py_trees Documentation

Release 2.1.6

Daniel Stonier

May 31, 2021

Guide

CHAPTER 1

Introduction

1.1 Quick Start

If you'd like to fast forward to some action, browse the *Demos* or read through the ROS2 Robotics Tutorials which incrementally create a significantly more complex behaviour tree for a robotics scenario (ROS2 knowledge not needed).

1.2 Background

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

Others include hierarchical finite state machines, task networks, and 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

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 Motivation

The use case that drove the early development of py_trees was robotics. In particular, the higher level decision making for a single robot, i.e. the scenario layer. For example, the scenario that enables a robot to navigate through a building to deliver a parcel and return to it's homebase safely.

In scope was any decision making that did not need a low-latency response (typically reactive safety control measures). This included docking/undocking processes, the initial localisation dance, topological path planning, navigation context switching, LED and sound interactions, elevator entry/exit decisions.

Also driving requirements was the need to offload scenario development to non-control engineers (juniors, interns, SWE's) and ensure they could develop and debug as rapidly as possible.

Behaviour trees turned out to be a perfect fit after attempts with finite state machines became entangled in wiring complexity as the problem grew in scope.

1.4 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 do **not need to be real time reactive**.

Rapid Development: Python was chosen as the language of choice since it enables a faster a cycle of development as well as a shorter learning curve (critical if you would like to shift the burden away from c++ control engineers to juniors/interns/software engineers).

Medium Scale: Robotic scenarios for a single robot tend to be, maximally in the order of hundreds of behaviours. This is in contrast to game NPC's which need to be executing thousands of behaviours and/or trees and consequently, frequently run into problems of scale. This tends to influence the language of choice (c++) and the tree design. Our requirements are somewhat more modest, so this permits some flexibility in the design, e.g. python as a language of choice.

Not Real Time Reactive: If low latency control measures, particularly for safety are needed, they are best handled directly inside the control layer, or even better, at an embedded level. This is not dissimilar to the way the human nervous system operates. All other decision making needs only to operate at a latency of ~50-200ms to negate any discernable delay observed by humans interacting with the robot.

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..

1.5 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 example:

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import py_trees
import random


class Foo(py_trees.behaviour.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):
        """
        When is this called?
          This function should be either manually called by your program
          to setup this behaviour alone, or more commonly, via
```

(continues on next page)

(continued from previous page)

```
            :meth:`~py_trees.behaviour.Behaviour.setup_with_descendants`
            or :meth:`~py_trees.trees.BehaviourTree.setup`, both of which
            will iterate over this behaviour, it's children (it's children's
            children ...) calling :meth:`~py_trees.behaviour.Behaviour.setup`
            on each in turn.

            If you have vital initialisation necessary to the success
            execution of your behaviour, put a guard in your
            :meth:`~py_trees.behaviour.Behaviour.initialise` method
            to protect against entry without having been setup.

        What to do here?
            Delayed one-time initialisation that would otherwise interfere
            with offline rendering of this behaviour in a tree to dot graph
            or validation of the behaviour's configuration.

            Good examples include:

            - Hardware or driver initialisation
            - Middleware initialisation (e.g. ROS pubs/subs/services)
            - A parallel checking for a valid policy configuration after
              children have been added or removed
        """
        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.common.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.common.Status.RUNNING
        elif decision:
            self.feedback_message = "We are not bar!"
            return py_trees.common.Status.SUCCESS
        else:
            self.feedback_message = "Uh oh"
```

(continues on next page)

(continued from previous page)

```
                return py_trees.common.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
        """
        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

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

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 a lot 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 catastrophic 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 a lot 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.

CHAPTER 3

Composites

Composites are responsible for directing the path traced through the tree on a given tick (execution). They are the **factories** (Sequences and Parallels) and **decision makers** (Selectors) of a behaviour tree.

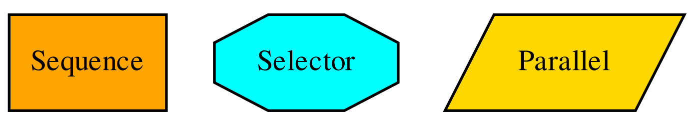


Fig. 1: PyTree Composites

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.

Most any desired functionality can be authored with a combination of these three composites. In fact, it is precisely this feature that makes behaviour trees attractive - it breaks down complex decision making logic to just three primitive elements. It is possible and often desirable to extend this set with custom composites of your own, but think carefully before you do - in almost every case, a combination of the existing composites will serve and as a result, you will merely compound the complexity inherent in your tree logic. This this makes it confoundingly difficult to design, introspect and debug. As an example, design sessions often revolve around a sketched graph on a whiteboard. When these graphs are composed of just five elements (Selectors, Sequences, Parallels, Decorators and Behaviours), it is very easy to understand the logic at a glance. Double the number of fundamental elements and you may as well be back at the terminal parsing code.

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

The basic operational modes of the three composites in this library are as follows:

- *Selector*: select a child to execute based on cascading priorities
- *Sequence*: execute children sequentially
- *Parallel*: execute children concurrently

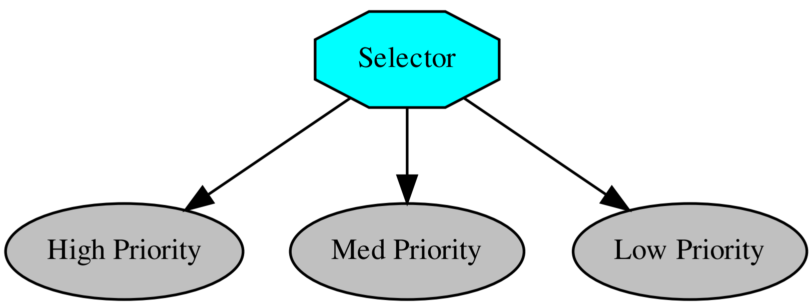
This library does provide some flexibility in *how* each composite is implemented without breaking the fundamental nature of each (as described above). Selectors and Sequences can be configured with or without memory (resumes or resets if children are RUNNING) and the results of a parallel can be configured to wait upon all children completing, succeed on one, all or a subset thereof.

Tip: Follow the links in each composite's documentation to the relevant demo programs.

3.1 Selector

class py_trees.composites.**Selector**(*name='Selector'*, *memory=False*, *children=None*)

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.

See also:

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

Parameters

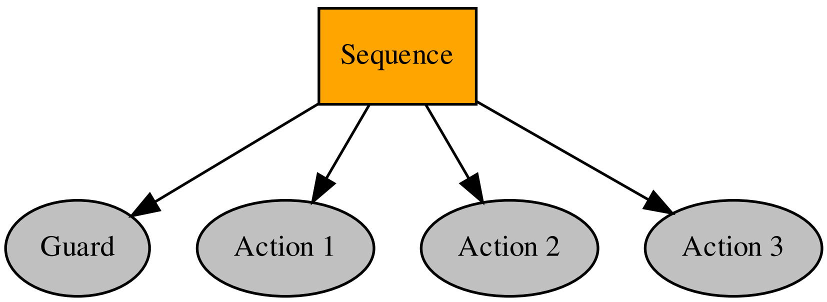
- **name** (str) – the composite behaviour name

- **memory** (bool) – if *RUNNING* on the previous tick, resume with the *RUNNING* child
- **children** ([*Behaviour*]) – list of children to add

3.2 Sequence

class py_trees.composites.**Sequence**(*name='Sequence'*, *memory=True*, *children=None*)

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.

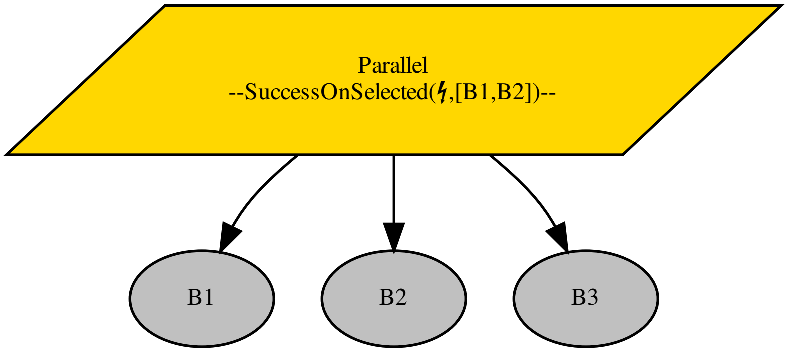
Parameters

- **name** (str) – the composite behaviour name
- **memory** (bool) – if *RUNNING* on the previous tick, resume with the *RUNNING* child
- **children** (Optional[List[*Behaviour*]]) – list of children to add

3.3 Parallel

class py_trees.composites.**Parallel**(*name=<Name.AUTO_GENERATED: 'AUTO_GENERATED'>*, *policy=<py_trees.common.ParallelPolicy.SuccessOnAll object>*, *children=None*)

Parallels enable a kind of concurrency



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

- Parallels will return *FAILURE* if any child returns *FAILURE*
- Parallels with policy *SuccessOnAll* only returns *SUCCESS* if **all** children return *SUCCESS*
- Parallels with policy *SuccessOnOne* return *SUCCESS* if **at least one** child returns *SUCCESS* and others are *RUNNING*
- Parallels with policy *SuccessOnSelected* only returns *SUCCESS* if a **specified subset** of children return *SUCCESS*

Policies *SuccessOnAll* and *SuccessOnSelected* may be configured to be *synchronised* in which case children that tick with *SUCCESS* will be skipped on subsequent ticks until the policy criteria is met, or one of the children returns status *FAILURE*.

Parallels with policy *SuccessOnSelected* will check in both the `setup()` and `tick()` methods to to verify the selected set of children is actually a subset of the children of this parallel.

See also:

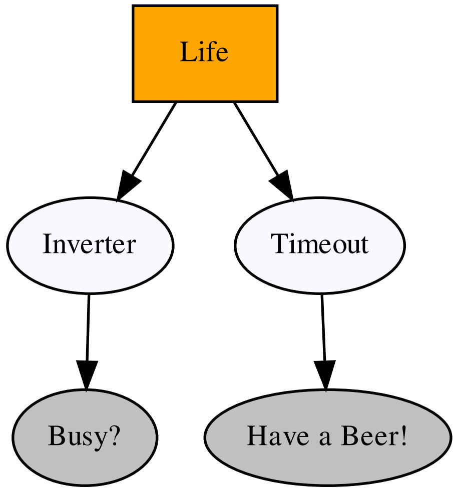
- *Context Switching Demo*

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). That is, they provide a means for behaviours to wear different 'hats' and this combinatorially expands the capabilities of your behaviour library.

An example:




```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

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.EternalGuard*
- *py_trees.decorators.Inverter*
- *py_trees.decorators.OneShot*
- *py_trees.decorators.StatusToBlackboard*
- *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*

Decorators for Blocking Behaviours

It is worth making a note of the effect of decorators on behaviours that return *RUNNING* for some time before finally returning *SUCCESS* or *FAILURE* (blocking behaviours) since the results are often at first, surprising.

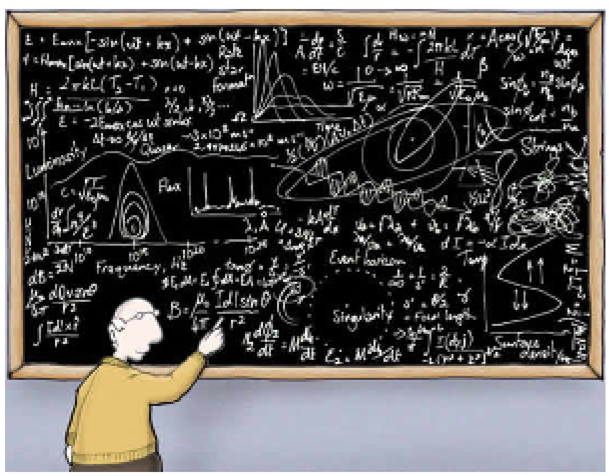
A decorator, such as *py_trees.decorators.RunningIsSuccess()* on a blocking behaviour will immediately terminate the underlying child and re-intialise on it's next tick. This is necessary to ensure the underlying child isn't left in a dangling state (i.e. *RUNNING*), but is often not what is being sought.

The typical use case being attempted is to convert the blocking behaviour into a non-blocking behaviour. If the underlying child has no state being modified in either the *initialise()* or *terminate()* methods (e.g. machinery is entirely launched at init or setup time), then conversion to a non-blocking representative of the original succeeds. Otherwise, another approach is needed. Usually this entails writing a non-blocking counterpart, or combination of behaviours to affect the non-blocking characteristics.

CHAPTER 5

Blackboards

Blackboards are not a necessary component of behaviour tree implementations, but are nonetheless, a fairly common mechanism for sharing data between behaviours in the tree. See, for example, the design notes for blackboards in Unreal Engine.



Implementations vary widely depending on the needs of the framework using them. The simplest implementations take the form of a key-value store with global access, while more rigorous implementations scope access or form a secondary graph overlaying the tree connecting data ports between behaviours.

The *'Zen of PyTrees'* is to enable rapid development, yet be rich enough so that *all* of the magic is exposed for debugging purposes. The first implementation of a blackboard was merely a global key-value store with an api that lent itself to ease of use, but did not expose the data sharing between behaviours which meant any tooling used to introspect or visualise the tree, only told half the story.

The current implementation adopts a strategy similar to that of a filesystem. Each client (subsequently behaviour) registers itself for read/write access to keys on the blackboard. This is less to do with permissions and more to do with tracking users of keys on the blackboard - extremely helpful with debugging.

The alternative approach of layering a secondary data graph with parameter and input-output ports on each behaviour was discarded as being too heavy for the zen requirements of py_trees. This is in part due to the wiring costs, but also due to complexity arising from a tree's partial graph execution (a feature which makes trees different from most computational graph frameworks) and not to regress on py_trees' capability to dynamically insert and prune subtrees on the fly.

A high-level list of existing / planned features:

- [+] Centralised key-value store
- [+] Client connections with namespaced read/write access to the store
- [+] Integration with behaviours for key-behaviour associations (debugging)
- [+] Activity stream that logs read/write operations by clients
- [+] Exclusive locks for writing
- [+] Framework for key remappings

The primary user-facing interface with the blackboard is via the Client.

class py_trees.blackboard.**Client**(*, *name=None*, *namespace=None*)

Client to the key-value store for sharing data between behaviours.

Examples

Blackboard clients will accept a user-friendly name or create one for you if none is provided. Regardless of what name is chosen, clients are always uniquely identified via a uuid generated on construction.

```
provided = py_trees.blackboard.Client(name="Provided")
print(provided)
generated = py_trees.blackboard.Client()
print(generated)
```

Fig. 1: Client Instantiation

Register read/write access for keys on the blackboard. Note, registration is not initialisation.

```
blackboard = py_trees.blackboard.Client(name="Client")
blackboard.register_key(key="foo", access=py_trees.common.Access.WRITE)
blackboard.register_key(key="bar", access=py_trees.common.Access.READ)
blackboard.foo = "foo"
print(blackboard)
```

Keys and clients can make use of namespaces, designed by the '/' char. Most methods permit a flexible expression of either relative or absolute names.

```
Blackboard Client
  Client Data
    name              : Client
    namespace         : /
    unique_identifier : de9cff53-a556-4891-8551-a34495925f73
    read              : {'/bar'}
    write             : {'/foo'}
  Variables
    /foo : foo
    /bar : -
```

Fig. 2: Variable Read/Write Registration

```python
blackboard = py_trees.blackboard.Client(name="Global")
parameters = py_trees.blackboard.Client(name="Parameters", namespace="parameters")

blackboard.register_key(key="foo", access=py_trees.common.Access.WRITE)
blackboard.register_key(key="/bar", access=py_trees.common.Access.WRITE)
blackboard.register_key(key="/parameters/default_speed", access=py_trees.common.Access.WRITE)
parameters.register_key(key="aggressive_speed", access=py_trees.common.Access.WRITE)

blackboard.foo = "foo"
blackboard.bar = "bar"
blackboard.parameters.default_speed = 20.0
parameters.aggressive_speed = 60.0

miss_daisy = blackboard.parameters.default_speed
van_diesel = parameters.aggressive_speed

print(blackboard)
print(parameters)
```

```
Blackboard Client
  Client Data
    name              : Global
    namespace         : /
    unique_identifier : 7b4b6fb3-d677-4e54-b0e9-3100c58fc236
    read              : set()
    write             : {'/foo', '/bar', '/parameters/default_speed'}
  Variables
    /foo                      : foo
    /bar                      : bar
    /parameters/default_speed : 20.0

Blackboard Client
  Client Data
    name              : Parameters
    namespace         : /parameters
    unique_identifier : e20f76b8-4767-4552-92b5-2535cd970d66
    read              : set()
    write             : {'/parameters/aggressive_speed'}
  Variables
    /parameters/aggressive_speed : 60.0
```

Fig. 3: Namespaces and Namespaced Clients

Disconnected instances will discover the centralised key-value store.

```python
def check_foo():
    blackboard = py_trees.blackboard.Client(name="Reader")
    blackboard.register_key(key="foo", access=py_trees.common.Access.READ)
    print("Foo: {}".format(blackboard.foo))


blackboard = py_trees.blackboard.Client(name="Writer")
blackboard.register_key(key="foo", access=py_trees.common.Access.WRITE)
blackboard.foo = "bar"
check_foo()
```

To respect an already initialised key on the blackboard:

```python
blackboard = Client(name="Writer")
blackboard.register_key(key="foo", access=py_trees.common.Access.READ)
result = blackboard.set("foo", "bar", overwrite=False)
```

Store complex objects on the blackboard:

```python
class Nested(object):
    def __init__(self):
        self.foo = None
        self.bar = None

    def __str__(self):
        return str(self.__dict__)


writer = py_trees.blackboard.Client(name="Writer")
writer.register_key(key="nested", access=py_trees.common.Access.WRITE)
reader = py_trees.blackboard.Client(name="Reader")
reader.register_key(key="nested", access=py_trees.common.Access.READ)

writer.nested = Nested()
writer.nested.foo = "I am foo"
writer.nested.bar = "I am bar"

foo = reader.nested.foo
print(writer)
print(reader)
```

Log and display the activity stream:

```
py_trees.blackboard.Blackboard.enable_activity_stream(maximum_size=100)
reader = py_trees.blackboard.Client(name="Reader")
reader.register_key(key="foo", access=py_trees.common.Access.READ)
writer = py_trees.blackboard.Client(name="Writer")
writer.register_key(key="foo", access=py_trees.common.Access.WRITE)
writer.foo = "bar"
writer.foo = "foobar"
unused_result = reader.foo
print(py_trees.display.unicode_blackboard_activity_stream())
py_trees.blackboard.Blackboard.activity_stream.clear()
```

Display the blackboard on the console, or part thereof:

```
writer = py_trees.blackboard.Client(name="Writer")
for key in {"foo", "bar", "dude", "dudette"}:
    writer.register_key(key=key, access=py_trees.common.Access.WRITE)

reader = py_trees.blackboard.Client(name="Reader")
for key in {"foo", "bar"}:
    reader.register_key(key="key", access=py_trees.common.Access.READ)

writer.foo = "foo"
writer.bar = "bar"
writer.dude = "bob"

# all key-value pairs
print(py_trees.display.unicode_blackboard())
# various filtered views
print(py_trees.display.unicode_blackboard(key_filter={"foo"}))
print(py_trees.display.unicode_blackboard(regex_filter="dud*"))
print(py_trees.display.unicode_blackboard(client_filter={reader.unique_identifier}
↪))
# list the clients associated with each key
print(py_trees.display.unicode_blackboard(display_only_key_metadata=True))
```

Behaviours are not automagically connected to the blackboard but you may manually attach one or more clients so that associations between behaviours and variables can be tracked - this is very useful for introspection and debugging.

Creating a custom behaviour with blackboard variables:

```
class Foo(py_trees.behaviour.Behaviour):

    def __init__(self, name):
        super().__init__(name=name)
        self.blackboard = self.attach_blackboard_client(name="Foo Global")
        self.parameters = self.attach_blackboard_client(name="Foo Params",␣
↪namespace="foo_parameters_")
        self.state = self.attach_blackboard_client(name="Foo State", namespace=
↪"foo_state_")
```

(continues on next page)

(continued from previous page)

```
        # create a key 'foo_parameters_init' on the blackboard
        self.parameters.register_key("init", access=py_trees.common.Access.READ)
        # create a key 'foo_state_number_of_noodles' on the blackboard
        self.state.register_key("number_of_noodles", access=py_trees.common.
↪Access.WRITE)

    def initialise(self):
        self.state.number_of_noodles = self.parameters.init

    def update(self):
        self.state.number_of_noodles += 1
        self.feedback_message = self.state.number_of_noodles
        if self.state.number_of_noodles > 5:
            return py_trees.common.Status.SUCCESS
        else:
            return py_trees.common.Status.RUNNING


# could equivalently do directly via the Blackboard static methods if
# not interested in tracking / visualising the application configuration
configuration = py_trees.blackboard.Client(name="App Config")
configuration.register_key("foo_parameters_init", access=py_trees.common.Access.
↪WRITE)
configuration.foo_parameters_init = 3

foo = Foo(name="The Foo")
for i in range(1, 8):
    foo.tick_once()
    print("Number of Noodles: {}".format(foo.feedback_message))
```

Rendering a dot graph for a behaviour tree, complete with blackboard variables:

```
# in code
py_trees.display.render_dot_tree(py_trees.demos.blackboard.create_root())
```

(continues on next page)

(continued from previous page)

```
# command line tools
py-trees-render --with-blackboard-variables py_trees.demos.blackboard.create_root
```

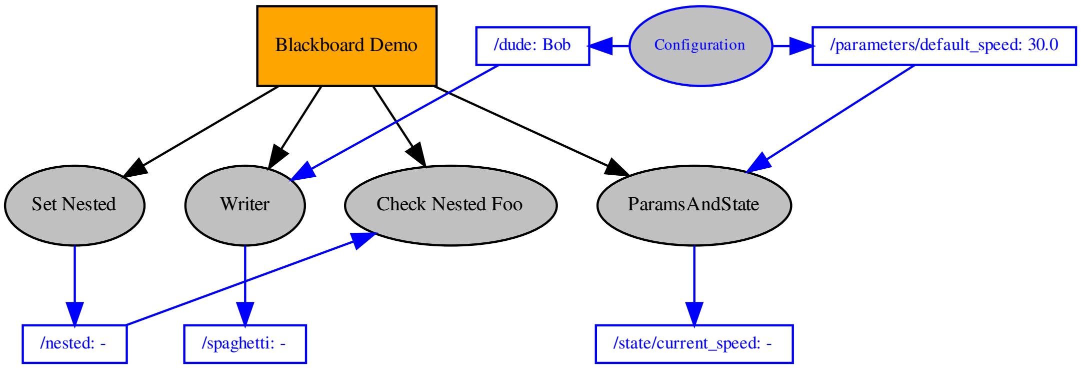


Fig. 4: Tree with Blackboard Variables

And to demonstrate that it doesn't become a tangled nightmare at scale, an example of a more complex tree:

Debug deeper with judicious application of the tree, blackboard and activity stream display methods around the tree tick (refer to `py_trees.visitors.DisplaySnapshotVisitor` for examplar code):

See also:

- *py-trees-demo-blackboard*
- *py-trees-demo-namespaces*
- *py-trees-demo-remappings*
- `py_trees.visitors.DisplaySnapshotVisitor`
- `py_trees.behaviours.SetBlackboardVariable`
- `py_trees.behaviours.UnsetBlackboardVariable`
- `py_trees.behaviours.CheckBlackboardVariableExists`
- `py_trees.behaviours.WaitForBlackboardVariable`
- `py_trees.behaviours.CheckBlackboardVariableValue`
- `py_trees.behaviours.WaitForBlackboardVariableValue`

Variables

- **name** (*str*) – client's convenient, but not necessarily unique identifier
- **namespace** (*str*) – apply this as a prefix to any key/variable name operations
- **unique_identifier** (*uuid.UUID*) – client's unique identifier
- **read** (*typing.Set[str]*) – set of absolute key names with read access
- **write** (*typing.Set[str]*) – set of absolute key names with write access

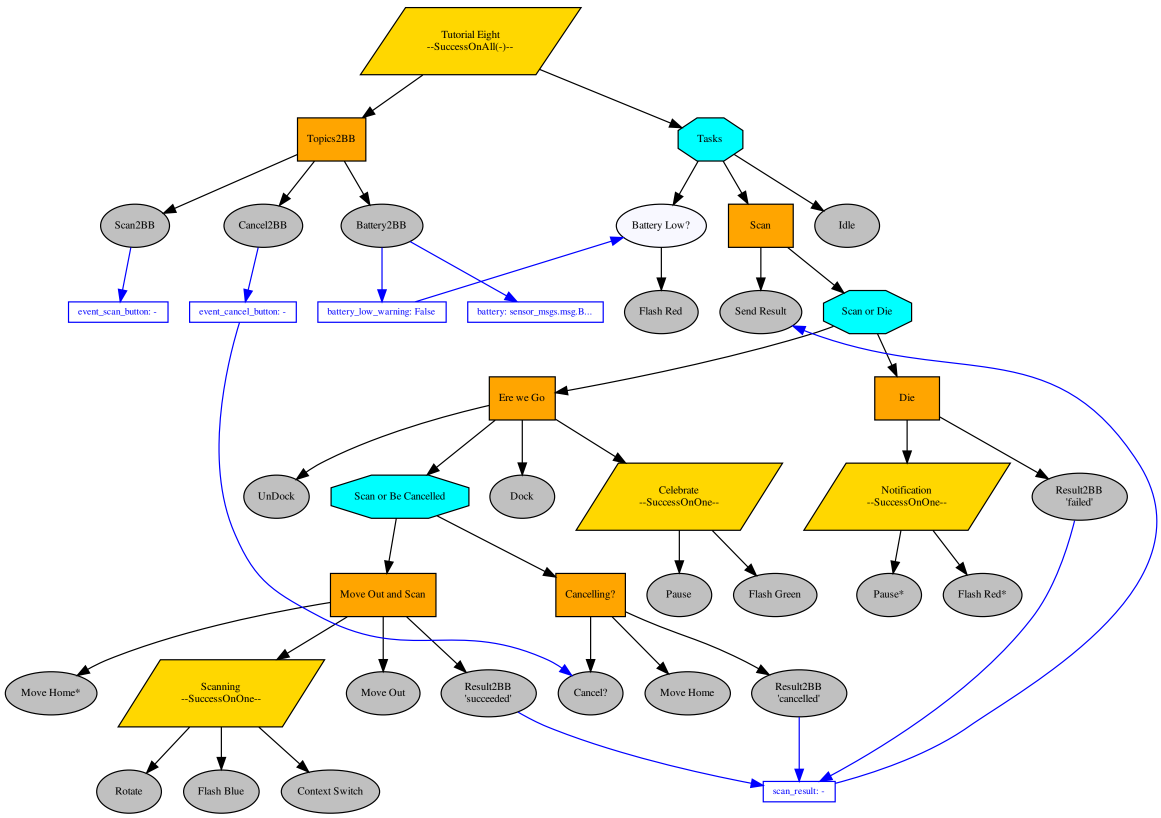


Fig. 5: A more complex tree

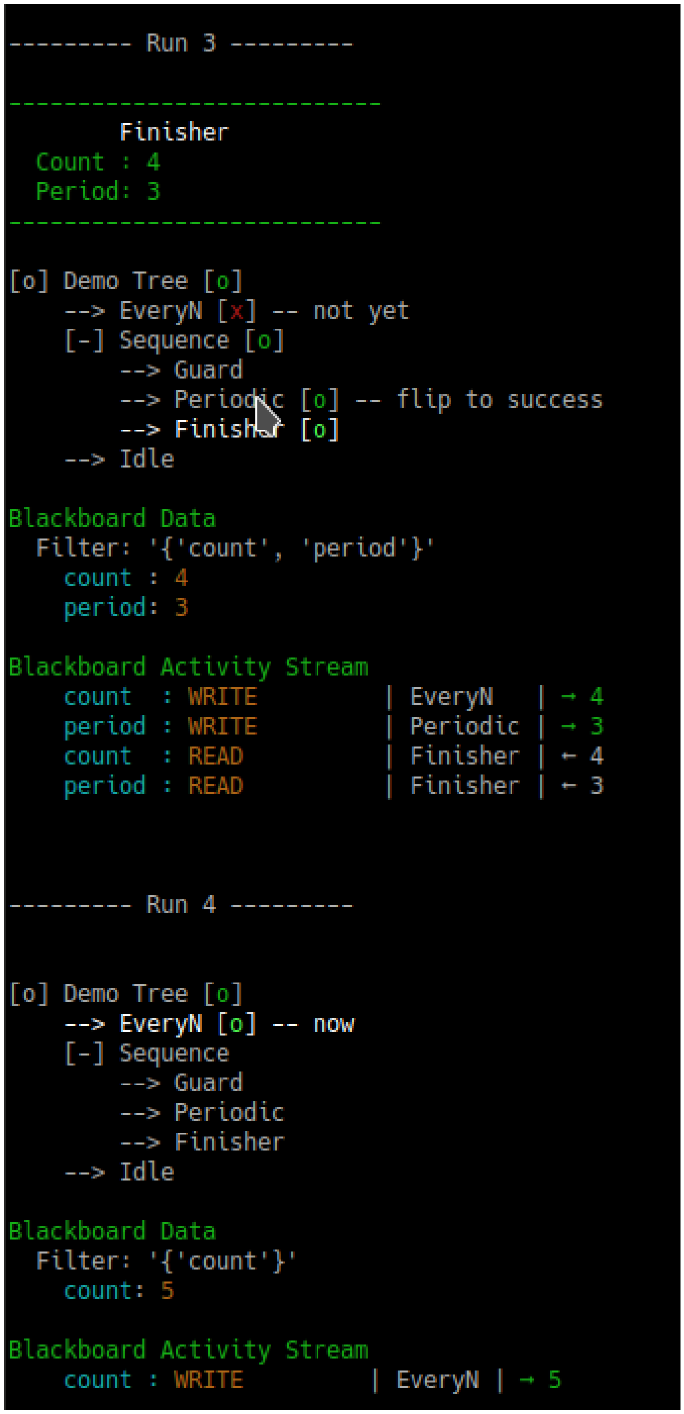


Fig. 6: Tree level debugging

- **exclusive** (*typing.Set[str]*) – set of absolute key names with exclusive write access
- **required** (*typing.Set[str]*) – set of absolute key names required to have data present
- **(typing.Dict[str, str]** (*remappings*) – client key names with blackboard remappings
- **(typing.Set[str]** (*namespaces*) – a cached list of namespaces this client accesses

CHAPTER 6

Idioms

A library of subtree creators that build complex patterns of behaviours representing common behaviour tree idioms.

Common decision making patterns can often be realised using a specific combination of fundamental behaviours and the blackboard. Even if this somewhat verbosely populates the tree, this is preferable to creating new composites types or overriding existing composites since this will increase tree logic complexity and/or bury details under the hood (both of which add an exponential cost to introspection/visualisation).

In this package these patterns will be referred to as **PyTree Idioms** and in this module you will find convenience functions that assist in creating them.

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

6.1 Either Or

idioms.**either_or**(*subtrees*, *name='Either Or'*, *namespace=None*)

Often you need a kind of selector that doesn't implement prioritisations, i.e. you would like different paths to be selected on a first-come, first-served basis.

```
task_one = py_trees.behaviours.TickCounter(name="Subtree 1", duration=2)
task_two = py_trees.behaviours.TickCounter(name="Subtree 2", duration=2)
either_or = py_trees.idioms.either_or(
    name="EitherOr",
    conditions=[
        py_trees.common.ComparisonExpression("joystick_one", "enabled", operator.
↪eq),
        py_trees.common.ComparisonExpression("joystick_two", "enabled", operator.
↪eq),
    ],
    subtrees=[task_one, task_two],
    namespace="either_or",
)
```

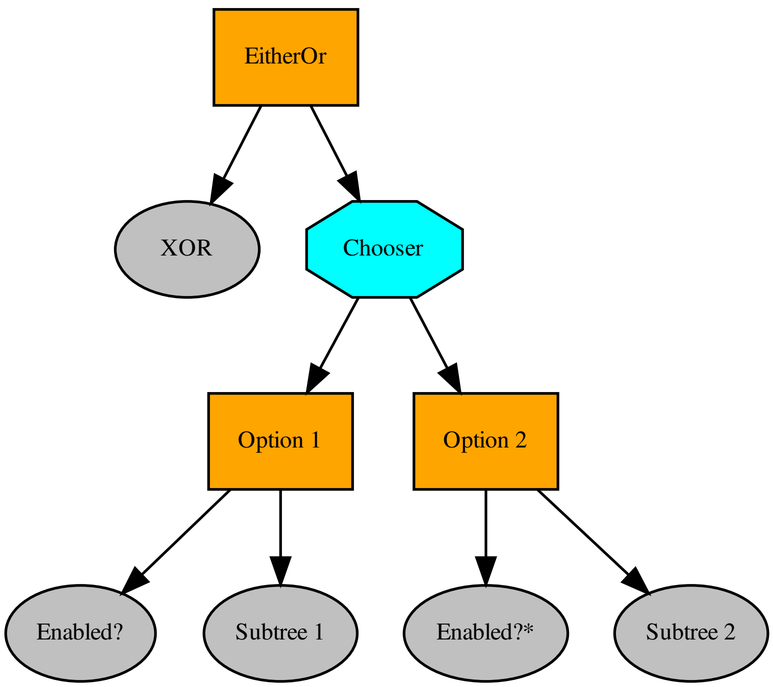


Fig. 1: Idiom - Either Or

Up front is an XOR conditional check which locks in the result on the blackboard under the specified namespace. Locking the result in permits the conditional variables to vary in future ticks without interrupting the execution of the chosen subtree (an example of a conditional variable may be one that has registered joystick button presses).

Once the result is locked in, the relevant subtree is activated beneath the selector. The children of the selector are, from left to right, not in any order of priority since the previous xor choice has been locked in and isn't revisited until the subtree executes to completion. Only one may be active and it cannot be interrupted by the others.

The only means of interrupting the execution is via a higher priority in the tree that this idiom is embedded in.

Parameters

- **`conditions`** (`List[ComparisonExpression]`) – list of triggers that ultimately select the subtree to enable
- **`subtrees`** (`List[Behaviour]`) – list of subtrees to tick from in the either_or operation
- **`name`** – the name to use for this idiom's root behaviour
- **`preemptible`** – whether the subtrees may preempt (interrupt) each other
- **`namespace`** (`Optional[str]`) – this idiom's private variables will be put behind this namespace

Raises ValueError if the number of conditions does not match the number of subtrees

If no namespace is provided, a unique one is derived from the idiom's name.

See also:

py-trees-demo-either-or

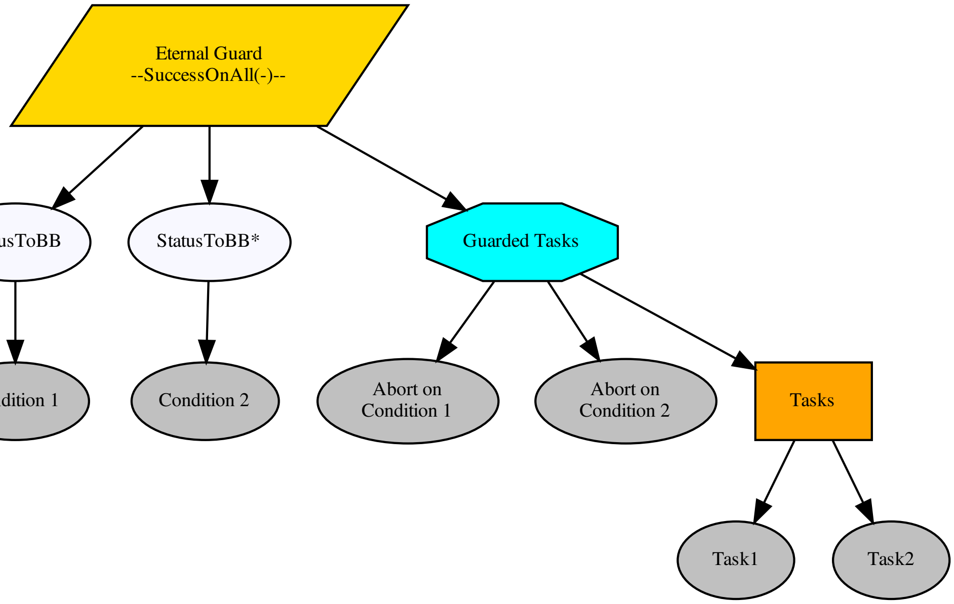
Todo: a version for which other subtrees can preempt (in an unprioritised manner) the active branch

Return type *`Behaviour`*

6.2 Eternal Guard

`idioms.`**`eternal_guard`**(*name='Eternal Guard'*, *conditions=[]*, *blackboard_namespace=None*)

The eternal guard idiom implements a stronger *guard* than the typical check at the beginning of a sequence of tasks. Here they guard continuously while the task sequence is being executed. While executing, if any of the guards should update with status *`FAILURE`*, then the task sequence is terminated.



Parameters

- **subtree** (*Behaviour*) – behaviour(s) that actually do the work
- **name** (str) – the name to use on the root behaviour of the idiom subtree
- **conditions** (List[*Behaviour*]) – behaviours on which tasks are conditional
- **blackboard_namespace** (Optional[str]) – applied to condition variable results stored on the blackboard (default: derived from the idiom name)

Return type *Behaviour*

Returns the root of the idiom subtree

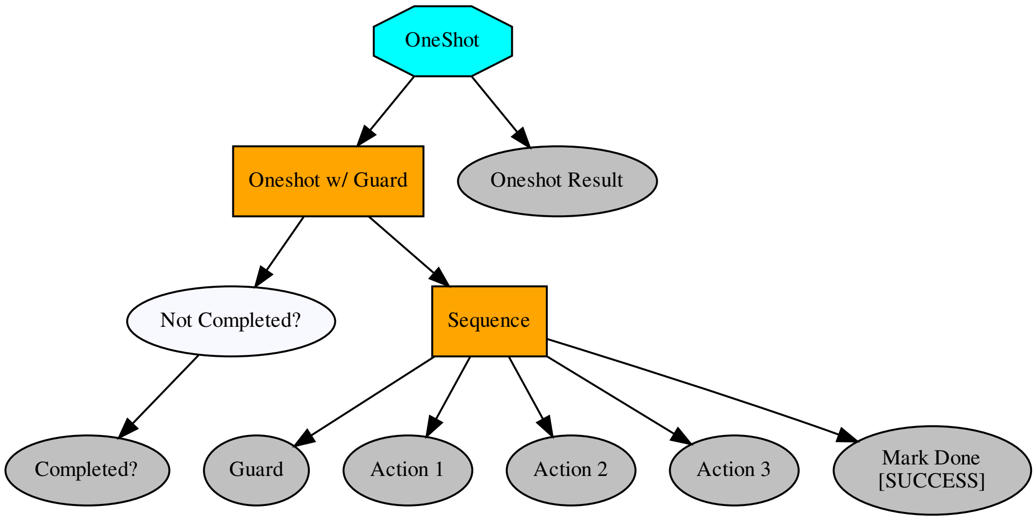
See also:

py_trees.decorators.EternalGuard

6.3 Oneshot

idioms.**oneshot**(*name='Oneshot'*, *variable_name='oneshot'*, *policy=<OneShotPolicy.ON_SUCCESSFUL_COMPLETION: [<Status.SUCCESS: 'SUCCESS'>]>*)

Ensure that a particular pattern is executed through to completion just once. Thereafter it will just rebound with the completion status.



Note: Set the policy to configure the oneshot to keep trying if failing, or to abort further attempts regardless of whether it finished with status `FAILURE`.

Parameters

- **behaviour** (*Behaviour*) – single behaviour or composited subtree to oneshot
- **name** (`str`) – the name to use for the oneshot root (selector)
- **variable_name** (`str`) – name for the variable used on the blackboard, may be nested
- **policy** (`OneShotPolicy`) – execute just once regardless of success or failure, or keep trying if failing

Returns the root of the oneshot subtree

Return type *Behaviour*

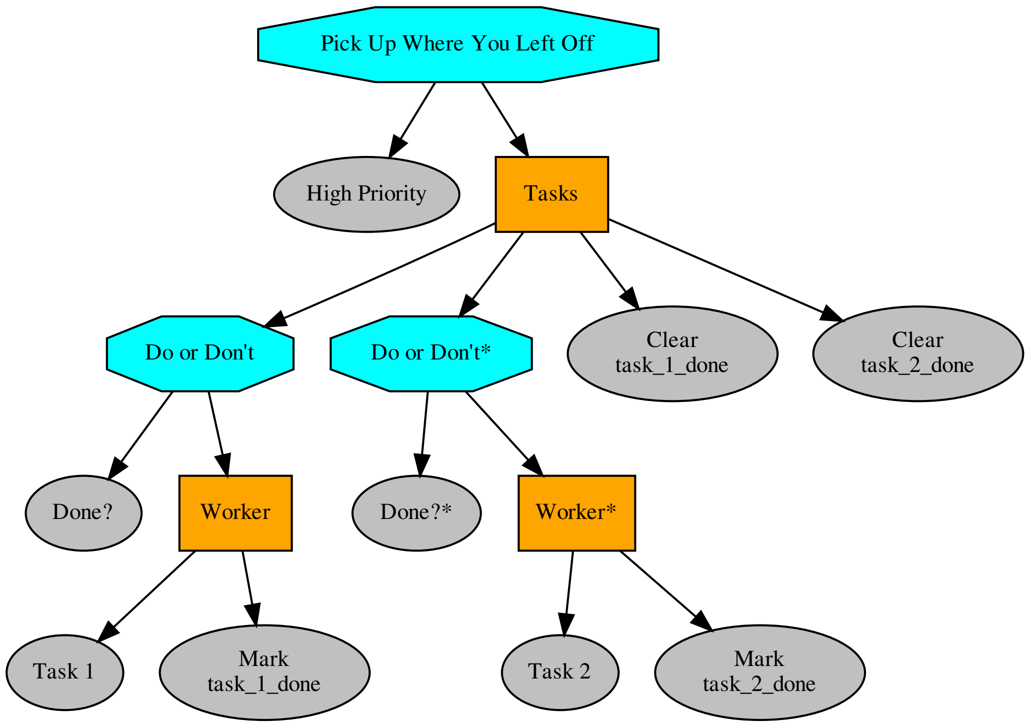
See also:

py_trees.decorators.OneShot

6.4 Pickup Where You left Off

`idioms.`**`pick_up_where_you_left_off`**(*tasks=[]*)

Rudely interrupted while enjoying a sandwich, a caveman (just because they wore loincloths does not mean they were not civilised), picks up his club and fends off the sabre-tooth tiger invading his sanctum as if he were swatting away a gnat. Task accomplished, he returns to the joys of munching through the layers of his sandwich.



Note: There are alternative ways to accomplish this idiom with their pros and cons.

a) The tasks in the sequence could be replaced by a factory behaviour that dynamically checks the state of play and spins up the tasks required each time the task sequence is first entered and invalidates/deletes them when it is either finished or invalidated. That has the advantage of not requiring much of the blackboard machinery here, but disadvantage in not making visible the task sequence itself at all times (i.e. burying details under the hood).

b) A new composite which retains the index between initialisations can also achieve the same pattern with fewer blackboard shenanigans, but suffers from an increased logical complexity cost for your trees (each new composite increases decision making complexity (O(n!)).

Parameters

- **name** (`str`) – the name to use for the task sequence behaviour
- **tasks** ([`Behaviour`]) – lists of tasks to be sequentially performed

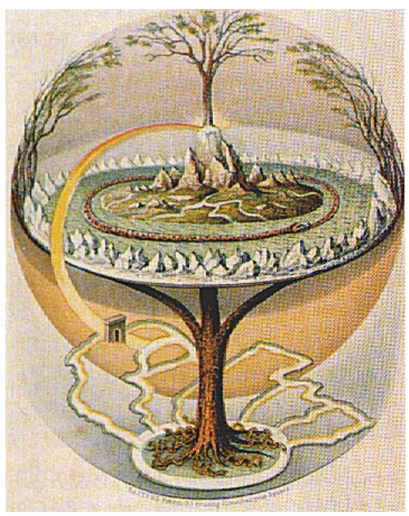
Returns root of the generated subtree

Return type `Behaviour`

CHAPTER 7

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.

7.1 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** – number of times the tree has been ticked.
- **root** – root node of the tree
- ***visitors*** – entities that visit traversed parts of the tree when it ticks
- **pre_tick_handlers** – functions that run before the entire tree is ticked
- **post_tick_handlers** – functions that run after the entire tree is ticked

Raises TypeError – if root variable is not an instance of *Behaviour*

7.2 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 python3
# -*- coding: utf-8 -*-

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=root
    )
    print(py_trees.display.unicode_tree(root=root))
    behaviour_tree.setup(timeout=15)

    def print_tree(tree):
        print(py_trees.display.unicode_tree(root=tree.root, show_status=True))

    try:
        behaviour_tree.tick_tock(
            period_ms=500,
            number_of_iterations=py_trees.trees.CONTINUOUS_TICK_TOCK,
            pre_tick_handler=None,
            post_tick_handler=print_tree
        )
```

(continues on next page)

(continued from previous page)

```
    except KeyboardInterrupt:
        behaviour_tree.interrupt()
```

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

7.3 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

```
        help='render dot tree to file with blackboard variables'
    )
    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):
    print("\n--------- Run %s ---------\n" % behaviour_tree.count)


class SuccessEveryN(py_trees.behaviours.SuccessEveryN):
```

Listing 2: pre-tick-handler-adding

```
    guard = py_trees.behaviours.Success("Guard")
    periodic_success = PeriodicSuccess()
```

7.4 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 8

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.

8.1 Ascii/Unicode Trees

You can obtain an ascii/unicode art representation of the tree on stdout via *py_trees.display.ascii_tree()* or *py_trees.display.unicode_tree()*:

py_trees.display.**ascii_tree**(*root*, *show_only_visited=False*, *show_status=False*, *visited={}*, *previously_visited={}*, *indent=0*)

Graffiti your console with ascii art for your trees.

Parameters

- **root** (*Behaviour*) – the root of the tree, or subtree you want to show
- **show_only_visited** (bool) – show only visited behaviours
- **show_status** (bool) – always show status and feedback message (i.e. for every element, not just those visited)
- **visited** (*dict*) – dictionary of (uuid.UUID) and status (*Status*) pairs for behaviours visited on the current tick
- **previously_visited** (*dict*) – dictionary of behaviour id/status pairs from the previous tree tick
- **indent** (int) – the number of characters to indent the tree

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

Return type str

See also:

py_trees.display.xhtml_tree(), *py_trees.display.unicode_tree()*

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.


```python
def post_tick_handler(snapshot_visitor, behaviour_tree):
    print(
        py_trees.display.unicode_tree(
            behaviour_tree.root,
            visited=snapshot_visitor.visited,
            previously_visited=snapshot_visitor.visited
        )
    )

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)
```

8.2 XHTML Trees

Similarly, *py_trees.display.xhtml_tree()* generates a static or runtime representation of the tree as an embeddeble XHTML snippet.

8.3 DOT Trees

API

A static representation of the tree as a dot graph is obtained via *py_trees.display.dot_tree()*. Should you wish to render the dot graph to dot/png/svg images, make use of *py_trees.display.render_dot_tree()*. Note that the dot graph representation does not generate runtime information for the tree (visited paths, status, . . .).

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. This is extremely useful when either designing your trees or auto-rendering dot graphs for documentation on CI.

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 9

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!

9.1 Behaviours

- Keep the constructor minimal so you can instantiate the behaviour for offline rendering
- Put hardware or other runtime specific initialisation in `setup()`
- 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 reusable concepts as subtrees (idioms)

9.2 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

9.3 Trees

- When designing your tree, stub them out with nonsense behaviours. Focus on descriptive names, composite types and render dot graphs to accelerate the design process (especially when collaborating).
- 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 1-500ms.

CHAPTER 10

Terminology

block

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*.

data gathering Caching events, notifications, or incoming data arriving asynchronously on the blackboard. This is a fairly common practice for behaviour trees which exist inside a complex system.

In most cases, data gathering is done either outside the tree, or at the front end of your tree under a parallel preceding the rest of the tree tick so that the ensuing behaviours work on a constant, consistent set of data. Even if the incoming data is not arriving asynchronously, this is useful conceptually and organisationally.

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 _:' \_,
    / (`---'\ `-.
 ,-`  _)    (_,
```

guard A guard is a behaviour at the start of a sequence that checks for a particular condition (e.g. is battery low?). If the check succeeds, then the door is opened to the rest of the work sequence.

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 11

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 12

Demos

12.1 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()

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/splintered-reality/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.common
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 += "* Kickstarts 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)

```python

    if py_trees.console.has_colours:
        banner_line = console.green + "*" * 79 + "\n" + console.reset
        s = "\n"
        s += banner_line
        s += console.bold_white + "Action Behaviour".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,
                                   )


def planning(pipe_connection):
    """
    Emulates an external process which might accept long running planning jobs.
    """
    idle = True
    percentage_complete = 0
    try:
        while(True):
            if pipe_connection.poll():
                pipe_connection.recv()
                percentage_complete = 0
                idle = False
            if not idle:
                percentage_complete += 10
                pipe_connection.send([percentage_complete])
                if percentage_complete == 100:
                    idle = True
            time.sleep(0.5)
    except KeyboardInterrupt:
        pass


class Action(py_trees.behaviour.Behaviour):
    """
    Connects to a subprocess to initiate a goal, and monitors the progress
```

(continues on next page)

(continued from previous page)

```python
    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):
        """
        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()

    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.common.Status.RUNNING
        if self.parent_connection.poll():
            self.percentage_completion = self.parent_connection.recv().pop()
            if self.percentage_completion == 100:
                new_status = py_trees.common.Status.SUCCESS
        if new_status == py_trees.common.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)

```
    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

    action = Action()
    action.setup()
    try:
        for unused_i in range(0, 12):
            action.tick_once()
            time.sleep(0.5)
        print("\n")
    except KeyboardInterrupt:
        pass
```

12.2 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()
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/splintered-reality/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"
    content += "initialisation, ticking and termination.\n"
    if py_trees.console.has_colours:
        banner_line = console.green + "*" * 79 + "\n" + console.reset
```

(continues on next page)

(continued from previous page)

```
        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):
        """
        No delayed initialisation required for this example.
        """
        self.logger.debug("%s.setup()" % (self.__class__.__name__))

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

(continues on next page)

(continued from previous page)

```
            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.common.Status.SUCCESS if self.counter == 3 else py_
↪trees.common.Status.RUNNING
        if new_status == py_trees.common.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
```

12.3 py-trees-demo-blackboard

Demonstrates usage of the blackboard and related behaviours.

A sequence is populated with a few behaviours that exercise reading and writing on the Blackboard in interesting ways.

```
usage: py-trees-demo-blackboard [-h] [-r | --render-with-blackboard-variables]
```

12.3.1 Named Arguments

-r, --render render dot tree to file

Default: False

--render-with-blackboard-variables render dot tree to file with blackboard variables

Default: False

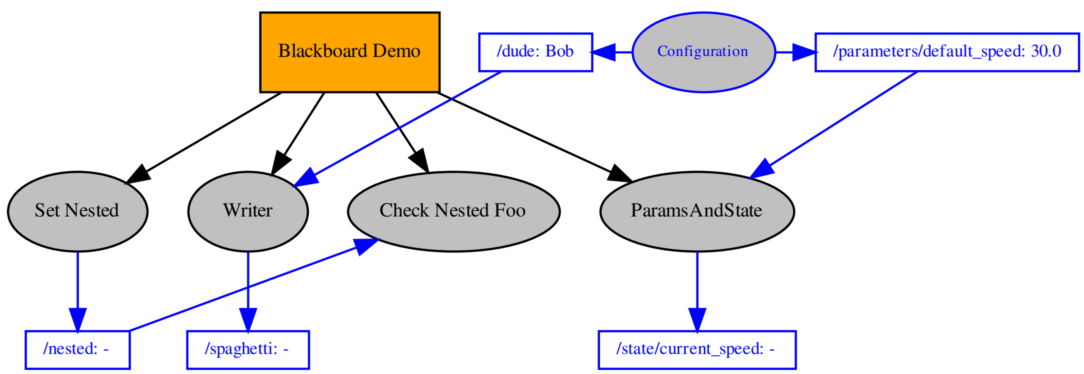


Fig. 1: Dot Graph

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*.

class py_trees.demos.blackboard.**Nested**

Bases: object

A more complex object to interact with on the blackboard.

__init__()

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

__str__()

Return str(self).

__weakref__

list of weak references to the object (if defined)

```
snorri > … > py_trees > doc > images > devel + 2 … 1 ⚑ 9 > 130 > py-trees-demo-blackboard

******************************************************************************
                                  Blackboard
******************************************************************************

Demonstrates usage of the blackboard and related behaviours.

A sequence is populated with a few behaviours that exercise
reading and writing on the Blackboard in interesting ways.

******************************************************************************

[DEBUG] Writer               : BlackboardWriter.__init__()

--------- Tick 0 ---------

[DEBUG] Blackboard Demo      : Sequence.tick()
[DEBUG] Blackboard Demo      : Sequence.tick() [!RUNNING->resetting child index]
[DEBUG] Set Nested           : SetBlackboardVariable.tick()
[DEBUG] Set Nested           : SetBlackboardVariable.stop(Status.INVALID->Status.SUCCESS)
[DEBUG] Writer               : BlackboardWriter.tick()
[DEBUG] Writer               : BlackboardWriter.update()
[DEBUG] Writer               : BlackboardWriter.stop(Status.INVALID->Status.SUCCESS)
[DEBUG] Check Nested Foo     : CheckBlackboardVariableValue.tick()
[DEBUG] Check Nested Foo     : CheckBlackboardVariableValue.update()
[DEBUG] Check Nested Foo     : CheckBlackboardVariableValue.stop(Status.INVALID->Status.SUCCESS)
[DEBUG] ParamsAndState       : ParamsAndState.tick()


[-] Blackboard Demo [*]
    --> Set Nested [o]
    --> Writer [o]
    --> Check Nested Foo [o] -- 'nested.foo' comparison succeeded [v: bar][e: bar]
    --> ParamsAndState [*]

-------------------------

Blackboard Data
    dude                     : Bob
    foo                      : -
    nested                   : {'foo': 'bar'}
    parameters_default_speed: 30.0
    spaghetti                : {'type': 'Gnocchi', 'quantity': 2}
    state_current_speed      : 31.0

-------------------------

Blackboard Clients
    dude                     : Configur... (w), Writer (r)
    foo                      : Unsetter (w)
    nested                   : Set Nested (w), Check Ne... (r)
    parameters_default_speed : Params (r), Configur... (w)
    spaghetti                : Writer (w)
    state_current_speed      : State (w)

-------------------------

Blackboard Activity Stream
    dude                     : INITIALISED   | Configuration    | → Bob
    parameters_default_speed : INITIALISED   | Configuration    | → 30.0
    nested                   : INITIALISED   | Set Nested       | → {'foo': 'bar'}
    dude                     : READ          | Writer           | ← Bob
    dudette                  : ACCESS_DENIED | Writer           | ✕ client has no read/write access
    dudette                  : ACCESS_DENIED | Writer           | ✕ client has no read/write access
    spaghetti                : INITIALISED   | Writer           | → {'type': 'Carbo...
    spaghetti                : WRITE         | Writer           | → {'type': 'Gnocc...
    spaghetti                : NO_OVERWRITE  | Writer           | ⦸ {'type': 'Gnocc...
    nested                   : READ          | Check Nested Foo | ← {'foo': 'bar'}
    parameters_default_speed : READ          | Params           | ← 30.0
    state_current_speed      : INITIALISED   | State            | → 30.0
```

Fig. 2: Console Screenshot

class py_trees.demos.blackboard.**ParamsAndState**(*name='ParamsAndState'*)

Bases: *py_trees.behaviour.Behaviour*

A more esotoric use of multiple blackboards in a behaviour to represent storage of parameters and state.

__init__(*name='ParamsAndState'*)

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

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!

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

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/splintered-reality/py_trees/devel/LICENSE
#
##############################################################################
# Documentation
##############################################################################

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

.. graphviz:: dot/demo-blackboard.dot
   :align: center
   :caption: Dot Graph
```

(continues on next page)

(continued from previous page)

```

.. figure:: images/blackboard_demo.png
   :align: center

   Console Screenshot
"""

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

import argparse
import operator
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 few behaviours that exercise\n"
    content += "reading and writing on the Blackboard in interesting ways.\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:
        return None


def command_line_argument_parser():
    parser = argparse.ArgumentParser(description=description(),
                                     epilog=epilog(),
                                     formatter_class=argparse.
↪RawDescriptionHelpFormatter,
```

(continues on next page)

(continued from previous page)

```
                                                    )
    render_group = parser.add_mutually_exclusive_group()
    render_group.add_argument('-r', '--render', action='store_true', help='render dot tree to file')
    render_group.add_argument(
        '--render-with-blackboard-variables',
        action='store_true',
        help='render dot tree to file with blackboard variables'
    )
    return parser


class Nested(object):
    """
    A more complex object to interact with on the blackboard.
    """
    def __init__(self):
        self.foo = "bar"

    def __str__(self):
        return str({"foo": self.foo})


class BlackboardWriter(py_trees.behaviour.Behaviour):
    """
    Custom writer that submits a more complicated variable to the blackboard.
    """
    def __init__(self, name="Writer"):
        super().__init__(name=name)
        self.blackboard = self.attach_blackboard_client()
        self.blackboard.register_key(key="dude", access=py_trees.common.Access.READ)
        self.blackboard.register_key(key="spaghetti", access=py_trees.common.Access.WRITE)

        self.logger.debug("%s.__init__()" % (self.__class__.__name__))

    def update(self):
        """
        Write a dictionary to the blackboard and return :data:`~py_trees.common.Status.SUCCESS`.
        """
        self.logger.debug("%s.update()" % (self.__class__.__name__))
        try:
            unused = self.blackboard.dude
        except KeyError:
            pass
        try:
            unused = self.blackboard.dudette
        except AttributeError:
            pass
        try:
            self.blackboard.dudette = "Jane"
        except AttributeError:
            pass
        self.blackboard.spaghetti = {"type": "Carbonara", "quantity": 1}
        self.blackboard.spaghetti = {"type": "Gnocchi", "quantity": 2}
        try:
```

(continues on next page)

(continued from previous page)

```
            self.blackboard.set("spaghetti", {"type": "Bolognese", "quantity": 3}, overwrite=False)
        except AttributeError:
            pass
        return py_trees.common.Status.SUCCESS


class ParamsAndState(py_trees.behaviour.Behaviour):
    """
    A more esotoric use of multiple blackboards in a behaviour to represent
    storage of parameters and state.
    """
    def __init__(self, name="ParamsAndState"):
        super().__init__(name=name)
        # namespaces can include the separator or may leave it out
        # they can also be nested, e.g. /agent/state, /agent/parameters
        self.parameters = self.attach_blackboard_client("Params", "parameters")
        self.state = self.attach_blackboard_client("State", "state")
        self.parameters.register_key(
            key="default_speed",
            access=py_trees.common.Access.READ
        )
        self.state.register_key(
            key="current_speed",
            access=py_trees.common.Access.WRITE
        )

    def initialise(self):
        try:
            self.state.current_speed = self.parameters.default_speed
        except KeyError as e:
            raise RuntimeError("parameter 'default_speed' not found [{}]".format(str(e)))

    def update(self):
        if self.state.current_speed > 40.0:
            return py_trees.common.Status.SUCCESS
        else:
            self.state.current_speed += 1.0
            return py_trees.common.Status.RUNNING


def create_root():
    root = py_trees.composites.Sequence("Blackboard Demo")
    set_blackboard_variable = py_trees.behaviours.SetBlackboardVariable(
        name="Set Nested", variable_name="nested", variable_value=Nested()
    )
    write_blackboard_variable = BlackboardWriter(name="Writer")
    check_blackboard_variable = py_trees.behaviours.CheckBlackboardVariableValue(
        name="Check Nested Foo",
        check=py_trees.common.ComparisonExpression(
            variable="nested.foo",
            value="bar",
            operator=operator.eq
        )
    )
    params_and_state = ParamsAndState()
```

(continues on next page)

(continued from previous page)

```
    root.add_children([
        set_blackboard_variable,
        write_blackboard_variable,
        check_blackboard_variable,
        params_and_state
    ])
    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
    py_trees.blackboard.Blackboard.enable_activity_stream(maximum_size=100)
    blackboard = py_trees.blackboard.Client(name="Configuration")
    blackboard.register_key(key="dude", access=py_trees.common.Access.WRITE)
    blackboard.register_key(key="/parameters/default_speed", access=py_trees.common.
↪Access.WRITE)
    blackboard.dude = "Bob"
    blackboard.parameters.default_speed = 30.0

    root = create_root()

    ####################
    # Rendering
    ####################
    if args.render:
        py_trees.display.render_dot_tree(root, with_blackboard_variables=False)
        sys.exit()
    if args.render_with_blackboard_variables:
        py_trees.display.render_dot_tree(root, with_blackboard_variables=True)
        sys.exit()

    ####################
    # Execute
    ####################
    root.setup_with_descendants()
    unset_blackboard = py_trees.blackboard.Client(name="Unsetter")
    unset_blackboard.register_key(key="foo", access=py_trees.common.Access.WRITE)
    print("\n--------- Tick 0 ---------\n")
    root.tick_once()
    print("\n")
    print(py_trees.display.unicode_tree(root, show_status=True))
    print("--------------------------\n")
    print(py_trees.display.unicode_blackboard())
    print("--------------------------\n")
    print(py_trees.display.unicode_blackboard(display_only_key_metadata=True))
    print("--------------------------\n")
    unset_blackboard.unset("foo")
    print(py_trees.display.unicode_blackboard_activity_stream())
```

12.4 py-trees-demo-blackboard-namespaces

Demonstrates usage of blackboard namespaces.

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

Fig. 3: Console Screenshot

`py_trees.demos.blackboard_namespaces.main()`

Entry point for the demo script.

Listing 4: py_trees/demos/blackboard_namespaces.py

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

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

.. figure:: images/blackboard_namespaces.png
   :align: center

   Console Screenshot
"""

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

(continues on next page)

(continued from previous page)

```python

import argparse
import py_trees

import py_trees.console as console

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


def description():
    content = "Demonstrates usage of blackboard namespaces.\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 + "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:
        return None


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


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


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

(continues on next page)

(continued from previous page)

```
    print(description())
    print("---------------------------------------------------------------------------
↪----")
    print("$ py_trees.blackboard.Client(name='Blackboard')")
    print("$ foo.register_key(key='dude', access=py_trees.common.Access.WRITE)")
    print("$ foo.register_key(key='/dudette', access=py_trees.common.Access.WRITE)")
    print("$ foo.register_key(key='/foo/bar/wow', access=py_trees.common.Access.WRITE)
↪")
    print("---------------------------------------------------------------------------
↪----")
    blackboard = py_trees.blackboard.Client(name="Blackboard")
    blackboard.register_key(key="dude", access=py_trees.common.Access.WRITE)
    blackboard.register_key(key="/dudette", access=py_trees.common.Access.WRITE)
    blackboard.register_key(key="/foo/bar/wow", access=py_trees.common.Access.WRITE)
    print(blackboard)
    print("---------------------------------------------------------------------------
↪----")
    print("$ blackboard.dude = 'Bob'")
    print("$ blackboard.dudette = 'Jade'")
    print("---------------------------------------------------------------------------
↪----")
    blackboard.dude = "Bob"
    blackboard.dudette = "Jade"
    print(py_trees.display.unicode_blackboard())
    print("---------------------------------------------------------------------------
↪----")
    print("$ blackboard.foo.bar.wow = 'foobar'")
    print("---------------------------------------------------------------------------
↪----")
    blackboard.foo.bar.wow = "foobar"
    print(py_trees.display.unicode_blackboard())
    print("---------------------------------------------------------------------------
↪----")
    print("$ py_trees.blackboard.Client(name='Foo', namespace='foo')")
    print("$ foo.register_key(key='awesome', access=py_trees.common.Access.WRITE)")
    print("$ foo.register_key(key='/brilliant', access=py_trees.common.Access.WRITE)")
    print("$ foo.register_key(key='/foo/clever', access=py_trees.common.Access.WRITE)
↪")
    print("---------------------------------------------------------------------------
↪----")
    foo = py_trees.blackboard.Client(name="Foo", namespace="foo")
    foo.register_key(key="awesome", access=py_trees.common.Access.WRITE)
    # TODO: should /brilliant be namespaced or go directly to root?
    foo.register_key(key="/brilliant", access=py_trees.common.Access.WRITE)
    # absolute names are ok, so long as they include the namespace
    foo.register_key(key="/foo/clever", access=py_trees.common.Access.WRITE)
    print(foo)
    print("---------------------------------------------------------------------------
↪----")
    print("$ foo.awesome = True")
    print("$ foo.set('/brilliant', False)")
    print("$ foo.clever = True")
    print("---------------------------------------------------------------------------
↪----")
    foo.awesome = True
    # Only accessable via set since it's not in the namespace
    foo.set("/brilliant", False)
```

(continues on next page)

(continued from previous page)

```
    # This will fail since it looks for the namespaced /foo/brilliant key
    # foo.brilliant = False
    foo.clever = True
    print(py_trees.display.unicode_blackboard())
```

12.5 py-trees-demo-blackboard-remappings

Demonstrates usage of blackbord remappings.

Demonstration is via an exemplar behaviour making use of remappings..

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

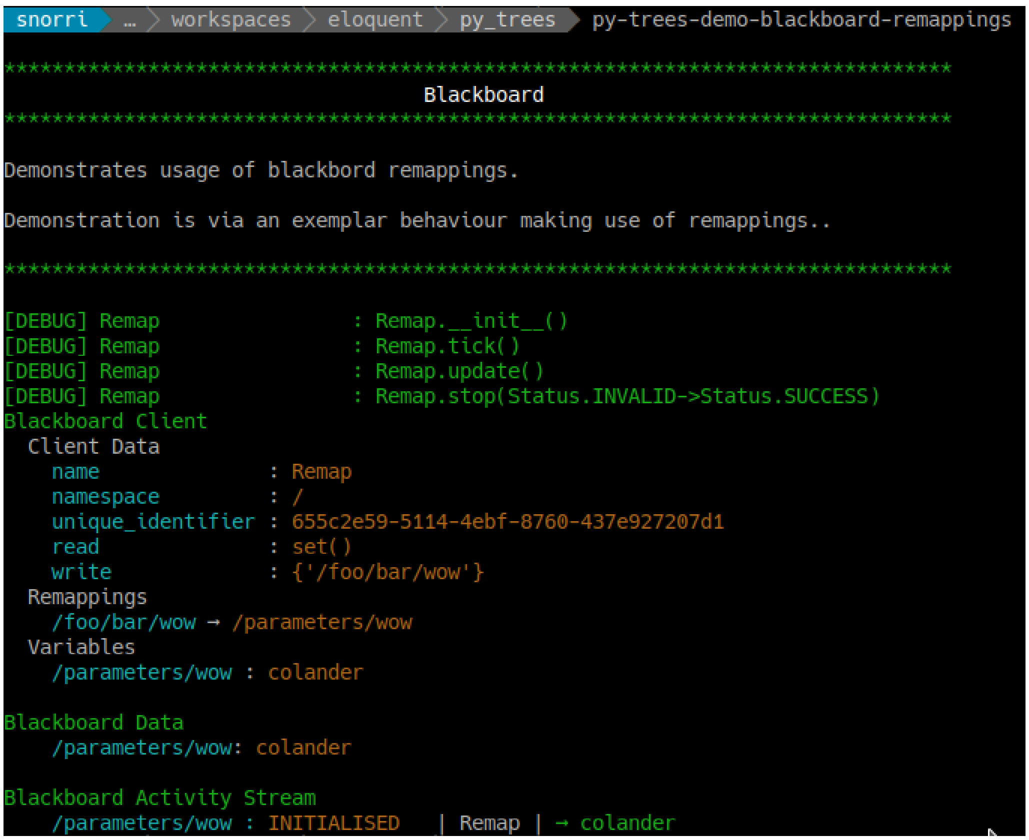


Fig. 4: Console Screenshot

class `py_trees.demos.blackboard_remappings.`**`Remap`**(*name*, *remap_to*)

Bases: *py_trees.behaviour.Behaviour*

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

`__init__`(*name*, *remap_to*)

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

`update`()

Write a dictionary to the blackboard and return *SUCCESS*.

`py_trees.demos.blackboard_remappings.main()`
Entry point for the demo script.

Listing 5: py_trees/demos/blackboard_remappings.py

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

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

.. figure:: images/blackboard_remappings.png
   :align: center

   Console Screenshot
"""

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

import argparse
import py_trees
import typing

import py_trees.console as console

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


def description():
    content = "Demonstrates usage of blackbord remappings.\n"
    content += "\n"
    content += "Demonstration is via an exemplar behaviour making use of remappings..
↪\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:
```

(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,
                                     )
    return parser


class Remap(py_trees.behaviour.Behaviour):
    """
    Custom writer that submits a more complicated variable to the blackboard.
    """
    def __init__(self, name: str, remap_to: typing.Dict[str, str]):
        super().__init__(name=name)
        self.logger.debug("%s.__init__()" % (self.__class__.__name__))
        self.blackboard = self.attach_blackboard_client()
        self.blackboard.register_key(
            key="/foo/bar/wow",
            access=py_trees.common.Access.WRITE,
            remap_to=remap_to["/foo/bar/wow"]
        )

    def update(self):
        """
        Write a dictionary to the blackboard and return :data:`~py_trees.common.
↪Status.SUCCESS`.
        """
        self.logger.debug("%s.update()" % (self.__class__.__name__))
        self.blackboard.foo.bar.wow = "colander"

        return py_trees.common.Status.SUCCESS

##############################################################################
# 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)

```
    py_trees.blackboard.Blackboard.enable_activity_stream(maximum_size=100)
    root = Remap(name="Remap", remap_to={"/foo/bar/wow": "/parameters/wow"})

    ####################
    # Execute
    ####################
    root.tick_once()
    print(root.blackboard)
    print(py_trees.display.unicode_blackboard())
    print(py_trees.display.unicode_blackboard_activity_stream())
```

12.6 py-trees-demo-context-switching

Demonstrates context switching with parallels and sequences.

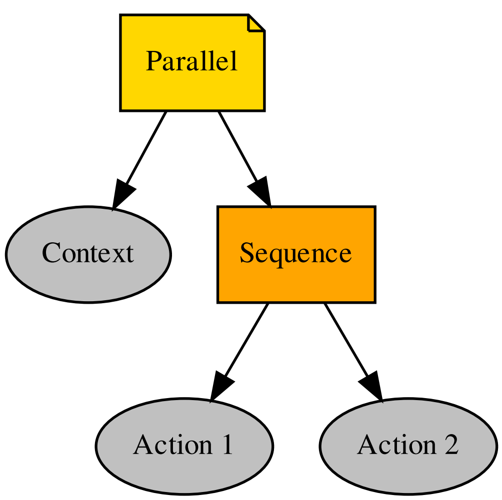
A context switching behaviour is run in parallel with a work sequence. Switching the context occurs in the initialise() and terminate() methods of the context switching behaviour. Note that whether the sequence results in failure or success, the context switch behaviour will always call the terminate() 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]
```

12.6.1 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 6: py_trees/demos/contex_switching.py

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

(continues on next page)

(continued from previous page)

```python

##############################################################################
# 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
        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


```

(continues on next page)

(continued from previous page)

```python
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 = "no context"

    def initialise(self):
        """
        Backup and set a new context.
        """
        self.logger.debug("%s.initialise()[switch context]" % (self.__class__.__name__))
        # Some actions that:
        #   1. retrieve the current context from somewhere
        #   2. cache the context internally
        #   3. apply a new context
        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.common.Status.RUNNING

    def terminate(self, new_status):
        """
        Restore the context with the previously backed up context.
        """
        self.logger.debug("%s.terminate()[%s->%s][restore context]" % (self.__class__.__name__, self.status, new_status))
        # Some actions that:
        #   1. restore the cached context
        self.feedback_message = "restored context"


def create_root():
    root = py_trees.composites.Parallel(name="Parallel", policy=py_trees.common.ParallelPolicy.SuccessOnOne())
    context_switch = ContextSwitch(name="Context")
    sequence = py_trees.composites.Sequence(name="Sequence")
    for job in ["Action 1", "Action 2"]:
        success_after_two = py_trees.behaviours.Count(name=job,
                                                      fail_until=0,
                                                      running_until=2,
                                                      success_until=10)
```

(continues on next page)

(continued from previous page)

```python
        sequence.add_child(success_after_two)
    root.add_child(context_switch)
    root.add_child(sequence)
    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

    root = create_root()

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

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

12.7 py-trees-demo-dot-graphs

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

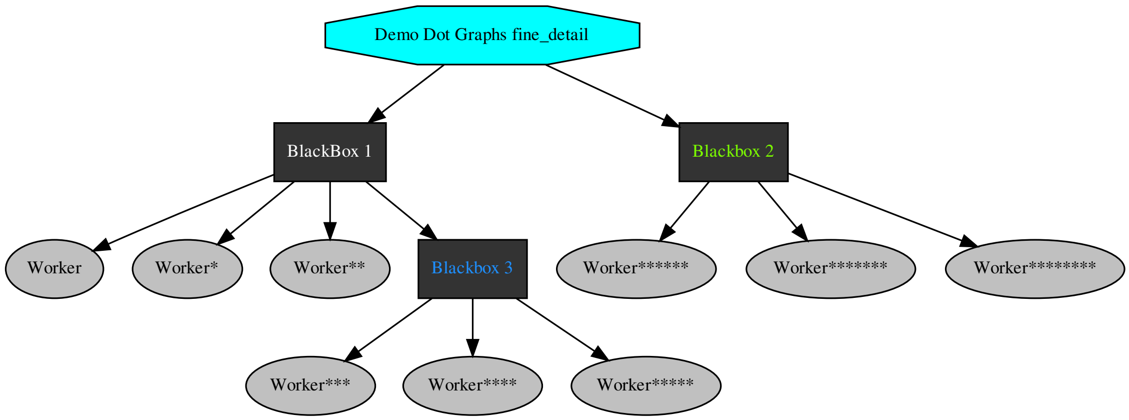
```
usage: py-trees-demo-dot-graphs [-h]
                                [-l {all,fine_detail,detail,component,big_picture}]
```

12.7.1 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 7: py_trees/demos/dot_graphs.py

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

"""
.. argparse::
   :module: py_trees.demos.dot_graphs
   :func: command_line_argument_parser
   :prog: py-trees-demo-dot-graphs

.. graphviz:: dot/demo-dot-graphs.dot

"""

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

import argparse
import subprocess
import py_trees

import py_trees.console as console

##############################################################################
# Classes
```

(continues on next page)

(continued from previous page)

```
##############################################################################


def description():
    name = "py-trees-demo-dot-graphs"
    content = "Renders a dot graph for a simple tree, with blackboxes.\n"
    if py_trees.console.has_colours:
        banner_line = console.green + "*" * 79 + "\n" + console.reset
        s = "\n"
        s += banner_line
        s += console.bold_white + "Dot Graphs".center(79) + "\n" + console.reset
        s += banner_line
        s += "\n"
        s += content
        s += "\n"
        s += console.white
        s += console.bold + "    Generate Full Dot Graph" + console.reset + "\n"
        s += "\n"
        s += console.cyan + "        {0}".format(name) + console.reset + "\n"
        s += "\n"
        s += console.bold + "    With Varying Visibility Levels" + console.reset + "\n"
        s += "\n"
        s += console.cyan + "        {0}".format(name) + console.yellow + " --level=all" + console.reset + "\n"
        s += console.cyan + "        {0}".format(name) + console.yellow + " --level=detail" + console.reset + "\n"
        s += console.cyan + "        {0}".format(name) + console.yellow + " --level=component" + console.reset + "\n"
        s += console.cyan + "        {0}".format(name) + console.yellow + " --level=big_picture" + console.reset + "\n"
        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('-l', '--level', action='store',
                        default='fine_detail',
                        choices=['all', 'fine_detail', 'detail', 'component', 'big_picture'],
                        help='visibility level')
```

(continues on next page)

(continued from previous page)

```
    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.
    """
    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("")
```

12.8 py-trees-demo-either-or

A demonstration of the 'either_or' idiom.

This behaviour tree pattern enables triggering of subtrees with equal priority (first in, first served).

EVENTS

- 3 : joystick one enabled, task one starts
- 5 : task one finishes
- 6 : joystick two enabled, task two starts
- 7 : joystick one enabled, task one ignored, task two continues
- 8 : task two finishes

```
usage: py-trees-demo-either-or [-h] [-r | -i]
```

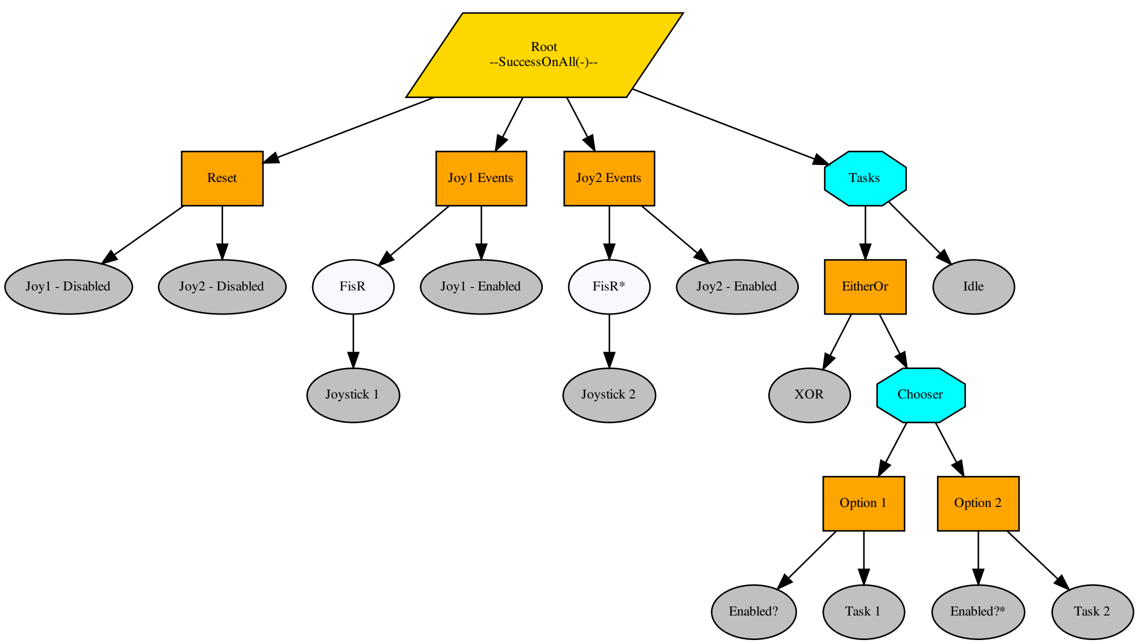
12.8.1 Named Arguments

-r, --render render dot tree to file

Default: False

-i, --interactive pause and wait for keypress at each tick

Default: False



`py_trees.demos.either_or.main()`

Entry point for the demo script.

Listing 8: py_trees/demos/either_or.py

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

"""
.. argparse::
   :module: py_trees.demos.either_or
   :func: command_line_argument_parser
   :prog: py-trees-demo-either-or

.. graphviz:: dot/demo-either-or.dot

.. image:: images/either_or.gif
"""

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

import argparse
import functools
import operator
import py_trees
import sys
import time

import py_trees.console as console

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


def description(root):
    content = "A demonstration of the 'either_or' idiom.\n\n"
    content += "This behaviour tree pattern enables triggering of subtrees\n"
    content += "with equal priority (first in, first served).\n"
    content += "\n"
    content += "EVENTS\n"
    content += "\n"
    content += " -  3 : joystick one enabled, task one starts\n"
    content += " -  5 : task one finishes\n"
    content += " -  6 : joystick two enabled, task two starts\n"
    content += " -  7 : joystick one enabled, task one ignored, task two continues\n"
    content += " -  8 : task two finishes\n"
    content += "\n"
    if py_trees.console.has_colours:
        banner_line = console.green + "*" * 79 + "\n" + console.reset
        s = "\n"
        s += banner_line
```

(continues on next page)

(continued from previous page)

```
        s += console.bold_white + "Either Or".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_root()),
                                     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):
    print("\n--------- Run %s ---------\n" % behaviour_tree.count)


def post_tick_handler(snapshot_visitor, behaviour_tree):
    print(
        "\n" + py_trees.display.unicode_tree(
            root=behaviour_tree.root,
            visited=snapshot_visitor.visited,
            previously_visited=snapshot_visitor.previously_visited
        )
    )
    print(py_trees.display.unicode_blackboard())


def create_root():
    trigger_one = py_trees.decorators.FailureIsRunning(
        name="FisR",
        child=py_trees.behaviours.SuccessEveryN(
            name="Joystick 1",
            n=4
        )
    )
    trigger_two = py_trees.decorators.FailureIsRunning(
```

(continues on next page)

(continued from previous page)

```
            name="FisR",
            child=py_trees.behaviours.SuccessEveryN(
                name="Joystick 2",
                n=7
            )
        )
        enable_joystick_one = py_trees.behaviours.SetBlackboardVariable(
            name="Joy1 - Enabled",
            variable_name="joystick_one",
            variable_value="enabled")
        enable_joystick_two = py_trees.behaviours.SetBlackboardVariable(
            name="Joy2 - Enabled",
            variable_name="joystick_two",
            variable_value="enabled")
        reset_joystick_one = py_trees.behaviours.SetBlackboardVariable(
            name="Joy1 - Disabled",
            variable_name="joystick_one",
            variable_value="disabled")
        reset_joystick_two = py_trees.behaviours.SetBlackboardVariable(
            name="Joy2 - Disabled",
            variable_name="joystick_two",
            variable_value="disabled")
        task_one = py_trees.behaviours.TickCounter(
            name="Task 1",
            duration=2,
            completion_status=py_trees.common.Status.SUCCESS
        )
        task_two = py_trees.behaviours.TickCounter(
            name="Task 2",
            duration=2,
            completion_status=py_trees.common.Status.SUCCESS
        )
        idle = py_trees.behaviours.Running(name="Idle")
        either_or = py_trees.idioms.either_or(
            name="Either Or",
            conditions=[
                py_trees.common.ComparisonExpression("joystick_one", "enabled", operator.
↪eq),
                py_trees.common.ComparisonExpression("joystick_two", "enabled", operator.
↪eq),
            ],
            subtrees=[task_one, task_two],
            namespace="either_or",
        )
        root = py_trees.composites.Parallel(
            name="Root",
            policy=py_trees.common.ParallelPolicy.SuccessOnAll(synchronise=False)
        )
        reset = py_trees.composites.Sequence(name="Reset")
        reset.add_children([reset_joystick_one, reset_joystick_two])
        joystick_one_events = py_trees.composites.Sequence(name="Joy1 Events")
        joystick_one_events.add_children([trigger_one, enable_joystick_one])
        joystick_two_events = py_trees.composites.Sequence(name="Joy2 Events")
        joystick_two_events.add_children([trigger_two, enable_joystick_two])
        tasks = py_trees.composites.Selector(name="Tasks")
        tasks.add_children([either_or, idle])
        root.add_children([reset, joystick_one_events, joystick_two_events, tasks])
```

(continues on next page)

(continued from previous page)

```python
    return root


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


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

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

    ####################
    # Tree Stewardship
    ####################
    behaviour_tree = py_trees.trees.BehaviourTree(root)
    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:
        py_trees.console.read_single_keypress()
    for unused_i in range(1, 11):
        try:
            behaviour_tree.tick()
            if args.interactive:
                py_trees.console.read_single_keypress()
            else:
                time.sleep(0.5)
        except KeyboardInterrupt:
            break
    print("\n")
```

12.9 py-trees-demo-logging

A demonstration of logging with trees.

This demo utilises a SnapshotVisitor to trigger a post-tick handler to dump a serialisation of the tree to a json log file.

This coupling of visitor and post-tick handler can be used for any kind of event handling - the visitor is the trigger and the post-tick handler the action. Aside from logging, the most common use case is to serialise the tree for messaging to a graphical, runtime monitor.

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

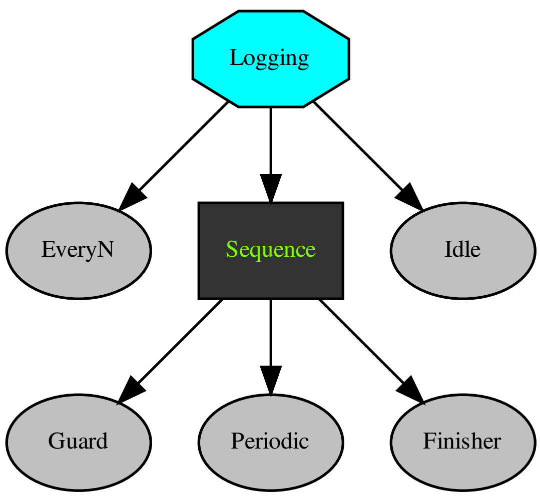
12.9.1 Named Arguments

-r, --render render dot tree to file

Default: False

-i, --interactive pause and wait for keypress at each tick

Default: False



py_trees.demos.logging.**logger**(*snapshot_visitor*, *behaviour_tree*)
A post-tick handler that logs the tree (relevant parts thereof) to a yaml file.

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

Listing 9: py_trees/demos/logging.py

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

(continues on next page)

(continued from previous page)

```python
# Documentation
##############################################################################

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

.. graphviz:: dot/demo-logging.dot

.. image:: images/logging.gif
"""

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

import argparse
import functools
import json
import py_trees
import sys
import time

import py_trees.console as console

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


def description(root):
    content = "A demonstration of logging with trees.\n\n"
    content += "This demo utilises a SnapshotVisitor to trigger\n"
    content += "a post-tick handler to dump a serialisation of the\n"
    content += "tree to a json log file.\n"
    content += "\n"
    content += "This coupling of visitor and post-tick handler can be\n"
    content += "used for any kind of event handling - the visitor is the\n"
    content += "trigger and the post-tick handler the action. Aside from\n"
    content += "logging, the most common use case is to serialise the tree\n"
    content += "for messaging to a graphical, runtime monitor.\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 + "Logging".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)

```python


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()),
                                     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 logger(snapshot_visitor, behaviour_tree):
    """
    A post-tick handler that logs the tree (relevant parts thereof) to a yaml file.
    """
    if snapshot_visitor.changed:
        print(console.cyan + "Logging.......................yes\n" + console.reset)
        tree_serialisation = {
            'tick': behaviour_tree.count,
            'nodes': []
        }
        for node in behaviour_tree.root.iterate():
            node_type_str = "Behaviour"
            for behaviour_type in [py_trees.composites.Sequence,
                                   py_trees.composites.Selector,
                                   py_trees.composites.Parallel,
                                   py_trees.decorators.Decorator]:
                if isinstance(node, behaviour_type):
                    node_type_str = behaviour_type.__name__
            node_snapshot = {
                'name': node.name,
                'id': str(node.id),
                'parent_id': str(node.parent.id) if node.parent else "none",
                'child_ids': [str(child.id) for child in node.children],
                'tip_id': str(node.tip().id) if node.tip() else 'none',
                'class_name': str(node.__module__) + '.' + str(type(node).__name__),
                'type': node_type_str,
                'status': node.status.value,
                'message': node.feedback_message,
                'is_active': True if node.id in snapshot_visitor.visited else False
            }
            tree_serialisation['nodes'].append(node_snapshot)
        if behaviour_tree.count == 0:
            with open('dump.json', 'w+') as outfile:
```

(continues on next page)

(continued from previous page)

```
                json.dump(tree_serialisation, outfile, indent=4)
        else:
            with open('dump.json', 'a') as outfile:
                json.dump(tree_serialisation, outfile, indent=4)
    else:
        print(console.yellow + "Logging.......................no\n" + console.reset)


def create_tree():
    every_n_success = py_trees.behaviours.SuccessEveryN("EveryN", 5)
    sequence = py_trees.composites.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.composites.Selector(name="Logging")
    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.
    """
    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)

    debug_visitor = py_trees.visitors.DebugVisitor()
    snapshot_visitor = py_trees.visitors.DisplaySnapshotVisitor()

    behaviour_tree.visitors.append(debug_visitor)
    behaviour_tree.visitors.append(snapshot_visitor)

```

(continues on next page)

(continued from previous page)

```
    behaviour_tree.add_post_tick_handler(functools.partial(logger, snapshot_visitor))

    behaviour_tree.setup(timeout=15)

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

12.10 py-trees-demo-selector

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

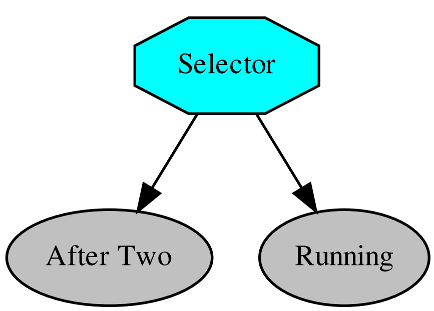
In this example the higher priority child is setup to fail initially, falling back to the continually running second child. On the third tick, the first child succeeds and cancels the hitherto running child.

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

12.10.1 Named Arguments

-r, --render render dot tree to file

Default: False



py_trees.demos.selector.**main**()

Entry point for the demo script.

Listing 10: py_trees/demos/selector.py

```
#!/usr/bin/env python
#
# License: BSD
#   https://raw.githubusercontent.com/splintered-reality/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_root():
    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

    root = create_root()

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

(continues on next page)

(continued from previous page)

```python

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

12.11 py-trees-demo-sequence

Demonstrates sequences in action.

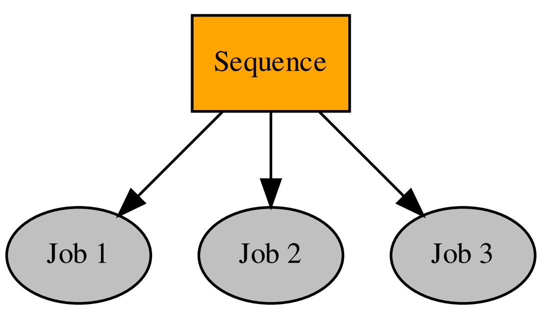
A sequence is populated with 2-tick jobs that are allowed to run through to completion.

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

12.11.1 Named Arguments

-r, --render render dot tree to file

Default: False



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

Listing 11: py_trees/demos/sequence.py

```
#!/usr/bin/env python
#
# License: BSD
#   https://raw.githubusercontent.com/splintered-reality/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)

```python


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

    root = create_root()

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

    ####################
    # Execute
    ####################
    root.setup_with_descendants()
```

(continues on next page)

(continued from previous page)

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

12.12 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 | --render-with-blackboard-variables | -i]
```

12.12.1 Named Arguments

-r, --render render dot tree to file

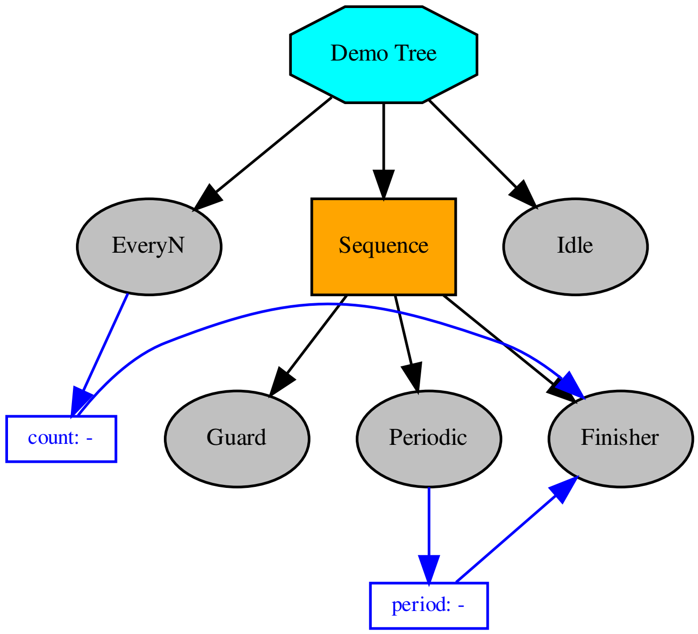
Default: False

--render-with-blackboard-variables render dot tree to file with blackboard variables

Default: False

-i, --interactive pause and wait for keypress at each tick

Default: False



class py_trees.demos.stewardship.**Finisher**

Bases: *py_trees.behaviour.Behaviour*

__init__()

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

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.demos.stewardship.**PeriodicSuccess**

Bases: *py_trees.behaviours.Periodic*

__init__()

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

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.demos.stewardship.**SuccessEveryN**

Bases: *py_trees.behaviours.SuccessEveryN*

__init__()

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

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

py_trees.demos.stewardship.**main**()

Entry point for the demo script.

Listing 12: py_trees/demos/stewardship.py

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

"""
.. argparse::
   :module: py_trees.demos.stewardship
   :func: command_line_argument_parser
```

(continues on next page)

(continued from previous page)

```
   :prog: py-trees-demo-tree-stewardship

.. graphviz:: dot/demo-tree-stewardship.dot

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

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

import argparse
import py_trees
import sys
import time

import py_trees.console as console

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


def description():
    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
```

(continues on next page)

(continued from previous page)

```
    else:
        return None


def command_line_argument_parser():
    parser = argparse.ArgumentParser(description=description(),
                                     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(
        '--render-with-blackboard-variables',
        action='store_true',
        help='render dot tree to file with blackboard variables'
    )
    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):
    print("\n--------- Run %s ---------\n" % behaviour_tree.count)


class SuccessEveryN(py_trees.behaviours.SuccessEveryN):
    def __init__(self):
        super().__init__(name="EveryN", n=5)
        self.blackboard = self.attach_blackboard_client(name=self.name)
        self.blackboard.register_key("count", access=py_trees.common.Access.WRITE)

    def update(self):
        status = super().update()
        self.blackboard.count = self.count
        return status


class PeriodicSuccess(py_trees.behaviours.Periodic):
    def __init__(self):
        super().__init__(name="Periodic", n=3)
        self.blackboard = self.attach_blackboard_client(name=self.name)
        self.blackboard.register_key("period", access=py_trees.common.Access.WRITE)

    def update(self):
        status = super().update()
        self.blackboard.period = self.period
        return status


class Finisher(py_trees.behaviour.Behaviour):
    def __init__(self):
        super().__init__(name="Finisher")
        self.blackboard = self.attach_blackboard_client(name=self.name)
        self.blackboard.register_key("count", access=py_trees.common.Access.READ)
        self.blackboard.register_key("period", access=py_trees.common.Access.READ)
```

(continues on next page)

(continued from previous page)

```python

    def update(self):
        print(console.green + "----------------------------" + console.reset)
        print(console.bold + "        Finisher" + console.reset)
        print(console.green + "  Count : {}".format(self.blackboard.count) + console.reset)
        print(console.green + "  Period: {}".format(self.blackboard.period) + console.reset)
        print(console.green + "----------------------------" + console.reset)
        return py_trees.common.Status.SUCCESS


def create_tree():
    every_n_success = SuccessEveryN()
    sequence = py_trees.composites.Sequence(name="Sequence")
    guard = py_trees.behaviours.Success("Guard")
    periodic_success = PeriodicSuccess()
    finisher = Finisher()
    sequence.add_child(guard)
    sequence.add_child(periodic_success)
    sequence.add_child(finisher)
    idle = py_trees.behaviours.Success("Idle")
    root = py_trees.composites.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.
    """
    args = command_line_argument_parser().parse_args()
    py_trees.logging.level = py_trees.logging.Level.DEBUG
    tree = create_tree()
    print(description())

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

    if args.render_with_blackboard_variables:
        py_trees.display.render_dot_tree(tree, with_blackboard_variables=True)
        sys.exit()

    ####################
    # Tree Stewardship
    ####################
    py_trees.blackboard.Blackboard.enable_activity_stream(100)
```

(continues on next page)

(continued from previous page)

```
    behaviour_tree = py_trees.trees.BehaviourTree(tree)
    behaviour_tree.add_pre_tick_handler(pre_tick_handler)
    behaviour_tree.visitors.append(py_trees.visitors.DebugVisitor())
    behaviour_tree.visitors.append(
        py_trees.visitors.DisplaySnapshotVisitor(
            display_blackboard=True,
            display_activity_stream=True)
    )
    behaviour_tree.setup(timeout=15)

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

12.13 py-trees-demo-pick-up-where-you-left-off

A demonstration of the 'pick up where you left off' idiom.

A common behaviour tree pattern that allows you to resume work after being interrupted by a high priority interrupt.

EVENTS

- 2 : task one done, task two running
- 3 : high priority interrupt
- 7 : task two restarts
- 9 : task two done

```
usage: py-trees-demo-pick-up-where-you-left-off [-h] [-r | -i]
```

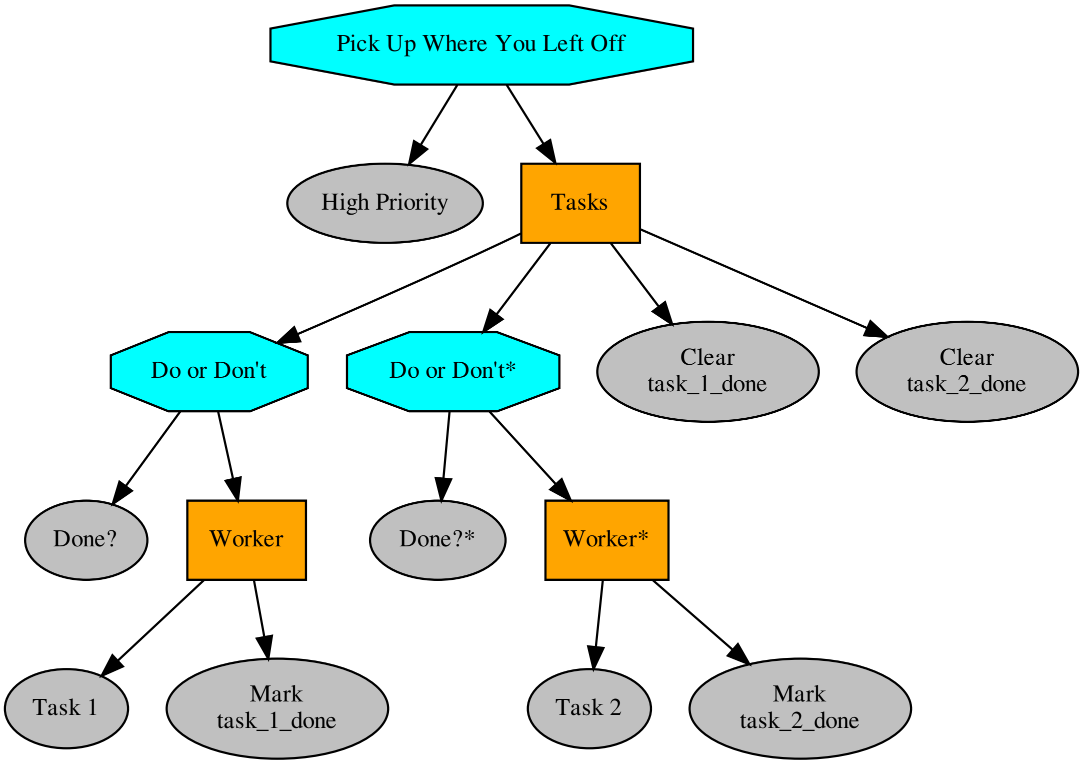
12.13.1 Named Arguments

-r, --render render dot tree to file

Default: False

-i, --interactive pause and wait for keypress at each tick

Default: False



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

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

py_trees.demos.pick_up_where_you_left_off.**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 13: py_trees/demos/pick_up_where_you_left_off.py

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

"""
.. argparse::
```

(continues on next page)

(continued from previous page)

```
    :module: py_trees.demos.pick_up_where_you_left_off
    :func: command_line_argument_parser
    :prog: py-trees-demo-pick-up-where-you-left-off

.. graphviz:: dot/pick_up_where_you_left_off.dot

.. image:: images/pick_up_where_you_left_off.gif
"""

##############################################################################
# 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 the 'pick up where you left off' idiom.\n\n"
    content += "A common behaviour tree pattern that allows you to resume\n"
    content += "work after being interrupted by a high priority interrupt.\n"
    content += "\n"
    content += "EVENTS\n"
    content += "\n"
    content += " -  2 : task one done, task two running\n"
    content += " -  3 : high priority interrupt\n"
    content += " -  7 : task two restarts\n"
    content += " -  9 : task two done\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 + "Pick Up Where you Left Off".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
```

(continues on next page)

(continued from previous page)

```python
    else:
        return None


def command_line_argument_parser():
    parser = argparse.ArgumentParser(description=description(create_root()),
                                     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.unicode_tree(
            root=behaviour_tree.root,
            visited=snapshot_visitor.visited,
            previously_visited=snapshot_visitor.previously_visited
        )
    )


def create_root():
    task_one = py_trees.behaviours.Count(
        name="Task 1",
        fail_until=0,
        running_until=2,
        success_until=10
    )
    task_two = py_trees.behaviours.Count(
        name="Task 2",
        fail_until=0,
        running_until=2,
        success_until=10
    )
    high_priority_interrupt = py_trees.decorators.RunningIsFailure(
        child=py_trees.behaviours.Periodic(
```

(continues on next page)

(continued from previous page)

```
            name="High Priority",
            n=3
        )
    )
    piwylo = py_trees.idioms.pick_up_where_you_left_off(
        name="Pick Up\nWhere You\nLeft Off",
        tasks=[task_one, task_two]
    )
    root = py_trees.composites.Selector(name="Root")
    root.add_children([high_priority_interrupt, piwylo])

    return root

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


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

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

    ####################
    # Tree Stewardship
    ####################
    behaviour_tree = py_trees.trees.BehaviourTree(root)
    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:
        py_trees.console.read_single_keypress()
    for unused_i in range(1, 11):
        try:
            behaviour_tree.tick()
            if args.interactive:
                py_trees.console.read_single_keypress()
            else:
```

(continues on next page)

(continued from previous page)

```
                time.sleep(0.5)
        except KeyboardInterrupt:
            break
    print("\n")
```

CHAPTER 13

Programs

13.1 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 --with-blackboard-variables
$ 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] [-b] [-v]
                       method
```

13.1.1 Positional Arguments

method space separated list of blackboard variables to watch

13.1.2 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: {}

-b, --with-blackboard-variables add nodes for the blackboard variables

Default: False

-v, --verbose embellish each node in the dot graph with extra information

Default: False

CHAPTER 14

Module API

14.1 py_trees

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

14.2 py_trees.behaviour

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

class py_trees.behaviour.**Behaviour**(*name=<Name.AUTO_GENERATED: 'AUTO_GENERATED'>*)

Bases: `object`

Defines the basic properties and methods required of a node in a behaviour tree. When implementing your own behaviour, subclass this class.

Parameters **name** (`Union[str, Name]`) – the behaviour name, defaults to auto-generating from the class name

Raises `TypeError` – if the provided name is not a string

Variables

- **id** (`uuid.UUID`) – automagically generated unique identifier for the behaviour
- **name** (`str`) – the behaviour name
- **blackboards** (*typing.List[py_trees.blackboard.Client]*) – collection of attached blackboard clients
- **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
- **logger** (`logging.Logger`) – a simple logging mechanism

- **feedback_message** (str) – improve debugging with a simple message
- **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*

attach_blackboard_client(*name=None*, *namespace=None*)

Create and attach a blackboard to this behaviour.

Parameters

- **name** (Optional[str]) – human-readable (not necessarily unique) name for the client
- **namespace** (Optional[str]) – sandbox the client to variables behind this namespace

Return type *Client*

Returns a handle to the attached blackboard client

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(***kwargs*)

Note: User Customisable Callback

Subclasses may override this method for any one-off delayed construction & validation that is necessary prior to ticking the tree. Such construction is best done here rather than in __init__ so that trees can be instantiated on the fly for easy rendering to dot graphs without imposing runtime requirements (e.g. establishing a middleware connection to a sensor or a driver to a serial port).

Equally as important, executing methods which validate the configuration of behaviours will increase confidence that your tree will successfully tick without logical software errors before actually ticking. This is useful both before a tree's first tick and immediately after any modifications to a tree has been made between ticks.

Tip: Faults are notified to the user of the behaviour via exceptions. Choice of exception to use is left to the user.

Warning: The kwargs argument is for distributing objects at runtime to behaviours before ticking. For example, a simulator instance with which behaviours can interact with the simulator's python api, a ros2 node for setting up communications. Use sparingly, as this is not proof against keyword conflicts amongst disparate libraries of behaviours.

Parameters ****kwargs** (`dict`) – distribute arguments to this behaviour and in turn, all of it's children

Raises `Exception` – if this behaviour has a fault in construction or configuration

See also:

py_trees.behaviour.Behaviour.shutdown()

setup_with_descendants()

Iterates over this child, it's children (it's children's children, . . .) calling the user defined *setup()* on each in turn.

shutdown()

Note: User Customisable Callback

Subclasses may override this method for any custom destruction of infrastructure usually brought into being in *setup()*.

Raises `Exception` – of whatever flavour the child raises when errors occur on destruction

See also:

py_trees.behaviour.Behaviour.setup()

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:** Override this method only in exceptional circumstances, prefer overriding *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

> **Warning:** Override this method only in exceptional circumstances, prefer overriding `update()` instead.

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.

14.3 py_trees.behaviours

A library of fundamental behaviours for use.

class py_trees.behaviours.**BlackboardToStatus**(*variable_name*, *name=<Name.AUTO_GENERATED: 'AUTO_GENERATED'>*)

Bases: `py_trees.behaviour.Behaviour`

This behaviour reverse engineers the `StatusToBlackboard` decorator. Used in conjuction with that decorator, this behaviour can be used to reflect the status of a decision elsewhere in the tree.

> **Note:** A word of caution. The consequences of a behaviour's status should be discernable upon inspection of the tree graph. If using StatusToBlackboard and BlackboardToStatus to reflect a behaviour's status across a tree, this is no longer true. The graph of the tree communicates the local consequences, but not the reflected consequences at the point BlackboardToStatus is used. A recommendation, use this class only where other options are infeasible or impractical.

Parameters

- **variable_name** (str) – name of the variable look for, may be nested, e.g. battery.percentage
- **name** (Union[str, *Name*]) – name of the behaviour

Raises

- KeyError – if the variable doesn't exist
- TypeError – if the variable isn't of type *Status*

update()

Check for existence.

Return type *Status*

Returns *SUCCESS* if key found, *FAILURE* otherwise.

class py_trees.behaviours.**CheckBlackboardVariableExists**(*variable_name*, *name=<Name.AUTO_GENERATED: 'AUTO_GENERATED'>*)

Bases: *py_trees.behaviour.Behaviour*

Check the blackboard to verify if a specific variable (key-value pair) exists. This is non-blocking, so will always tick with status *FAILURE SUCCESS*.

See also:

WaitForBlackboardVariable for the blocking counterpart to this behaviour.

Parameters

- **variable_name** (str) – name of the variable look for, may be nested, e.g. battery.percentage
- **name** (Union[str, *Name*]) – name of the behaviour

update()

Check for existence.

Return type *Status*

Returns *SUCCESS* if key found, *FAILURE* otherwise.

class py_trees.behaviours.**CheckBlackboardVariableValue**(*check*, *name=<Name.AUTO_GENERATED: 'AUTO_GENERATED'>*)

Bases: *py_trees.behaviour.Behaviour*

Inspect a blackboard variable and if it exists, check that it meets the specified criteria (given by operation type and expected value). This is non-blocking, so it will always tick with *SUCCESS* or *FAILURE*.

Parameters

- **check** (ComparisonExpression) – a comparison expression to check against
- **name** (Union[str, *Name*]) – name of the behaviour

Note: If the variable does not yet exist on the blackboard, the behaviour will return with status *FAILURE*.

Tip: The python operator module includes many useful comparison operations.

---

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.behaviours.**CheckBlackboardVariableValues**(*checks*, *operator*, *name=<Name.AUTO_GENERATED: 'AUTO_GENERATED'>*, *namespace=None*)

Bases: *py_trees.behaviour.Behaviour*

Apply a logical operation across a set of blackboard variable checks. This is non-blocking, so will always tick with status *FAILURE* or *SUCCESS*.

Parameters

- **checks** (List[ComparisonExpression]) – a list of comparison checks to apply to blackboard variables
- **logical_operator** – a logical check to apply across the results of the blackboard variable checks
- **name** (Union[str, *Name*]) – name of the behaviour
- **namespace** (Optional[str]) – optionally store results of the checks (boolean) under this namespace

---

Tip: The python operator module includes many useful logical operators, e.g. operator.xor.

Raises ValueError if less than two variable checks are specified (insufficient for logical operations)

update()

Applies comparison checks on each variable and a logical check across the complete set of variables.

Return type *Status*

Returns *FAILURE* if key retrieval or logical checks failed, *SUCCESS* otherwise.

class py_trees.behaviours.**Count**(*name='Count'*, *fail_until=3*, *running_until=5*, *success_until=6*, *reset=True*)

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.**Dummy**(*name='Dummy'*)
Bases: *py_trees.behaviour.Behaviour*

class py_trees.behaviours.**Failure**(*name='Failure'*)
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='Running'*)
Bases: *py_trees.behaviour.Behaviour*

class py_trees.behaviours.**SetBlackboardVariable**(*variable_name*, *variable_value*, *overwrite=True*, *name=<Name.AUTO_GENERATED: 'AUTO_GENERATED'>*)
Bases: *py_trees.behaviour.Behaviour*

Set the specified variable on the blackboard.

Parameters

- **variable_name** (str) – name of the variable to set, may be nested, e.g. battery.percentage
- **variable_value** (Union[Any, Callable[[], Any]]) – value of the variable to set
- **overwrite** (bool) – when False, do not set the variable if it already exists
- **name** (Union[str, *Name*]) – name of the behaviour

update()
Always return success.

Return type *Status*

Returns *FAILURE* if no overwrite requested and the variable exists, *SUCCESS* otherwise

class py_trees.behaviours.**StatusSequence**(*name*, *sequence*, *eventually*)
Bases: *py_trees.behaviour.Behaviour*

Cycle through the specified sequence of states.

Parameters

- **name** (str) – name of the behaviour
- **sequence** (List[*Status*]) – list of status values to cycle through

- **eventually** (Optional[*Status*]) – status to use eventually, None to re-cycle the sequence

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.**Success**(*name='Success'*)

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.decorators.FailureIsRunning()*

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.**TickCounter**(*duration*, *name=<Name.AUTO_GENERATED: 'AUTO_GENERATED'>*, *completion_status=<Status.SUCCESS: 'SUCCESS'>*)

Bases: *py_trees.behaviour.Behaviour*

A useful utility behaviour for demos and tests. Simply ticks with *RUNNING* for the specified number of ticks before returning the requested completion status (*SUCCESS* or *FAILURE*).

This behaviour will reset the tick counter when initialising.

Parameters

- **name** – name of the behaviour
- **duration** (int) – number of ticks to run
- **completion_status** (*Status*) – status to switch to once the counter has expired

initialise()

Reset the tick counter.

update()

Increment the tick counter and return the appropriate status for this behaviour based on the tick count.

Returns *RUNNING* while not expired, the given completion status otherwise

class py_trees.behaviours.**UnsetBlackboardVariable**(*key*, *name=<Name.AUTO_GENERATED: 'AUTO_GENERATED'>*)

Bases: *py_trees.behaviour.Behaviour*

Unset the specified variable (key-value pair) from the blackboard.

This always returns *SUCCESS* regardless of whether the variable was already present or not.

Parameters

- **key** (str) – unset this key-value pair
- **name** (Union[str, *Name*]) – name of the behaviour

update()

Unset and always return success.

Return type *Status*

Returns *SUCCESS*

class py_trees.behaviours.**WaitForBlackboardVariable**(*variable_name*, *name=<Name.AUTO_GENERATED: 'AUTO_GENERATED'>*)

Bases: *py_trees.behaviours.CheckBlackboardVariableExists*

Wait for the blackboard variable to become available on the blackboard. This is blocking, so it will tick with status *SUCCESS* if the variable is found, and *RUNNING* otherwise.

See also:

CheckBlackboardVariableExists for the non-blocking counterpart to this behaviour.

Parameters

- **variable_name** (str) – name of the variable to wait for, may be nested, e.g. battery.percentage
- **name** (Union[str, *Name*]) – name of the behaviour

update()

Check for existence, wait otherwise.

Return type *Status*

Returns *SUCCESS* if key found, *RUNNING* otherwise.

class py_trees.behaviours.**WaitForBlackboardVariableValue**(*check*, *name=<Name.AUTO_GENERATED: 'AUTO_GENERATED'>*)

Bases: *py_trees.behaviours.CheckBlackboardVariableValue*

Inspect a blackboard variable and if it exists, check that it meets the specified criteria (given by operation type and expected value). This is blocking, so it will always tick with *SUCCESS* or *RUNNING*.

See also:

CheckBlackboardVariableValue for the non-blocking counterpart to this behaviour.

Note: If the variable does not yet exist on the blackboard, the behaviour will return with status *RUNNING*.

Parameters

- **check** (ComparisonExpression) – a comparison expression to check against
- **name** (Union[str, *Name*]) – name of the behaviour

update()

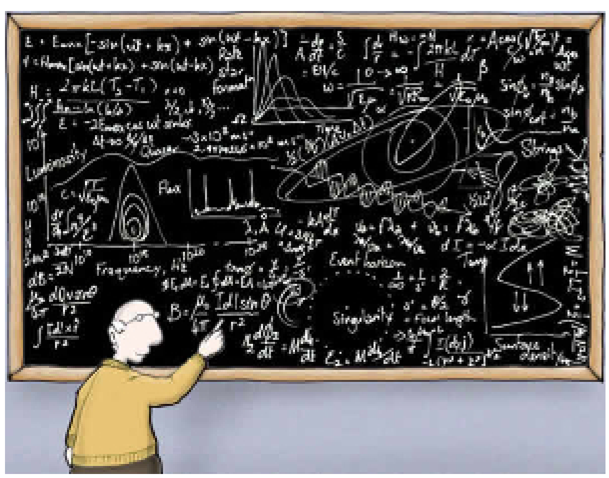
Check for existence, or the appropriate match on the expected value.

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

Return type *Status*

14.4 py_trees.blackboard

Blackboards are not a necessary component of behaviour tree implementations, but are nonetheless, a fairly common mechanism for sharing data between behaviours in the tree. See, for example, the design notes for blackboards in Unreal Engine.



Implementations vary widely depending on the needs of the framework using them. The simplest implementations take the form of a key-value store with global access, while more rigorous implementations scope access or form a secondary graph overlaying the tree connecting data ports between behaviours.

The *'Zen of PyTrees'* is to enable rapid development, yet be rich enough so that *all* of the magic is exposed for debugging purposes. The first implementation of a blackboard was merely a global key-value store with an api that lent itself to ease of use, but did not expose the data sharing between behaviours which meant any tooling used to introspect or visualise the tree, only told half the story.

The current implementation adopts a strategy similar to that of a filesystem. Each client (subsequently behaviour) registers itself for read/write access to keys on the blackboard. This is less to do with permissions and more to do with tracking users of keys on the blackboard - extremely helpful with debugging.

The alternative approach of layering a secondary data graph with parameter and input-output ports on each behaviour was discarded as being too heavy for the zen requirements of py_trees. This is in part due to the wiring costs, but also due to complexity arising from a tree's partial graph execution (a feature which makes trees different from most computational graph frameworks) and not to regress on py_trees' capability to dynamically insert and prune subtrees on the fly.

A high-level list of existing / planned features:

- [+] Centralised key-value store
- [+] Client connections with namespaced read/write access to the store
- [+] Integration with behaviours for key-behaviour associations (debugging)
- [+] Activity stream that logs read/write operations by clients
- [+] Exclusive locks for writing
- [+] Framework for key remappings

class py_trees.blackboard.**ActivityItem**(*key*, *client_name*, *client_id*, *activity_type*, *previous_value=None*, *current_value=None*)

Bases: object

Recorded data pertaining to activity on the blackboard.

Parameters

- **key** – name of the variable on the blackboard
- **client_name** (str) – convenient name of the client performing the operation
- **client_id** (UUID) – unique id of the client performing the operation
- **activity_type** (str) – type of activity
- **previous_value** (Optional[Any]) – of the given key (None if this field is not relevant)
- **current_value** (Optional[Any]) – current value for the given key (None if this field is not relevant)

__init__(*key*, *client_name*, *client_id*, *activity_type*, *previous_value=None*, *current_value=None*)

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

__weakref__

list of weak references to the object (if defined)

class py_trees.blackboard.**ActivityStream**(*maximum_size=500*)

Bases: object

Storage container with convenience methods for manipulating the stored activity stream.

Variables

- **(typing.List[*ActivityItem*]** (*data*) – list of activity items, earliest first
- **maximum_size** (*int*) – pop items if this size is exceeded

__init__(*maximum_size=500*)
Initialise the stream with a maximum storage limit.

Parameters **maximum_size** (`int`) – pop items from the stream if this size is exceeded

__weakref__
list of weak references to the object (if defined)

clear()
Delete all activities from the stream.

push(*activity_item*)
Push the next activity item to the stream.

Parameters **activity_item** (*ActivityItem*) – new item to append to the stream

class py_trees.blackboard.**ActivityType**
Bases: `enum.Enum`

An enumerator representing the operation on a blackboard variable

ACCESSED = 'ACCESSED'
Key accessed, either for reading, or modification of the value's internal attributes (e.g. foo.bar).

ACCESS_DENIED = 'ACCESS_DENIED'
Client did not have access to read/write a key.

INITIALISED = 'INITIALISED'
Initialised a key-value pair on the blackboard

NO_KEY = 'NO_KEY'
Tried to access a key that does not yet exist on the blackboard.

NO_OVERWRITE = 'NO_OVERWRITE'
Tried to write but variable already exists and a no-overwrite request was respected.

READ = 'READ'
Read from the blackboard

UNSET = 'UNSET'
Key was removed from the blackboard

WRITE = 'WRITE'
Wrote to the blackboard.

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

Centralised key-value store for sharing data between behaviours. This class is a coat-hanger for the centralised data store, metadata for it's administration and static methods for interacting with it.

This api is intended for authors of debugging and introspection tools on the blackboard. Users should make use of the *Client*.

Variables

- **Blackboard.clients** (*typing.Dict[uuid.UUID, str]*) – client uuid-name registry
- **Blackboard.storage** (*typing.Dict[str, typing.Any]*) – key-value data store
- **Blackboard.metadata** (*typing.Dict[str, KeyMetaData]*) – key associated metadata

- **Blackboard.activity_stream** (ActivityStream) – logged activity
- **Blackboard.separator** (*char*) – namespace separator character

__weakref__
list of weak references to the object (if defined)

static absolute_name(*namespace*, *key*)
Generate the fully qualified key name from namespace and name arguments.

Examples

```
'/' + 'foo'  = '/foo'
'/' + '/foo' = '/foo'
'/foo' + 'bar' = '/foo/bar'
'/foo/' + 'bar' = '/foo/bar'
'/foo' + '/foo/bar' = '/foo/bar'
'/foo' + '/bar' = '/bar'
'/foo' + 'foo/bar' = '/foo/foo/bar'
```

Parameters

- **namespace** (str) – namespace the key should be embedded in
- **key** (str) – key name (relative or absolute)

Return type str

Returns the absolute name

> **Warning:** To expedite the method call (it's used with high frequency in blackboard key lookups), no checks are made to ensure the namespace argument leads with a "/". Nor does it check that a name in absolute form is actually embedded in the specified namespace, it just returns the given (absolute) name directly.

static clear()
Completely clear all key, value and client information from the blackboard. Also deletes the activity stream.

static disable_activity_stream()
Disable logging of activities on the blackboard

static enable_activity_stream(*maximum_size=500*)
Enable logging of activities on the blackboard.

Parameters **maximum_size** (int) – pop items from the stream if this size is exceeded

Raises RuntimeError if the activity stream is already enabled

static exists(*name*)
Check if the specified variable exists on the blackboard.

Parameters **name** (str) – name of the variable, can be nested, e.g. battery.percentage

Raises AttributeError – if the client does not have read access to the variable

Return type bool

static get(*variable_name*)
Extract the value associated with the given a variable name, can be nested, e.g. battery.percentage. This

differs from the client get method in that it doesn't pass through the client access checks. To be used for utility tooling (e.g. display methods) and not by users directly.

Parameters **variable_name** (`str`) – of the variable to get, can be nested, e.g. battery.percentage

Raises `KeyError` – if the variable or it's nested attributes do not yet exist on the blackboard

Return type `Any`

Returns The stored value for the given variable

static key(*variable_name*)

Extract the key for an arbitrary blackboard variable, keeping in mind that blackboard variable names can be pointing to a nested attribute within a key.

Example: '/foo/bar.woohoo -> /foo/bar'.

Parameters **variable_name** (`str`) – blackboard variable name - can be nested, e.g. battery.percentage

Return type `str`

Returns name of the underlying key

static key_with_attributes(*variable_name*)

Extract the key for an arbitrary blackboard variable, keeping in mind that blackboard variable names can be pointing to a nested attribute within a key,

Example: '/foo/bar.woohoo -> (/foo/bar'. 'woohoo')

Parameters **variable_name** (`str`) – blackboard variable name - can be nested, e.g. battery.percentage

Return type `Tuple[str, str]`

Returns a tuple consisting of the key and it's attributes (in string form)

static keys()

Get the set of blackboard keys.

Return type `Set[str]`

Returns the complete set of keys registered by clients

static keys_filtered_by_clients(*client_ids*)

Get the set of blackboard keys filtered by client unique identifiers.

Parameters **client_ids** (`Union[Set[UUID], List[UUID]]`) – set of client uuid's.

Return type `Set[str]`

Returns subset of keys that have been registered by the specified clients

static keys_filtered_by_regex(*regex*)

Get the set of blackboard keys filtered by regex.

Parameters **regex** (`str`) – a python regex string

Return type `Set[str]`

Returns subset of keys that have been registered and match the pattern

static relative_name(*namespace*, *key*)

Examples

```
'/' + 'foo'  = '/foo'
'/' + '/foo' = '/foo'
'/foo' + 'bar' = '/foo/bar'
'/foo/' + 'bar' = '/foo/bar'
'/foo' + '/bar' => KeyError('/bar' is not in 'foo')
'/foo' + 'foo/bar' = '/foo/foo/bar'
```

Parameters

- **namespace** (str) – namespace the key should be embedded in
- **key** (str) – key name (relative or absolute)

Return type str

Returns the absolute name

Raises KeyError if the key is not in the specified namespace

> **Warning:** To expedite the method call (it's used with high frequency in blackboard key lookups), no checks are made to ensure the namespace argument leads with a "/"

static set (*variable_name*, *value*)

Set the value associated with the given a variable name, can be nested, e.g. battery.percentage. This differs from the client get method in that it doesn't pass through the client access checks. To be used for utility tooling (e.g. display methods) and not by users directly.

Parameters **variable_name** (str) – of the variable to set, can be nested, e.g. battery.percentage

Raises AttributeError – if it is attempting to set a nested attribute tha does not exist.

static unset (*key*)

For when you need to completely remove a blackboard variable (key-value pair), this provides a convenient helper method.

Parameters **key** (str) – name of the variable to remove

Returns True if the variable was removed, False if it was already absent

class py_trees.blackboard.**Client** (*, *name=None*, *namespace=None*)

Bases: object

Client to the key-value store for sharing data between behaviours.

Examples

Blackboard clients will accept a user-friendly name or create one for you if none is provided. Regardless of what name is chosen, clients are always uniquely identified via a uuid generated on construction.

```
provided = py_trees.blackboard.Client(name="Provided")
print(provided)
generated = py_trees.blackboard.Client()
print(generated)
```

Register read/write access for keys on the blackboard. Note, registration is not initialisation.

Fig. 1: Client Instantiation

```
blackboard = py_trees.blackboard.Client(name="Client")
blackboard.register_key(key="foo", access=py_trees.common.Access.WRITE)
blackboard.register_key(key="bar", access=py_trees.common.Access.READ)
blackboard.foo = "foo"
print(blackboard)
```

Fig. 2: Variable Read/Write Registration

Keys and clients can make use of namespaces, designed by the '/' char. Most methods permit a flexible expression of either relative or absolute names.

```
blackboard = py_trees.blackboard.Client(name="Global")
parameters = py_trees.blackboard.Client(name="Parameters", namespace="parameters")

blackboard.register_key(key="foo", access=py_trees.common.Access.WRITE)
blackboard.register_key(key="/bar", access=py_trees.common.Access.WRITE)
blackboard.register_key(key="/parameters/default_speed", access=py_trees.common.
↪Access.WRITE)
parameters.register_key(key="aggressive_speed", access=py_trees.common.Access.
↪WRITE)

blackboard.foo = "foo"
blackboard.bar = "bar"
blackboard.parameters.default_speed = 20.0
parameters.aggressive_speed = 60.0

miss_daisy = blackboard.parameters.default_speed
van_diesel = parameters.aggressive_speed
```

(continues on next page)

(continued from previous page)

```
print(blackboard)
print(parameters)
```

Fig. 3: Namespaces and Namespaced Clients

Disconnected instances will discover the centralised key-value store.

```
def check_foo():
    blackboard = py_trees.blackboard.Client(name="Reader")
    blackboard.register_key(key="foo", access=py_trees.common.Access.READ)
    print("Foo: {}".format(blackboard.foo))


blackboard = py_trees.blackboard.Client(name="Writer")
blackboard.register_key(key="foo", access=py_trees.common.Access.WRITE)
blackboard.foo = "bar"
check_foo()
```

To respect an already initialised key on the blackboard:

```
blackboard = Client(name="Writer")
blackboard.register_key(key="foo", access=py_trees.common.Access.READ)
result = blackboard.set("foo", "bar", overwrite=False)
```

Store complex objects on the blackboard:

```
class Nested(object):
    def __init__(self):
        self.foo = None
        self.bar = None

    def __str__(self):
        return str(self.__dict__)


writer = py_trees.blackboard.Client(name="Writer")
```

(continues on next page)

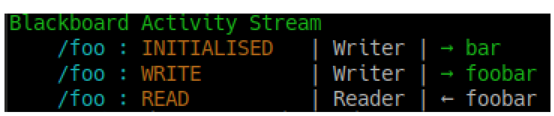
(continued from previous page)

```
writer.register_key(key="nested", access=py_trees.common.Access.WRITE)
reader = py_trees.blackboard.Client(name="Reader")
reader.register_key(key="nested", access=py_trees.common.Access.READ)

writer.nested = Nested()
writer.nested.foo = "I am foo"
writer.nested.bar = "I am bar"

foo = reader.nested.foo
print(writer)
print(reader)
```

```
Blackboard Client
  Client Data
    name              : Writer
    namespace         : /
    unique_identifier : 8d42f132-6b1f-4c1d-b149-6a32e0d19ef9
    read              : set()
    write             : {'/nested'}
  Variables
    /nested : {'foo': 'I am foo', 'bar': 'I am bar'}

Blackboard Client
  Client Data
    name              : Reader
    namespace         : /
    unique_identifier : d8523f5a-03a9-44e4-98a7-790d0f65ba16
    read              : {'/nested'}
    write             : set()
  Variables
    /nested : {'foo': 'I am foo', 'bar': 'I am bar'}
```

Log and display the activity stream:

```
py_trees.blackboard.Blackboard.enable_activity_stream(maximum_size=100)
reader = py_trees.blackboard.Client(name="Reader")
reader.register_key(key="foo", access=py_trees.common.Access.READ)
writer = py_trees.blackboard.Client(name="Writer")
writer.register_key(key="foo", access=py_trees.common.Access.WRITE)
writer.foo = "bar"
writer.foo = "foobar"
unused_result = reader.foo
print(py_trees.display.unicode_blackboard_activity_stream())
py_trees.blackboard.Blackboard.activity_stream.clear()
```



```
Blackboard Activity Stream
    /foo : INITIALISED    | Writer | → bar
    /foo : WRITE          | Writer | → foobar
    /foo : READ           | Reader | ← foobar
```

Display the blackboard on the console, or part thereof:

```
writer = py_trees.blackboard.Client(name="Writer")
for key in {"foo", "bar", "dude", "dudette"}:
    writer.register_key(key=key, access=py_trees.common.Access.WRITE)

reader = py_trees.blackboard.Client(name="Reader")
for key in {"foo", "bar"}:
    reader.register_key(key="key", access=py_trees.common.Access.READ)
```

(continues on next page)

(continued from previous page)

```
writer.foo = "foo"
writer.bar = "bar"
writer.dude = "bob"

# all key-value pairs
print(py_trees.display.unicode_blackboard())
# various filtered views
print(py_trees.display.unicode_blackboard(key_filter={"foo"}))
print(py_trees.display.unicode_blackboard(regex_filter="dud*"))
print(py_trees.display.unicode_blackboard(client_filter={reader.unique_identifier}
↪))
# list the clients associated with each key
print(py_trees.display.unicode_blackboard(display_only_key_metadata=True))
```

Behaviours are not automagically connected to the blackboard but you may manually attach one or more clients so that associations between behaviours and variables can be tracked - this is very useful for introspection and debugging.

Creating a custom behaviour with blackboard variables:

```
class Foo(py_trees.behaviour.Behaviour):

    def __init__(self, name):
        super().__init__(name=name)
        self.blackboard = self.attach_blackboard_client(name="Foo Global")
        self.parameters = self.attach_blackboard_client(name="Foo Params",␣
↪namespace="foo_parameters_")
        self.state = self.attach_blackboard_client(name="Foo State", namespace=
↪"foo_state_")

        # create a key 'foo_parameters_init' on the blackboard
        self.parameters.register_key("init", access=py_trees.common.Access.READ)
        # create a key 'foo_state_number_of_noodles' on the blackboard
        self.state.register_key("number_of_noodles", access=py_trees.common.
↪Access.WRITE)
```

(continues on next page)

(continued from previous page)

```
    def initialise(self):
        self.state.number_of_noodles = self.parameters.init

    def update(self):
        self.state.number_of_noodles += 1
        self.feedback_message = self.state.number_of_noodles
        if self.state.number_of_noodles > 5:
            return py_trees.common.Status.SUCCESS
        else:
            return py_trees.common.Status.RUNNING


# could equivalently do directly via the Blackboard static methods if
# not interested in tracking / visualising the application configuration
configuration = py_trees.blackboard.Client(name="App Config")
configuration.register_key("foo_parameters_init", access=py_trees.common.Access.
↪WRITE)
configuration.foo_parameters_init = 3

foo = Foo(name="The Foo")
for i in range(1, 8):
    foo.tick_once()
    print("Number of Noodles: {}".format(foo.feedback_message))
```

Rendering a dot graph for a behaviour tree, complete with blackboard variables:

```
# in code
py_trees.display.render_dot_tree(py_trees.demos.blackboard.create_root())
# command line tools
py-trees-render --with-blackboard-variables py_trees.demos.blackboard.create_root
```

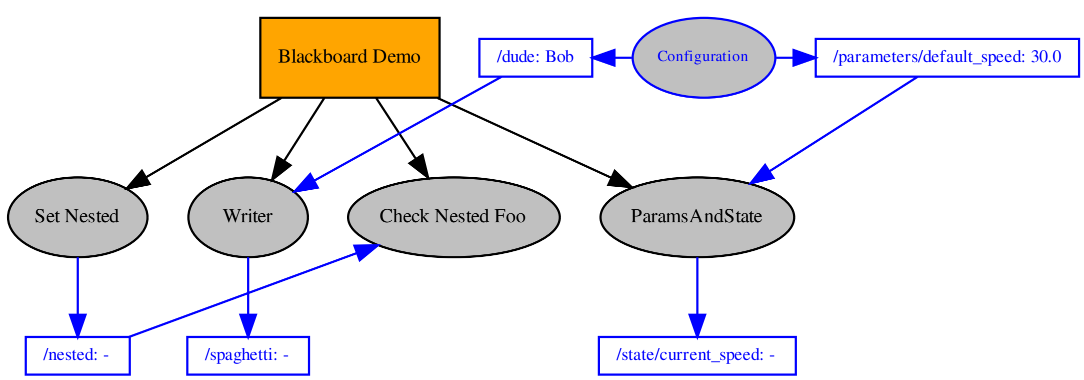


Fig. 4: Tree with Blackboard Variables

And to demonstrate that it doesn't become a tangled nightmare at scale, an example of a more complex tree:

Debug deeper with judicious application of the tree, blackboard and activity stream display methods around the tree tick (refer to `py_trees.visitors.DisplaySnapshotVisitor` for examplar code):

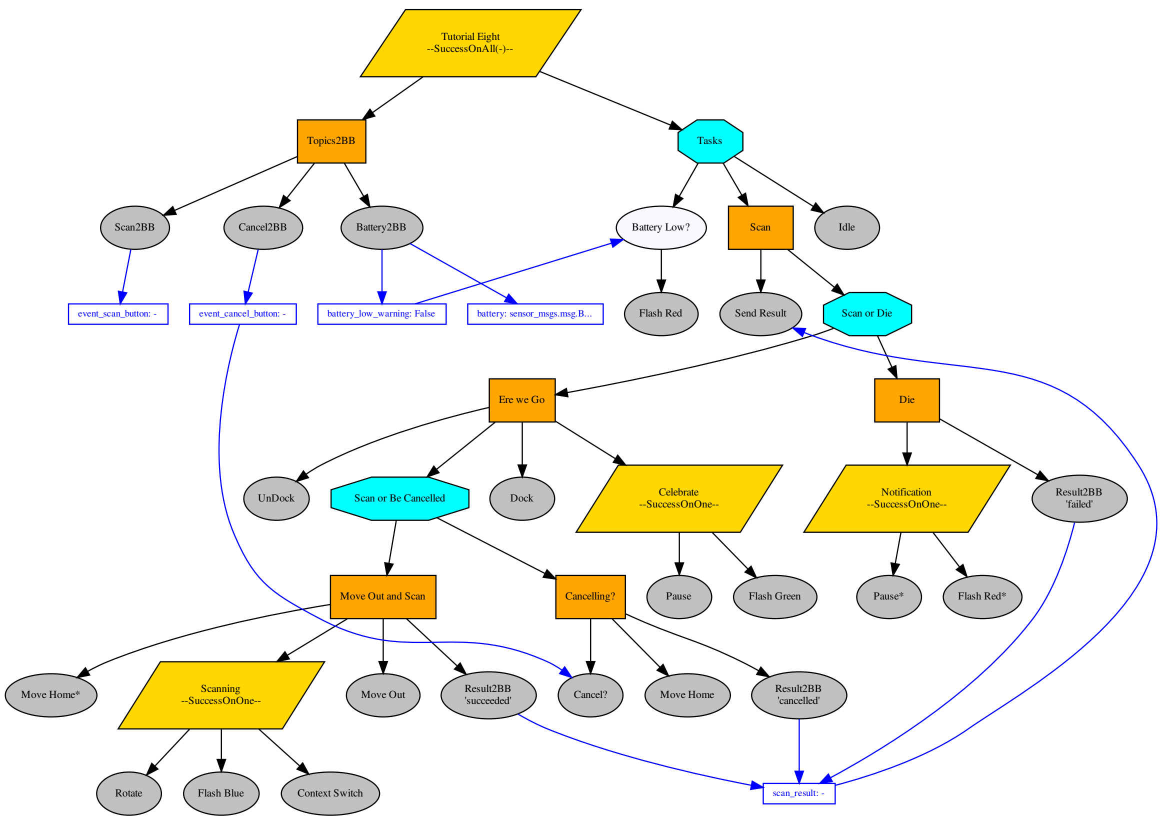


Fig. 5: A more complex tree

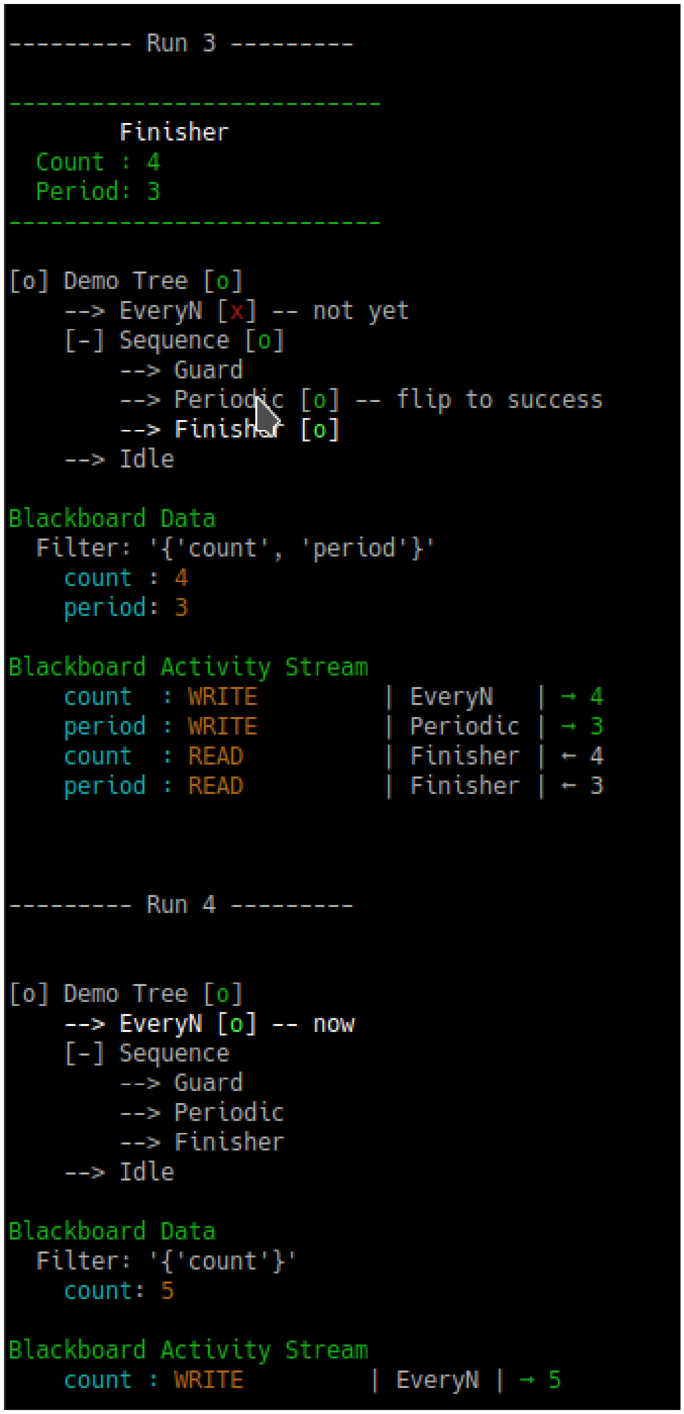


Fig. 6: Tree level debugging

See also:

- *py-trees-demo-blackboard*
- *py-trees-demo-namespaces*
- *py-trees-demo-remappings*
- `py_trees.visitors.DisplaySnapshotVisitor`
- `py_trees.behaviours.SetBlackboardVariable`
- `py_trees.behaviours.UnsetBlackboardVariable`
- `py_trees.behaviours.CheckBlackboardVariableExists`
- `py_trees.behaviours.WaitForBlackboardVariable`
- `py_trees.behaviours.CheckBlackboardVariableValue`
- `py_trees.behaviours.WaitForBlackboardVariableValue`

Variables

- **name** (*str*) – client's convenient, but not necessarily unique identifier
- **namespace** (*str*) – apply this as a prefix to any key/variable name operations
- **unique_identifier** (*uuid.UUID*) – client's unique identifier
- **read** (*typing.Set[str]*) – set of absolute key names with read access
- **write** (*typing.Set[str]*) – set of absolute key names with write access
- **exclusive** (*typing.Set[str]*) – set of absolute key names with exclusive write access
- **required** (*typing.Set[str]*) – set of absolute key names required to have data present
- **(typing.Dict[str, str]** (*remappings*) – client key names with blackboard remappings
- **(typing.Set[str]** (*namespaces*) – a cached list of namespaces this client accesses

__getattr__(*name*)

Convenience attribute style referencing with checking against permissions.

Raises

- `AttributeError` – if the client does not have read access to the variable
- `KeyError` – if the variable does not yet exist on the blackboard

__init__(**, name=None, namespace=None*)

Parameters

- **name** (`Optional`[`str`]) – client's convenient identifier (stringifies the uuid if None)
- **namespace** (`Optional`[`str`]) – prefix to apply to key/variable name operations
- **read** – list of keys for which this client has read access
- **write** – list of keys for which this client has write access
- **exclusive** – list of keys for which this client has exclusive write access

Raises

- `TypeError` – if the provided name is not of type str
- `ValueError` – if the unique identifier has already been registered

`__setattr__`(*name*, *value*)
Convenience attribute style referencing with checking against permissions.

Raises `AttributeError` – if the client does not have write access to the variable

`__str__`()
Return str(self).

`__weakref__`
list of weak references to the object (if defined)

`absolute_name`(*key*)
Generate the fully qualified key name for this key.

```
blackboard = Client(name="FooBar", namespace="foo")
blackboard.register_key(key="bar", access=py_trees.common.Access.READ)
print("{}".format(blackboard.absolute_name("bar")))  # "/foo/bar"
```

Parameters **key** (`str`) – name of the key

Return type `str`

Returns the absolute name

Raises `KeyError` – if the key is not registered with this client

`exists`(*name*)
Check if the specified variable exists on the blackboard.

Parameters **name** (`str`) – name of the variable to get, can be nested, e.g. battery.percentage

Raises `AttributeError` – if the client does not have read access to the variable

Return type `bool`

`get`(*name*)
Method based accessor to the blackboard variables (as opposed to simply using '.<name>').

Parameters **name** (`str`) – name of the variable to get, can be nested, e.g. battery.percentage

Raises

- `AttributeError` – if the client does not have read access to the variable
- `KeyError` – if the variable or it's nested attributes do not yet exist on the blackboard

Return type `Any`

`id`()
The unique identifier for this client.

Return type `UUID`

Returns The uuid.UUID object

`is_registered`(*key*, *access=None*)
Check to see if the specified key is registered.

Parameters

- **key** (str) – in either relative or absolute form
- **access** (Union[None, Access]) – access property, if None, just checks for registration, regardless of property

Return type bool

Returns if registered, True otherwise False

register_key (*key*, *access*, *required=False*, *remap_to=None*)
Register a key on the blackboard to associate with this client.

Parameters

- **key** (str) – key to register
- **access** (Access) – access level (read, write, exclusive write)
- **required** (bool) – if true, check key exists when calling *verify_required_keys_exist()*
- **remap_to** (Optional[str]) – remap the key to this location on the blackboard

Note the remap simply changes the storage location. From the perspective of the client, access via the specified 'key' remains the same.

Raises

- AttributeError if exclusive write access is requested, but write access has already been given to another client
- TypeError if the access argument is of incorrect type

set (*name*, *value*, *overwrite=True*)
Set, conditionally depending on whether the variable already exists or otherwise.

This is most useful when initialising variables and multiple elements seek to do so. A good policy to adopt for your applications in these situations is a first come, first served policy. Ensure global configuration has the first opportunity followed by higher priority behaviours in the tree and so forth. Lower priority behaviours would use this to respect the pre-configured setting and at most, just validate that it is acceptable to the functionality of it's own behaviour.

Parameters

- **name** (str) – name of the variable to set
- **value** (Any) – value of the variable to set
- **overwrite** (bool) – do not set if the variable already exists on the blackboard

Return type bool

Returns success or failure (overwrite is False and variable already set)

Raises

- AttributeError – if the client does not have write access to the variable
- KeyError – if the variable does not yet exist on the blackboard

unregister (*clear=True*)
Unregister this blackboard client and if requested, clear key-value pairs if this client is the last user of those variables.

Parameters **clear** (bool) – remove key-values pairs from the blackboard

unregister_all_keys(*clear=True*)
Unregister all keys currently registered by this blackboard client and if requested, clear key-value pairs if this client is the last user of those variables.

Parameters **clear** (bool) – remove key-values pairs from the blackboard

unregister_key(*key*, *clear=True*, *update_namespace_cache=True*)
Unegister a key associated with this client.

Parameters

- **key** (str) – key to unregister
- **clear** (bool) – remove key-values pairs from the blackboard
- **update_namespace_cache** (bool) – disable if you are batching

A method that batches calls to this method is *unregister_all_keys()*.

Raises KeyError if the key has not been previously registered

unset(*key*)
For when you need to completely remove a blackboard variable (key-value pair), this provides a convenient helper method.

Parameters **key** (str) – name of the variable to remove

Returns True if the variable was removed, False if it was already absent

verify_required_keys_exist()
En-masse check of existence on the blackboard for all keys that were hitherto registered as 'required'.

Raises: KeyError if any of the required keys do not exist on the blackboard

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

Stores the aggregated metadata for a key on the blackboard.

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

__weakref__
list of weak references to the object (if defined)

14.5 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.

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.**Duration**
Bases: `enum.Enum`

Naming conventions.

INFINITE = inf
INFINITE oft used for perpetually blocking operations.

UNTIL_THE_BATTLE_OF_ALFREDO = inf
UNTIL_THE_BATTLE_OF_ALFREDO is an alias for *INFINITE*.

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

Naming conventions.

AUTO_GENERATED = 'AUTO_GENERATED'
AUTO_GENERATED leaves it to the behaviour to generate a useful, informative name.

class py_trees.common.**ParallelPolicy**
Configurable policies for *Parallel* behaviours.

class SuccessOnAll(*synchronise=True*)
Return *SUCCESS* only when each and every child returns *SUCCESS*. If synchronisation is requested, any children that tick with *SUCCESS* will be skipped on subsequent ticks until the policy criteria is met, or one of the children returns status *FAILURE*.

class SuccessOnOne
Return *SUCCESS* so long as at least one child has *SUCCESS* and the remainder are *RUNNING*

class SuccessOnSelected(*children*, *synchronise=True*)
Return *SUCCESS* so long as each child in a specified list returns *SUCCESS*. If synchronisation is requested, any children that tick with *SUCCESS* will be skipped on subsequent ticks until the policy criteria is met, or one of the children returns status *FAILURE*.

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*

14.6 py_trees.composites

Composites are responsible for directing the path traced through the tree on a given tick (execution). They are the **factories** (Sequences and Parallels) and **decision makers** (Selectors) of a behaviour tree.

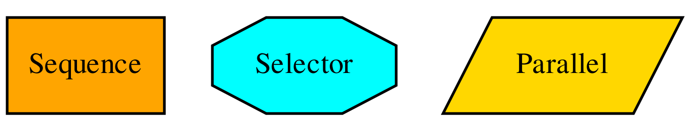


Fig. 7: PyTree Composites

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.

Most any desired functionality can be authored with a combination of these three composites. In fact, it is precisely this feature that makes behaviour trees attractive - it breaks down complex decision making logic to just three primitive elements. It is possible and often desirable to extend this set with custom composites of your own, but think carefully before you do - in almost every case, a combination of the existing composites will serve and as a result, you will merely compound the complexity inherent in your tree logic. This this makes it confoundingly difficult to design, introspect and debug. As an example, design sessions often revolve around a sketched graph on a whiteboard. When these graphs are composed of just five elements (Selectors, Sequences, Parallels, Decorators and Behaviours), it is very easy to understand the logic at a glance. Double the number of fundamental elements and you may as well be back at the terminal parsing code.

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

The basic operational modes of the three composites in this library are as follows:

- `Selector`: select a child to execute based on cascading priorities
- `Sequence`: execute children sequentially
- `Parallel`: execute children concurrently

This library does provide some flexibility in *how* each composite is implemented without breaking the fundamental nature of each (as described above). Selectors and Sequences can be configured with or without memory (resumes or resets if children are RUNNING) and the results of a parallel can be configured to wait upon all children completing, succeed on one, all or a subset thereof.

Tip: Follow the links in each composite's documentation to the relevant demo programs.

---

class `py_trees.composites.`**`Composite`**(*name=<Name.AUTO_GENERATED: 'AUTO_GENERATED'>, children=None*)

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

`__init__`(*name=<Name.AUTO_GENERATED: 'AUTO_GENERATED'>, children=None*)

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

`add_child`(*child*)

Adds a child.

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

Raises

- `TypeError` – if the child is not an instance of `Behaviour`
- `RuntimeError` – if the child already has a parent

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

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()

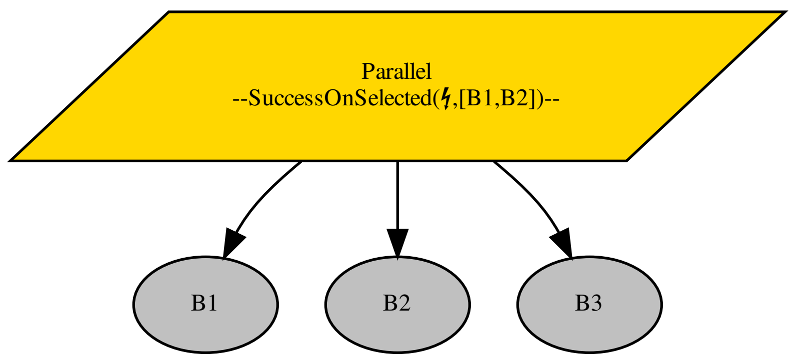
Recursive function to extract the last running node of the tree.

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

class py_trees.composites.**Parallel**(*name=<Name.AUTO_GENERATED: 'AUTO_GENERATED'>, policy=<py_trees.common.ParallelPolicy.SuccessOnAll object>, children=None*)

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 parallelism).

- Parallels will return *FAILURE* if any child returns *FAILURE*
- Parallels with policy *SuccessOnAll* only returns *SUCCESS* if **all** children return *SUCCESS*
- Parallels with policy *SuccessOnOne* return *SUCCESS* if **at least one** child returns *SUCCESS* and others are *RUNNING*
- Parallels with policy *SuccessOnSelected* only returns *SUCCESS* if a **specified subset** of children return *SUCCESS*

Policies *SuccessOnAll* and *SuccessOnSelected* may be configured to be *synchronised* in which case children that tick with *SUCCESS* will be skipped on subsequent ticks until the policy criteria is met, or one of the children returns status *FAILURE*.

Parallels with policy *SuccessOnSelected* will check in both the *setup()* and *tick()* methods to to verify the selected set of children is actually a subset of the children of this parallel.

See also:

- *Context Switching Demo*

__init__(*name=<Name.AUTO_GENERATED: 'AUTO_GENERATED'>, policy=<py_trees.common.ParallelPolicy.SuccessOnAll object>, children=None*)

Parameters

- **name** (`str`) – the composite behaviour name
- **policy** (*ParallelPolicy*) – policy to use for deciding success or otherwise
- **children** ([*Behaviour*]) – list of children to add

setup(***kwargs*)

Detect before ticking whether the policy configuration is invalid.

Parameters ****kwargs** (`dict`) – distribute arguments to this behaviour and in turn, all of it's children

Raises

- `RuntimeError` – if the parallel's policy configuration is invalid
- `Exception` – be ready to catch if any of the children raise an exception

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

For interrupts or any of the termination conditions, ensure that any running children are stopped.

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

Raises `RuntimeError` – if the policy configuration was invalid

validate_policy_configuration()

Policy configuration can be invalid if:

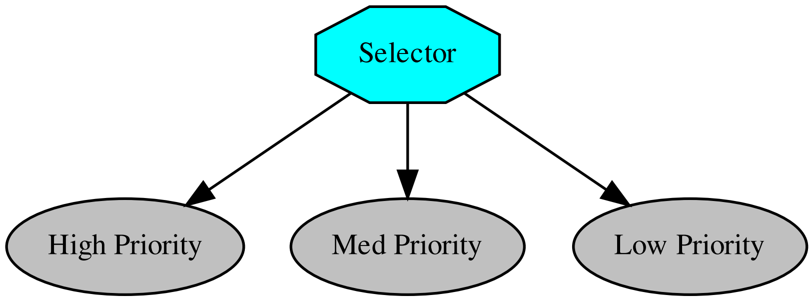
- Policy is SuccessOnSelected and no behaviours have been specified
- Policy is SuccessOnSelected and behaviours that are not children exist

Raises `RuntimeError` – if policy configuration was invalid

class py_trees.composites.**Selector**(*name='Selector', memory=False, children=None*)

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.

See also:

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

Parameters

- **name** (str) – the composite behaviour name
- **memory** (bool) – if *RUNNING* on the previous tick, resume with the *RUNNING* child
- **children** ([*Behaviour*]) – list of children to add

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

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.

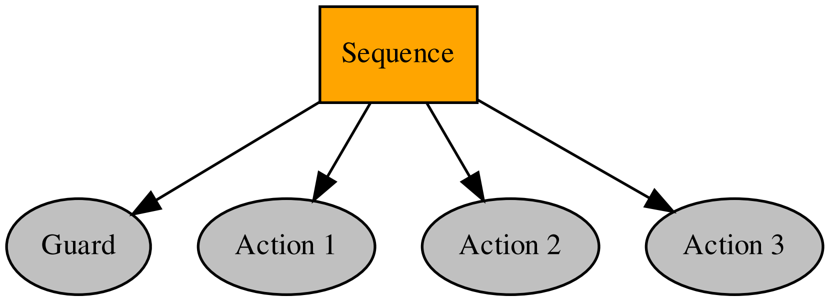
Parameters **new_status** (*Status*) – the composite is transitioning to this new status

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'*, *memory=True*, *children=None*)
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
- **memory** (`bool`) – if *RUNNING* on the previous tick, resume with the *RUNNING* child
- **children** (`Optional[List[Behaviour]]`) – list of children to add

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

tick()
Tick over the children.

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

14.7 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.**define_symbol_or_fallback**(*original*, *fallback*, *encoding='UTF-8'*)
Return the correct encoding according to the specified encoding. Used to make sure we get an appropriate symbol, even if the shell is merely ascii as is often the case on, e.g. Jenkins CI.

Parameters

- **original** (str) – the unicode string (usually just a character)
- **fallback** (str) – the fallback ascii string
- **encoding** (str, optional) – the encoding to check against.

Returns either original or fallback depending on whether exceptions were thrown.

Return type str

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

py_trees.console.**has_unicode**(*encoding='UTF-8'*)
Define whether the specified encoding has unicode symbols. Usually used to check if the stdout is capable or otherwise (e.g. Jenkins CI can often be configured with unicode disabled).

Parameters **encoding** (str, optional) – the encoding to check against.

Returns true if capable, false otherwise

Return type bool

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

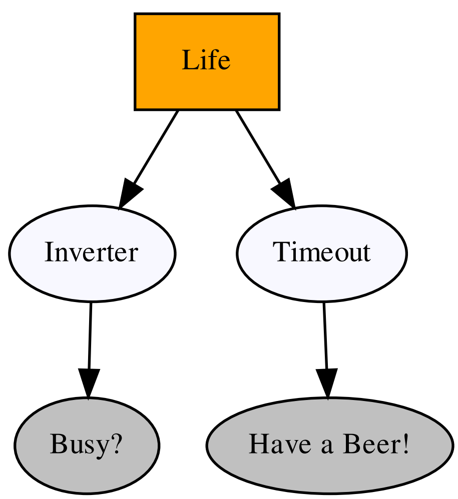
Return type int

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

14.8 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). That is, they provide a means for behaviours to wear different 'hats' and this combinatorially expands the capabilities of your behaviour library.

An example:




```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import py_trees.decorators
import py_trees.display

```

(continues on next page)

(continued from previous page)

```
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.EternalGuard*
- *py_trees.decorators.Inverter*
- *py_trees.decorators.OneShot*
- *py_trees.decorators.StatusToBlackboard*
- *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*

Decorators for Blocking Behaviours

It is worth making a note of the effect of decorators on behaviours that return *RUNNING* for some time before finally returning *SUCCESS* or *FAILURE* (blocking behaviours) since the results are often at first, surprising.

A decorator, such as *py_trees.decorators.RunningIsSuccess()* on a blocking behaviour will immediately terminate the underlying child and re-intialise on it's next tick. This is necessary to ensure the underlying child isn't left in a dangling state (i.e. *RUNNING*), but is often not what is being sought.

The typical use case being attempted is to convert the blocking behaviour into a non-blocking behaviour. If the underlying child has no state being modified in either the *initialise()* or *terminate()* methods (e.g. machinery is entirely launched at init or setup time), then conversion to a non-blocking representative of the original succeeds. Otherwise, another approach is needed. Usually this entails writing a non-blocking counterpart, or combination of behaviours to affect the non-blocking characteristics.

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.

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*

Return type *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

Parameters

- **child** (*Behaviour*) – the child to be decorated
- **name** – the decorator name

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

stop(*new_status*)
As with other composites, it checks if the child is running and stops it if that is the case.

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

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

class py_trees.decorators.**EternalGuard**(***, *child*, *condition*, *blackboard_keys=[]*, *name=<Name.AUTO_GENERATED: 'AUTO_GENERATED'>*)
Bases: *py_trees.decorators.Decorator*

A decorator that continually guards the execution of a subtree. If at any time the guard's condition check fails, then the child behaviour/subtree is invalidated.

Note: This decorator's behaviour is stronger than the *guard* typical of a conditional check at the beginning of a sequence of tasks as it continues to check on every tick whilst the task (or sequence of tasks) runs.

Parameters

- **child** (*Behaviour*) – the child behaviour or subtree
- **condition** (Any) – a functional check that determines execution or not of the subtree
- **blackboard_keys** (Union[List[str], Set[str]]) – provide read access for the conditional function to these keys

- **name** (Union[str, *Name*]) – the decorator name

Examples:

Simple conditional function returning True/False:

```
def check():
     return True

foo = py_trees.behaviours.Foo()
eternal_guard = py_trees.decorators.EternalGuard(
    name="Eternal Guard",
    condition=check,
    child=foo
)
```

Simple conditional function returning SUCCESS/FAILURE:

```
def check():
     return py_trees.common.Status.SUCCESS

foo = py_trees.behaviours.Foo()
eternal_guard = py_trees.decorators.EternalGuard(
    name="Eternal Guard",
    condition=check,
    child=foo
)
```

Conditional function that makes checks against data on the blackboard (the blackboard client with pre-configured access is provided by the EternalGuard instance):

```
def check(blackboard):
     return blackboard.velocity > 3.0

foo = py_trees.behaviours.Foo()
eternal_guard = py_trees.decorators.EternalGuard(
    name="Eternal Guard",
    condition=check,
    blackboard_keys={"velocity"},
    child=foo
)
```

See also:

py_trees.idioms.eternal_guard()

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

update()

The update method is only ever triggered in the child's post-tick, which implies that the condition has already been checked and passed (refer to the *tick()* method).

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*

Return type *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*

Return type *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.

update()
Flip *FAILURE* and *SUCCESS*

Returns the behaviour's new status *Status*

Return type *Status*

class py_trees.decorators.**OneShot**(*child*, *name=<Name.AUTO_GENERATED: 'AUTO_GENERATED'>*, *policy=<OneShotPolicy.ON_SUCCESSFUL_COMPLETION: [<Status.SUCCESS: 'SUCCESS'>]>*)
Bases: *py_trees.decorators.Decorator*

A decorator that implements the oneshot pattern.

This decorator ensures that the underlying child is ticked through to completion just once and while doing so, will return with the same status as it's child. Thereafter it will return with the final status of the underlying child.

Completion status is determined by the policy given on construction.

- With policy `ON_SUCCESSFUL_COMPLETION`, the oneshot will activate only when the underlying child returns *SUCCESS* (i.e. it permits retries).
- With policy `ON_COMPLETION`, the oneshot will activate when the child returns *SUCCESS* || *FAILURE*.

See also:

py_trees.idioms.oneshot()

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`

Return type `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`

Return type `Status`

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

Bases: `py_trees.decorators.Decorator`

Reflect the status of the decorator's child to the blackboard.

Parameters

- **child** (`Behaviour`) – the child behaviour or subtree
- **variable_name** (`str`) – name of the blackboard variable, may be nested, e.g. foo.status
- **name** (`Union[str, Name]`) – the decorator name

update()

Reflect the decorated child's status to the blackboard and return

Returns: the decorated child's 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`

Return type `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*

Return type *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.

initialise()

Reset the feedback message and finish time on behaviour entry.

update()

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

14.9 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_blackboard**(*key_filter=None*, *regex_filter=None*, *client_filter=None*, *keys_to_highlight=[]*, *display_only_key_metadata=False*, *indent=0*)

Graffiti your console with ascii art for your blackboard.

Parameters

- **key_filter** (Union[Set[str], List[str], None]) – filter on a set/list of blackboard keys
- **regex_filter** (Optional[str]) – filter on a python regex str
- **client_filter** (Union[Set[UUID], List[UUID], None]) – filter on a set/list of client uuids
- **keys_to_highlight** (List[str]) – list of keys to highlight
- **display_only_key_metadata** (bool) – read/write access, … instead of values
- **indent** (int) – the number of characters to indent the blackboard

Return type str

Returns a unicoded blackboard (i.e. in string form)

See also:

py_trees.display.unicode_blackboard()

Note: registered variables that have not yet been set are marked with a '-'

---

py_trees.display.**ascii_symbols = {'space': ' ', 'left_arrow': '<-', 'right_arrow': '->**

Symbols for a non-unicode, non-escape sequence capable console.

py_trees.display.**ascii_tree**(*root*, *show_only_visited=False*, *show_status=False*, *visited={}*, *previously_visited={}*, *indent=0*)

Graffiti your console with ascii art for your trees.

Parameters

- **root** (*Behaviour*) – the root of the tree, or subtree you want to show
- **show_only_visited** (bool) – show only visited behaviours
- **show_status** (bool) – always show status and feedback message (i.e. for every element, not just those visited)
- **visited** (*dict*) – dictionary of (uuid.UUID) and status (*Status*) pairs for behaviours visited on the current tick
- **previously_visited** (*dict*) – dictionary of behaviour id/status pairs from the previous tree tick
- **indent** (int) – the number of characters to indent the tree

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

Return type str

See also:

py_trees.display.xhtml_tree(), *py_trees.display.unicode_tree()*

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.unicode_tree(
            behaviour_tree.root,
            visited=snapshot_visitor.visited,
            previously_visited=snapshot_visitor.visited
        )
    )

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.**dot_tree**(*root*, *visibility_level=<VisibilityLevel.DETAIL: 1>*, *collapse_decorators=False*, *with_blackboard_variables=False*, *with_qualified_names=False*)

Paint your tree on a pydot graph.

See also:

render_dot_tree().

Parameters

- **root** (*Behaviour*) – the root of a tree, or subtree
- **visibility_level** (*optional*) – collapse subtrees at or under this level
- **collapse_decorators** (*optional*) – only show the decorator (not the child), defaults to False
- **with_blackboard_variables** (*optional*) – add nodes for the blackboard variables
- **with_qualified_names** (*optional*) – print the class information for each behaviour in each node, defaults to False

Returns graph

Return type pydot.Dot

Examples

```
# convert the pydot graph to a string object
print("{}".format(py_trees.display.dot_graph(root).to_string()))
```

py_trees.display.**render_dot_tree**(*root*, *visibility_level=<VisibilityLevel.DETAIL: 1>*, *collapse_decorators=False*, *name=None*, *target_directory='/home/docs/checkouts/readthedocs.org/user_builds/py-trees/checkouts/release-2.1.x/doc'*, *with_blackboard_variables=False*, *with_qualified_names=False*)

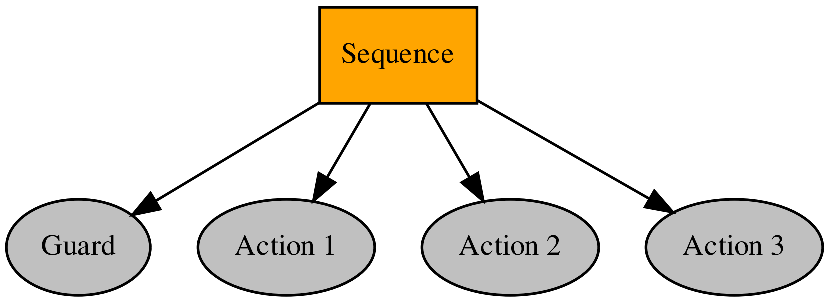
Render the dot tree to .dot, .svg, .png. files in the current working directory. These will be named with the root behaviour name.

Parameters

- **root** (*Behaviour*) – the root of a tree, or subtree
- **visibility_level** (*VisibilityLevel*) – collapse subtrees at or under this level
- **collapse_decorators** (bool) – only show the decorator (not the child)
- **name** (Optional[str]) – name to use for the created files (defaults to the root behaviour name)
- **target_directory** (str) – default is to use the current working directory, set this to redirect elsewhere
- **with_blackboard_variables** (bool) – add nodes for the blackboard variables
- **with_qualified_names** (bool) – print the class names of each behaviour in the dot node

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.`**`unicode_blackboard`**(*key_filter=None*, *regex_filter=None*, *client_filter=None*, *keys_to_highlight=[]*, *display_only_key_metadata=False*, *indent=0*)

Graffiti your console with unicode art for your blackboard.

Parameters

- **key_filter** (`Union`[`Set`[`str`], `List`[`str`], `None`]) – filter on a set/list of blackboard keys
- **regex_filter** (`Optional`[`str`]) – filter on a python regex str
- **client_filter** (`Union`[`Set`[`UUID`], `List`[`UUID`], `None`]) – filter on a set/list of client uuids
- **keys_to_highlight** (`List`[`str`]) – list of keys to highlight
- **display_only_key_metadata** (`bool`) – read/write access, . . . instead of values
- **indent** (`int`) – the number of characters to indent the blackboard

Return type `str`

Returns a unicoded blackboard (i.e. in string form)

See also:

py_trees.display.ascii_blackboard()

Note: registered variables that have not yet been set are marked with a '-'

py_trees.display.**unicode_blackboard_activity_stream**(*activity_stream=None*, *indent=0*, *show_title=True*)

Pretty print the blackboard stream to console.

Parameters

- **activity_stream** (Optional[List[*ActivityItem*]]) – the log of activity, if None, get the entire activity stream
- **indent** (int) – the number of characters to indent the blackboard
- **show_title** (bool) – include the title in the output

py_trees.display.**unicode_symbols = {'space': ' ', 'left_arrow': '←', 'right_arrow':**

Symbols for a unicode, escape sequence capable console.

py_trees.display.**unicode_tree**(*root*, *show_only_visited=False*, *show_status=False*, *visited={}*, *previously_visited={}*, *indent=0*)

Graffiti your console with unicode art for your trees.

Parameters

- **root** (*Behaviour*) – the root of the tree, or subtree you want to show
- **show_only_visited** (bool) – show only visited behaviours
- **show_status** (bool) – always show status and feedback message (i.e. for every element, not just those visited)
- **visited** (*dict*) – dictionary of (uuid.UUID) and status (*Status*) pairs for behaviours visited on the current tick
- **previously_visited** (*dict*) – dictionary of behaviour id/status pairs from the previous tree tick
- **indent** (int) – the number of characters to indent the tree

Returns a unicode tree (i.e. in string form)

Return type str

See also:

py_trees.display.ascii_tree(), *py_trees.display.xhtml_tree()*

py_trees.display.**xhtml_symbols = {'space': '<text> </text>', 'left_arrow': '<text>&#**

Symbols for embedding in html.

py_trees.display.**xhtml_tree**(*root*, *show_only_visited=False*, *show_status=False*, *visited={}*, *previously_visited={}*, *indent=0*)

Paint your tree on an xhtml snippet.

Parameters

- **root** (*Behaviour*) – the root of the tree, or subtree you want to show
- **show_only_visited** (bool) – show only visited behaviours
- **show_status** (bool) – always show status and feedback message (i.e. for every element, not just those visited)

- **visited** (*dict*) – dictionary of (uuid.UUID) and status (*Status*) pairs for behaviours visited on the current tick
- **previously_visited** (*dict*) – dictionary of behaviour id/status pairs from the previous tree tick
- **indent** (int) – the number of characters to indent the tree

Returns an ascii tree (i.e. as a xhtml snippet)

Return type str

See also:

py_trees.display.ascii_tree(), *py_trees.display.unicode_tree()*

Examples:

```
import py_trees
a = py_trees.behaviours.Success()
b = py_trees.behaviours.Success()
c = c = py_trees.composites.Sequence(children=[a, b])
c.tick_once()

f = open('testies.html', 'w')
f.write('<html><head><title>Foo</title><body>')
f.write(py_trees.display.xhtml_tree(c, show_status=True))
f.write("</body></html>")
```

14.10 py_trees.idioms

A library of subtree creators that build complex patterns of behaviours representing common behaviour tree idioms.

py_trees.idioms.**either_or**(*conditions*, *subtrees*, *name='Either Or'*, *namespace=None*)

Often you need a kind of selector that doesn't implement prioritisations, i.e. you would like different paths to be selected on a first-come, first-served basis.

```
task_one = py_trees.behaviours.TickCounter(name="Subtree 1", duration=2)
task_two = py_trees.behaviours.TickCounter(name="Subtree 2", duration=2)
either_or = py_trees.idioms.either_or(
    name="EitherOr",
    conditions=[
        py_trees.common.ComparisonExpression("joystick_one", "enabled", operator.
↪eq),
        py_trees.common.ComparisonExpression("joystick_two", "enabled", operator.
↪eq),
    ],
    subtrees=[task_one, task_two],
    namespace="either_or",
)
```

Up front is an XOR conditional check which locks in the result on the blackboard under the specified namespace. Locking the result in permits the conditional variables to vary in future ticks without interrupting the execution of the chosen subtree (an example of a conditional variable may be one that has registered joystick button presses).

Once the result is locked in, the relevant subtree is activated beneath the selector. The children of the selector are, from left to right, not in any order of priority since the previous xor choice has been locked in and isn't revisited until the subtree executes to completion. Only one may be active and it cannot be interrupted by the others.

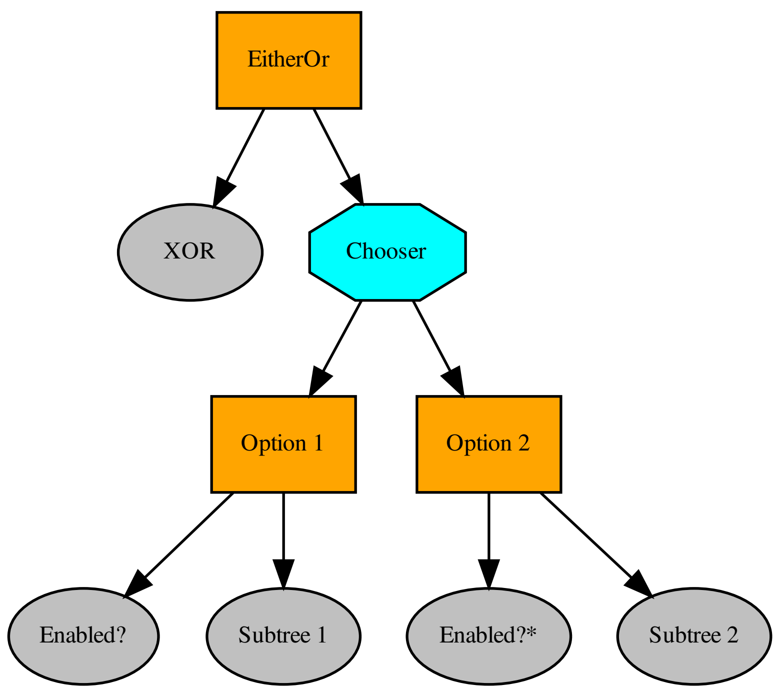


Fig. 8: Idiom - Either Or

The only means of interrupting the execution is via a higher priority in the tree that this idiom is embedded in.

Parameters

- **conditions** (List[ComparisonExpression]) – list of triggers that ultimately select the subtree to enable
- **subtrees** (List[*Behaviour*]) – list of subtrees to tick from in the either_or operation
- **name** – the name to use for this idiom's root behaviour
- **preemptible** – whether the subtrees may preempt (interrupt) each other
- **namespace** (Optional[str]) – this idiom's private variables will be put behind this namespace

Raises ValueError if the number of conditions does not match the number of subtrees

If no namespace is provided, a unique one is derived from the idiom's name.

See also:

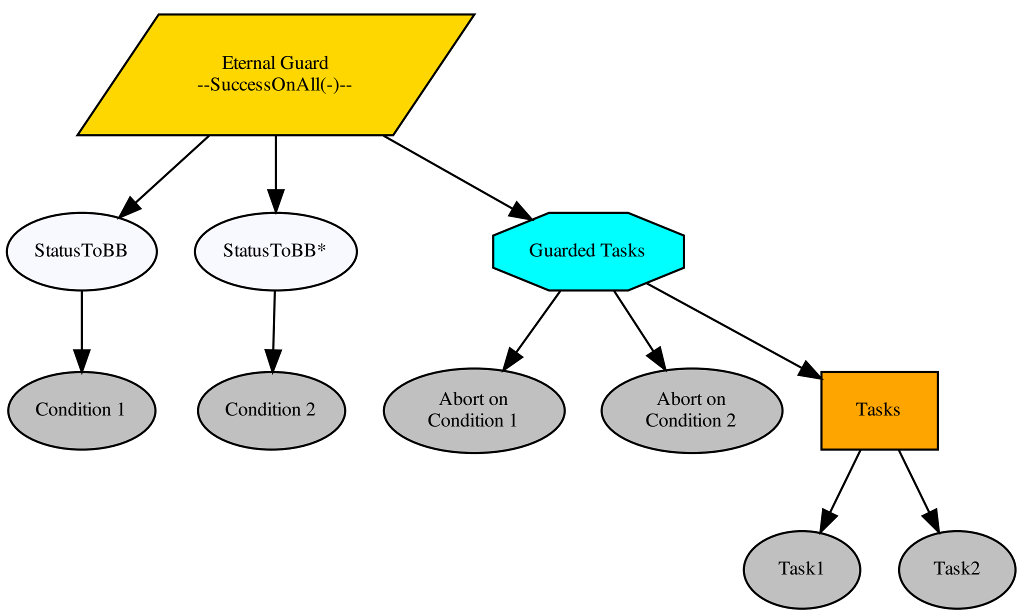
py-trees-demo-either-or

Todo: a version for which other subtrees can preempt (in an unprioritised manner) the active branch

Return type *Behaviour*

py_trees.idioms.**eternal_guard**(*subtree*, *name='Eternal Guard'*, *conditions=[]*, *blackboard_namespace=None*)

The eternal guard idiom implements a stronger *guard* than the typical check at the beginning of a sequence of tasks. Here they guard continuously while the task sequence is being executed. While executing, if any of the guards should update with status FAILURE, then the task sequence is terminated.



Parameters

- **subtree** (*Behaviour*) – behaviour(s) that actually do the work
- **name** (str) – the name to use on the root behaviour of the idiom subtree
- **conditions** (List[*Behaviour*]) – behaviours on which tasks are conditional
- **blackboard_namespace** (Optional[str]) – applied to condition variable results stored on the blackboard (default: derived from the idiom name)

Return type *Behaviour*

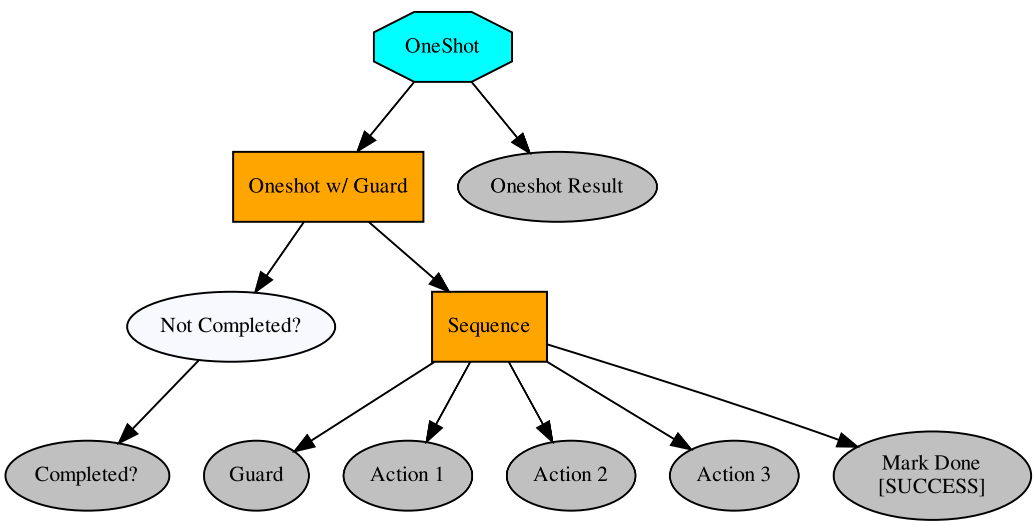
Returns the root of the idiom subtree

See also:

py_trees.decorators.EternalGuard

py_trees.idioms.**oneshot**(*behaviour*, *name='Oneshot'*, *variable_name='oneshot'*, *policy=<OneShotPolicy.ON_SUCCESSFUL_COMPLETION: [<Status.SUCCESS: 'SUCCESS'>]>*)

Ensure that a particular pattern is executed through to completion just once. Thereafter it will just rebound with the completion status.



Note: Set the policy to configure the oneshot to keep trying if failing, or to abort further attempts regardless of whether it finished with status FAILURE.

Parameters

- **behaviour** (*Behaviour*) – single behaviour or composited subtree to oneshot
- **name** (str) – the name to use for the oneshot root (selector)
- **variable_name** (str) – name for the variable used on the blackboard, may be nested
- **policy** (OneShotPolicy) – execute just once regardless of success or failure, or keep trying if failing

Returns the root of the oneshot subtree

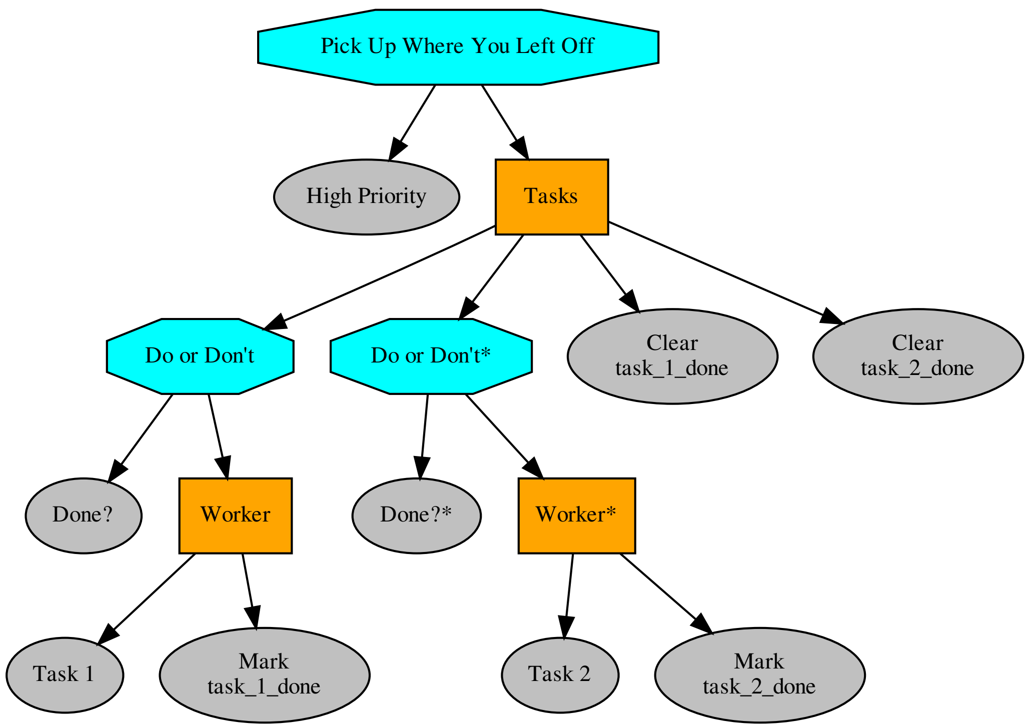
Return type *Behaviour*

See also:

py_trees.decorators.OneShot

`py_trees.idioms.`**`pick_up_where_you_left_off`**(*name='Pickup Where You Left Off Idiom'*, *tasks=[]*)

Rudely interrupted while enjoying a sandwich, a caveman (just because they wore loincloths does not mean they were not civilised), picks up his club and fends off the sabre-tooth tiger invading his sanctum as if he were swatting away a gnat. Task accomplished, he returns to the joys of munching through the layers of his sandwich.



Note: There are alternative ways to accomplish this idiom with their pros and cons.

a) The tasks in the sequence could be replaced by a factory behaviour that dynamically checks the state of play and spins up the tasks required each time the task sequence is first entered and invalidates/deletes them when it is either finished or invalidated. That has the advantage of not requiring much of the blackboard machinery here, but disadvantage in not making visible the task sequence itself at all times (i.e. burying details under the hood).

b) A new composite which retains the index between initialisations can also achieve the same pattern with fewer blackboard shenanigans, but suffers from an increased logical complexity cost for your trees (each new composite increases decision making complexity (O(n!)).

Parameters

- **name** (str) – the name to use for the task sequence behaviour
- **tasks** ([*Behaviour*]) – lists of tasks to be sequentially performed

Returns root of the generated subtree

Return type *Behaviour*

14.11 py_trees.meta

Meta methods to create behaviours without needing to create the behaviours themselves.

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

14.12 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)

Raises TypeError – if the provided duration is not a real number

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

Tip: Use the *RunningIsFailure()* 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*)

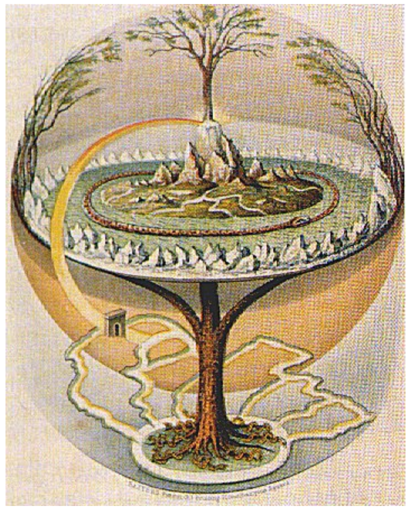
Clear the expected finishing time.

update()

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

14.13 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** – number of times the tree has been ticked.
- **root** – root node of the tree
- ***visitors*** – entities that visit traversed parts of the tree when it ticks
- **pre_tick_handlers** – functions that run before the entire tree is ticked
- **post_tick_handlers** – functions that run after the entire tree is ticked

Raises TypeError – if root variable is not an instance of *Behaviour*

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

add_visitor(*visitor*)
Trees can run multiple visitors on each behaviour as they tick through a tree.

Parameters visitor (*VisitorBase*) – sub-classed instance of a visitor

See also:

DebugVisitor, *SnapshotVisitor*, *DisplaySnapshotVisitor*

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 sucess or failure (parent not found) of the operation

Return type bool

Raises TypeError – 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 RuntimeError – 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 sucess or failure of the operation

Return type bool

setup (*timeout=<Duration.INFINITE: inf>*, *visitor=None*, ***kwargs*)
Crawls across the tree calling *setup()* on each behaviour.

Visitors can optionally be provided to provide a node-by-node analysis on the result of each node's *setup()* before the next node's *setup()* is called. This is useful on trees with relatively long setup times to progressively report out on the current status of the operation.

Parameters

- **timeout** (float) – time (s) to wait (use common.Duration.INFINITE to block indefinitely)
- **visitor** (*VisitorBase*) – runnable entities on each node after it's setup
- ****kwargs** (dict) – distribute arguments to this behaviour and in turn, all of it's children

Raises

- Exception – be ready to catch if any of the behaviours raise an exception
- RuntimeError – in case setup() times out

shutdown ()
Crawls across the tree calling *shutdown()* on each behaviour.

Raises Exception – be ready to catch if any of the behaviours raise an exception

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 (*period_ms*, *number_of_iterations=-1*, *pre_tick_handler=None*, *post_tick_handler=None*)

Tick continuously with period as specified. Depending on the implementation, the period may be more or less accurate and may drift in some cases (the default implementation here merely assumes zero time in tick and sleeps for this duration of time and consequently, will drift).

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

- **period_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()`

py_trees.trees.**setup** (*root*, *timeout=<Duration.INFINITE: inf>*, *visitor=None*, ***kwargs*)

Crawls across a (sub)tree of behaviours calling `setup()` on each behaviour.

Visitors can optionally be provided to provide a node-by-node analysis on the result of each node's `setup()` before the next node's `setup()` is called. This is useful on trees with relatively long setup times to progressively report out on the current status of the operation.

Parameters

- **root** (`Behaviour`) – unmanaged (sub)tree root behaviour
- **timeout** (`Union[float, Duration]`) – time (s) to wait (use common.Duration.INFINITE to block indefinitely)
- **visitor** (`Optional[VisitorBase]`) – runnable entities on each node after it's setup
- ****kwargs** – dictionary of arguments to distribute to all behaviours in the (sub) tree

Raises

- `Exception` – be ready to catch if any of the behaviours raise an exception
- `RuntimeError` – in case setup() times out

14.14 py_trees.utilities

Assorted utility functions.

class py_trees.utilities.**Process** (**args*, ***kwargs*)

Bases: `multiprocessing.context.Process`

run()
Method to be run in sub-process; can be overridden in sub-class

py_trees.utilities.**get_fully_qualified_name**(*instance*)
Get at the fully qualified name of an object, e.g. an instance of a *Sequence* becomes 'py_trees.composites.Sequence'.

Parameters **instance** (object) – an instance of any class

Returns the fully qualified name

Return type str

py_trees.utilities.**get_valid_filename**(*s*)
Return the given string converted to a string that can be used for a clean filename (without extension). Remove leading and trailing spaces; convert other spaces and newlines to underscores; and remove anything that is not an alphanumeric, dash, underscore, or dot.

```
>>> utilities.get_valid_filename("john's portrait in 2004.jpg")
'johns_portrait_in_2004.jpg'
```

Parameters **program** (str) – string to convert to a valid filename

Returns a representation of the specified string as a valid filename

Return type str

py_trees.utilities.**is_primitive**(*incoming*)
Check if an incoming argument is a primitive type with no esoteric accessors (e.g. class attributes or container [] accessors.

Parameters **incoming** (Any) – the instance to check

Return type bool

Returns True or false, depending on the check against the reserved primitives

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.

```
@static_variables(counter=0)
def foo():
    foo.counter += 1
    print("Counter: {}".format(foo.counter))
```

py_trees.utilities.**truncate**(*original*, *length*)
Provide an elided version of the string for which the last three characters are dots if the original string does not fit within the specified length.

Parameters

- **original** (str) – string to elide
- **length** (int) – constrain the elided string to this

Return type str

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

14.15 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.**DisplaySnapshotVisitor**(*display_only_visited_behaviours=False, display_blackboard=False, display_activity_stream=False*)

Bases: *py_trees.visitors.SnapshotVisitor*

Visit the tree, capturing the visited path, it's changes since the last tick and additionally print the snapshot to console.

Parameters

- **display_blackboard** (bool) – print to the console the relevant part of the blackboard associated with behaviours on the visited path
- **display_activity_stream** (bool) – print to the console a log of the activity on the blackboard over the last tick

finalise()

Override this method if any work needs to be performed after ticks (i.e. showing data).

initialise()

Switch running to previously running and then reset all other variables. This should 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.**SnapshotVisitor**

Bases: *py_trees.visitors.VisitorBase*

Visits the ticked part of a tree, checking off the status against the set of status results recorded in the previous tick. If there has been a change, it flags it. This is useful for determining when to trigger, e.g. logging.

Variables

- **changed** (*Bool*) – flagged if there is a difference in the visited path or *Status* of any behaviour on the path
- **visited** (*dict*) – dictionary of behaviour id (uuid.UUID) and status (*Status*) pairs from the current tick
- **previously_visited** (*dict*) – dictionary of behaviour id (uuid.UUID) and status (*Status*) pairs from the previous tick
- **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
- **visited_blackboard_ids** (*typing.Set[uuid.UUID]*) – blackboard client id's on the visited path
- **visited_blackboard_keys** (*typing.Set[str]*) – blackboard variable keys on the visited path

See also:

The *py-trees-demo-logging* program demonstrates use of this visitor to trigger logging of a tree serialisation.

initialise()

Switch running to previously running and then reset all other variables. This should 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

finalise()

Override this method if any work needs to be performed after ticks (i.e. showing data).

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 15

Release Notes

15.1 Forthcoming

- …

15.2 2.1.6 (2021-05-31)

- [tests] mypy conformance, #327
- [composites] show ghost states for sequence children, #330 (reverts behaviour introduced in #325)

15.3 2.1.5 (2021-05-09)

- [composites] sequences w/o memory, #325
- [composites] selectors with memory, #324
- [display] unicode trees are unicode, not ascii #324

15.4 2.1.4 (2021-02-24)

- [behaviours] added BlackboardToStatus, #320
- [idioms] bugfix lost append in blackboard names for either_or, #319
- [docs] contributing guidelines added, #315

15.5 2.1.3 (2020-12-13)

- [blackboard] convenience *absolute_name()* api for blackboard clients, #306

15.6 2.1.2 (2020-11-05)

- [sequences] bugfix current child setting whilst moving through children, #304

15.7 2.1.1 (2020-08-11)

- [behaviours] generators for variable values in SetBlackboardVariable, #300

15.8 2.1.0 (2020-07-20)

- [behaviours] CheckBlackboardVariableValue, WaitForBlackboardVariableValue use comparison expressins, #296
- [idioms] eternal_guard uses blackboard_names (previously blackboard_name_prefix), #295
- [composites] Chooser deprecated, #293
- [composites] chain-able add_children for all composites, #290

15.9 2.0.16 (2020-08-11)

- [behaviours] generators for variable values in SetBlackboardVariable, #300

15.10 2.0.15 (2020-05-14)

- [composites] fix current_child problems on dynamic child insertion/removal, #289

15.11 2.0.14 (2020-04-26)

- [blackboard] bugfix metadata not created on static set, #286

15.12 2.0.13 (2020-03-24)

- [composites] parallel bugfix to invalidate correctly, #285

15.13 2.0.12 (2020-03-10)

- [idioms] the either_or pattern, designed to replace the Chooser, #283
- [behaviours] TickCounter, a timer based on tree ticks, #283
- [behaviours] CheckBlackboardVariableValues, logical checks across multiple values, #283
- [common] ComparisonExpression, a more concise way of storing checkers, #283
- [composites] protect against adding a child to multiple parents, #281

15.14 2.0.11 (2020-03-01)

- [display] completely remove blackboard variables from the dot layout, #280

15.15 2.0.10 (2020-02-27)

- [trees] setup timeout error with last behaviour name included in the error message, #279
- [blackboard] rooted variables in namespaced clients working as designed, fixed docs, #277

15.16 2.0.9 (2020-02-14)

- [demos] display modes demonstrating usage of various options, #275
- [display] enforce left to right ordering of children in dot graphs

15.17 2.0.8 (2020-02-01)

- [display] bugfix and use elipsis on ascii visited only display modes, #273

15.18 2.0.7 (2020-01-24)

- [display] option for only visited behaviours in text tree snapshot displays, #272

15.19 2.0.5 (2019-12-25)

- [display] enum agnostic display for local and remote activity view displays

15.20 2.0.4 (2019-11-25)

- [display] optional show_title in unicode_blackboard_activity_stream

15.21 2.0.3 (2019-11-24)

- [trees] revert to using user signals if available to avoid shenanigans with SIGINT, #264
- [trees] play nicely, reset signal handlers after setup, #262
- [visitors] bugfix the snapshot visitor to look for exclusive write keys as well

15.22 2.0.1 (2019-11-19)

- [blackboard] static methods have a namespace too (root), use absolute names, #261
- [blackboard] do not register keys on the client when xclusive write aborts the process, #261

15.23 2.0.x (2019-11-15) - Blackboards v2!

The *2.0.x* release wraps up the experimental blackboard improvements being rolled out in *1.3.x* and *1.4.x*. At this point, the changes to the blackboard framework are so extensive it makes sense to release it with a major version bump and to consider the *1.2.x* release as the official goto release for the *1.x.y* series.

New Features

- [blackboard] exclusive write access, #260
- [blackboard] key remappings, #259
- [blackboard] formalise namespaces with separators, #256
- [blackboard] distinguish primitives vs nested for refined read activity detection, #255

See the 1.3.x and 1.4.x changelog notes for additional details.

15.24 1.4.x (2019-11-07)

Breaking API

- [blackboard] fixed read/write ambiguity, now use `py_trees.common.Access`, #250

```
# Previously
self.blackboard.register_key(key="foo", write=True)
# Now
self.blackboard.register_key(key="foo", access=py_trees.common.Access.WRITE)
```

- [blackboard] drop `SubBlackboard`, it has problems, #249

New Features

- [blackboard] namespaced blackboard clients, #250

```
# Previously, a single blackboard client exists per behaviour
# Now, no blackboard client on construction, instead attach on demand:
self.blackboard = self.attach_blackboard_client(name="Foo")
self.parameters = self.attach_blackboard_client(
    name="FooParams",
    namespace="parameters_foo_"
```

(continues on next page)

(continued from previous page)

```
)
self.state = self.attach_blackboard_client(
    name="FooState",
    namespace="state_foo_"
)
# create a local key 'speed' that maps to 'state_foo_speed' on the blackboard
self.state.register_key(key="speed", access=py_trees.common.Access.WRITE)
self.state.speed = 30.0
```

- [blackboard] required keys and batch verification method, #254

```
self.blackboard = self.attach_blackboard_client(name="Foo")
self.blackboard.register_key(name="foo", access=py_trees.common.Access.READ,␣
↪required=True)
# ...
self.verify_required_keys_exist()  # KeyError if any required keys do not yet exist␣
↪on the blackboard
```

- [visitors] `SnapshotVisitor` tracking blackboards on the visited path, #250

```
# Previously tangled in DisplaySnapshotVisitor:
display_snapshot_visitor.visited.keys()  # blackboard client uuid's (also behaviour␣
↪uuid's), typing.Set[uuid.UUID]
display_snapshot_visitor.visited_keys  # blackboard keys, typing.Set[str]
# Now in SnapshotVisitor:
snapshot_visitor.visited_blackboard_client_ids  # typing.Set[uuid.UUID]
snapshot_visitor.visited_blackboard_keys  # typing.Set[str]
```

15.25 1.3.3 (2019-10-15)

- [blackboard] client `Blackboard.unregister_key()` method

15.26 1.3.2 (2019-10-15)

- [blackboard] global `Blackboard.clear()` method

15.27 1.3.1 (2019-10-15)

- [blackboard] don't do any copying, just pass handles around, #239
- [blackboard] client `exists()` method, #238
- [blackboard] global `Blackboard.set()` method
- [blackboard] client `Blackboard.unset()` method, #239

15.28 1.3.x (2019-10-03)

Breaking API

- [decorators] updated `EternalGuard` to accommodate new blackboard variable tracking mechanisms
- [behaviours] blackboard behaviours decoupled - `CheckBlackboardVariableExists`, `WaitForBlackboardVariable`
- [behaviours] blackboard behaviours decoupled - `CheckBlackboardVariableValue`, `WaitForBlackboardVariableValue`
- [behaviours] blackboard behaviours dropped use of the largely redundant `ClearingPolicy`
- [visitors] collapsed `SnapshotVisitor` and `WindsOfChangeVisitor` functionality, #228

New Features

- [blackboard] read/write access configuration for clients on blackboard keys
- [blackboard] log the activity on the blackboard
- [display] dot graphs now have an option to display blackboard variables
- [display] unicode to console the entire blackboard key-value store
- [display] unicode to console the blackboard activity stream
- [visitors] new `DisplaySnapshotVisitor` to simplify collection/printing the tree to console, #228

Bugfixes

- [infra] only require test html reports on circle ci builds (saves a dependency requirement), #229

15.29 1.2.2 (2019-08-06)

- [trees] standalone `setup()` method with timer for use on unmanaged trees, #198
- [examples] fix api in `skeleton_tree.py`, #199

15.30 1.2.1 (2019-05-21)

- [decorators] `StatusToBlackboard` reflects the status of it's child to the blackboard, #195
- [decorators] `EternalGuard` decorator that continuously guards a subtree (c.f. Unreal conditions), #195
- [idioms] `eternal_guard` counterpart to the decorator whose conditions are behaviours, #195

15.31 1.2.x (2019-04-28)

Breaking API

- [trees] removes the curious looking and unused `destroy()` method, #193
- [display] `ascii_tree` -> `ascii_tree/unicode_tree()`, no longer subverts the choice depending on your stdout, #192
- [display] `dot_graph` -> `dot_tree` for consistency with the text tree methods, #192

New Features

- [behaviour] `shutdown()` method to compliment `setup()`, #193
- [decorators] `StatusToBlackboard` reflects the status of it's child to the blackboard, #195

- [decorators] EternalGuard decorator that continuously guards a subtree (c.f. Unreal conditions), #195
- [display] xhtml_tree provides an xhtml compatible equivalent to the ascii_tree representation, #192
- [idioms] eternal_guard counterpart to the decorator whose conditions are behaviours, #195
- [trees] walks the tree calling shutdown() on each node in it's own shutdown() method, #193
- [visitors] get a finalise() method called immediately prior to post tick handlers, #191

15.32 1.1.0 (2019-03-19)

Breaking API

- [display] print_ascii_tree -> ascii_tree, #178.
- [display] generate_pydot_graph -> dot_graph, #178.
- [trees] tick_tock(sleep_ms, ..) -> tick_tock(period_ms, . . .), #182.

New Features

- [trees] add missing add_visitor() method
- [trees] flexible setup() for children via kwargs
- [trees] convenience method for ascii tree debugging
- [display] highlight the tip in ascii tree snapshots

Bugfixes

- [trees] threaded timers for setup (avoids multiprocessing problems)
- [behaviour|composites] bugfix tip behaviour, add tests
- [display] correct first indent when non-zero in ascii_tree
- [display] apply same formatting to root as children in ascii_tree

15.33 1.0.7 (2019-xx-yy)

- [display] optional arguments for generate_pydot_graph

15.34 1.0.6 (2019-03-06)

- [decorators] fix missing root feedback message in ascii graphs

15.35 1.0.5 (2019-02-28)

- [decorators] fix timeout bug that doesn't respect a child's last tick

15.36 1.0.4 (2019-02-26)

- [display] drop spline curves, it's buggy with graphviz 2.38

15.37 1.0.3 (2019-02-13)

- [visitors] winds of change visitor and logging demo

15.38 1.0.2 (2019-02-13)

- [console] fallbacks for unicode chars when (UTF-8) encoding cannot support them

15.39 1.0.1 (2018-02-12)

- [trees] don't use multiprocess on setup if not using timeouts

15.40 1.0.0 (2019-01-18)

Breaking API

- [behaviour] setup() no longer returns a boolean, catch exceptions instead, #143.
- [behaviour] setup() no longer takes timeouts, responsibility moved to BehaviourTree, #148.
- [decorators] new-style decorators found in py_trees.decorators
- [decorators] new-style decorators stop their running child on completion (SUCCESS||FAILURE)
- [decorators] old-style decorators in py_trees.meta deprecated

New Features

- [blackboard] added a method for clearing the entire blackboard (useful for tests)
- [composites] raise TypeError when children's setup methods don't return a bool (common mistake)
- [composites] new parallel policies, SuccessOnAll, SuccessOnSelected
- [decorators] oneshot policies for activating on completion or *successful* completion only
- [meta] behaviours from functions can now automagically generate names

15.41 0.8.x (2018-10-18)

Breaking API

- Lower level namespace types no longer exist (PR117), e.g. `py_trees.Status` -> `py_trees.common.Status`
- Python2 support dropped

New Features

- [idioms] 'Pick Up Where You Left Off'
- [idioms] 'OneShot'

15.42 0.8.0 (2018-10-18)

- [infra] shortcuts to types in __init__.py removed (PR117)
- [bugfix] python3 rosdeps
- [idioms] pick_up_where_you_left_off added

15.43 0.7.5 (2018-10-10)

- [idioms] oneshot added
- [bugfix] properly set/reset parents when replacing/removing children in composites

15.44 0.7.0 (2018-09-27)

- [announce] python3 only support from this point forward
- [announce] now compatible for ros2 projects

15.45 0.6.5 (2018-09-19)

- [bugfix] pick up missing feedback messages in inverters
- [bugfix] eliminate costly/spammy blackboard variable check feedback message

15.46 0.6.4 (2018-09-19)

- [bugfix] replace awkward newlines with spaces in ascii trees

15.47 0.6.3 (2018-09-04)

- [bugfix] don't send the parellel's status to running children, invalidate them instead

15.48 0.6.2 (2018-08-31)

- [bugfix] oneshot now reacts to priority interrupts correctly

15.49 0.6.1 (2018-08-20)

- [bugfix] oneshot no longer permanently modifies the original class

15.50 0.6.0 (2018-05-15)

- [infra] python 2/3 compatibility

15.51 0.5.10 (2017-06-17)

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

15.52 0.5.9 (2017-03-25)

- [docs] bugfix image links and rewrite the motivation

15.53 0.5.8 (2017-03-19)

- [infra] setup.py tests_require, not test_require

15.54 0.5.7 (2017-03-01)

- [infra] update maintainer email

15.55 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

15.56 0.5.4 (2017-02-22)

- [infra] handle pypi/catkin conflicts with install_requires

15.57 0.5.2 (2017-02-22)

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

15.58 0.5.1 (2017-02-21)

- [infra] pypi package enabled

15.59 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

15.60 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

15.61 0.3.0 (2016-08-25)

- `failure_is_running decorator` (meta).

15.62 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

15.63 0.1.2 (2015-11-16)

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

15.64 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 16

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index

Symbols

A

B

C

D

E

F

G

O

P

R

S

T

U

V

W

X