asyncframes Documentation

Release 2.2.0

Sebastian Klaassen

Feb 18, 2019

Contents

Version	asyncframes v1	asyncframes v2
Docs		
Download		
Source		
Status		
License		

---

asyncframes is a coroutine library for Python and a reference implementation of the *Frame Hierarchy Programming Model* (FHM). The goal of FHM is to help programmers design clean and scalable parallel programs. The main features of *asyncframes* are:

- Hierarchical code design
- Inherent and scalable parallelism
- Architecture independence
- Extensibility through frame classes (a class whose lifetime is bound to the execution of a frame)

CHAPTER 1

Introduction

In the *Frame Hierarchy Programming Model* (FHM) a program is represented as a dynamic tree of *frames*. A frame is a suspendable function (a coroutine) with an object oriented context (the frame class) that only exists until the function returns. Frames can be used to represent both temporal processes (using the coroutine) and physical or conceptual objects (using the frame class).

Each FHM program has exactly one root frame. The root frame can recursively spawn child frames. Each child frame runs in parallel unless it's awaiting another frame or an awaitable event. Frames of type `Frame` run on a single thread. They use cooperative multitasking to simulate parallelism. Frames of type `PFrame` run on any of the threads available in the event loop's thread pool. `Frame` and `PFrame` are frame classes. They can be sub-classed to create specialized frame classes with encapsulated data.

1.1 Programming with asyncframes

1.1.1 Frame Hierarchy Model vs. Object Oriented Programming

The most common form of Object Oriented Programming (OOP) is class-oriented programming. In this form programs are designed using **classes** and **objects**. A **class** defines the structure of a conceptual entity. After a class is defined, the programmer can create one or more **objects** of that entity. These objects are also known as instances of the class.

The **state** and **behavior** of objects are defined within the class by creating variables and methods respectively. Different objects of the same class can contain different data (i.e. values), but their state (i.e. variables) and behavior (i.e. methods) are the same. Dynamic languages, like Python, allow manipulation of state and behavior at runtime.

In the Frame Hierarchy Model (FHM) programs are designed using **frame classes**, **frame instances** and **frames**. The **frame class** defines static state and behavior, similar to the class in OOP. The **frame** defines dynamic state and behavior, that is specific to a single instance of a frame class. After a frame is defined, the programmer can create one or more **frame instances** of it, similar to objects in OOP.

Any class deriving from one of the fundamental frame classes *Frame*, *PFrame* or *DFrame* is by definition a frame class. Frame instances are created by instantiating frames and frames are created by instantiating frames classes. Frames can be created without any general state or behavior by directly instantiating one of the fundamental frame classes.

```
class ButtonFrame(asyncframes.Frame):
    """An example frame class."""

@ButtonFrame
async def button_frame():
    """An example frame."""

@Frame
async def helper_frame():
    """An example frame without a frame class."""

button = button_frame(): # An example frame instance.
```

Note: The differences between *Frame*, *PFrame* and *DFrame* are explained in chapter *Parallel Programming with asyncframes*.

In a way, FHM adds another layer between classes and objects. The additional third layer may seem to add complexity to the programming model, but it can be strictly separated by the following principle:

Tip: State and behavior that is general enough to be applicable to different programs should be defined via **frame classes**. State and behavior that is specific to a single program should be defined via **frames**.

Ideally most frame classes should be defined in separate Python packages, so they can be reused across projects (see `examples/frame_libraries` in the git repository).

1.1.2 Example

To illustrate the differences between frame classes, frame instances and frames, let's consider a simple use case:

We would like to create a button in a user interface that prints the line "Hello World!" when clicked.

Creating frame classes for Gtk.Window and Gtk.Button

First we create a frame class that represents a GTK window:

```
1 class GtkFrame(FrameMeta, type(GObject)):
2     pass
3
4 class Window(Frame, Gtk.Window, metaclass=GtkFrame):
5     def __init__(self, *args, **kwargs):
6         Frame.__init__(self)
7         Gtk.Window.__init__(self, *args, **kwargs)
8         self.connect("destroy", lambda _: self.remove())
```

The class `Window` is a frame class because it derives from *`asyncframes.Frame`*. In line 8, we connect the window's destroy event to the *`asyncframes.Awaitable.remove()`* method. This will remove the frame when the user closes the window. The rest of this code snippet is required to enable multiple inheritance in Python:

Lines 1 and 2 declare a metaclass that derives from both the metaclass of `Frame` (i.e. `FrameMeta`) and `Gtk.Window` (i.e. `type(GObject)`). Lines 6 and 7 call the constructors of both base classes. Note that we are pass through any arguments of the window frame class to the GTK window. We will use this later to pass a title string to the window.

Now let's create another frame class for buttons:

```
class Button(Frame, Gtk.Button, metaclass=GtkFrame):
    def __init__(self, *args, **kwargs):
        Frame.__init__(self)
        Gtk.Button.__init__(self, *args, **kwargs)
        find_parent(Window).add(self)

        self.clicked = Event("Button.clicked")
        def send_clicked_event(*args):
            self.clicked.send(args)
        self.connect("clicked", send_clicked_event)
```

The `Button` frame class derives from both `Frame` and `Gtk.Button`. After calling the constructors of both base classes, we add the button to its window. Remember that any FHM program consists of a hierarchy of frames. To find the window this button belongs to, we use the function `asyncframes.find_parent(parenttype)` to search the hierarchy for the closest ancestor of type `Window`. Finally, in lines 7 through 10, we create an `asyncframes.Event` and connect it to the *clicked* event of the GTK button.

Creating a FHM program using Window and Button

Let's use `Window` and `Button` frame classes to create a simple GUI application:

```
@Window(title="Button Example")
async def main_frame(self):
    @Button("Click Here")
    async def button_frame(self):
        self.show()
        while True:
            await self.clicked
            print("Hello World!")
    button = button_frame()

    self.set_default_size(280, 40)
    self.set_border_width(8)
    self.show()
    await hold()

loop = glib_eventloop.EventLoop()
loop.run(main_frame)
```

We start by creating a main frame of type `Window` in lines 1 and 2. Since our `Window` frame class is forwarding all arguments to the underlying `Gtk.Window`, we can pass the window title when creating the frame. The `self` argument is optional in asyncframes. It refers to the frame just like the `self` argument on a Python method.

In lines 11 to 13 we use `self` to call methods of the `Gtk.Window`. Line 11 resizes the window, line 12 adds padding around our button and line 13 displays the window.

Line 14 is important to keep the `main_frame` from going out of scope. Frame classes are removed when they go out of scope and the window is part of our `main_frame`. Accordingly, without `await hold()` our application would close the window and exit immediately.

Note: `await hold()` is semantically equivalent to `await sleep(sys.float_info.max)`

The last thing to define is our button. FHM allows us to create the button and define its entire life cycle in a single block of code (lines 3 to 9). If we were to completely remove all code related this button in the future, we would only

need to remove or comment-out these lines. Lines 3 to 8 define the button frame and line 9 creates a frame instance.

Note: We define `button_frame` inside `main_frame` to emphasize that this button is a child of the window in the frame hierarchy. This is not a requirement. The button's position in the hierarchy only depends on where the frame instance is created (line 9). Accordingly, it wouldn't affect the application if we defined `button_frame` outside `main_frame`.

Similar to the window title, we pass the button text as an argument when creating the button frame (line 3).

Lines 5 to 8 define the behavior of the button. In our case we start by making the button visible (line 5) and then we print "Hello World!" (line 8) every time (line 6) the button is clicked (line 7).

Running the example

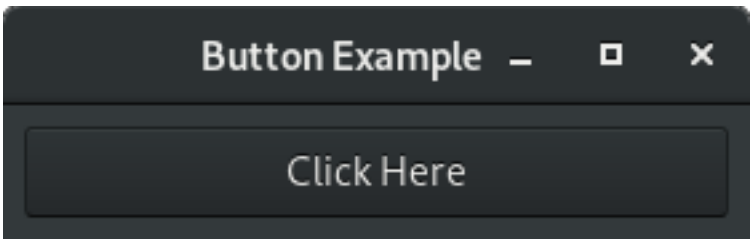
To run an FHM application that uses GTK, we need to invoke the main frame from an eventloop that is implemented on top of the GLib event system:

```
1 loop = glib_eventloop.EventLoop()
2 loop.run(main_frame)
```

This example requires asyncframes and GTK for Python:

```
pip install asyncframes pygobject
```

The created window will look like this:



Whenever the button is pressed, "Hello World!" will be displayed in the output terminal or console.

1.2 Parallel Programming with asyncframes

Parallel programming is to write fragments of code that can be executed in parallel. It is used to either speed up code execution or to circumvent blocking operations.

1.2.1 Types of Parallelism

There are many types of parallelism. For asyncframes we distinguish the following three types:

1. Cooperative multitasking
2. Shared memory parallelism
3. Distributed memory parallelism

If you understand the differences between these types of parallelism and know about the implications of their implementation in Python, feel free to move on to the next section.

Cooperative multitasking does not actually execute code in parallel. Instead, It allows a program to pause the execution of a function and execute other parts of the program before returning. In Python, such functions can be implemented using generators (functions that use the `yield` keyword) or coroutines (functions defined using `async def`, that can use the `await` keyword). We will only focus on coroutines here. By using coroutines on top of an event loop, we can implement a parallel programming environment, where the eventloop acts as the task scheduler and individual coroutines act as tasks that periodically yield execution using the `await` keyword. Since this environment never actually switches between CPU threads, it doesn't come with any of the usual caveats of parallel programming, like nondeterministic execution, dead locks and race conditions. However, cooperative multitasking doesn't run faster than serial code and blocking a single coroutine will block the entire program.

Shared memory parallelism is employed when separate execution contexts (i.e. threads) execute code in parallel that accesses a shared pool of memory. In modern computers this is utilized by running separate CPU cores or hardware threads in parallel. In contrast to cooperative multitasking, this type of parallelism does run code in parallel. It therefore requires much more careful code design to avoid dead locks and race conditions, while rewarding the programmer with parallel speedup and non-blocking execution. In Python shared memory parallelism is limited by the Global Interpreter Lock (GIL). The GIL is a mechanism that only allows one thread to interpret Python code at a time. This prohibits parallel speedups, but it doesn't affect the non-blocking behavior of multi-threaded Python code. Since this limitation is not part of the Python standard, it may not apply to all Python distributions and it may even be removed in a future release of CPython. In terms of the Frame Hierarchy Programming Model, we assume that shared memory parallelism *can* result in faster code, and it should be preferred to cooperative multitasking for thread-safe frames.

Distributed memory parallelism is employed when threads cannot access memory of other threads without using specialized memory transfer mechanisms. In modern computers such threads are known as processes. They can either run on the same machine, using memory separated by the operating system, or on physically separate machines. In either case we should assume inter-process communication to be much slower than interactions between shared memory threads. The main advantage of distributed memory parallelism is that it is much more scalable than shared memory parallelism. A modern supercomputer, for example, has thousands of compute nodes with physically separated memory, while the CPUs on each node only employ a small number of hardware threads. In Python multi-processing runs multiple instances of the program. Each process runs a separate Python interpreter, which allows speedup through parallel execution without being affected by the previously mentioned limitations of the GIL. Distributed frames can take advantage of this speedup, as long as they are thread-safe and they don't access global variables of other processes.

Important: Distributed frames aren't implemented in asyncframes v2.2. This feature is under active development and will be added in a future release.

1.2.2 Parallel Programming using Frames

The following table summarizes the three types of parallelism of the previous section from a software engineering perspective:

Type of parallelism	Implementation in asyncframes	Perks		Requirements	
		Blocking operations	Parallel speedup	Thread-safety	Localized memory
Cooperative multitasking	*Frame*				
Shared memory	*PFrame*[1]	✓		✓	
Distributed memory	*DFrame*[2]	✓	✓	✓	✓

In the Frame Hierarchy Programming Model, parallelism is implemented according to the "concurrency by default" paradigm. By default every frame is maximally parallel (*DFrame*), but the programmer can reduce the degree of parallelism by employing restrictions. *PFrames* are like *DFrames*, but with the restriction of running on the same *process* as their parent frame. *Frames* are like *PFrames*, but with the restriction of running on the same *thread* as their parent frame.

The main advantage of the restriction model is that parallel software can be designed iteratively. The entire program can first be designed using only *Frames* (except blocking operations, which should always be placed inside *PFrames* or *DFrames*; see table). Once completed, the programmer can assure thread-safety of individual frames, promote them to *PFrames* and rerun all unit tests. If the program still produces deterministic correct results (note that multithreading can lead to non-deterministic errors, which only fail with a certain probability!), the programmer can assure individual frames don't access global memory of other processes and further promote them to *DFrames*.

Reasons to choose higher degrees of parallelism

In general it should be the goal of any frame hierarchy program to promote as many frames as possible to higher degrees of parallelism. Only then can an optimized scheduler efficiently distribute frames across available threads and processes in a transparent and scalable manner. Keep in mind that any *PFrame can* be executed on the same thread as it's parent frame if the scheduler decides that this is the most efficient thing to do. For example, if all other available threads are busy. It can even execute different parts of a single frame on different threads. The fewer restrictions are enforced, the more freedom is granted to the scheduler to efficiently parallelize a program.

How to make parts of a program singlethreaded

There are situations where multithreading should be avoided. For example, many user interface libraries, like Qt, are strictly singlethreaded. By only using *Frames* to interact with the user interface, this restriction is satisfied. Programmers can still create *PFrames* or *DFrames* in response to a user interface event, for example to execute a computationally expensive operation in parallel, as long as these parallel frames don't directly access the user interface.

It's important to note that asyncframes doesn't use a master thread. Whenever an eventloop runs a *Frame*, this frame will run on the thread that executed the `EventLoop.run()` command. However, this doesn't mean that all *Frames* always run on that same thread. If a *Frame* is created from a *Pframe*, it will run on whatever thread the *Pframe* was running on when it created the *Frame*. This way, a frame hierarchy program can contain multiple serial parts that run on different threads. For example, a program can utilize a singlethreaded user interface library and a singlethreaded database library on different threads. Of course, these concepts also apply to processes if the frame hierarchy contains *DFrames*.

[1] *PFrames* require asyncframes v2.0 or above.
[2] *DFrames* aren't implemented in asyncframes v2.2. This feature is under development and will be added in a future release.

How to disable multithreading entirely

When `EventLoop.run()` is called, asyncframes allocates multiple threads. The number of allocated threads can be controlled with the `num_threads` parameter. By default asyncframes will allocate as many threads as there are available hardware threads on the CPU. To run a program entirely singlethreaded, set the `num_threads` parameter to 1. In this scenario, asyncframes will never run any other threads, even if *PFrames* or *DFrames* are used. This is because in the restriction model *PFrames* are free to run on any available thread, but there is only on thread available.

1.2.3 Example

To illustrate the differences between *Frames* and *PFrames*, let's run multiple counters in parallel using blocking operations.

The following frame prints the result of a call to `printfunc` three times every 0.3 seconds after an initial delay:

```
@asyncframes.Frame
async def frame_counter(delay, printfunc, printfunc_args):
    time.sleep(delay)
    for _ in range(3):
        time.sleep(0.3)
        print(printfunc(*printfunc_args), end='', flush=True)
```

We also create a parallel frame with the same content:

```
@asyncframes.PFrame
async def pframe_counter(delay, printfunc, printfunc_args):
    time.sleep(delay)
    for _ in range(3):
        time.sleep(0.3)
        print(printfunc(*printfunc_args), end='', flush=True)
```

The frame `count_using_frames` creates three *Frame*-based counters, each starting 0.1 seconds after the previous counter. Again, we also create a *PFrame*-based version, named `count_using_pframes`:

```
@asyncframes.Frame
async def count_using_frames(printfunc):
    counters = [frame_counter(delay=0.1 * i, printfunc=printfunc, printfunc_args=(i,
↪)) for i in range(3)]
    await asyncframes.all_(*counters)
    print()

@asyncframes.Frame
async def count_using_pframes(printfunc):
    counters = [pframe_counter(delay=0.1 * i, printfunc=printfunc, printfunc_args=(i,
↪)) for i in range(3)]
    await asyncframes.all_(*counters)
    print()
```

Let's see what happens if we run three blocking counters using *Frames*. The first counter prints the character *a*, the second one prints *b* and the third one prints *c*:

```
>>> loop.run(count_using_frames, printfunc=lambda i: "abc"[i])
aaabbbccc
```

As we learned in the previous section, *Frames* always run on the same thread as their parent frame. We don't use any *PFrames*, so that thread is the main thread (thread 0). Since we never call `await` inside `frame_counter`, the main thread is blocked until the counter returns, before starting the next counter.

We can visualize which thread each counter runs on, using the `get_current_eventloop_index()` function:

```
>>> loop.run(count_using_frames, printfunc=lambda i: asyncframes.get_current_
↪eventloop_index())
000000000
```

Now let's repeat the experiment using *PFrames*:

```
>>> loop.run(count_using_frames, printfunc=lambda i: "abc"[i])
abcabcabc
```

Each counter still blocks until it is done, but because we now create *PFrame*-based counters, asyncframes can distribute them over individual threads in the thread pool:

```
>>> loop.run(count_using_frames, printfunc=lambda i: asyncframes.get_current_
↪eventloop_index())
123123123
```

Note that the used threads are threads 1 through 3. That's because thread 0 is used to run the parent frame (`count_using_pframes`).

Finally, let's restrict the thread pool to three threads:

```
>>> loop.run(count_using_frames, printfunc=lambda i: "abc"[i], num_threads=3)
abababccc
```

We notice that the first two counters run in parallel, but the third one is blocked. Let's see which threads were involved in this behavior:

```
>>> loop.run(count_using_frames, printfunc=lambda i: asyncframes.get_current_
↪eventloop_index(), num_threads=3)
121212111
```

We use 3 threads. Thread 0 runs the parent frame (`count_using_pframes`), threads 1 runs the *a* counter and thread 2 runs the *b* counter. The *c* counter can't start until a thread becomes available. The first thread to become available is thread 1, after the *a* counter finishes.

Note: We used blocking sleep operations here for illustrative purposes only. In production code one should use `await asyncframes.sleep()` instead.

1.3 API

1.3.1 asyncframes module

class asyncframes.**all_**(**awaitables*)

Bases: *asyncframes.Awaitable*

An awaitable that blocks the awaiting frame until all passed awaitables have woken up.

Parameters **awaitables** (*Awaitable[]*) – A list of all awaitables to await.

class asyncframes.**animate**(*seconds*, *callback*, *interval=0.0*)

Bases: *asyncframes.Event*

An awaitable event used for periodically calling a callback function for the specified amount of time.

Parameters

- **seconds** (*float*) – The duration of the animation.
- **callback** (*Callable[float, None]*) – The function to be called on every iteration. The first parameter of *callback* indicates animation progress between 0 and 1.
- **interval** (*float, optional*) – Defaults to 0.0. The minimum time in seconds between two consecutive calls of the callback.

class asyncframes.**any_**(**awaitables*)
Bases: *asyncframes.Awaitable*

An awaitable that blocks the awaiting frame until any of the passed awaitables wakes up.

Parameters **awaitables** (*Awaitable[]*) – A list of all awaitables to await.

class asyncframes.**Awaitable**(*name*, *singleshot*, *lifebound*)
Bases: collections.abc.Awaitable

An awaitable frame or event.

Every node in the frame hierarchy is a subclass of *Awaitable*. An awaitable has a *__name__*, a parent awaitable (None, if the awaitable is the main frame), a list of child awaitables and a result, that gets set when the awaitable finishes.

Parameters **name** (*str*) – The name of the awaitable.

remove()
Remove this awaitable from the frame hierarchy.

Returns

An awaitable event.

The remove event returns True once the awaitable has been removed or False if the request was either canceled, or the awaitable had already been removed before.

Return type *Event*

removed
Boolean property, indicating whether this awaitable has been removed from the frame hierarchy.

class asyncframes.**AbstractEventLoop**
Bases: object

Abstract base class of event loops.

postevent(*eventsource*, *event*, *delay=0*)

run(*frame*, **frameargs*, *num_threads=0*, ***framekwargs*)

static sendevent(*eventsource*, *event*, *process_counter=None*, *blocking=False*)

class asyncframes.**Event**(*name*, *singleshot=False*, *lifebound=False*)
Bases: *asyncframes.Awaitable*

An awaitable event.

Instantiate or overload this class to implement new events. Each type of event should be emitted by exactly one event class. For example, key-up and key-down events should be implemented by two separate events. Events represent leave nodes in the frame hierarchy.

Parameters

- **name** (*str*) – The name of the event.

- **singleshot** (*bool, optional*) – Defaults to False. If True, removes the event after it has been woken.

post(*args=None*, *delay=0*)
Enqueue an event in the event loop.

Parameters

- **args** (*optional*) – Defaults to None. Event arguments, for example, the progress value on a progress-update event.
- **delay** (*float, optional*) – Defaults to 0. The time in seconds to wait before posting the event.

send(*args=None*)
Dispatch and immediately process an event.

Parameters **args** (*optional*) – Defaults to None. Event arguments, for example, the progress value on a progress-update event.

asyncframes.**find_parent**(*parenttype*)
Find parent frame of given type.

Recursively search the frame hierarchy for the closest ancestor of the given type.

Parameters **parenttype** (*type*) – The frame class type to search for.

Returns The closest ancestor of type `parenttype` or None if none was found.

Return type *Frame*

class asyncframes.**Frame**(*startup_behaviour=<FrameStartupBehaviour.delayed: 1>*, *thread_idx=None*)
Bases: *asyncframes.Awaitable*

An object within the frame hierarchy.

This class represents the default frame class. All other frame classes have to be derived from *Frame*.

A frame is an instance of a frame class. Use the nested Factory class to create frames.

The factory class is created by decorating a function or coroutine with `@FRAME`, where `FRAME` is the frame class.

Example:

```
class MyFrameClass(Frame):
    pass

@MyFrameClass
async def my_frame_factory():
    pass

assert(type(my_frame_factory) == MyFrameClass.Factory)
my_frame = my_frame_factory()
assert(type(my_frame) == MyFrameClass)
```

Parameters

- **startup_behaviour** (*FrameStartupBehaviour, optional*) – Defaults to FrameStartupBehaviour.delayed. Controls whether the frame is started immediately or queued on the eventloop.

- **thread_idx** (*int, optional*) – Defaults to None. If set, forces the scheduler to affiliate this frame with the given thread.

free
Event – An event that fires just before the frame is removed.

ready
Event – An event that fires the first time the frame is suspended (using `await`) or goes out of scope.

Raises `ValueError` – If *thread_idx* is outside the range of allocated threads.

The number of allocated threads is controlled by the *num_threads* parameter of *AbstractEventLoop.run()*.

class Factory(*framefunc*, *frameclassargs*, *frameclasskwargs*)
Bases: `object`

A frame function declared in the context of a frame class.

Parameters

- **framefunc** (*Callable*) – The function or coroutine that describes the frame's behaviour.
- **frameclassargs** (*tuple*) – Positional arguments to the frame class.
- **frameclasskwargs** (*dict*) – Keyword arguments to the frame class.

frameclass
alias of *Frame*

create(*framefunc*, **frameargs*, ***framekwargs*)
Start the frame function with the given arguments.

Parameters **framefunc** (*function*) – A coroutine or regular function controlling the behaviour of this frame. If *framefunc* is a coroutine, then the frame only exists until the coroutine exits.

class asyncframes.**FrameMeta**
Bases: `abc.ABCMeta`

class asyncframes.**FrameStartupBehaviour**
Bases: `enum.Enum`

An enumeration.

delayed = 1

immediate = 2

class asyncframes.**FreeEventArgs**
Bases: `object`

Event arguments returned by the *Frame.free* event.

cancel
bool – Setting this to True, cancels the event.

asyncframes.**get_current_eventloop_index**()
Get the thread index of the currently active event loop.

Returns The thread index of the current event loop or None if no event loop is currently active.

Return type int

exception asyncframes.**InvalidOperationException**(*msg*)
Bases: Exception

Raised when operations are performed out of context.

Parameters **msg** (*str*) – Human readable string describing the exception.

class asyncframes.**hold**
Bases: *asyncframes.Event*

An awaitable event used for suspending execution indefinitely.

Frames are automatically removed when the frame coroutine finishes. If you would like the frame to remain open until it is removed, write *await hold()* at the end of the coroutine.

class asyncframes.**PFrame**(*startup_behaviour=<FrameStartupBehaviour.delayed: 1>*, *thread_idx=None*)
Bases: *asyncframes.Frame*

A parallel *Frame* that can run on any thread.

Multithreading can be enabled for any frame by changing its base class to *PFrame*.

The only difference between *Frame* and *PFrame* is that instances of *PFrame* are not restricted to run on the same thread as their parent frame.

Parameters

- **startup_behaviour** (FrameStartupBehaviour, *optional*) – Defaults to FrameStartupBehaviour.delayed. Controls whether the frame is started immediately or queued on the eventloop.
- **thread_idx** (*int, optional*) – Defaults to None. If set, forces the scheduler to affiliate this frame with the given thread.

Raises ValueError – If *thread_idx* is outside the range of allocated threads.

The number of allocated threads is controlled by the *num_threads* parameter of *AbstractEventLoop.run()*.

Factory
alias of *PFrame.Factory*

class asyncframes.**Primitive**(*owner*)
Bases: object

An object owned by a frame of the specified frame class.

A primitive has to be created within the frame function of its owner or within the frame function of any child frame of its owning frame class. If it is created within a child frame, it will still be registered with the closest parent of the owning frame class.

Parameters **owner** (*class*) – The owning frame class.

Raises

- TypeError – Raised if owner is not a frame class.
- Exception – Raised if a primitive is created outside the frame function of its owning frame class.

remove()
Remove this primitive from its owner.

Returns If True, this event was removed. If False the request was either canceled, or the event had already been removed before

Return type bool

class asyncframes.**sleep**(*seconds=0.0*)
Bases: *asyncframes.Event*

An awaitable event used for suspending execution by the specified amount of time.

A duration of 0 seconds will resume the awaiting frame as soon as possible. This is useful to implement non-blocking loops.

Parameters **seconds** (*float, optional*) – Defaults to 0. The duration to wait.

1.3.2 asyncframes.asyncio_eventloop module

class asyncframes.asyncio_eventloop.**EventLoop**
Bases: *asyncframes.AbstractEventLoop*

An implementation of AbstractEventLoop based on asyncio.

1.3.3 asyncframes.glib_eventloop module

1.3.4 asyncframes.pyqt5_eventloop module

CHAPTER 2

Indices and tables

- genindex

Python Module Index

a

Index