found, this directly calls the parent's *insert_child()* method using the child and index arguments.

Parameters

- **child** (*Behaviour*) – subtree to insert
- **unique_id** (*uuid.UUID*) – unique id of the parent
- **index** (int) – insert the child at this index, pushing all children after it back one.

Returns suceess or failure (parent not found) of the operation

Return type bool

Raises AssertionError – if the parent is not a *Composite*

Todo: Could use better, more informative error handling here. Especially if the insertion has its own error handling (e.g. index out of range). Could also use a different api that relies on the id of the sibling node it should be inserted before/after.

interrupt()
Interrupt tick-tock if it is tick-tocking. Note that this will permit a currently executing tick to finish before interrupting the tick-tock.

prune_subtree(*unique_id*)
Prune a subtree given the unique id of the root of the subtree.

Parameters **unique_id** (*uuid.UUID*) – unique id of the subtree root

Returns success or failure of the operation

Return type bool

Raises AssertionError – if unique id is the behaviour tree's root node id

replace_subtree(*unique_id*, *subtree*)
Replace the subtree with the specified id for the new subtree. This is a common pattern where we'd like to swap out a whole sub-behaviour for another one.

Parameters

- **unique_id** (*uuid.UUID*) – unique id of the parent
- **subtree** (*Behaviour*) – root behaviour of the subtree

Raises AssertionError: if unique id is the behaviour tree's root node id

Returns suceess or failure of the operation

Return type bool

setup(*timeout*)

Relays to calling the setup() method on the root behaviour. This in turn should get recursively called down through the entire tree.

Parameters **timeout** (float) – time to wait (0.0 is blocking forever)

Returns suceess or failure of the operation

Return type bool

tick(*pre_tick_handler=None*, *post_tick_handler=None*)
Tick the tree just once and run any handlers before and after the tick. This optionally accepts some one-shot handlers (c.f. those added by *add_pre_tick_handler()* and *add_post_tick_handler()* which will be automatically run every time).

The handler functions must have a single argument of type *BehaviourTree*.

Parameters

- **pre_tick_handler** (func) – function to execute before ticking
- **post_tick_handler** (func) – function to execute after ticking

tick_tock(*sleep_ms*, *number_of_iterations=-1*, *pre_tick_handler=None*, *post_tick_handler=None*)
Tick continuously with a sleep interval as specified. This optionally accepts some handlers that will be used for the duration of this tick tock (c.f. those added by *add_pre_tick_handler()* and *add_post_tick_handler()* which will be automatically run every time).

The handler functions must have a single argument of type *BehaviourTree*.

Parameters

- **sleep_ms** (float) – sleep this much between ticks (milliseconds)
- **number_of_iterations** (int) – number of iterations to tick-tock
- **pre_tick_handler** (func) – function to execute before ticking
- **post_tick_handler** (func) – function to execute after ticking

tip()
Get the *tip* of the tree. This corresponds to the the deepest node that was running before the subtree traversal reversed direction and headed back to this node.

Returns child behaviour, itself or None if its status is *INVALID*

Return type *Behaviour* or None

See also:

tip()

13.13 py_trees.utilities

Assorted utility functions.

py_trees.utilities.**static_variables**(***kwargs*)
This is a decorator that can be used with python methods to attach initialised static variables to the method.

py_trees.utilities.**which**(*program*)
Wrapper around the command line 'which' program.

Parameters **program** (str) – name of the program to find.

Returns path to the program or None if it doesnt exist.

Return type str

13.14 py_trees.visitors

Visitors are entities that can be passed to a tree implementation (e.g. *BehaviourTree*) and used to either visit each and every behaviour in the tree, or visit behaviours as the tree is traversed in an executing tick. At each behaviour, the visitor runs its own method on the behaviour to do as it wishes - logging, introspecting, etc.

Warning: Visitors should not modify the behaviours they visit.

class py_trees.visitors.**DebugVisitor**
Bases: *py_trees.visitors.VisitorBase*

Picks up and logs feedback messages and the behaviour's status. Logging is done with the behaviour's logger.

run(*behaviour*)
This method gets run as each behaviour is ticked. Override it to perform some activity - e.g. introspect the behaviour to store/process logging data for visualisations.

Parameters **behaviour** (*Behaviour*) – behaviour that is ticking

class py_trees.visitors.**SnapshotVisitor**(*full=False*)
Bases: *py_trees.visitors.VisitorBase*

Visits the tree in tick-tock, recording runtime information for publishing the information as a snapshot view of the tree after the iteration has finished.

Parameters **full** (bool) – flag to indicate whether it should be used to visit only traversed nodes or the entire tree

Variables

- **nodes** (*dict*) – dictionary of behaviour id (uuid.UUID) and status (*Status*) pairs
- **running_nodes** (*[uuid.UUID]*) – list of id's for behaviours which were traversed in the current tick
- **previously_running_nodes** (*[uuid.UUID]*) – list of id's for behaviours which were traversed in the last tick

See also:

This visitor is used with the *BehaviourTree* class to collect information and *ascii_tree()* to display information.

initialise()
Switch running to previously running and then reset all other variables. This will get called before a tree ticks.

run(*behaviour*)
This method gets run as each behaviour is ticked. Catch the id and status and store it. Additionally add it to the running list if it is *RUNNING*.

Parameters **behaviour** (*Behaviour*) – behaviour that is ticking

class py_trees.visitors.**VisitorBase**(*full=False*)
Bases: object

Parent template for visitor types.

Visitors are primarily designed to work with *BehaviourTree* but they can be used in the same way for other tree custodian implementations.

Parameters **full** (bool) – flag to indicate whether it should be used to visit only traversed nodes or the entire tree

Variables **full** (bool) – flag to indicate whether it should be used to visit only traversed nodes or the entire tree

initialise()
Override this method if any resetting of variables needs to be performed between ticks (i.e. visitations).

run(*behaviour*)
This method gets run as each behaviour is ticked. Override it to perform some activity - e.g. introspect the behaviour to store/process logging data for visualisations.

Parameters **behaviour** (*Behaviour*) – behaviour that is ticking

CHAPTER 14

Changelog

14.1 Forthcoming

14.2 0.6.9 (2021-01-10)

- [docs] fix some warnings
- [decorators] setting the child's parent as the decorator

14.3 0.6.7 (2019-02-13)

- [decorators] default option for collapsing decorators (resolves py_trees_ros bug)

14.4 0.6.6 (2019-02-13)

[decorators] new-style decorators can be found in py_trees.decorators [decorators] new-style decorators now stop their running child on completion (SUCCESS||FAILURE) [decorators] onshot now activates upon *successful completion* (SUCCESS only), previously on *any completion* (SUCCESS||FAILURE) [meta] behaviours from functions can now automagically generate names

14.5 0.6.5 (2018-09-19)

- Inverters bugfix - pick up missing feedback messages
- Eliminate costly blackboard variable check feedback message

14.6 0.6.4 (2018-09-19)

- Ascii tree bugfix - replace awkward newlines with spaces

14.7 0.6.3 (2018-09-04)

- Parallels bugfix - don't send own status to running children, invalidate them instead

14.8 0.6.2 (2018-08-31)

- Oneshot bugfix - react to priority interrupts correctly

14.9 0.6.1 (2018-08-20)

- Oneshot bugfix - no longer permanently modifies the original class

14.10 0.6.0 (2018-05-15)

- Python 2/3 compatibility

14.11 0.5.10 (2017-06-17)

- [meta] add children monkeypatching for composite imposters
- [blackboard] check for nested variables in WaitForBlackboard

14.12 0.5.9 (2017-03-25)

- [docs] bugfix image links and rewrite the motivation

14.13 0.5.8 (2017-03-19)

- [infra] setup.py tests_require, not test_require

14.14 0.5.7 (2017-03-01)

- [infra] update maintainer email

14.15 0.5.5 (2017-03-01)

- [docs] many minor doc updates
- [meta] bugfix so that imposter now ticks over composite children
- [trees] method for getting the tip of the tree
- [programs] py-trees-render program added

14.16 0.5.4 (2017-02-22)

- [infra] handle pypi/catkin conflicts with install_requires

14.17 0.5.2 (2017-02-22)

- [docs] disable colour when building
- [docs] sidebar headings
- [docs] dont require project installation

14.18 0.5.1 (2017-02-21)

- [infra] pypi package enabled

14.19 0.5.0 (2017-02-21)

- [ros] components moved to py_trees_ros
- [timeout] bugfix to ensure timeout decorator initialises properly
- [docs] rolled over with napolean style
- [docs] sphinx documentation updated
- [imposter] make sure tip() drills down into composites
- [demos] re-organised into modules

14.20 0.4.0 (2017-01-13)

- [trees] add pre/post handlers after setup, just in case setup fails
- [introspection] do parent lookups so you can crawl back up a tree
- [blackboard] permit init of subscriber2blackboard behaviours
- [blackboard] watchers
- [timers] better feedback messages
- [imposter] ensure stop() directly calls the composited behaviour

14.21 0.3.0 (2016-08-25)

- `failure_is_running decorator` (meta).

14.22 0.2.0 (2016-06-01)

- do terminate properly amongst relevant classes
- blackboxes
- chooser variant of selectors
- bugfix the decorators
- blackboard updates on change only
- improved dot graph creation
- many bugfixes to composites
- subscriber behaviours
- timer behaviours

14.23 0.1.2 (2015-11-16)

- one shot sequences
- abort() renamed more appropriately to stop()

14.24 0.1.1 (2015-10-10)

- lots of bugfixing stabilising py_trees for the spain field test
- complement decorator for behaviours
- dot tree views
- ascii tree and tick views
- use generators and visitors to more efficiently walk/introspect trees
- a first implementation of behaviour trees in python

CHAPTER 15

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index

Symbols

A

B

C

O

P

R

S

T

U

V

W