evergreen Documentation

Release 0.2.0

Saúl Ibarra Corretgé

March 31, 2014

CHAPTER 1

Overview

Evergreen is a Python library to help you write multitasking and I/O driven applications in a cooperative way. Unlike when using threads, where the execution model is preemptive and thus not controlled by *you*, using a cooperative model allows you to choose *what* runs and *when*. Evergreen will make this easier.

CHAPTER 2

Features

- Cooperative multitasking abstractions similar to threads
- Multiple synchronization primitives
- Event loop driven scheduler
- Non-blocking I/O
- Convenience APIs for writing network server software
- Cooperative *concurrent.futures* style APIs
- Cooperative versions of certain standard library modules
- As little magic as possible

CHAPTER 3

Documentation

3.1 Design

The following sections contain an explanation of the design of the different parts that compose evergreen. Evergreen was inspired by similar libraries such as Gevent and Eventlet, but some of the key ideas are different:

- Limit the amount of 'magic' to the minimum possible
- Cooperative tasks should look like threads
- APIs for dealing with tasks should mimic those used in threading
- Task scheduling has to be predictable and consistent, but without being exposed to the user
- The event loop (or hub or reactor) is a first class citizen and it's not hidden

3.1.1 Event loop

The event loop can be considered the central point of evergreen, it deals with timers, I/O and task scheduling (described later). The event loop API is heavily inspired by PEP 3156, so it's possible that in the future the event loop implementation evergreen uses can be replaced. At the moment evergreen uses puyv as the underlying event loop.

In evergreen only one loop may exist per thread, and it has to be manually created for threads other than the main thread. This would be the structure of a normal program using evergreen:

```
import evergreen

# Create the global loop
loop = evergreen.EventLoop()

# Setup tasks
...

# Start the loop
loop.run()
```

No tasks will start working until *loop.run* is called unless a blocking operation is performed, in which case the loop is automatically started. For long running processes such as servers, it's advised to explicitly create the event loop, setup tasks and manually run it. Small scripts can rely on the fact that the main thread's loop is automatically created and run when an opperation cooperatively blocks.

3.1.2 Tasks

The cooperative task abstraction provided by evergreen (through the `Task` class). The public API for this class mimics that of a *threading.Thread* thread, but it's scheduled cooperatively.

Tasks are implemented using the fibers library.

Here are some 'rules' that apply to tasks:

- Tasks don't yield control to each other, they must always yield control to the loop, or schedule a switch to the desired task in the loop, this ensures predictable execution order and fairness.
- In order to exchange values between tasks any of the provided synchronization primitives should be used, the tasks themselves don't provide any means to do it.

3.1.3 Scheduling

The scheduler has no public interface. You interact with it by switching execution to the loop. In fact, there is no single object representing the scheduler, its behavior is implemented by the Task, Future and other classes using only the public interface of the event loop.

The easiest way to suspend the execution of the current task and yield control to the loop so that other tasks can run is to use:

```
evergreen.sleep(0)
```

The only functions that suspend the current task are those which 'block', for example lock or socket functions.

3.2 API documentation

Documentation for the modules composing the public API for Evergreen.

3.2.1 Modules

The Event Loop

The event loop is the main entity in evergreen, together with tasks. It takes care of running all scheduled operations and provides time based callback scheduling as well as I/O readyness based callback scheduling.

class `evergreen.loop.`**`EventLoop`**

This is the main class that sets things in motion in evergreen. I runs scheduled tasks, timers and I/O operations. Only one event loop may exist per thread and it needs to be explicitly created for threads other than the main thread. The current loop can be accessed with *evergreen.current.loop*.

classmethod **`current`**`()`

Returns the event loop instance running in the current thread.

`call_soon``(`*callback*, **args*, ***kw*`)`

Schedule the given callback to be called as soon as possible. Returns a *Handler* object which can be used to cancel the callback.

`call_from_thread``(`*callback*, **args*, ***kw*`)`

Schedule the given callback to be called by the loop thread. This is the only thread-safe function on the loop. Returns a *Handler* object which can be used to cancel the callback.

call_later(*delay*, *callback*, **args*, ***kw*)
Schedule the given callback to be called after the given amount of time. Returns a *Handler* object which can be used to cancel the callback.

call_at(*when*, *callback*, **args*, ***kw*)
Schedule the given callback to be called at the given time. Returns a *Handler* object which can be used to cancel the callback.

time()
Returns the current time.

add_reader(*fd*, *callback*, **args*, ***kw*)
Create a handler which will call the given callback when the given file descriptor is ready for reading.

remove_reader(*fd*)
Remove the read handler for the given file descriptor.

add_writer(*fd*, *callback*, **args*, ***kw*)
Create a handler which will call the given callback when the given file descriptor is ready for writing.

remove_writer(*fd*)
Remove the write handler for the given file descriptor.

add_signal_handler(*signum*, *callback*, **args*, ***kw*)
Create a handler which will run the given callback when the specified signal is captured. Multiple handlers for the same signal can be added. If the handler is cancelled, only *that* particular handler is removed.

remove_signal_handler(*signum*)
Remove all handlers for the specified signal.

switch()
Switch task execution to the loop's main task. If the loop wasn't started yet it will be started at this point.

run()
Start running the event loop. It will be automatically stopped when there are no more scheduled tasks or callbacks to run.

Note: Once the loop has been stopped it cannot be started again.

---

run_forever()
Similar to *run* but it will not stop be stopped automatically even if all tasks are finished. The loop will be stopped when *stop()* is called. Useful for long running processes such as servers.

stop()
Stop the event loop.

destroy()
Free all resources associated with an event loop. The thread local storage is also emptied, so after destroying a loop a new one can be created on the same thread.

class evergreen.loop.**Handler**
This is an internal class which is returned by many of the *EventLoop* methods and provides a way to cancel scheduled callbacks.

Note: This class should not be instantiated by user applications, the loop itself uses it to wrap callbacks and return it to the user.

---

cancel()
Cancels the handle, preventing its callback from being executed, if it wasn't executed yet.

> **Warning:** Like every API method other than *EventLoop.call_from_thread*, this function is not thread safe, it must be called from the event loop thread.

Finding the 'current loop'

evergreen provides a convenience mechanism to get a reference to the loop running in the current thread:

```
current_loop = evergreen.current.loop
```

If a loop was not explicitly created in the current thread `RuntimeError` is raised.

Handling signals

While the *signal* module works just fine, it's better to use the signal handling functions provided by the *EventLoop*. It allows adding multiple handlers for the same signal, from different threads and the handlers are called in the appropriate thread (where they were added from).

Task: a cooperative *thread*

The tasks module provides one of the most important pieces of evergreen, the *Task* class along with some utility functions. The *Task* class encapsulates a unit of cooperative work and it has an API which is very similar to the *Thread* class in the standard library.

`evergreen.tasks.`**`sleep`**(*seconds*)
Suspend the current task until the given amount of time has elapsed.

`evergreen.tasks.`**`spawn`**(*func*, **args*, ***kwargs*)
Create a *Task* object to run *func(*args, **kwargs)* and start it. Returns the *Task* object.

`evergreen.tasks.`**`task`**()
Decorator to run the decorated function in a *Task*.

class `evergreen.tasks.`**`Task`**(*target=None*, *args=()*, *kwargs={}*)
Runs the given target function with the specified arguments as a cooperative task.

classmethod **`current`**()
Returns the current running task instance.

`start`()
Schedules the task to be run.

`run`()
Main method the task will execute. Subclasses may want to override this method instead of passing arguments to __init__.

`join`(*timeout=None*)
Wait until the task finishes for the given amount of time. Returns a boolean flag indicating if the task finished the work or not.

`kill`(*typ=TaskExit*[, *value*[, *tb*]])
Raises the given exception (*TaskExit* by default) in the task. If the task wasn't run yet it will be raised the moment it runs. If the task was already running, it will be raised when it yields control.

Calling this function doesn't unschedule the current task.

exception evergreen.tasks.**TaskExit**
Exception used to kill a single task. It does not propagate.

Synchronization primitives: Channel

Channels are the simplest mechanism for 2 tasks to exchange data.

class evergreen.channel.**Channel**
A synchronous communication pipe between 2 tasks.

send(*data*)
Send data over the channel. The calling task will be blocked if there is no task waiting for the data on the other side.

send_exception(*exc_type*, *exc_value=None*, *exc_tb=None*)
Send the given exception. It will be raised on the receiving task.

receive()
Wait for data to arrive on the channel.

Synchronization primitives: Event

This is one of the simplest mechanisms for communication between tasks: one task signals an event and other threads wait for it.

class evergreen.event.**Event**
This class implements a cooperative version of *threading.Event*.

is_set()
Returns *True* if the flag is set, *False* otherwise.

set()
Set the internal flag to true. All tasks waiting for it to become true are awakened. Tasks that call wait() once the flag is true will not block at all.

clear()
Reset the internal flag to false. Subsequently, tasks calling wait() will block until set() is called to set the internal flag to true again.

wait([*timeout*])
Block until the internal flag is true. If the internal flag is true on entry, return immediately. Otherwise, block until another task calls set() to set the flag to true, or until the optional timeout occurs. The internal flag is returned on exit.

Synchronization primitives: locks

This module implements synchronization primitives to be used with cooperative tasks, in an analogous and API compatible way as *threading* module's primitives which are used with threads.

class evergreen.locks.**Semaphore**([*value*])
A semaphore manages an internal counter which is decremented by each `acquire()` call and incremented by each `release()` call. The counter can never go below zero; when `acquire()` finds that it is zero, it blocks, waiting until some other task calls `release()`.

The optional argument gives the initial *value* for the internal counter; it defaults to `1`. If the *value* given is less than 0, `ValueError` is raised.

acquire([*blocking*])
Acquire a semaphore.

When invoked without arguments: if the internal counter is larger than zero on entry, decrement it by one and return immediately. If it is zero on entry, block, waiting until some other task has called `release()` to make it larger than zero. This is done with proper interlocking so that if multiple `acquire()` calls are blocked, `release()` will wake exactly one of them up. The implementation may pick one at random, so the order in which blocked tasks are awakened should not be relied on. Returns true (or blocks indefinitely).

When invoked with *blocking* set to false, do not block. If a call without an argument would block, return false immediately; otherwise, do the same thing as when called without arguments, and return true.

When invoked with a *timeout* other than None, it will block for at most *timeout* seconds. If acquire does not complete successfully in that interval, return false. Return true otherwise.

release()
Release a semaphore, incrementing the internal counter by one. When it was zero on entry and another task is waiting for it to become larger than zero again, wake up that task.

class `evergreen.locks.`**BoundedSemaphore**([*value*])
Class implementing bounded semaphore objects. A bounded semaphore checks to make sure its current value doesn't exceed its initial value. If it does, `ValueError` is raised. In most situations semaphores are used to guard resources with limited capacity. If the semaphore is released too many times it's a sign of a bug. If not given, *value* defaults to 1.

class `evergreen.locks.`**RLock**
This class implements reentrant lock objects. A reentrant lock must be released by the task that acquired it. Once a task has acquired a reentrant lock, the same task may acquire it again without blocking; the task must release it once for each time it has acquired it.

acquire(*blocking=True*, *timeout=None*)
Acquire a lock, blocking or non-blocking.

When invoked without arguments: if this task already owns the lock, increment the recursion level by one, and return immediately. Otherwise, if another task owns the lock, block until the lock is unlocked. Once the lock is unlocked (not owned by any task), then grab ownership, set the recursion level to one, and return. If more than one task is blocked waiting until the lock is unlocked, only one at a time will be able to grab ownership of the lock. There is no return value in this case.

When invoked with the *blocking* argument set to true, do the same thing as when called without arguments, and return true.

When invoked with the *blocking* argument set to false, do not block. If a call without an argument would block, return false immediately; otherwise, do the same thing as when called without arguments, and return true.

When invoked with the floating-point *timeout* argument set to a positive value, block for at most the number of seconds specified by *timeout* and as long as the lock cannot be acquired. Return true if the lock has been acquired, false if the timeout has elapsed.

release()
Release a lock, decrementing the recursion level. If after the decrement it is zero, reset the lock to unlocked (not owned by any task), and if any other tasks are blocked waiting for the lock to become unlocked, allow exactly one of them to proceed. If after the decrement the recursion level is still nonzero, the lock remains locked and owned by the calling task.

Only call this method when the calling task owns the lock. A `RuntimeError` is raised if this method is called when the lock is unlocked.

There is no return value.

class evergreen.locks.**Condition**(*lock=None*)

This class implements condition variable objects. A condition variable allows one or more tasks to wait until they are notified by another task.

If the *lock* argument is given and not None, it must be a Semaphore or RLock object, and it is used as the underlying lock. Otherwise, a new RLock object is created and used as the underlying lock.

acquire(**args*)

Acquire the underlying lock. This method calls the corresponding method on the underlying lock; the return value is whatever that method returns.

release()

Release the underlying lock. This method calls the corresponding method on the underlying lock; there is no return value.

wait(*timeout=None*)

Wait until notified or until a timeout occurs. If the calling task has not acquired the lock when this method is called, a RuntimeError is raised.

This method releases the underlying lock, and then blocks until it is awakened by a notify() or notify_all() call for the same condition variable in another task, or until the optional timeout occurs. Once awakened or timed out, it re-acquires the lock and returns.

When the *timeout* argument is present and not None, it should be a floating point number specifying a timeout for the operation in seconds (or fractions thereof).

When the underlying lock is an RLock, it is not released using its release() method, since this may not actually unlock the lock when it was acquired multiple times recursively. Instead, an internal interface of the RLock class is used, which really unlocks it even when it has been recursively acquired several times. Another internal interface is then used to restore the recursion level when the lock is reacquired.

The return value is True unless a given *timeout* expired, in which case it is False.

notify(*n=1*)

By default, wake up one task waiting on this condition, if any. If the calling task has not acquired the lock when this method is called, a RuntimeError is raised.

This method wakes up at most *n* of the tasks waiting for the condition variable; it is a no-op if no tasks are waiting.

The current implementation wakes up exactly *n* tasks, if at least *n* tasks are waiting. However, it's not safe to rely on this behavior. A future, optimized implementation may occasionally wake up more than *n* tasks.

Note: an awakened task does not actually return from its wait() call until it can reacquire the lock. Since notify() does not release the lock, its caller should.

notify_all()

Wake up all tasks waiting on this condition. This method acts like notify(), but wakes up all waiting tasks instead of one. If the calling task has not acquired the lock when this method is called, a RuntimeError is raised.

Synchronization primitives: queues

The queue module implements multi-producer, multi-consumer queues, useful for exchanging data between tasks.

This module implements API compatible cooperative versions of the different queue implementations that can be found in the Python standard library.

The module implements three types of queue, which differ only in the order in which the entries are retrieved. In a FIFO queue, the first tasks added are the first retrieved. In a LIFO queue, the most recently added entry is the first

retrieved (operating like a stack). With a priority queue, the entries are kept sorted (using the *heapq* module) and the lowest valued entry is retrieved first.

class `evergreen.queue.`**`Queue`**(*maxsize*)
Constructor for a FIFO queue. *maxsize* is an integer that sets the upper limit on the number of items that can be placed in the queue. Insertion will block once this size has been reached, until queue items are consumed. If *maxsize* is less than or equal to zero, the queue size is infinite.

class `evergreen.queue.`**`PriorityQueue`**(*maxsize*)
Constructor for a LIFO queue. *maxsize* is an integer that sets the upper limit on the number of items that can be placed in the queue. Insertion will block once this size has been reached, until queue items are consumed. If *maxsize* is less than or equal to zero, the queue size is infinite.

class `evergreen.queue.`**`LifoQueue`**(*maxsize*)
Constructor for a priority queue. *maxsize* is an integer that sets the upper limit on the number of items that can be placed in the queue. Insertion will block once this size has been reached, until queue items are consumed. If *maxsize* is less than or equal to zero, the queue size is infinite.

The lowest valued entries are retrieved first (the lowest valued entry is the one returned by `sorted(list(entries))[0]`). A typical pattern for entries is a tuple in the form: `(priority_number, data)`.

Queue Objects

Queue objects (`Queue`, `LifoQueue`, or `PriorityQueue`) provide the public methods described below.

`Queue.`**`qsize`**`()`
Return the approximate size of the queue. Note, qsize() > 0 doesn't guarantee that a subsequent get() will not block, nor will qsize() < maxsize guarantee that put() will not block.

`Queue.`**`empty`**`()`
Return `True` if the queue is empty, `False` otherwise. If empty() returns `True` it doesn't guarantee that a subsequent call to put() will not block. Similarly, if empty() returns `False` it doesn't guarantee that a subsequent call to get() will not block.

`Queue.`**`full`**`()`
Return `True` if the queue is full, `False` otherwise. If full() returns `True` it doesn't guarantee that a subsequent call to get() will not block. Similarly, if full() returns `False` it doesn't guarantee that a subsequent call to put() will not block.

`Queue.`**`put`**(*item*[, *block*[, *timeout*]])
Put *item* into the queue. If optional args *block* is true and *timeout* is None (the default), block if necessary until a free slot is available. If *timeout* is a positive number, it blocks at most *timeout* seconds and raises the `Full` exception if no free slot was available within that time. Otherwise (*block* is false), put an item on the queue if a free slot is immediately available, else raise the `Full` exception (*timeout* is ignored in that case).

`Queue.`**`put_nowait`**(*item*)
Equivalent to `put(item, False)`.

`Queue.`**`get`**([*block*[, *timeout*]])
Remove and return an item from the queue. If optional args *block* is true and *timeout* is None (the default), block if necessary until an item is available. If *timeout* is a positive number, it blocks at most *timeout* seconds and raises the `Empty` exception if no item was available within that time. Otherwise (*block* is false), return an item if one is immediately available, else raise the `Empty` exception (*timeout* is ignored in that case).

`Queue.`**`get_nowait`**`()`
Equivalent to `get(False)`.

Two methods are offered to support tracking whether enqueued tasks have been fully processed by daemon consumer tasks.

`Queue.task_done()`
Indicate that a formerly enqueued task is complete. Used by queue consumer tasks. For each `get()` used to fetch a task, a subsequent call to `task_done()` tells the queue that the processing on the task is complete.

If a `join()` is currently blocking, it will resume when all items have been processed (meaning that a `task_done()` call was received for every item that had been `put()` into the queue).

Raises a `ValueError` if called more times than there were items placed in the queue.

`Queue.join()`
Blocks until all items in the queue have been gotten and processed.

The count of unfinished tasks goes up whenever an item is added to the queue. The count goes down whenever a consumer task calls `task_done()` to indicate that the item was retrieved and all work on it is complete. When the count of unfinished tasks drops to zero, `join()` unblocks.

Example of how to wait for enqueued tasks to be completed

```
def worker():
    while True:
        item = q.get()
        do_work(item)
        q.task_done()

q = Queue()
for i in range(num_worker_tasks):
    t = Task(target=worker)
    t.start()

for item in source():
    q.put(item)
q.join()       # block until all tasks are done
```

Exceptions

exception `evergreen.queue.Empty`
Exception raised when non-blocking `get()` (or `get_nowait()`) is called on a `Queue` object which is empty.

exception `evergreen.queue.Full`
Exception raised when non-blocking `put()` (or `put_nowait()`) is called on a `Queue` object which is full.

Task local storage

This module provides the equivalent to *threading.local* but applying the concept to tasks.

class `evergreen.local.local`
A class that represents task-local data. Task-local data are data whose values are task specific. To manage task-local data, just create an instance of `local` (or a subclass) and store attributes on it

```
mydata = local()
mydata.x = 1
```

The instance's values will be different for separate tasks.

Managing timeouts

Timeout objects allow to stop a task after a given amount of time. This is useful to abort a network connection if the response is taking too long to arrive, for example.

class `evergreen.timeout.`**`Timeout`**(*[seconds[, exception]]*)
Raises *exception* in the current task after *timeout* seconds.

When *exception* is omitted or `None`, the `Timeout` instance itself is raised. If *seconds* is None or < 0, the timer is not scheduled, and is only useful if you're planning to raise it directly.

`Timeout` objects are context managers, and so can be used in with statements. When used in a with statement, if *exception* is `False`, the timeout is still raised, but the context manager suppresses it, so the code outside the with-block won't see it.

`start`()
Start the Timeout object.

`cancel`()
Prevent the Timeout from raising, if hasn't done so yet.

Futures

This module implements an (almost) API compatible *concurrent.futures* implementation which is cooperative.

class `evergreen.futures.`**`Future`**
The Future class encapsulates the asynchronous execution of a callable. Future instances are created by Executor.submit().

`cancel`()
Attempt to cancel the call. If the call is currently being executed and cannot be cancelled then the method will return False, otherwise the call will be cancelled and the method will return True.

`cancelled`
Return True if the call was successfully cancelled.

`done`
Return True if the call was successfully cancelled or finished running.

`get`(*timeout=None*)
Return the value returned by the call. If the call hasn't yet completed then this method will wait up to timeout seconds. If the call hasn't completed in timeout seconds, then a TimeoutError will be raised. timeout can be an int or float. If timeout is not specified or None, there is no limit to the wait time.

If the future is cancelled before completing then CancelledError will be raised.

If the call raised, this method will raise the same exception.

`add_done_callback`(*func*)
Attaches the callable func to the future. func will be called, with the future as its only argument, when the future is cancelled or finishes running.

Added callables are called in the order that they were added and are always called in a thread belonging to the process that added them. If the callable raises a Exception subclass, it will be logged and ignored. If the callable raises a BaseException subclass, the behavior is undefined.

If the future has already completed or been cancelled, func will be called immediately.

class `evergreen.futures.`**`Executor`**
An abstract class that provides methods to execute calls asynchronously. It should not be used directly, but through its concrete subclasses.

submit (*fn*, **args*, ***kwargs*)
Schedules the callable, *fn*, to be executed as `fn(*args **kwargs)` and returns a `Future` object representing the execution of the callable.

```
with ThreadPoolExecutor(max_workers=1) as executor:
    future = executor.submit(pow, 323, 1235)
    print(future.result())
```

map (*func*, **iterables*, *timeout=None*)
Equivalent to `map(func, *iterables)` except *func* is executed asynchronously and several calls to *func* may be made concurrently. The returned iterator raises a `TimeoutError` if `__next__()` is called and the result isn't available after *timeout* seconds from the original call to `Executor.map()`. *timeout* can be an int or a float. If *timeout* is not specified or `None`, there is no limit to the wait time. If a call raises an exception, then that exception will be raised when its value is retrieved from the iterator.

shutdown (*wait=True*)
Signal the executor that it should free any resources that it is using when the currently pending futures are done executing. Calls to `Executor.submit()` and `Executor.map()` made after shutdown will raise `RuntimeError`.

If *wait* is `True` then this method will not return until all the pending futures are done executing and the resources associated with the executor have been freed. If *wait* is `False` then this method will return immediately and the resources associated with the executor will be freed when all pending futures are done executing. Regardless of the value of *wait*, the entire Python program will not exit until all pending futures are done executing.

You can avoid having to call this method explicitly if you use the *with* statement, which will shutdown the `Executor` (waiting as if `Executor.shutdown()` were called with *wait* set to `True`)

```
import shutil
with ThreadPoolExecutor(max_workers=4) as e:
    e.submit(shutil.copy, 'src1.txt', 'dest1.txt')
    e.submit(shutil.copy, 'src2.txt', 'dest2.txt')
    e.submit(shutil.copy, 'src3.txt', 'dest3.txt')
    e.submit(shutil.copy, 'src3.txt', 'dest4.txt')
```

class `evergreen.futures.`**TaskPoolExecutor** (*max_workers*)
An `Executor` subclass that uses a pool of at most *max_workers* tasks to execute calls concurrently.

class `evergreen.futures.`**ThreadPoolExecutor** (*max_workers*)
An `Executor` subclass that uses a pool of at most *max_workers* threads to execute calls asynchronously.

`evergreen.futures.`**wait** (*fs*, *timeout=None*, *return_when=ALL_COMPLETED*)
Wait for the `Future` instances (possibly created by different `Executor` instances) given by *fs* to complete. Returns a named 2-tuple of sets. The first set, named `done`, contains the futures that completed (finished or were cancelled) before the wait completed. The second set, named `not_done`, contains uncompleted futures.

timeout can be used to control the maximum number of seconds to wait before returning. *timeout* can be an int or float. If *timeout* is not specified or `None`, there is no limit to the wait time.

return_when indicates when this function should return. It must be one of the following constants:

Constant	Description
`FIRST_COMPLETED`	The function will return when any future finishes or is cancelled.
`FIRST_EXCEPTION`	The function will return when any future finishes by raising an exception. If no future raises an exception then it is equivalent to `ALL_COMPLETED`.
`ALL_COMPLETED`	The function will return when all futures finish or are cancelled.

`evergreen.futures.`**as_completed**()
Returns an iterator over the `Future` instances (possibly created by different `Executor` instances) given

by *fs* that yields futures as they complete (finished or were cancelled). Any futures that completed before `as_completed()` is called will be yielded first. The returned iterator raises a `TimeoutError` if `__next__()` is called and the result isn't available after *timeout* seconds from the original call to `as_completed()`. *timeout* can be an int or float. If *timeout* is not specified or `None`, there is no limit to the wait time.

Exceptions

exception `evergreen.futures.`**`CancelledError`**

exception `evergreen.futures.`**`TimeoutError`**

Future class API changes

The future class in this module doesn't conform 100% to the API exposed by the equivalent class in the *concurrent.futures* module from the standard library, though they are pretty minor. Here is the list of changes:

- *cancelled* and *done* are properties, not functions
- *result* function is called *get*
- there is no *exception* function
- there is no *running* function
- futures can only be used once, after the result (or exception) is fetched from a future, it will raise `RuntimeError` if `get()` is called again on it

I/O utilities

The io module provides utility classes for writing cooperative servers and clients in an easy way.

Note: This module is still quite experimental, API changes are expected.

---

class `evergreen.io.`**`BaseStream`**

Basic class for defining a stream-like transport.

`closed`

Returns True if the stream is closed, False otherwise. Once the stream is closed an exception will be raised if any operation is attempted.

`read_bytes`(*nbytes*)

Read the specified amount of bytes (at most) from the stream.

`read_until`(*delimiter*)

Read until the specified delimiter is found.

`read_until_regex`(*regex*)

Read until the given regular expression is matched.

`write`(*data*)

Write data on the stream. Return True if data was flushed to the underlying resource and False in case the data was buffered and will be sent later.

`shutdown`()

Close the write side of a stream and flush the pending data.

close()
Close the stream. All further operations will raise an exception. Any buffered data will be lost.

_set_connected()
This method is part of the internal API. It sets the stream state to connected. Before a stream is connected all write operations will be buffered and flushed once the stream is connected.

class evergreen.io.**StreamServer**
Base class for writing servers which use a stream-like transport.

bind(*address*)
Bind the server to the specified address. The address will be different depending on the particular server implementation.

serve([*backlog*])
Start listening for incoming connections. The caller will block until the server is stopped with a call to close.

close()
Close the server. All active connections are also closed.

handle_connection(*connection*)
Abstract method which subclasses need to implement in order handle incoming connections.

connections
List of currently active connections.

class evergreen.io.**StreamConnection**
Base class representing a connection handled by a *StreamServer*.

server
Reference to the *StreamServer* which accepted the connection.

close()
Close the connection.

_set_accepted(*server*)
Internal API method: sets the connection state to accepted.

exception evergreen.io.**StreamError**
Base class for stream related errors.

class evergreen.io.**TCPClient**
Class representing a TCP client.

sockname
Returns the local address.

peername
Returns the remote endpoint's address.

connect(*target*[, *source_address*])
Start an outgoing connection towards the specified target. If *source_address* is specified the socket will be bound to it, else the system will pick an appropriate one.

class evergreen.io.**TCPServer**
Class representing a TCP server.

sockname
Returns the local address where the server is listening.

class evergreen.io.**TCPConnection**
Class representing a TCP connection handled by a TCP server.

sockname
Returns the local address.

peername
Returns the remote endpoint's address.

exception evergreen.io.**TCPError**
Class for representing all TCP related errors.

class evergreen.io.**PipeClient**
Class representing a named pipe client.

connect(*target*)
Connects to the specified named pipe.

class evergreen.io.**PipeServer**
Class representing a named pipe server.

pipename
Returns the name of the pipe to which the server is bound.

class evergreen.io.**PipeConnection**
Class representing a connection to a named pipe server.

open(*fd*)
Opens the given file descriptor (or Windows HANDLE) and allows for using it as a regular pipe stream.

class evergreen.io.**PipeStream**
Class representing generic pipe stream. Currently it can only be used to open an arbitrary file descriptor such as */dev/net/tun* and treat it as a pipe stream.

exception evergreen.io.**PipeError**
Class for representing all Pipe related errors.

class evergreen.io.**TTYStream**(*fd*, *readable*)
Class representing a TTY stream. The specified *fd* is opened as a TTY, so make sure it's already a TTY. If you plan on reading from this stream specify *readable* as True.

winsize
Returns the current window size.

set_raw_mode(*enable*)
If set to *True*, sets this TTY handle in raw mode.

class evergreen.io.**StdinStream**
Convenience class to use stdin as a cooperative stream.

class evergreen.io.**StdoutStream**
Convenience class to use stdout as a cooperative stream.

class evergreen.io.**StderrStream**
Convenience class to use stderr as a cooperative stream.

exception evergreen.io.**TTYError**
Class for representing all TTY related errors.

class evergreen.io.**UDPEndpoint**
Class representing a UDP endpoint. UDP endpoints can be both servers and clients.

exception evergreen.io.**UDPError**
Class for representing all UDP related errors.

sockname
Returns the local address.

bind(*address*)
Bind the endpoint to the specified IPv4 or IPv6 address.

send(*data*, *address*)
Write data to the specified address.

receive()
Wait for incoming data. The return value is a tuple consisting of the received data and the source IP address where it was received from.

close()
Close the stream. All further operations will raise an exception.

errno.**errorcode**()
Mapping between errno codes and their names.

errno.**strerror**(*errorno*)
Returns error string representation.

errno.**EXXX**
All error number constants are defined in the errno submodule.

Monkeypatching support

Evergreen supports monkeypatching certain modules to make them cooperative. By monkeypatching, some modules which block are replaced with API compatible versions which cooperatively yield.

While evergreen doesn't encourage this practice, because it leads to unexpected behavior depending on how modules are used, limited support is provided for some common modules:

- socket
- select
- time

This module provides several functions to monkeypatch modules.

evergreen.patcher.**patch**(***modules*)
Globally patches the given modules to make them cooperative. Example:

```
import evergreen.patcher

patcher.patch(socket=True, select=True, time=True)
```

evergreen.patcher.**is_patched**(*module*)
Returns true if the given module is currently monkeypatched, false otherwise. *Module* can be either the module object or its name.

evergreen.patcher.**import_patched**(*module*, ***additional_modules*)
Imports a module and ensures that it uses the cooperative versions of the specified modules, or all of the supported ones in case no *additional_modules* is supplied. Example:

```
import evergreen.patcher

SocketServer = patcher.import_patched('SocketServer')
```

evergreen.patcher.**original**(*module*)
Returns an un-patched version of a module.

Extending evergreen

evergreen provides a friendly way to import extensions that users may implement. This mechanism has been borrowed from Flask :-)

There are no rules in how modules should be named, but if your module happens to be named *evergreen-foo* you can import the module like this

```
from evergreen.ext import foo
```

instead of doing

```
import evergreen_foo
```

Standard library compatible cooperative modules

This module contains several API compatible and cooperative modules with some standard Python library modules.

Only a subset of those is provided, and it's **not** evergreen's intention to eventually provide alternatives to every module in the standard library.

Provided modules:

- socket
- select
- time

```
from evergreen.lib import socket

# use the socket as it was a 'normal' one
...
```

3.3 Examples

3.3.1 Crawler

A cooperative i/o library wouldn't be such without a "crawler" example:

```
from evergreen import futures, patcher
urllib2 = patcher.import_patched('urllib2')

urls = ["http://google.com",
        "http://yahoo.com",
        "http://bing.com"]


def fetch(url):
    return urllib2.urlopen(url).read()

executor = futures.TaskPoolExecutor(100)
for body in executor.map(fetch, urls):
    print("got body {}".format(len(body)))
```

3.3.2 Echo server

```
import sys

import evergreen
from evergreen.io import tcp

loop = evergreen.EventLoop()


class EchoServer(tcp.TCPServer):

    @evergreen.task
    def handle_connection(self, connection):
        print('client connected from {}'.format(connection.peername))
        while True:
            data = connection.read_until('\n')
            if not data:
                break
            connection.write(data)
        print('connection closed')


def main():
    server = EchoServer()
    port = int(sys.argv[1] if len(sys.argv) > 1 else 1234)
    server.bind(('0.0.0.0', port))
    print ('listening on {}'.format(server.sockname))
    server.serve()


evergreen.spawn(main)
loop.run()
```

CHAPTER 4

Indices and tables

- *genindex*
- *modindex*
- *search*

e