base.py

```python
# AsynQueue:
# Asynchronous task queueing based on the Twisted framework, with task
# prioritization and a powerful worker interface.
#
# Copyright (C) 2006-2007, 2015 by Edwin A. Suominen,
# http://edsuom.com/AsynQueue
#
# See edsuom.com for API documentation as well as information about
# Ed's background and other projects, software and otherwise.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the
# License. You may obtain a copy of the License at
#
#   http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing,
# software distributed under the License is distributed on an "AS
# IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either
# express or implied. See the License for the specific language
# governing permissions and limitations under the License.

"""
The L{TaskQueue} and its immediate support staff.
"""

import heapq, logging
from contextlib import contextmanager

from zope.interface import implements
from twisted.python.failure import Failure
from twisted.internet import reactor, interfaces, defer
# Use C Deferreds if possible, for efficiency
try:
    from twisted.internet import cdefer
except:
    pass
else:
    defer.Deferred = cdefer.Deferred

import errors, tasks, iteration
from info import Info


class Priority(object):
    """
    I provide simple, asynchronous access to a priority heap.
    """
    def __init__(self):
        self.heap = []
        self.pendingGetCalls = []

    def shutdown(self):
```

```
        """
        Shuts down the priority heap, firing errbacks of the deferreds of
        any L{get} requests that will not be fulfilled.
        """
        if self.pendingGetCalls:
            msg = "No more items forthcoming"
            theFailure = Failure(errors.QueueRunError(msg))
            for d in self.pendingGetCalls:
                d.errback(theFailure)

    def get(self):
        """
        Gets an item with the highest priority (lowest value) from the
        heap, returning a C{Deferred} that fires when the item becomes
        available.
        """
        if len(self.heap):
            d = defer.succeed(heapq.heappop(self.heap))
        else:
            d = defer.Deferred()
            self.pendingGetCalls.insert(0, d)
        return d

    def put(self, item):
        """
        Adds the supplied I{item} to the heap, firing the oldest getter
        deferred if any L{get} calls are pending.
        """
        heapq.heappush(self.heap, item)
        if len(self.pendingGetCalls):
            d = self.pendingGetCalls.pop()
            d.callback(heapq.heappop(self.heap))

    def cancel(self, selector):
        """
        Removes all pending items from the heap that the supplied I{selector}
        function selects. The function must take an item as its sole argument
        and return C{True} if it selects the item for queue removal.
        """
        for item in self.heap:
            if selector(item):
                self.heap.remove(item)
        # Fix up the possibly mangled heap list
        heapq.heapify(self.heap)


class LoadInfoProducer(object):
    """
    Produces task queue loading information.

    I produce information about the current load of a
    L{TaskQueue}. The information consists of the number of tasks
    currently queued, and is written as a single integer to my
    consumers as a single integer whenever a task is queued up and
    again when it is completed.
```

```python
    @ivar consumer: A list of the consumers for whom I'm producing
      information.
    """
    implements(interfaces.IPushProducer)

    def __init__(self):
        self.queued = 0
        self.producing = True
        self.consumers = []

    def registerConsumer(self, consumer):
        """
        Call this with a provider of C{IConsumer} and I'll produce for it
        in addition to any others already registered with me.
        """
        try:
            consumer.registerProducer(self, True)
        except RuntimeError:
            # I must have already been registered with this consumer
            return
        self.consumers.append(consumer)

    def shutdown(self):
        """
        Stop me from producing and unregister any consumers I have.
        """
        self.producing = False
        for consumer in self.consumers:
            consumer.unregisterProducer()

    def oneLess(self):
        self._update(-1)

    def oneMore(self):
        self._update(+1)

    def _update(self, increment):
        self.queued += increment
        if self.queued < 0:
            self.queued = 0
        if self.producing:
            for consumer in self.consumers:
                consumer.write(self.queued)

    #--- IPushProducer implementation -----------------------------------------

    def pauseProducing(self):
        self.producing = False

    def resumeProducing(self):
        self.producing = True

    def stopProducing(self):
        self.shutdown()
```

```python
class Queue(object):
    """
    I am an asynchronous priority queue. Construct me with an item
    handler that can be called with each item from the queue and
    call L{shutdown} when I'm done.

    Put anything you like in the queue except C{None} objects. Those
    are reserved for triggering a queue shutdown.

    You will probably use a L{TaskQueue} instead of me directly.
    """
    def __init__(self, handler, timeout=None):
        """
        Starts up a deferred-yielding loop that runs the queue. This
        method can only be run once, by the constructor upon
        instantiation.
        """
        @defer.inlineCallbacks
        def runner():
            while True:
                self._runFlag = True
                item = yield self.heap.get()
                if item is None:
                    break
                self.loadInfoProducer.oneLess()
                yield self.handler(item)
            # Clean up after the loop exits
            result = yield self.handler.shutdown(timeout)
            self.heap.shutdown()
            defer.returnValue(result)

        if self.isRunning():
            raise errors.QueueRunError(
                "Startup only occurs upon instantiation")
        self.heap = Priority()
        self.handler = handler
        self.loadInfoProducer = LoadInfoProducer()
        # Start my loop
        self._d = runner()

    def isRunning(self):
        """
        Returns C{True} if the queue is running, C{False} otherwise.
        """
        return getattr(self, '_runFlag', False)

    def shutdown(self):
        """
        Initiates a shutdown of the queue by putting a lowest-possible
        priority C{None} object onto the priority heap.

        @return: A deferred that fires when my handler has shut down,
          with a list of any items left unhandled in the queue.
        """
        if self.isRunning():
            self.heap.put(None)
```

```
            d = self._d
        else:
            d = defer.succeed([])
        self._runFlag = False
        return d

    def put(self, item):
        """
        Put an item into my heap
        """
        self.heap.put(item)
        self.loadInfoProducer.oneMore()

    def cancelSeries(self, series):
        """
        Cancels any pending items in the specified I{series},
        unceremoniously removing them from the queue.
        """
        self.heap.cancel(
            lambda item: getattr(item, 'series', None) == series)

    def cancelAll(self):
        """
        Cancels all pending items, unceremoniously removing them from the
        queue.
        """
        self.heap.cancel(lambda item: True)

    def subscribe(self, consumer):
        """
        Subscribes the supplied provider of C{IConsumer} to updates on the
        number of items queued whenever it goes up or down.

        The figure is the integer number of calls currently pending,
        i.e., the number of items that have been queued up but haven't
        yet been handled plus those that have been called but haven't
        yet returned a result.
        """
        if interfaces.IConsumer.providedBy(consumer):
            self.loadInfoProducer.registerConsumer(consumer)
        else:
            raise errors.ImplementationError(
                "Object doesn't provide the IConsumer interface")


class TaskQueue(object):
    """
    I am a task queue for dispatching arbitrary callables to be run by
    one or more worker objects.

    You can construct me with one or more workers, or you can attach
    them later with L{attachWorker}, in which you'll receive an ID
    that you can use to detach the worker.

    @keyword timeout: A number of seconds after which to more
      drastically terminate my workers if they haven't gracefully shut
```

```
      down by that point.

    @keyword warn: Merely log errors via an 'asynqueue' logger with
      ERROR events. The default is to stop the reactor and raise an
      exception when an error is encountered.

    @keyword verbose: Provide detailed info about tasks that are logged
      or result in errors.

    @keyword spew: Log all task calls, whether they raise errors or
      not. Can generate huge logs! Implies verbose=True.

    @keyword returnFailure: If a task raises an exception, call its
      errback with a Failure. Default is to either log an error (if
      'warn' is set) or stop the queue.
    """
    def __init__(self, *args, **kw):
        # Options
        self.timeout = kw.get('timeout', None)
        self.warn = kw.get('warn', False)
        self.spew = kw.get('spew', False)
        self.returnFailure = kw.get('returnFailure', False)
        if self.warn or self.spew:
            self.logger = logging.getLogger('asynqueue')
            if self.spew:
                self.logger.setLevel(logging.INFO)
        if kw.get('verbose', False) or self.spew:
            self.info = Info(remember=True)
        # Bookkeeping
        self.tasksBeingRetried = []
        # Tools
        self.th = tasks.TaskHandler()
        self.taskFactory = tasks.TaskFactory()
        # Attach any workers provided now
        for worker in args:
            self.attachWorker(worker)
        # Start things up with my very own live asynchronous queue
        # using a TaskHandler
        self.q = Queue(self.th, self.timeout)
        # Provide for a clean shutdown
        self._triggerID = reactor.addSystemEventTrigger(
            'before', 'shutdown', self.shutdown)

    def __len__(self):
        """
        Returns my "length" as the number of workers currently at my
        disposal.
        """
        return len(self.th.roster())

    def isRunning(self):
        """
        Returns C{True} if my task handler and queue are running,
        C{False} otherwise.
        """
        return self.th.isRunning and self.q.isRunning()
```

```python
    def shutdown(self):
        """
        You must call this and wait for the C{Deferred} it returns when
        you're done with me. Calls L{Queue.shutdown}, among other
        things.
        """
        def cleanup(stuff):
            if hasattr(self, '_triggerID'):
                reactor.removeSystemEventTrigger(self._triggerID)
                del self._triggerID
            if hasattr(self, '_dc') and self._dc.active():
                self._dc.cancel()
            for dc in tasks.Task.timeoutCalls:
                if dc.active():
                    dc.cancel()
            return stuff

        if not self.isRunning():
            return defer.succeed(None)
        return self.th.shutdown().addCallback(
            lambda _: self.q.shutdown()).addCallback(cleanup)

    def attachWorker(self, worker):
        """
        Registers a new provider of C{IWorker} for working on tasks from
        the queue.

        @return: A C{Deferred} that fires with an integer ID uniquely
        identifying the worker.

        @see: L{tasks.TaskHandler.hire}.
        """
        return self.th.hire(worker)

    def _getWorkerID(self, workerOrID):
        if workerOrID in self.th.workers:
            return workerOrID
        for thisID, worker in self.th.workers.iteritems():
            if worker == workerOrID:
                return thisID

    def detachWorker(self, workerOrID, reassign=False, crash=False):
        """
        Detaches and terminates the worker supplied or specified by its
        ID, returning a C{Deferred} that fires with a list of tasks
        left unfinished by the worker.

        If I{reassign} is set C{True}, any tasks left unfinished by
        the worker are put into new assignments for other or future
        workers. Otherwise, they are returned via the deferred's
        callback.

        @see: L{tasks.TaskHandler.terminate}.
        """
        ID = self._getWorkerID(workerOrID)
```

```
        if ID is None:
            return defer.succeed([])
        if crash:
            return self.th.terminate(ID, crash=True, reassign=reassign)
        return self.th.terminate(ID, self.timeout, reassign=reassign)

    def qualifyWorker(self, worker, series):
        """
        Adds the specified I{series} to the qualifications of the supplied
        I{worker}.
        """
        if series not in worker.iQualified:
            worker.iQualified.append(series)
            self.th.assignmentFactory.request(worker, series)

    def workers(self, ID=None):
        """
        Returns the worker object specified by I{ID}, or C{None} if that
        worker is not employed with me.

        If no ID is specified, a list of the workers currently
        attached, in no particular order, will be returned instead.
        """
        if ID is None:
            return self.th.workers.values()
        return self.th.workers.get(ID, None)

    def taskDone(self, statusResult, task, **kw):
        """
        Processes the status/result tuple from a worker running a
        task. You don't need to call this directly.

          - B{e}: An B{e}xception was raised; the result is a
            pretty-printed traceback string. If the keyword
            'returnFailure' was set for my constructor or this task, I
            will make it into a failure so the task's errback is
            triggered.

          - B{r}: The task B{r}an fine, the result is the return value
            of the call.

          - B{i}: Ran fine, but the result was an B{i}terable other
            than a standard Python one. So my result is a Deferator
            that yields deferreds to the worker's iterations, or, if
            you specified a consumer, an IterationProducer registered
            with the consumer that needs to get running to write
            iterations to it. If the iterator was empty, the result is
            just an empty list.

          - B{c}: Ran fine (on an AMP server), but the result is being
            B{c}hunked because it was too big for a single return
            value. So the result is a deferred that will eventually
            fire with the result after all the chunks of the return
            value have arrived and been magically pieced together and
            unpickled.
```

```
      - B{t}: The task B{t}imed out. I'll try to re-run it, once.

      - B{n}: The task returned [n]othing, as will I.

      - B{d}: The task B{d}idn't run, probably because there was a
        disconnection. I'll re-run it.
    """
    @contextmanager
    def taskInfo(ID):
        if hasattr(self, 'logger'):
            if ID:
                taskInfo = self.info.aboutCall(ID)
                self.info.forgetID(ID)
                yield taskInfo
            else:
                # Why do logging without an info object?
                yield "TASK"
        else:
            yield None
        if self.spew:
            taskInfo += " -> {}".format(result)
            self.logger.info(taskInfo)

    def retryTask():
        self.tasksBeingRetried.append(task)
        task.rush()
        self.q.put(task)
        return task.reset().addCallback(self.taskDone, task, **kw)

    status, result = statusResult
    # Deal with any info for this task call
    with taskInfo(kw.get('ID', None)) as prefix:
        if status == 'e':
            # There was an error...
            if prefix:
                # ...log it
                self.logger.error("{}: {}".format(prefix, result))
            if kw.get('rf', False):
                # ...just return the Failure
                result = Failure(errors.WorkerError(result))
            elif not self.warn:
                # ...stop the reactor
                import sys
                sys.stderr.write("\nERROR: {}".format(result))
                print "\nShutting down in one second!\n"
                self._dc = reactor.callLater(1.0, reactor.stop)
            return result
    if status in "rc":
        # A plain result, or a deferred to a chunked one
        return result
    if status == 'i':
        # An iteration, possibly an IterationConsumer that we need
        # to run now
        if kw.get('consumer', None):
            if hasattr(result, 'run'):
                return result.run()
```

```
                # Nothing to produce from an empty iterator, consider
                # the iterations "done" right away.
                return defer.succeed(None)
            return result
        if status == 't':
            # Timed out. Try again, once.
            if task in self.tasksBeingRetried:
                self.tasksBeingRetried.remove(task)
                return Failure(
                    errors.TimeoutError(
                        "Timed out after two tries, gave up"))
            return retryTask()
        if status == 'n':
            # None object
            return
        if status == 'd':
            # Didn't run. Try again, hopefully with a different worker.
            return retryTask()
        return Failure(
            errors.WorkerError("Unknown status '{}'".format(status)))

    def newTask(self, func, args, kw):
        """
        Make a new L{tasks.Task} object from a func-args-kw combo. You
        won't call this directly.
        """
        if not self.isRunning():
            text = Info().setCall(func, args, kw).aboutCall()
            raise errors.QueueRunError(text)
        # Some parameters just for me, not for the task
        niceness = kw.pop('niceness',       0     )
        series   = kw.pop('series',         None  )
        timeout  = kw.pop('timeout',        None  )
        doLast   = kw.pop('doLast',         False )
        rf       = kw.pop('returnFailure', self.returnFailure )
        task = self.taskFactory.new(func, args, kw, niceness, series, timeout)
        # Workers have to honor the consumer and doNext keywords, too
        if kw.get('doNext', False):
            task.rush()
        elif doLast:
            task.relax()
        kwTD = { 'rf': rf, 'consumer': kw.get('consumer', None) }
        if hasattr(self, 'info'):
            kwTD['ID'] = self.info.setCall(func, args, kw).ID
        task.addCallback(self.taskDone, task, **kwTD)
        return task

    def call(self, func, *args, **kw):
        """
        Queues up a function call.

        Puts a call to I{func} with any supplied arguments and
        keywords into the pipeline. This is perhaps the B{single most
        important method} of the AsynQueue API.

        Scheduling of the call is impacted by the I{niceness} keyword
```

```
        that can be included in addition to any keywords for the
        call. As with UNIX niceness, the value should be an integer
        where 0 is normal scheduling, negative numbers are higher
        priority, and positive numbers are lower priority.

        Tasks in a series of tasks all having niceness N+10 are
        dequeued and run at approximately half the rate of tasks in
        another series with niceness N.

        @return: A C{Deferred} to the eventual result of the call when
          it is eventually pulled from the pipeline and run.

        @keyword niceness: Scheduling niceness, an integer between -20
          and 20, with lower numbers having higher scheduling priority
          as in UNIX C{nice} and C{renice}.

        @keyword series: A hashable object uniquely identifying a
          series for this task. Tasks of multiple different series
          will be run with somewhat concurrent scheduling between the
          series even if they are dumped into the queue in big
          batches, whereas tasks within a single series will always
          run in sequence (except for niceness adjustments).

        @keyword doNext: Set C{True} to assign highest possible
          priority, even higher than a deeply queued task with
          niceness = -20.

        @keyword doLast: Set C{True} to assign priority so low that
          any other-priority task gets run before this one, no matter
          how long this task has been queued up.

        @keyword timeout: A timeout interval in seconds from when a
          worker gets a task assignment for the call, after which the
          call will be retried.

        @keyword consumer: An implementor of L{interfaces.IConsumer}
          that will receive iterations if the result of the call is an
          interator. In such case, the returned result is a deferred
          that fires (with a reference to the consumer) when the
          iterations have all been produced.

        @keyword returnFailure: If a task raises an exception, call
          its errback with a Failure. Default is to either log an
          error (if 'warn' is set) or stop the queue.
        """
        task = self.newTask(func, args, kw)
        self.q.put(task)
        return task.d

    def update(self, func, *args, **kw):
        """
        Sets an update task from I{func} with any supplied arguments and
        keywords to be run directly on all current and future
        workers. Returns a C{Deferred} to the result of the call on
        all current workers, though there is no mechanism for
        obtaining such results for new hires, so it's probably best
```

```
    not to rely too much on them.

    The updates are run directly via L{tasks.TaskHandler.update},
    not through the queue. Because of the disconnect between
    queued and direct calls, it is likely but not guaranteed that
    any jobs you have queued when this method is called will run
    on a particular worker B{after} this update is run. Wait for
    the C{Deferred} from this method to fire before queuing any
    jobs that need the update to be in place before running.

    If you don't want the task saved to the update list, but only
    run on the workers currently attached, set the I{ephemeral}
    keyword C{True}.
    """
    if 'consumer' in kw:
        raise ValueError(
            "Can't supply a consumer for an update because there "+\
            "may be multiple iteration producers")
    ephemeral = kw.pop('ephemeral', False)
    task = self.newTask(func, args, kw)
    return self.th.update(task, ephemeral)
```

errors.py

```
# AsynQueue:
# Asynchronous task queueing based on the Twisted framework, with task
# prioritization and a powerful worker interface.
#
# Copyright (C) 2006-2007, 2015 by Edwin A. Suominen,
# http://edsuom.com/AsynQueue
#
# See edsuom.com for API documentation as well as information about
# Ed's background and other projects, software and otherwise.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the
# License. You may obtain a copy of the License at
#
#   http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing,
# software distributed under the License is distributed on an "AS
# IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either
# express or implied. See the License for the specific language
# governing permissions and limitations under the License.

"""
Custom Exceptions.
"""

from zope.interface import Invalid


class QueueRunError(Exception):
    """
    An attempt was made to dispatch tasks when the dispatcher isn't running.
    """

class ImplementationError(Exception):
    """
    There was a problem implementing the required interface.
    """

class NotReadyError(ImplementationError):
    """
    You shouldn't have called yet!
    """

class InvariantError(Invalid):
    """
    An invariant of the IWorker provider did not meet requirements.
    """
    def __repr__(self):
        return "InvariantError(%r)" % self.args
```

```python
class TimeoutError(Exception):
    """
    A local worker took too long to provide a result.
    """

class WorkerError(Exception):
    """
    A worker ran into an exception trying to run a task.
    """

class ThreadError(Exception):
    """
    A function call in a thread raised an exception.
    """
```

info.py

```python
# AsynQueue:
# Asynchronous task queueing based on the Twisted framework, with task
# prioritization and a powerful worker interface.
#
# Copyright (C) 2006-2007, 2015 by Edwin A. Suominen,
# http://edsuom.com/AsynQueue
#
# See edsuom.com for API documentation as well as information about
# Ed's background and other projects, software and otherwise.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the
# License. You may obtain a copy of the License at
#
#   http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing,
# software distributed under the License is distributed on an "AS
# IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either
# express or implied. See the License for the specific language
# governing permissions and limitations under the License.

"""
Information about callables and what happens to them.

My L{Info} object is a flexible info provider, with several methods
for offering you text info about a call. Construct it with a function
object and any args and keywords if you want the info to include that
particular function call, or you can set it (and change it) later with
L{Info.setCall}.

The L{Info.nn} method may be useful in the still non-working L{wire}
module for separating a call into a namespace and attribute name. It's
not used yet, though, there or anywhere else in C{AsynQueue}.

Another useful object for development are my L{showResult} and
L{whichThread} decorator functions, which you can use together.
"""

import cPickle as pickle
import sys, traceback, inspect, threading
from contextlib import contextmanager

from twisted.internet import defer
from twisted.python import reflect


def hashIt(*args):
    """
    Returns a pretty much unique 32-bit hash for pretty much any
    python object.
    """
    total = 0L
```

```
    for x in args:
        if isinstance(x, dict):
            for k, key in enumerate(sorted(x.keys())):
                total += hashIt(k, key, x[key])
        elif isinstance(x, (list, tuple)):
            for k, value in enumerate(x):
                total += hashIt(k, value)
        else:
            try:
                thisHash = hash(x)
            except:
                try:
                    thisHash = hash(pickle.dumps(x))
                except:
                    thisHash = 0
            total += thisHash
    return hash(total)


SR_STUFF = [0, None, False]
def showResult(f):
    """
    Use as a decorator to print info about the function and its
    result. Follows deferred results.
    """
    def substitute(self, *args, **kw):
        def msg(result, callInfo):
            resultInfo = str(result)
            if len(callInfo) + len(resultInfo) > 70:
                callInfo += "\n"
            print "\n{} -> {}".format(callInfo, resultInfo)
            return result

        SR_STUFF[0] += 1
        callInfo = "{:03d}: {}".format(
            SR_STUFF[0],
            SR_STUFF[1].setCall(
                instance=self, args=args, kw=kw).aboutCall())
        result = f(self, *args, **kw)
        if isinstance(result, defer.Deferred):
            return result.addBoth(msg, callInfo)
        return msg(result, callInfo)

    SR_STUFF[1] = Info(whichThread=SR_STUFF[2]).setCall(f)
    substitute.func_name = f.func_name
    return substitute

def whichThread(f):
    """
    Use as a decorator (after showResult) to include the current
    thread in the info about the function.
    """
    SR_STUFF[2] = True
    return f
```

```python
class Converter(object):
    """
    I provide a bunch of methods for converting objects.
    """
    def strToFQN(self, x):
        """
        Returns the fully qualified name of the supplied string if it can
        be imported and then reflected back into the FQN, or
        C{None} if not.
        """
        try:
            obj = reflect.namedObject(x)
            fqn = reflect.fullyQualifiedName(obj)
        except:
            return
        return fqn

    def objToPickle(self, x):
        """
        Returns a string of the pickled object or C{None} if it couldn't
        be pickled and unpickled back again.
        """
        try:
            xp = pickle.dumps(x)
            pickle.loads(xp)
        except:
            return
        return xp

    def objToFQN(self, x):
        """
        Returns the fully qualified name of the supplied object if it can
        be reflected into an FQN and back again, or C{None} if
        not.
        """
        try:
            fqn = reflect.fullyQualifiedName(x)
            reflect.namedObject(fqn)
        except:
            return
        return fqn

    def processObject(self, x):
        """
        Attempts to convert the supplied object to a pickle and, failing
        that, to a fully qualified name.
        """
        pickled = self.objToPickle(x)
        if pickled:
            return pickled
        return self.objToFQN(x)


class InfoHolder(object):
    """
    An instance of me is yielded by L{Info.context}, for you to call
```

```
    about info concerning a particular saved function call.
    """
    def __init__(self, info, ID):
        self.info = info
        self.ID = ID
    def getInfo(self, name):
        return self.info.getInfo(self.ID, name)
    def nn(self, raw=False):
        return self.info.nn(self.ID, raw)
    def aboutCall(self):
        return self.info.aboutCall(self.ID)
    def aboutException(self, exception=None):
        return self.info.aboutCall(self.ID, exception)
    def aboutFailure(self, failureObj):
        return self.info.aboutFailure(failureObj, self.ID)


class Info(object):
    """
    Provides detailed info about function/method calls.

    I provide text (picklable) info about a call. Construct me with a
    function object and any args and keywords if you want the info to
    include that particular function call, or you can set it (and
    change it) later with L{setCall}.
    """
    def __init__(self, remember=False, whichThread=False):
        self.cv = Converter()
        self.lastMetaArgs = None
        if remember:
            self.pastInfo = {}
        self.whichThread = whichThread

    def setCall(self, *metaArgs, **kw):
        """
        Sets my current f-args-kw tuple, returning a reference to myself
        to allow easy method chaining.

        The function I{f} must be an actual callable object if you
        want to use L{nn}. Otherwise it can also be a string depicting
        a callable.

        You can specify I{args} with a second argument (as a list or
        tuple), and I{kw} with a third argument (as a C{dict}). If you are
        only specifying a single arg, you can just provide it as your
        second argument to this method call without wrapping it in a
        list or tuple. I try to be flexible.

        If you've set a function name and want to add a sequence of
        args or a dict of keywords, you can do it by supplying the
        I{args} or I{kw} keywords. You can also set a class instance
        at that time with the I{instance} keyword.

        To sum up, here are the numbers of arguments you can provide:

          1. A single argument with a callable object or string
```

```
            depicting a callable.

          2. Two arguments: the callable I{f} plus a single
            argument or list of arguments to I{f}.

          3. Three arguments: I{f}, I{args}, and a dict
            of keywords for the callable.

        @param metaArgs: 1-3 arguments as specified above.

        @keyword args: A sequence of arguments for the callable I{f}
          or one previously set.

        @keyword kw: A dict of keywords for the callable I{f} or one
          previously set.

        @keyword instance: An instance of a class of which the
          callable I{f} is a method.
        """
        if metaArgs:
            if metaArgs == self.lastMetaArgs and not hasattr(self, 'pastInfo'):
                # We called this already with the same metaArgs and
                # without any pastInfo to reckon with, so there's
                # nothing to do.
                return self
            # Starting over with a new f
            callDict = {'f': metaArgs[0], 'fs': self._funcText(metaArgs[0])}
            args = metaArgs[1] if len(metaArgs) > 1 else []
            if not isinstance(args, (tuple, list)):
                args = [args]
            callDict['args'] = args
            callDict['kw'] = metaArgs[2] if len(metaArgs) > 2 else {}
            callDict['instance'] = None
            if self.whichThread:
                callDict['thread'] = threading.current_thread().name
            self.callDict = callDict
        elif hasattr(self, 'callDict'):
            # Adding to an existing f
            for name in ('args', 'kw', 'instance'):
                if name in kw:
                    self.callDict[name] = kw[name]
        else:
            raise ValueError(
                "You must supply at least a new function/string "+\
                "or keywords adding args, kw to a previously set one")
        if hasattr(self, 'currentID'):
            del self.currentID
        # Runs the property getter
        self.ID
        if metaArgs:
            # Save metaArgs to ignore repeated calls with the same metaArgs
            self.lastMetaArgs = metaArgs
        return self

    @property
    def ID(self):
```

```
        """
        Returns a unique ID for my current callable.
        """
        if hasattr(self, 'currentID'):
            return self.currentID
        if hasattr(self, 'callDict'):
            thisID = hashIt(self.callDict)
            if hasattr(self, 'pastInfo'):
                self.pastInfo[thisID] = {'callDict': self.callDict}
        else:
            thisID = None
        self.currentID = thisID
        return thisID

    def forgetID(self, ID):
        """
        Use this whenever info won't be needed anymore for the specified
        call ID, to avoid memory leaks.
        """
        if ID in getattr(self, 'pastInfo', {}):
            del self.pastInfo[ID]

    @contextmanager
    def context(self, *metaArgs, **kw):
        """
        Context manager for setting and getting call info.

        Call this context manager method with info about a particular call
        (same format as L{setCall} uses) and it yields an
        L{InfoHolder} object keyed to that call. It lets you get info
        about the call inside the context, without worrying about the
        ID or calling L{forgetID}, even after I have been used for
        other calls outside the context.
        """
        if not hasattr(self, 'pastInfo'):
            raise Exception(
                "Can't use a context manager without saving call info")
        ID = self.setCall(*metaArgs, **kw).ID
        yield InfoHolder(self, ID)
        self.forgetID(ID)

    def getInfo(self, ID, name, nowForget=False):
        """
        Provides info about a call.

        If the supplied name is 'callDict', returns the f-args-kw-instance
        dict for my current callable. The value of I{ID} is ignored in
        such case. Otherwise, returns the named information attribute
        for the previous call identified with the supplied ID.

        @param ID: ID of a previous call, ignored if I{name} is 'callDict'

        @param name: The name of the particular type of info requested.

        @type name: str
```

```
        @param nowForget: Set C{True} to remove any reference to this
          ID or callDict after the info is obtained.

        """
        def getCallDict():
            if hasattr(self, 'callDict'):
                result = self.callDict
                if nowForget:
                    del self.callDict
            else:
                result = None
            return result

        if hasattr(self, 'pastInfo'):
            if ID is None and name == 'callDict':
                return getCallDict()
            if ID in self.pastInfo:
                x = self.pastInfo[ID]
                if nowForget:
                    del self.pastInfo[ID]
                return x.get(name, None)
            return None
        if name == 'callDict':
            return getCallDict()
        return None

    def saveInfo(self, name, x, ID=None):
        if ID is None:
            ID = self.ID
        if hasattr(self, 'pastInfo'):
            self.pastInfo.setdefault(ID, {})[name] = x
        return x

    def nn(self, ID=None, raw=False):
        """
        Namespace-name parser

        For my current callable or a previous one identified by I{ID},
        returns a 3-tuple namespace-ID-name combination suitable for
        sending to a process worker via L{pickle}.

        The first element: If the callable is a method, a pickled or
        fully qualified name (FQN) version of its parent object. This
        is C{None} if the callable is a standalone function.

        The second element: If the callable is a method, the
        callable's name as an attribute of the parent object. If it's
        a standalone function, the pickled or FQN version. If nothing
        works, this element will be C{None} along with the first one.

        @param ID: Previous callable

        @type ID: int

        @param raw: Set C{True} to return the raw parent (or
          function) object instead of a pickle or FQN. All the type
```

```
          checking and round-trip testing still will be done.
        """
        if ID:
            pastInfo = self.getInfo(ID, 'wireVersion')
            if pastInfo:
                return pastInfo
        result = None, None
        callDict = self.getInfo(ID, 'callDict')
        if not callDict:
            # No callable set
            return result
        func = callDict['f']
        if isinstance(func, (str, unicode)):
            # A callable defined as a string can only be a function
            # name, return its FQN or None if that doesn't work
            result = None, self.cv.strToFQN(func)
        elif inspect.ismethod(func):
            # It's a method, so get its parent
            parent = getattr(func, 'im_self', None)
            if parent:
                processed = self.cv.processObject(parent)
                if processed:
                    # Pickle or FQN of parent, method name
                    if raw:
                        processed = parent
                    result = processed, func.__name__
        if result == (None, None):
            # Couldn't get or process a parent, try processing the
            # callable itself
            processed = self.cv.processObject(func)
            if processed:
                # None, pickle or FQN of callable
                if raw:
                    processed = func
                result = None, processed
        return self.saveInfo('wireVersion', result, ID)

    def aboutCall(self, ID=None, nowForget=False):
        """
        Returns an informative string describing my current function call
        or a previous one identified by ID.
        """
        if ID:
            pastInfo = self.getInfo(ID, 'aboutCall', nowForget)
            if pastInfo:
                return pastInfo
        callDict = self.getInfo(ID, 'callDict')
        if not callDict:
            return ""
        func, args, kw = [callDict[x] for x in ('f', 'args', 'kw')]
        instance = callDict.get('instance', None)
        text = repr(instance) + "." if instance else ""
        text += self._funcText(func) + "("
        if args:
            text += ", ".join([str(x) for x in args])
```

```
        for name, value in kw.iteritems():
            text += ", {}={}".format(name, value)
        text += ")"
        if 'thread' in callDict:
            text += " <Thread: {}>".format(callDict['thread'])
        return self.saveInfo('aboutCall', text, ID)

    def aboutException(self, ID=None, exception=None, nowForget=False):
        """
        Returns an informative string describing an exception raised from
        my function call or a previous one identified by ID, or one
        you supply (as an instance, not a class).
        """
        if ID:
            pastInfo = self.getInfo(ID, 'aboutException', nowForget)
            if pastInfo:
                return pastInfo
        if exception:
            lineList = ["Exception '{}'".format(repr(exception))]
        else:
            stuff = sys.exc_info()
            lineList = ["Exception '{}'".format(stuff[1])]
        callInfo = self.aboutCall()
        if callInfo:
            lineList.append(
                " doing call '{}':".format(callInfo))
        self._divider(lineList)
        if not exception:
            lineList.append("".join(traceback.format_tb(stuff[2])))
            del stuff
        text = self._formatList(lineList)
        return self.saveInfo('aboutException', text, ID)

    def aboutFailure(self, failureObj, ID=None, nowForget=False):
        """
        Returns an informative string describing a Twisted failure raised
        from my function call or a previous one identified by ID. You
        can use this as an errback.
        """
        if ID:
            pastInfo = self.getInfo(ID, 'aboutFailure', nowForget)
            if pastInfo:
                return pastInfo
        lineList = ["Failure '{}'".format(failureObj.getErrorMessage())]
        callInfo = self.aboutCall()
        if callInfo:
            lineList.append(
                " doing call '{}':".format(callInfo))
        self._divider(lineList)
        lineList.append(failureObj.getTraceback(detail='verbose'))
        text = self._formatList(lineList)
        return self.saveInfo('aboutFailure', text, ID)

    def _divider(self, lineList):
        N_dashes = max([len(x) for x in lineList]) + 1
        if N_dashes > 79:
```

```
            N_dashes = 79
        lineList.append("-" * N_dashes)

    def _formatList(self, lineList):
        lines = []
        for line in lineList:
            newLines = line.split(':')
            for newLine in newLines:
                for reallyNewLine in newLine.split('\\n'):
                    lines.append(reallyNewLine)
        return "\n".join(lines)

    def _funcText(self, func):
        if isinstance(func, (str, unicode)):
            return func
        if callable(func):
            text = getattr(func, '__name__', None)
            if text:
                return text
            if inspect.ismethod(func):
                text = "{}.{}".format(func.im_self, text)
                return text
        try:
            func = str(func)
        except:
            func = repr(func)
        return "{}[Not Callable!]".format(func)
```

interfaces.py

```python
# AsynQueue:
# Asynchronous task queueing based on the Twisted framework, with task
# prioritization and a powerful worker interface.
#
# Copyright (C) 2006-2007, 2015 by Edwin A. Suominen,
# http://edsuom.com/AsynQueue
#
# See edsuom.com for API documentation as well as information about
# Ed's background and other projects, software and otherwise.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the
# License. You may obtain a copy of the License at
#
#   http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing,
# software distributed under the License is distributed on an "AS
# IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either
# express or implied. See the License for the specific language
# governing permissions and limitations under the License.

"""
The worker interface.
"""

from zope.interface import invariant, Interface, Attribute
from twisted.internet.interfaces import IConsumer

import errors


class IWorker(Interface):
    """
    Provided by worker objects that can have tasks assigned to them for
    processing.

    All worker objects are considered qualified to run tasks of the
    default C{None} series. To indicate that subclasses or subclass
    instances are qualified to run tasks of user-defined series in
    addition to the default, the hashable object that identifies the
    additional series must be listed in the C{cQualified} or C{iQualified}
    class or instance attributes, respectively.
    """
    cQualified = Attribute(
        """
        A class-attribute list containing all series for which all instances
        of the subclass are qualified to run tasks.
        """)

    iQualified = Attribute(
```

```
    """
    An instance-attribute list containing all series for which the
    subclass instance is qualified to run tasks.
    """)

def _check_qualifications(ob):
    """
    Qualification attributes must be present as lists.
    """
    for attrName in ('cQualified', 'iQualified'):
        x = getattr(ob, attrName, None)
        if not isinstance(x, list):
            raise errors.InvariantError(ob)
invariant(_check_qualifications)

def setResignator(callableObject):
    """
    Registers the supplied I{callableObject} to be called if the
    worker deems it necessary to resign, e.g., a remote connection
    has been lost.
    """

def run(task):
    """
    Adds the task represented by the specified I{task} object to the list
    of tasks pending for this worker, to be run however and whenever
    the worker sees fit. However, workers are expected to run
    highest-priority tasks before anything else they have lined up in
    their mini-queues.

    Unless the worker is constructed with a C{raw=True} keyword or
    the task includes C{raw=True}, an iterator resulting from the
    task is converted into an instance of
    L{iteration.Deferator}. The underlying iteration (possibly
    across a pipe or wire) must be handled transparently to the
    user. If the task has a I{consumer} keyword set to an
    implementor of C{IConsumer}, an L{iteration.IterationProducer}
    coupled to that consumer will be the end result instead.

    Make sure that any callbacks you add to the task's internal
    deferred object C{task.d} return the callback argument. Otherwise,
    the result of your task will be lost in the callback chain.

    @return: A C{Deferred} that fires when the worker is ready to
      be assigned another task.
    """

def stop():
    """
    Attempts to gracefully shut down the worker, returning a
    C{Deferred} that fires when the worker is done with all
    assigned tasks and will not cause any errors if the reactor is
    stopped or its object is deleted.
```

```
        The C{Deferred} returned by your implementation of this method
        must not fire until B{after} the results of all pending tasks
        have been obtained. Thus the deferred must be chained to each
        C{task.d} somehow.

        Make sure that any callbacks you add to the task's internal
        deferred object C{task.d} return the callback argument. Otherwise,
        the result of your task will be lost in the callback chain.
        """

    def crash():
        """
        Takes drastic action to shut down the worker, rudely and
        synchronously.

        @return: A list of I{task} objects, one for each task left
          uncompleted. You shouldn't have to call this method if no
          tasks are left pending; the L{stop} method should be enough
          in that case.

        """
```

iteration.py

```
# AsynQueue:
# Asynchronous task queueing based on the Twisted framework, with task
# prioritization and a powerful worker interface.
#
# Copyright (C) 2006-2007, 2015 by Edwin A. Suominen,
# http://edsuom.com/AsynQueue
#
# See edsuom.com for API documentation as well as information about
# Ed's background and other projects, software and otherwise.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the
# License. You may obtain a copy of the License at
#
#   http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing,
# software distributed under the License is distributed on an "AS
# IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either
# express or implied. See the License for the specific language
# governing permissions and limitations under the License.

"""
Iteration, Twisted style!

This module contains multitudes; consider it carefully. It provides a
way of dealing with iterations asynchronously. The L{Deferator} yields
C{Deferred} objects, an asynchronous version of an iterator.

Even cooler is L{IterationProducer}, which I{produces} iterations to
an implementor of C{twisted.internet.interfaces.IConsumer}. You can
make one out of an iterator with L{iteratorToProducer}.

The L{Delay} object is also very useful, both as a
Deferred-after-delay callable and a way to get a C{Deferred} that
fires when an event occurs. This is the key to getting
L{process.ProcessWorker} to work so nicely via Python's standard
multiprocessing module.
"""

import time, inspect

from zope.interface import implements
from twisted.internet import defer, reactor
from twisted.python.failure import Failure
from twisted.internet.interfaces import IPushProducer, IConsumer

import errors
# Almost everybody in the package imports this module, so it can
# import very little from the package.


def deferToDelay(delay):
```

```
    """
    Returns a C{Deferred} that fires after the specified I{delay} (in
    seconds).
    """
    return Delay(delay)()

def isIterator(x):
    """
    @see: L{Deferator.isIterator}
    """
    return Deferator.isIterator(x)


class Delay(object):
    """
    I let you delay things and wait for things that may take a while,
    in Twisted fashion.

    Perhaps a bit more suited to the L{util} module, but that would
    require this module to import it, and it imports this module.

    With event delays of 100 ms to 1 second (in
    L{process.ProcessWorker}, setting I{backoff} to 1.10 seems more
    efficient than 1.05 or 1.20, with the (initial) I{interval} of 50
    ms. However, you may want to tune things for your application and
    system.

    @ivar interval: The initial event-checking interval, in seconds.
    @type interval: float
    @ivar backoff: The backoff exponent.
    @type backoff: float
    """
    interval = 0.001
    backoff = 1.10

    def __init__(self, interval=None, backoff=None, timeout=None):
        if interval:
            self.interval = interval
        if backoff:
            self.backoff = backoff
        if timeout:
            self.timeout = timeout
        if self.backoff < 1.0 or self.backoff > 1.3:
            raise ValueError(
                "Unworkable backoff {:f}, keep it in 1.0-1.3 range".format(
                    self.backoff))

    def __call__(self, delay=None):
        """
        Returns a C{Deferred} that fires after my default delay interval
        or one you specify. You can have it fire in the next reactor
        iteration by setting I{delay} to zero (not C{None}, as that
        will use the default delay instead).

        The default interval is 10ms unless you override that in by
        setting my I{interval} attribute to something else.
```

```python
        """
        if delay is None:
            delay = self.interval
        d = defer.Deferred()
        reactor.callLater(delay, d.callback, None)
        return d

    @defer.inlineCallbacks
    def untilEvent(self, eventChecker):
        """
        Returns a C{Deferred} that fires when a call to the supplied
        event-checking callable returns an affirmative (not C{None},
        C{False}, etc.) result, or until the optional timeout limit is
        reached. The result of the C{Deferred} is C{True} if the event
        actually happened, or C{False} if a timeout occurred.

        The event checker should B{not} return a C{Deferred}. I call
        the event checker less and less frequently as the wait goes
        on, depending on the backoff exponent (default is C{1.04}).

        @param eventChecker: A no-argument callable that returns an
          immediate boolean value indicating if an event occurred.
        """
        if not callable(eventChecker):
            raise TypeError("You must supply a callable event checker")

        # We do two very quick checks before entering the delay loop,
        # to minimize overhead when dealing with very fast events.
        if eventChecker():
            # First, if the event has happened right away, we don't
            # enter the loop at all.
            defer.returnValue(True)
        else:
            t0 = time.time()
            interval = self.interval
            # Second, we "wait" until the very next reactor iteration
            # to do another check, as the first and possibly only loop
            # iteration.
            yield self(0)
            while True:
                if eventChecker():
                    defer.returnValue(True)
                    break
                if hasattr(self, 'timeout') and time.time()-t0 > self.timeout:
                    defer.returnValue(False)
                    break
                # No response yet, check again after the poll interval,
                # which increases exponentially so that each incremental
                # delay is somewhat proportional to the amount of time
                # spent waiting thus far.
                yield self(interval)
                interval *= self.backoff


class Deferator(object):
    """
```

B{Defer}red-yielding iterB{ator}.

Use an instance of me in place of a task result that is an iterable other than one of Python's built-in containers (C{list}, C{dict}, etc.). I yield deferreds to the next iteration of the result and maintain an internal deferred that fires when the iterations are done or terminated cleanly with a call to my L{stop} method. The deferred fires with C{True} if the iterations were completed, or C{False} if not, i.e., a stop was done.

Access the done-iterating deferred via my I{d} attribute. I also try to provide access to its methods attributes and attributes as if they were my own.

When the deferred from my first L{next} call fires, with the first iteration of the underlying (possibly remote) iterable, you can call L{next} again to get a deferred to the next one, and so on, until I raise L{StopIteration} just like a regular iterable.

B{NOTE}: There are two very important rules. First, you B{must} wrap my iteration in a L{defer.inlineCallbacks} loop or otherwise wait for each yielded deferred to fire before asking for the next one. Second, you must call the L{stop} method of the Deferator (or the deferreds it yields) before doing a C{break} or C{return} to prematurely terminate the loop.

Good behavior looks something like this::

```
@defer.inlineCallbacks
def printItems(self, ID):
    for d in Deferator("remoteIterator", getMore, ID):
        listItem = yield d
        print listItem
        if listItem == "Danger Will Robinson":
            d.stop()
            # You still have to break out of the loop after calling
            # the deferator's stop method
            return
```

Instantiate me with a string representation of the underlying iterable (or the object itself, if it's handy) and a function (along with any args and kw) that returns a deferred to a 3-tuple containing (1) the next value yielded from the task result, (2) a Bool indicating if this value is valid or a bogus first one from an empty iterator, and (3) a Bool indicating whether there are more iterations left.

This requires your get-more function to be one step ahead somehow, returning C{False} as its status indicator when the I{next} call would raise L{StopIteration}. Use L{Prefetcherator.getNext} after setting the prefetcherator up with a suitable iterator or next-item callable.

The object (or string representation) isn't strictly needed; it's for informative purposes in case an error gets propagated back somewhere. You can cheat and just use C{None} for the first

```
    constructor argument. Or you can supply a Prefetcherator as the
    first and sole argument, or an iterator for which a
    L{Prefetcherator} will be constructed internally.
    """
    builtIns = (
        str, unicode,
        list, tuple, bytearray, buffer, dict, set, frozenset)

    @classmethod
    def isIterator(cls, obj):
        """
        Tells you if I{obj} is a suitable iterator.

        @return: C{True} if the object is an iterator suitable for use
          with me, C{False} otherwise.
        """
        if isinstance(obj, cls.builtIns):
            return False
        if inspect.isgenerator(obj) or inspect.isgeneratorfunction(obj):
            return True
        for attrName in ('__iter__', 'next'):
            if not callable(getattr(obj, attrName, None)):
                return False
        return True

    def __init__(self, objOrRep, *args, **kw):
        self.d = defer.Deferred()
        self.moreLeft = True
        if isinstance(objOrRep, (str, unicode)):
            # Use supplied string representation
            self.representation = objOrRep.strip('<>')
        else:
            # Use repr of the object itself
            self.representation = repr(objOrRep)
        if args:
            # A callTuple was supplied
            self.callTuple = args[0], args[1:], kw
            return
        if isinstance(objOrRep, Prefetcherator):
            # A Prefetcherator was supplied
            self.callTuple = (objOrRep.getNext, [], {})
            return
        if self.isIterator(objOrRep):
            # An iterator was supplied for which I will make my own
            # Prefetcherator
            pf = Prefetcherator()
            if pf.setup(objOrRep):
                self.callTuple = (pf.getNext, [], {})
                return
        # Nothing worked; make me equivalent to an empty iterator
        self.moreLeft = False
        self.representation = repr([])
        # The non-existent iteration was "complete" since nothing was
        # terminated prematurely.
        self._callback(True)
```

```
    def __repr__(self):
        """
        We all want to be nicely represented.
        """
        return "<Deferator wrapping of\n  <{}>,\nat 0x{}>".format(
            self.representation, format(id(self), '012x'))

    def __getattr__(self, name):
        """
        Provides access to my done-iterating deferred's attributes as if
        they were my own.
        """
        if name == 'd':
            raise AttributeError("No internal deferred is defined!")
        return getattr(self.d, name)

    def _callback(self, wasCompleteIteration):
        if not self.d.called:
            self.d.callback(wasCompleteIteration)

    # Iterator implementation
    #--------------------------------------------------------------------------

    def __iter__(self):
        """
        One of two methods needed for me to act like an iterator.
        """
        return self

    def next(self):
        """
        One of two methods needed for me to act like an iterator. The
        result (unless C{StopIteration} is raised) is a C{Deferred} to
        the underlying iterator's next value, not the value itself.

        Cool, huh? It took a B{lot} of work to figure this out. You
        have to play nice, too, calling this method again only after
        the C{Deferred} fires and calling L{stop} if you want to break
        out of the iterations early.
        """
        def gotNext(result):
            value, isValid, self.moreLeft = result
            return value

        if self.moreLeft:
            if hasattr(self, 'dIterate') and not self.dIterate.called:
                raise errors.NotReadyError(
                    "You didn't wait for the last deferred to fire!")
            f, args, kw = self.callTuple
            self.dIterate = f(*args, **kw).addCallback(gotNext)
            self.dIterate.stop = self.stop
            return self.dIterate
        if hasattr(self, 'dIterate'):
            del self.dIterate
        self._callback(True)
        raise StopIteration
```

```
    def stop(self):
        """
        You must call this to cleanly break out of a loop of my iterations.

        Not part of the official iterator implementation, but
        necessary for a Twisted way of iterating. You need a way of
        letting whatever is producing the iterations know that there
        won't be any more of them.

        For convenience, each C{Deferred} that I yield while iterating
        has a reference to this method via its own C{stop} attribute.
        """
        self.moreLeft = False
        self._callback(False)


class Prefetcherator(object):
    """
    I prefetch iterations from an iterator, providing a L{getNext}
    method suitable for L{Deferator}.

    You can supply an ID for me, purely to provide a more informative
    representation and something you can retrieve via my I{ID}
    attribute.
    """
    __slots__ = ['ID', 'nextCallTuple', 'lastFetch']

    def __init__(self, ID=None):
        self.ID = ID

    def __repr__(self):
        """
        An informative representation. You may thank me for having this
        during development.
        """
        text = "<Prefetcherator instance '{}'".format(self.ID)
        if self.isBusy():
            text += "\n with nextCallTuple '{}'>".format(
                repr(self.nextCallTuple))
        else:
            text += ">"
        return text

    def isBusy(self):
        """
        @return: C{True} if I've been set up with a call to L{setup} and
        am still running whatever iteration that involved.
        """
        return hasattr(self, 'nextCallTuple')

    def setup(self, *args, **kw):
        """
        Sets me up with an attempt at an initial prefetch.

        Set me up with a new iterator, or the callable for an
```

```
    iterator-like-object, along with any args or keywords. Does a
    first prefetch.

    @return: A C{Deferred} that fires with C{True} if all goes
      well or C{False} otherwise.
    """
    def parseArgs():
        if not args:
            return False
        if Deferator.isIterator(args[0]):
            iterator = args[0]
            if not hasattr(iterator, 'next'):
                iterator = iter(iterator)
            if not hasattr(iterator, 'next'):
                raise AttributeError(
                    "Can't get a nextCallTuple from so-called "+\
                    "iterator '{}'".format(repr(args[0])))
            self.nextCallTuple = (iterator.next, [], {})
            return True
        if callable(args[0]):
            self.nextCallTuple = (args[0], args[1:], kw)
            return True
        return False

    def done(result):
        self.lastFetch = result
        return result[1]

    if self.isBusy() or not parseArgs():
        return defer.succeed(False)
    return self._tryNext().addCallback(done)

def _tryNext(self):
    """
    Returns a deferred that fires with the value from my
    I{nextCallTuple} along with a C{bool} indicating if it's a
    valid value. Deletes the I{nextValue} reference after it
    returns with a failure.
    """
    def done(value):
        return value, True
    def oops(failureObj):
        del self.nextCallTuple
        return None, False
    if not hasattr(self, 'nextCallTuple'):
        return defer.succeed((None, False))
    f, args, kw = self.nextCallTuple
    return defer.maybeDeferred(f, *args, **kw).addCallbacks(done, oops)

def getNext(self):
    """
    Prefetch analog to C{next} on a regular iterator.

    Gets the next value from my current iterator, or a deferred value
    from my current nextCallTuple, returning it along with a Bool
    indicating if this is a valid value and another one indicating
```

```
        if more values are left.

        Once a prefetch returns a bogus value, the result of this call
        will remain (None, False, False), until a new iterator or
        nextCallable is set.

        Use this method as the callable (second constructor argument)
        of L{Deferator}.
        """
        def done(thisFetch):
            nextIsValid = thisFetch[1]
            if not nextIsValid:
                if hasattr(self, 'lastFetch'):
                    del self.lastFetch
                # This call's value is valid, but there's no more
                return value, True, False
            # This call's value is valid and there is more to come
            result = value, True, True
            self.lastFetch = thisFetch
            return result

        value, isValid = getattr(self, 'lastFetch', (None, False))
        if not isValid:
            # The last prefetch returned a bogus value, and obviously
            # no more are left now.
            return defer.succeed((None, False, False))
        # The prefetch of this call's value was valid, so try a
        # prefetch for a possible next call after this one.
        return self._tryNext().addCallback(done)


class ListConsumer(object):
    """
    Bare-bones iteration consumer.

    I am a bare-bones iteration consumer that accumulates iterations
    as list items, processing each item by running it through
    L{processItem}, which you of course can override in your
    subclass. It can return a C{Deferred}.

    Call my instance to get a C{Deferred} that fires with the
    underlying list when the producer unregisters.

    Set any attributes you want me to have using keywords.
    """
    implements(IConsumer)

    def __init__(self, **kw):
        self.x = {}
        self.count = 0
        self.dPending = []
        self.dp = defer.Deferred()
        for name, value in kw.iteritems():
            setattr(self, name, value)

    def __call__(self):
```

```
        """
        Call to get a (deferred) list of what I consumed.
        """
        def done(null):
            return [self.x[key] for key in sorted(self.x.keys())]

        dList = [d for d in self.dPending if not d.called]
        if hasattr(self, 'dp'):
            # "Wait" for the producer to unregister and fire its
            # deferred
            dList.append(self.dp)
        return defer.DeferredList(dList).addCallback(done)

    def registerProducer(self, producer, streaming):
        """
        L{IConsumer} implementation.
        """
        if hasattr(self, 'producer'):
            raise RuntimeError()
        if not streaming:
            raise TypeError("I only work with push producers")
        # Create a deferred that will be fired when production is done
        self.producer = producer

    def unregisterProducer(self):
        """
        L{IConsumer} implementation.
        """
        if hasattr(self, 'producer'):
            del self.producer
        if hasattr(self, 'dp') and not self.dp.called:
            self.dp.callback(None)

    def write(self, data):
        """
        Records data such that it will be returned in the order written,
        even if L{processItem} takes a different amount of time for
        each.
        """
        def doneProcessing(result, k):
            if hasattr(self, 'producer'):
                self.producer.resumeProducing()
            self.x[k] = result
            self.dPending.remove(d)

        self.count += 1
        self.producer.pauseProducing()
        d = defer.maybeDeferred(self.processItem, data).addCallback(
            doneProcessing, self.count)
        self.dPending.append(d)

    def processItem(self, item):
        """
        Process list items as they come in.

        Override this to do special processing on each item as it arrives,
```

```
        returning the (possibly deferred) value of the item that
        should actually get appended to the list.
        """
        return item


class IterationProducer(object):
    """
    Producer of iterations from a L{Deferator}.

    I am a producer of iterations from a L{Deferator}. Get me running
    with a call to L{run}, which returns a deferred that fires when
    I'm done iterating or when the consumer has stopped me, whichever
    comes first.
    """
    implements(IPushProducer)

    def __init__(self, dr, consumer=None):
        if not isinstance(dr, Deferator):
            raise TypeError("Object {} is not a Deferator".format(repr(dr)))
        self.dr = dr
        self.delay = Delay()
        if consumer is not None:
            self.registerConsumer(consumer)

    def deferUntilDone(self):
        """
        Returns a deferred that fires (with a reference to my consumer)
        when I am done producing iterations.

        """
        d = defer.Deferred().addCallback(lambda _: self.consumer)
        self.dr.chainDeferred(d)
        return d

    def registerConsumer(self, consumer):
        """
        How could we push to a consumer without knowing what it is?
        """
        if not IConsumer.providedBy(consumer):
            raise errors.ImplementationError(
                "Object {} isn't a consumer".format(repr(consumer)))
        try:
            consumer.registerProducer(self, True)
        except RuntimeError:
            # Ignore any exception raised from a consumer already
            # having registered me.
            pass
        self.consumer = consumer

    @defer.inlineCallbacks
    def run(self):
        """
        Produces my iterations, returning a C{Deferred} that fires (with a
        reference to my consumer) when done.
        """
```

```python
        if not hasattr(self, 'consumer'):
            raise AttributeError("Can't run without a consumer registered")
        self.paused = False
        self.running = True
        for d in self.dr:
            # Pause/stop opportunity after the last item write (if
            # any) and before the deferred fires
            if not self.running:
                break
            if self.paused:
                yield self.delay.untilEvent(lambda: not self.paused)
            item = yield d
            # Another pause/stop opportunity before the item write
            if not self.running:
                break
            if self.paused:
                yield self.delay.untilEvent(lambda: not self.paused)
            # Write the item and do the next iteration
            self.consumer.write(item)
        # Done with the iteration, and with producer/consumer
        # interaction
        self.consumer.unregisterProducer()
        defer.returnValue(self.consumer)

    def pauseProducing(self):
        """
        L{IPushProducer} implementation.
        """
        self.paused = True

    def resumeProducing(self):
        """
        L{IPushProducer} implementation.
        """
        self.paused = False

    def stopProducing(self):
        """
        L{IPushProducer} implementation.
        """
        self.running = False
        self.dr.stop()


@defer.inlineCallbacks
def iteratorToProducer(iterator, consumer=None, wrapper=None):
    """
    Makes an iterator into an L{IterationProducer}.

    Converts a possibly slow-running iterator into a Twisted-friendly
    producer, returning a deferred that fires with the producer when
    it's ready. If the the supplied object is not a suitable iterator
    (perhaps empty), the result will be C{None}.

    If a consumer is not supplied, whatever consumer gets this must
    register with the producer by calling its non-interface method
```

```
    L{IterationProducer.registerConsumer} and then its
    L{IterationProducer.run} method to start the iteration/production.

    If you supply a consumer, those two steps will be done
    automatically, and this method will fire with a C{Deferred} that
    fires when the iteration/production is done.
    """
    result = None
    if Deferator.isIterator(iterator):
        pf = Prefetcherator()
        ok = yield pf.setup(iterator)
        if ok:
            if wrapper:
                if callable(wrapper):
                    args = (wrapper, pf.getNext)
                else:
                    result = Failure(TypeError(
                        "Wrapper '{}' is not a callable".format(
                            repr(wrapper))))
            else:
                args = (pf.getNext,)
            dr = Deferator(repr(iterator), *args)
            result = IterationProducer(dr, consumer)
            if consumer:
                yield result.run()
    defer.returnValue(result)
```

misc.py

```
# AsynQueue:
# Asynchronous task queueing based on the Twisted framework, with task
# prioritization and a powerful worker interface.
#
# Copyright (C) 2006-2007, 2015 by Edwin A. Suominen,
# http://edsuom.com/AsynQueue
#
# See edsuom.com for API documentation as well as information about
# Ed's background and other projects, software and otherwise.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the
# License. You may obtain a copy of the License at
#
#   http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing,
# software distributed under the License is distributed on an "AS
# IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either
# express or implied. See the License for the specific language
# governing permissions and limitations under the License.

"""
Miscellaneous stuff that is needed for testing and nothing else.
"""

from asynqueue.wire import WireWorkerUniverse


class TestMethods:
    chars = "abcdefghijklmnopqrst"

    def add(self, x, y):
        return x+y
    def divide(self, x, y):
        return x/y
    def setStuff(self, N1, N2):
        self.stuff = []
        for j in xrange(N1):
            self.stuff.append("".join(
                 [self.chars[k % len(self.chars)]
                  for k in xrange(N2)]))
    def getStuff(self):
        return self.stuff
    def stuffSize(self):
        return len(self.stuff)
    def stufferator(self):
        for chunk in self.stuff:
```

```
            yield chunk
    def blockingTask(self, x, delay):
        import time
        time.sleep(delay)
        return 2*x


class TestUniverse(TestMethods, WireWorkerUniverse):
    pass
```

process.py

```python
# AsynQueue:
# Asynchronous task queueing based on the Twisted framework, with task
# prioritization and a powerful worker interface.
#
# Copyright (C) 2006-2007, 2015 by Edwin A. Suominen,
# http://edsuom.com/AsynQueue
#
# See edsuom.com for API documentation as well as information about
# Ed's background and other projects, software and otherwise.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the
# License. You may obtain a copy of the License at
#
#   http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing,
# software distributed under the License is distributed on an "AS
# IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either
# express or implied. See the License for the specific language
# governing permissions and limitations under the License.

"""
An implementor of the C{IWorker} interface using
(I{gasp! Twisted heresy!}) Python's standard-library
multiprocessing.

@see: L{ProcessQueue} and L{ProcessWorker}.
"""
from time import time
import multiprocessing as mp

from zope.interface import implements
from twisted.internet import defer
from twisted.python.failure import Failure

from base import TaskQueue
from interfaces import IWorker
import errors, util, iteration, info


class ProcessQueue(TaskQueue):
    """
    A L{TaskQueue} that runs tasks on one or more subordinate Python
    processes.

    I am a L{TaskQueue} for dispatching picklable or keyword-supplied
    callables to be run by workers from a pool of I{N} worker
    processes, the number I{N} being specified as the sole argument of
    my constructor.
    """
    @staticmethod
    def cores():
```

```
        """
        @return: The number of CPU cores available.

        @rtype: int
        """
        return ProcessWorker.cores()

    def __init__(self, N, **kw):
        """
        @param N: The number of workers (subordinate Python processes)
        initially in the queue.

        @type N: int

        @param kw: Keywords for the regular L{TaskQueue} constructor,
          except I{callStats}. That enables callStats on each worker.
        """
        callStats = kw.pop('callStats', False)
        TaskQueue.__init__(self, **kw)
        for null in xrange(N):
            worker = ProcessWorker(callStats=callStats)
            self.attachWorker(worker)

    def stats(self):
        """
        Only call this if you've constructed me with
        C{callStats=True}. Returns a C{Deferred} that fires with a
        concatenated list of lists from calls to
        L{ProcessWorker.stats} on each of my workers.
        """
        dList = []
        result = []
        for worker in self.th.roster('process'):
            dList.append(worker.stats().addCallback(result.extend))
        return defer.DeferredList(dList).addCallback(lambda _: result)


class ProcessWorker(object):
    """
    I implement an L{IWorker} that runs tasks in a dedicated worker
    process.

    You can also supply a I{series} keyword containing a list of one
    or more task series that I am qualified to handle.

    B{Note:} Each task's callable is pickled along with its arguments
    to be sent over the interprocess pipe. Thus it must be something
    that can be reconstructed at the process, i.e., a method of an
    instance of a class that is importable by the process. Keep this
    in mind if you get errors like this::

      cPickle.PicklingError:
        Can't pickle <type 'function'>: attribute lookup
        __builtin__.function failed
```

```
Tuning the I{backoff} coefficient
=================================

I did some testing of backoff coefficients with unit tests, where
the reactor wasn't doing much other than running the asynqueue and
Twisted's trial test runner.

With tasks whose completion time range from 0 to 1 second, backoff
of B{1.10} was significantly more efficent than 1.15, even more so
than 1.20.

Backoff of 1.05 was somewhat more efficient than 1.10 with
completion times ranging from 0 to 200 ms: 96.7% process/worker
efficiency vs. 94.5%, with the mean overhead cut in half to around
3.3ms. But that's not the whole story: with a constant completion
time of 100ms, 1.05 was actually B{less} efficient: 95.5% vs. 99%,
and mean overhead B{increased} from around 0.5 ms to around 4 ms!

After 100 ms, there will have been 7 checks, with the check
interval finally doubling to 2.1 ms. Things take off rapidly;
reaching 200 ms takes just another 4 checks, and the interval is
then 3.1 ms.

It's only the calls that take longer that benefit from a smaller
backoff. Going from 1.05 to 1.10 decreased the efficiency of
1-second calls from 97.5% to 94.3% because the overhead doubled
from 25 ms to 60 ms. After so many event checks, the check
interval had increased considerably, enough to add some
significant dead time after the calls were done. By the time the
second is up, there will have been 48 checks and the check
interval will be 0.1 second.

A backoff of 1.10 is a bit magic numerically in that the current
check interval is always about one tenth the amount of time that
has passed since the first check. For a backoff of 1.05, the
interval is half that. (It takes 81 checks to reach 1 second
instead of 48.)

The cost of having too many checks is considerable, though, and
must be worse with a busy reactor, so a backoff less than 1.10
with an (initial) interval of 0.001 isn't recommended. But you
might consider tuning it for your application and system.

@ivar interval: The initial event-checking interval, in seconds.
@type interval: float
@ivar backoff: The backoff exponent.
@type backoff: float

@cvar cQualified: 'process', 'local'
"""
backoff = 1.10 # This is the iteration.Delay default, anyhow

implements(IWorker)
cQualified = ['process', 'local']

@staticmethod
```

```
def cores():
    """
    @return: The number of CPU cores available.

    @rtype: int
    """
    return mp.cpu_count()

def __init__(self, series=[], raw=False, callStats=False):
    """
    Constructs me with a L{ThreadLooper} and an empty list of tasks.

    @param series: A list of one or more task series that this
      particular instance of me is qualified to handle.

    @param raw: Set C{True} if you want raw iterators to be
      returned instead of L{iteration.Deferator} instances. You
      can override this in with the same keyword set C{False} in a
      call.

    @param callStats: Set C{True} if you want stats kept about how
      long the calls took to send and to run on the process. Might
      add significant memory usage and slow things down a bit
      overall if there are lots of calls. Obtain a list of the
      call times here and on the process (2-tuples) with the
      L{stats} method.
    """
    self.tasks = []
    self.iQualified = series
    self.callStats = callStats
    if callStats:
        self.callTimes = []
    # Tools
    self.delay = iteration.Delay(backoff=self.backoff)
    self.dLock = util.DeferredLock()
    # Multiprocessing with (Gasp! Twisted heresy!) standard lib Python
    self.cMain, cProcess = mp.Pipe()
    pu = ProcessUniverse(raw, callStats)
    self.process = mp.Process(target=pu.loop, args=(cProcess,))
    self.process.start()

def _killProcess(self):
    self.cMain.close()
    self.process.terminate()

def next(self, ID):
    """
    Do a next call of the iterator held by my process, over the pipe
    and in Twisted fashion.

    @param ID: A unique identifier for the iterator.
    """
    def gotLock(null):
        self.cMain.send(ID)
        return self.delay.untilEvent(
            self.cMain.poll).addCallback(responseReady)
```

```
        def responseReady(waitStatus):
            if not waitStatus:
                raise errors.TimeoutError(
                    "Timeout waiting for next iteration from process")
            result = self.cMain.recv()
            self.dLock.release()
            if result[1]:
                return result[0]
            return Failure(StopIteration)
        return self.dLock.acquire(vip=True).addCallback(gotLock)

    def stats(self):
        """
        Assembles and returns a (deferred) list of call times. Each list
        item is a 2-tuple. The first element is the time it took to
        get the result from the process after sending the call to it,
        and the second element is how long the process took to run on
        the process.
        """
        def gotProcessTimes(pTimes):
            result = []
            for k, pTime in enumerate(pTimes):
                result.append((self.callTimes[k], pTime))
            return result
        return self.next("").addCallback(gotProcessTimes)

    # Implementation methods
    # ----------------------------------------------------------------------------

    def setResignator(self, callableObject):
        self.dLock.addStopper(callableObject)

    @defer.inlineCallbacks
    def run(self, task):
        """
        Sends the I{task} callable and args, kw to the process (must all
        be picklable) and polls the interprocess connection for a
        result, with exponential backoff.

        I{This actually works very well, O ye Twisted event-purists.}
        """
        if task is None:
            # A termination task, do after pending tasks are done
            yield self.dLock.acquire()
            self.cMain.send(None)
            # Wait (a very short amount of time) for the process loop
            # to exit
            self.process.join()
            self.dLock.release()
        else:
            # A regular task
            self.tasks.append(task)
            yield self.dLock.acquire(task.priority <= -20)
            # Our turn!
            #------------------------------------------------------------------------
            consumer = task.callTuple[2].pop('consumer', None)
```

```
            if self.callStats:
                t0 = time()
            self.cMain.send(task.callTuple)
            # "Wait" here (in Twisted-friendly fashion) for a response
            # from the process
            yield self.delay.untilEvent(self.cMain.poll)
            if self.callStats:
                self.callTimes.append(time()-t0)
            status, result = self.cMain.recv()
            self.dLock.release()
            if status == 'i':
                # What we get from the process is an ID to an iterator
                # it is holding onto, but we need to hook up to it
                # with a Prefetcherator and then make a Deferator,
                # which we will either return to the caller or couple
                # to a consumer provided by the caller.
                ID = result
                pf = iteration.Prefetcherator(ID)
                ok = yield pf.setup(self.next, ID)
                if ok:
                    result = iteration.Deferator(pf)
                    if consumer:
                        result = iteration.IterationProducer(result, consumer)
                else:
                    # The process returned an iterator, but it's not
                    # one I could prefetch from. Probably empty.
                    result = []
            if task in self.tasks:
                self.tasks.remove(task)
            task.callback((status, result))

    def stop(self):
        """
        @return: A C{Deferred} that fires when the task loop has ended and
          its process terminated.
        """
        def terminationTaskDone(null):
            self._killProcess()
            return self.dLock.stop()
        return self.run(None).addCallback(terminationTaskDone)

    def crash(self):
        self._killProcess()
        return self.tasks


class ProcessUniverse(object):
    """
    Each process for a L{ProcessWorker} lives in one of these.
    """
    def __init__(self, raw=False, callStats=False):
        self.iterators = {}
        self.runner = util.CallRunner(raw, callStats)

    def loop(self, connection):
        """
```

```python
        Runs a loop in a dedicated process that waits for new tasks. The
        loop exits when a C{None} object is supplied as a task.

        @param connection: The sub-process end of an interprocess
        connection.
        """
        while True:
            # Wait here for the next call
            callSpec = connection.recv()
            if callSpec is None:
                # Termination call, no reply expected; just exit the
                # loop
                break
            elif isinstance(callSpec, str):
                if callSpec == "":
                    # A blank string is a request for stats. Bummer
                    # that this check runs everytime an iteration
                    # happens, but a simple string comparison
                    # operation shouldn't slow things down measurably
                    connection.send(
                        (getattr(self.runner, 'callTimes'), True))
                else:
                    # A next-iteration call
                    connection.send(self.next(callSpec))
            else:
                # A task call
                status, result = self.runner(callSpec)
                if status == 'i':
                    # Due to the pipe between worker and process, we
                    # hold onto the iterator here and just
                    # return an ID to it
                    ID = str(info.hashIt(result))
                    self.iterators[ID] = result
                    result = ID
                connection.send((status, result))
        # Broken out of loop, ready for the process to end
        connection.close()

    def next(self, ID):
        """
        My L{loop} calls this when the interprocess pipe sends it a string
        identifier for one of the iterators I have pending.

        @return: A 2-tuple with the specified iterator's next value,
          and a bool indicating if the value was valid or a bogus
          C{None} resulting from a C{StopIteration} error or a
          non-existent iterator.
        @rtype: tuple
        """
        if ID in self.iterators:
            try:
                value = self.iterators[ID].next()
            except StopIteration:
                del self.iterators[ID]
                return None, False
            return value, True
```

```python
        return None, False
```

tasks.py

```python
# AsynQueue:
# Asynchronous task queueing based on the Twisted framework, with task
# prioritization and a powerful worker interface.
#
# Copyright (C) 2006-2007, 2015 by Edwin A. Suominen,
# http://edsuom.com/AsynQueue
#
# See edsuom.com for API documentation as well as information about
# Ed's background and other projects, software and otherwise.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the
# License. You may obtain a copy of the License at
#
#   http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing,
# software distributed under the License is distributed on an "AS
# IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either
# express or implied. See the License for the specific language
# governing permissions and limitations under the License.

"""
Task management for the task queue workers. The star of this show
is L{TaskHandler}, which is what turns L{base.PriorityQueue} into a
L{base.TaskQueue}.

Be sure to call the L{TaskQueue.shutdown} method (or that of your
subclass, e.g., L{threads.ThreadQueue}) before you shut down your
Twisted reactor.
"""
from contextlib import contextmanager

from twisted.internet import defer, reactor
# Use C Deferreds if possible, for efficiency
try:
    from twisted.internet import cdefer
except:
    pass
else:
    defer.Deferred = cdefer.Deferred

from info import Info
from interfaces import IWorker
from errors import ImplementationError


class Task(object):
    """
    I represent a task that has been dispatched to a queue for running with a
    given scheduling I{niceness}.

    I generate a C{Deferred} that you fire by calling either my L{callback} or
```

```
    L{errback} with a result or failure, respectively, when the the task is
    finally run and its result is obtained. You can call the deferred's
    versions of those methods directly, but my versions deal with things like
    repeated callbacks, which happen sometimes with task timeouts.

    @ivar d: A C{Deferred} to the eventual result of the task.

    @ivar series: A hashable object identifying the series of which this task
      is a part.

    """
    info = Info()
    timeoutCalls = []

    def __init__(self, f, args, kw, priority, series, timeout=None):
        if not isinstance(args, (tuple, list)):
            raise TypeError("Second argument 'args' isn't a sequence")
        if not isinstance(kw, dict):
            raise TypeError("Third argument 'kw' isn't a dict")
        self.callTuple = (f, args, kw)
        self.priority = priority
        self.series = series
        self.d = defer.Deferred()
        self.callbacks = []
        self.timeout = timeout

    def startTimer(self):
        if self.timeout:
            self.callID = reactor.callLater(self.timeout, self.timedout)
            self.timeoutCalls.append(self.callID)
        else:
            self.callID = None

    def _cancelTimeout(self):
        if getattr(self, 'callID', None):
            if self.callID in self.timeoutCalls:
                self.timeoutCalls.remove(self.callID)
            if self.callID.active():
                self.callID.cancel()
            self.callID = None

    def addCallback(self, f, *args, **kw):
        callTuple = (f, args, kw)
        self.callbacks.append(callTuple)
        self.d.addCallback(f, *args, **kw)

    def callback(self, result):
        self._cancelTimeout()
        if not self.d.called:
            self.d.callback(result)

    def errback(self, result):
        self._cancelTimeout()
        self.d.errback(result)

    def timedout(self):
```

```
        if not self.d.called:
            self.d.callback(
                ('t', "Timeout after {:f} seconds".format(self.timeout)))
        self.callID = None

    def reset(self):
        self.d = defer.Deferred()
        return self.d

    def rush(self):
        self.priority = -1000000

    def relax(self):
        self.priority = 1000000

    def copy(self):
        """
        Returns a functional copy of me with all necessary attributes and
        callbacks pre-added.
        """
        args = list(self.callTuple)
        args.append(self.priority)
        args.append(self.series)
        args.append(self.timeout)
        newTask = Task(*args)
        for f, args, kw in self.callbacks:
            newTask.addCallback(f, *args, **kw)
        return newTask

    def __repr__(self):
        """
        Gives me an informative string representation.
        """
        func = self.callTuple[0]
        args = ", ".join([str(x) for x in self.callTuple[1]])
        kw = "".join(
            [", %s=%s" % item for item in self.callTuple[2].iteritems()])
        if func.__class__.__name__ == "function":
            funcName = func.__name__
        elif callable(func):
            funcName = "%s.%s" % (func.__class__.__name__, func.__name__)
        else:
            funcName = "<worker call> "
            args = ("%s, " % func) + args
        return "Task: %s(%s%s)" % (funcName, args, kw)

    def __cmp__(self, other):
        """
        Numeric comparisons between tasks are based on their priority, with
        higher (lower-numbered) priorities being considered 'less' and thus
        sorted first.

        A task will always have a higher priority, i.e., be comparatively
        I{less}, than a C{None} object, which is used as a shutdown signal
        instead of a task.
        """
```

```
        if other is None:
            return -1
        return cmp(self.priority, other.priority)


class TaskFactory(object):
    """
    I generate L{Task} instances with the right priority setting for effective
    scheduling between tasks in one or more concurrently running task series.
    """
    TaskClass = Task

    def __init__(self, klass=None):
        self.seriesNumbers = {}
        if klass:
            self.TaskClass = klass

    def new(self, func, args, kw, niceness, series=None, timeout=None):
        """
        Call this to obtain a L{Task} instance that will run in the specified
        I{series} at a priority reflecting the specified I{niceness}.

        The equation for priority has been empirically determined as follows::

            p = k * (1 + nn**2)

        where C{k} is an iterator that increments for each new task and C{nn}
        is niceness normalized from -20...+20 to the range 0...2.

        @param func: A callable object to run as the task, the result of which
          will be sent to the callback for the deferred of the task returned by
          this method when it fires.

        @param args: A tuple containing any arguments to include in the call.

        @param kw: A dict containing any keywords to include in the call.

        """
        if not isinstance(niceness, int) or abs(niceness) > 20:
            raise ValueError(
                "Niceness must be an integer between -20 and +20")
        positivized = niceness + 20
        priority = self._serial(series) * (1 + (float(positivized)/10)**2)
        return self.TaskClass(func, args, kw, priority, series, timeout)

    def _serial(self, series):
        """
        Maintains serial numbers for tasks in one or more separate series, such
        that the numbers in each series increment independently except that any
        new series starts at a value greater than the maximum serial number
        currently found in any series.
        """
        if series not in self.seriesNumbers:
            eachSeries = [0] + self.seriesNumbers.values()
            maxCurrentSN = max(eachSeries)
            self.seriesNumbers[series] = maxCurrentSN
```

```
        self.seriesNumbers[series] += 1
        return float(self.seriesNumbers[series])


class Assignment(object):
    """
    I represent the assignment of a single task to whichever worker object
    accepts me. Deep down, my real role is to provide something to fire the
    callback of a deferred with instead of just another deferred.

    @ivar d: A C{Deferred} that is instantiated for a given instance
      of me, which fires when a worker accepts the assigment
      represented by that instance.
    """
    # We go through a lot of these objects and they're small, so let's make
    # them cheap to build
    __slots__ = ['task', 'd']

    def __init__(self, task):
        self.task = task
        self.d = defer.Deferred()

    def accept(self, worker):
        """
        Called when the worker accepts the assignment, firing my
        C{Deferred}.

        @return: Another C{Deferred} that fires when the worker is
          ready to accept B{another} assignment following this one.
        """
        self.d.callback(None)
        self.task.startTimer()
        return worker.run(self.task)


class AssignmentFactory(object):
    """
    I generate L{Assignment} instances for workers to handle
    particular tasks.
    """
    def __init__(self):
        self.waiting = {}
        self.pending = {}
        self.broadcast = {}

    def cancelRequests(self, worker):
        """
        Cancel this worker's assignment requests
        """
        for series, dList in getattr(worker, 'assignments', {}).iteritems():
            requestsWaiting = self.waiting.get(series, [])
            for d in dList:
                if d in requestsWaiting:
                    requestsWaiting.remove(d)

    def request(self, worker, series):
```

```
        """
        Called to request a new assignment in the specified I{series} of tasks
        for the supplied I{worker}.

        When a new assignment in the series is finally ready in the
        queue for this worker, the C{Deferred} for the assignment
        request will fire with an instance of L{Assignment} that has
        been constructed with the task to be assigned.

        If the worker is still gainfully employed when it accepts the
        assignment, and is not just wrapping up its work after having
        been fired, the worker will request another assignment when it
        finishes the task.
        """
        def accept(assignment, d_request):
            worker.assignments[series].remove(d_request)
            if isinstance(assignment, Assignment):
                d = assignment.accept(worker)
                if worker.hired:
                    d.addCallback(lambda _: self.request(worker, series))
                return d

        assignments = getattr(worker, 'assignments', {})
        if self.pending.get(series, []):
            d = defer.succeed(self.pending[series].pop(0))
        else:
            d = defer.Deferred()
            self.waiting.setdefault(series, []).append(d)
        assignments.setdefault(series, []).append(d)
        worker.assignments = assignments
        # The callback is added to the deferred *after* being appended to the
        # worker's assignments list for this series. That way, we know that the
        # callback will be able to remove the deferred even if the deferred
        # fires immediately due to the queue having a surplus of assignments.
        d.addCallback(accept, d)

    def new(self, task):
        """
        Creates and queues a new assignment for the supplied I{task},
        returning a deferred that fires when the assignment has been
        accepted.
        """
        series = task.series
        assignment = Assignment(task)
        if self.waiting.get(series, []):
            self.waiting[series].pop(0).callback(assignment)
        else:
            self.pending.setdefault(series, []).append(assignment)
        return assignment.d


class TaskHandler(object):
    """
    I am a Queue handler that manages one or more providers of
    L{IWorker}.
```

```
    When a new worker is hired with my L{hire} method, I run the
    L{AssignmentFactory.request} method to request that the worker be
    assigned a task from the queue of each task series for which it is
    qualified.

    When the worker finally gets the assignment, it fires the
    L{Assignment} object's internal deferred with a reference to
    itself. That is my cue to have the worker run the assigned task
    and request another assignment of a task in the same series when
    it's done, unless I've terminated the worker in the meantime.

    Each worker object maintains a dictionary of deferreds for each of
    its outstanding assignment requests so that I can cancel them if I
    terminate the worker. Then I can effectively cancel the assignment
    requests by firing the deferreds with fake, no-task
    assignments. See my L{terminate} method.

    @ivar workers: A C{dict} of worker objects that are currently
      employed by me, keyed by a unique integer ID code for each
      worker.
    """
    def __init__(self):
        self.isRunning = True
        self.workers = {}
        self.laborPools = {}
        self.updateTasks = []
        self.assignmentFactory = AssignmentFactory()

    def shutdown(self, timeout=None):
        """
        Shutdown all my workers, then fire them, in turn.

        @return: A C{Deferred} that fires when my entire work force
          has been terminated. The deferred result is a list of all
          tasks, if any, that were left unfinished by the work force.
        """
        def gotResults(results):
            # Why not just return the result? Don't remember.
            unfinishedTasks = []
            for result in results:
                unfinishedTasks.extend(result)
            self.isRunning = False
            return unfinishedTasks

        dList = []
        for workerID in self.workers.keys():
            d = self.terminate(workerID, timeout=timeout)
            dList.append(d)
        return defer.gatherResults(dList).addCallback(gotResults)

    def hire(self, worker):
        """
        Adds a new worker to my work force.

        Makes sure that there is an assignment request queue for each
        task series for which the worker is qualified, then has the
```

```
        new worker request an initial assignment from each queue.

        The method generates an integer ID uniquely identifying the
        worker, and gives the worker an I{ID} attribute with the ID
        for its own reference.

        @return: A C{Deferred} that fires with the worker's ID when it
          has been hired and is ready for assignments.
        """
        @defer.inlineCallbacks
        def readyToRun():
            # Run any relevant update tasks
            for task in self.updateTasks:
                if task.series in qualifications:
                    yield worker.run(task.copy())
            # Now ready for assignments
            for series in qualifications:
                self.assignmentFactory.request(worker, series)
                if series is not None:
                    self.laborPools.setdefault(series, []).append(worker)
            defer.returnValue(workerID)

        if not IWorker.providedBy(worker):
            raise ImplementationError(
                "'%s' doesn't provide the IWorker interface" % worker)
        IWorker.validateInvariants(worker)
        worker.hired = True
        worker.assignments = {}
        # Qualifications
        qualifications = [None]
        if hasattr(worker, 'cQualified'):
            qualifications.extend(worker.cQualified)
        if hasattr(worker, 'iQualified'):
            qualifications.extend(worker.iQualified)
        # ID Badge
        workerID = worker.ID = getattr(self, '_workerCounter', 0) + 1
        self._workerCounter = workerID
        self.workers[workerID] = worker
        # Exit process
        worker.setResignator(
            lambda : self.terminate(worker.ID, crash=True, reassign=True))
        # Welcome aboard! Start by running any update tasks
        return readyToRun()

    def terminate(self, workerID, timeout=None, crash=False, reassign=False):
        """
        Removes a worker from my work force, canceling all of its unfullfilled
        assignment requests back from the queue and then attempting to shut it
        down gracefully via its C{stop} method.

        The I{timeout} keyword can be set to a number of seconds after which
        the worker will be terminated rudely via its C{crash} method if it
        hasn't shut down gracefully by then. If the I{crash} keyword is set
        C{True}, the worker is crashed right away without waiting for it to run
        through its pending tasks.
```

```
        @return: A C{Deferred} that fires when the worker has been
          removed, gracefully or not, with a list of any tasks left
          unfinished and not reassigned.
        """
        def crashTheWorker(worker, d):
            unfinished = worker.crash()
            # Fire deferred with list of unfinished tasks
            if not d.called:
                d.callback(unfinished)

        def stopped(result):
            if callID.active():
                callID.cancel()
                # No tasks left unfinished if deferred fires normally
                return []
            return result

        def reassignTasks(tasks):
            for task in tasks:
                self.assignmentFactory.new(task)
            return []

        worker = self.workers.pop(workerID, None)
        if worker is None:
            return defer.succeed([])
        worker.hired = False
        self.assignmentFactory.cancelRequests(worker)
        for series, workerList in self.laborPools.iteritems():
            if worker in workerList:
                workerList.remove(worker)
        if crash:
            d = defer.succeed(worker.crash())
        else:
            d = worker.stop()
            if timeout:
                callID = reactor.callLater(timeout, crashTheWorker, worker, d)
                d.addCallback(stopped)
            else:
                # No tasks left unfinished if deferred fires without timeout
                d.addCallback(lambda _: [])
        if reassign:
            d.addCallback(reassignTasks)
        return d

    def roster(self, series=None):
        """
        Returns a list of the workers who are qualified to run the
        specified series, or all my workers if no series specified.
        """
        if series is None:
            return self.workers.values()
        return self.laborPools.get(series, [])

    def update(self, task, ephemeral=False):
        """
        Updates my workforce with the supplied task, calling identical
```

```
        copies of each one directly (I have no need of or reference
        to TaskQueue) to all current workers who are qualified to run
        the task. Saves the task for sending to qualified new hires as
        well.

        Returns a deferred that fires when when the task has run on
        all current workers, with a list of the results from each
        run. Note that there is no mechanism for obtaining such
        results for new hires, so it's probably best not to rely too
        much on them.

        If you don't want the task saved to the update list, but only
        run on my current workers, set the ephemeral to C{True}.
        """
        if not ephemeral:
            self.updateTasks.append(task)
        dList = []
        for worker in self.roster(task.series):
            # The "ready for another assignment" deferred that's
            # returned from calling the worker's run method is
            # irrelevant to doing updates outside the queue. We ignore
            # it in favor of a new deferred that fires when all
            # results have been obtained from the workers.
            newTask = task.copy()
            worker.run(newTask)
            dList.append(newTask.d)
        return defer.gatherResults(dList)

    def __call__(self, task):
        """
        Generates a new assignment for the supplied I{task}. This is the
        handler for an item of L{base.Queue}.

        If the worker that runs the task is still working for me when it
        becomes ready for another task following this one, an assignment
        request will be entered for it to obtain another task of the same
        series.

        @return: A C{Deferred} that fires when the task has been
          assigned to a worker and it has accepted the assignment.
        """
        return self.assignmentFactory.new(task)
```

threads.py

```
# AsynQueue:
# Asynchronous task queueing based on the Twisted framework, with task
# prioritization and a powerful worker interface.
#
# Copyright (C) 2006-2007, 2015 by Edwin A. Suominen,
# http://edsuom.com/AsynQueue
#
# See edsuom.com for API documentation as well as information about
# Ed's background and other projects, software and otherwise.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the
# License. You may obtain a copy of the License at
#
#   http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing,
# software distributed under the License is distributed on an "AS
# IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either
# express or implied. See the License for the specific language
# governing permissions and limitations under the License.

"""
L{ThreadQueue}, L{ThreadWorker} and their support staff. Also, a
cool implementation of the oft-desired C{deferToThread}, in
L{ThreadQueue.deferToThread}.
"""

import threading

from zope.interface import implements
from twisted.internet import defer, reactor
from twisted.python.failure import Failure
from twisted.internet.interfaces import IConsumer, IPushProducer


from base import TaskQueue
from interfaces import IWorker
import errors, util, iteration


_DTL = [None]
def deferToThread(*fargs, **kw):
    """
    Module-level function that lets you call a function in a dedicated
    thread and get a C{Deferred} to its result, with no fuss on your
    part. The thread will remain alive and will be used for further
    calls to this function and this function only.

    Call with I{f}, I{*args}, and I{**kw} as usual.

    This is AsynQueue's single-threaded, queued, I{doNext}-able,
    L{iteration.Deferator}-able answer to Twisted's C{deferToThread}.
```

```
    If you expect a deferred iterator as your result (an instance of
    L{iteration.Deferator}), supply an L{IConsumer} implementor via
    the I{consumer} keyword. Each iteration will be written to it, and
    the deferred will fire when the iterations are done. Otherwise,
    the deferred will fire with an L{iteration.Deferator}.

    If you want to kill the dedicated thread, just call this function
    with no arguments, not even a callable object I{f}. A C{Deferred}
    will be returned that fires when the thread is gone.
    """
    if not fargs:
        tl = _DTL[0]
        if tl is None:
            return defer.succeed(None)
        return tl.stop()
    if _DTL[0] is None:
        tl = _DTL[0] = ThreadLooper()
        reactor.addSystemEventTrigger('before', 'shutdown', tl.stop)
    return _DTL[0].deferToThread(*fargs, **kw)


class ThreadQueue(TaskQueue):
    """
    I am a L{TaskQueue} for dispatching arbitrary callables to be run
    by a single worker thread.
    """
    def __init__(self, **kw):
        raw = kw.pop('raw', False)
        TaskQueue.__init__(self, **kw)
        self.worker = ThreadWorker(raw=raw)
        self.d = self.attachWorker(self.worker)

    def deferToThread(self, f, *args, **kw):
        """
        Runs the f-args-kw call in my dedicated worker thread, skipping
        past the queue. As with a regular L{TaskQueue.call}, returns a
        C{Deferred} that fires with the result and deals with
        iterators.
        """
        return util.callAfterDeferred(
            self, 'd', self.worker.t.deferToThread, f, *args, **kw)


class ThreadWorker(object):
    """
    I implement an L{IWorker} that runs tasks in a dedicated worker
    thread.

    @cvar cQualified: Task series all instances of me are qualified to
      perform.

    @ivar iQualified: Task series one instance of me is qualified to
      perform. Usually left blank, unless you want only some workers
      doing certain tasks.
    """
```

```
    implements(IWorker)
    cQualified = ['thread', 'local']

    def __init__(self, series=[], raw=False):
        """
        Constructs me with a L{ThreadLooper} and an empty list of tasks.

        @param series: A list of one or more task series that this
          particular instance of me is qualified to handle.

        @param raw: Set C{True} if you want raw iterators to be
          returned instead of L{iteration.Deferator} instances. You
          can override this in with the same keyword set C{False} in a
          call.
        """
        self.tasks = []
        self.iQualified = series
        self.t = ThreadLooper(raw)

    def setResignator(self, callableObject):
        self.t.dLock.addStopper(callableObject)

    def run(self, task):
        """
        Returns a C{Deferred} that fires only after the threaded call is
        done.

        I do basic FIFO queuing of calls to this method, but priority
        queuing is above my paygrade and you'd best honor my deferred
        and let someone like L{tasks.TaskHandler} only call this
        method when I say I'm ready.

        One simple thing I B{will} do is apply the I{doNext} keyword
        to any task with the highest priority, -20 or lower (for a
        L{base.TaskQueue.call} with its own I{doNext} keyword set). If
        you call this method one task at a time like you're supposed
        to, even that won't make a difference, except that it will cut
        in front of any existing call with I{doNext} set. So use
        judiciously.
        """
        def done(statusResult):
            if task in self.tasks:
                self.tasks.remove(task)
            if statusResult[0] == 'i':
                # What we got is a Deferator, but if a consumer was
                # supplied, we need to couple an IterationProducer to
                # it and fire the task callback with the deferred from
                # running the producer.
                if consumer:
                    dr = statusResult[1]
                    ip = iteration.IterationProducer(dr, consumer)
                    statusResult = ('i', ip)
            task.d.callback(statusResult)

        self.tasks.append(task)
        f, args, kw = task.callTuple
```

```
        consumer = kw.pop('consumer', None)
        if task.priority <= -20:
            kw['doNext'] = True
        return self.t.call(f, *args, **kw).addCallback(done)

    def stop(self):
        """
        @return: A C{Deferred} that fires when the task loop has ended and
          its thread has terminated.
        """
        return self.t.stop()

    def crash(self):
        """
        Unfortunately, a thread can only terminate itself, so calling
        this method only forces firing of the deferred returned from a
        previous call to L{stop} and returns the task that hung the
        thread.
        """
        self.t.stop()
        return self.tasks


class ThreadLooper(object):
    """
    I run function calls in a dedicated thread.

    Each call returns a C{Deferred} to its eventual result, which is a
    2-tuple containing the status of the last call and its result
    according to the format of L{util.CallRunner}.

    If the result is an iterable other than one of Python's built-in
    ones, the C{Deferred} fires with an instance of
    L{iteration.Prefetcherator} instead. Couple it to your own
    deferator to iterate over the underlying iterable running in my
    thread. You can disable this behavior by setting C{raw=True} in
    the constructor, or enable/disable it on an individual call by
    setting raw=True/False.

    @ivar timeout: The wait timeout, which defaults to 60 (one
      minute). This is just how long the thread loop waits before
      checking for a pending deferred and firing it with a timeout
      error. Otherwise, it simply waits another minute, and it can do
      that forever with no problem.
    """
    timeout = 60

    def __init__(self, raw=False):
        # Just a simple attribute to indicate if the thread loop is
        # running, mostly for unit testing
        self.threadRunning = True
        # Tools
        self.runner = util.CallRunner(raw)
        self.dLock = util.DeferredLock()
        self.event = threading.Event()
        self.thread = threading.Thread(name=repr(self), target=self.loop)
```

```
        self.thread.start()

    def loop(self):
        """
        Runs a loop in a dedicated thread that waits for new tasks. The loop
        exits when a C{None} object is supplied as a task.
        """
        def callback(status, result):
            reactor.callFromThread(self.d.callback, (status, result))

        self.threadRunning = True
        while True:
            # Wait here for my main-thread caller to release me for
            # another call
            self.event.wait(self.timeout)
            # For Python 2.7 and above, we could have just done
            # if not self.event.wait(...):
            if not self.event.isSet():
                # Timed out waiting for the next call. If there indeed
                # was one, we need to let the caller know. That
                # shouldn't ever happen, though.
                if hasattr(self, 'd') and not self.d.called:
                    callback('e', "Thread timed out waiting for this call!")
                continue
            if self.callTuple is None:
                # Shutdown was requested
                break
            status, result = self.runner(self.callTuple)
            # We are about to call back the shared deferred, so clear
            # the event to force me to wait for the next call at the
            # top of the loop. The main thread will not set the event
            # again until the callback is done, so this is safe.
            self.event.clear()
            # OK, now call the shared deferred
            callback(status, result)
        # Broken out of loop, the thread now dies
        self.threadRunning = False

    @defer.inlineCallbacks
    def call(self, f, *args, **kw):
        """
        Runs the supplied callable function with any args and keywords in
        a dedicated thread, returning a deferred that fires with a
        status/result tuple.

        Calls are done in the order received, unless you set
        C{doNext=True}.

        Set C{raw=True} to have a raw iterator returned instead of a
        Deferator, or C{raw=False} to have a L{Deferator} returned
        instead of a raw iterator, contrary to the instance-wide
        default set with the constructor keyword 'raw'.
        """
        yield self.dLock.acquire(kw.pop('doNext', False))
        self.callTuple = f, args, kw
        self.d = defer.Deferred()
```

```
        # The callTuple is set for this call along with the deferred
        # to be called back with its result, so release the thread to
        # work on it, firing this deferred's callback with its result.
        self.event.set()
        statusResult = yield self.d
        # The deferred lock is released after the call is done so
        # that another call can proceed. This is NOT the same as
        # the event used as a threading lock. It keeps the main
        # thread from setting that event before the thread loop is
        # ready for that.
        self.dLock.release()
        status, result = statusResult
        if status == 'i':
            ID = str(hash(result))
            pf = iteration.Prefetcherator(ID)
            ok = yield pf.setup(result)
            if ok:
                # OK, we can iterate this
                result = iteration.Deferator(
                    repr(pf), self.deferToThread, pf.getNext)
            else:
                # An iterator, but not one we could prefetch
                # from. Probably empty.
                result = []
        # Not an iterator, at least not one being specially
        # processed; we already have our result
        defer.returnValue((status, result))

    def dr2ip(self, dr, consumer=None):
        """
        Converts a L{Deferator} into an L{IterationProducer}, with a
        consumer registered if you supply one. Then each iteration
        will be written to your consumer, and the deferred returned
        will fire when the iterations are done. Otherwise, the
        deferred will fire with an L{iteration.IterationProducer} and
        you will have to register with and run it yourself.
        """
        ip = iteration.IterationProducer(dr)
        if consumer:
            ip.registerConsumer(consumer)
            return ip.run()
        return ip

    def deferToThread(self, f, *args, **kw):
        """
        My single-threaded, queued, doNext-able, Deferator-able answer to
        Twisted's deferToThread.

        If you expect a deferred iterator as your result (an instance
        of L{iteration.Deferator}), supply an L{IConsumer} implementor
        via the I{consumer} keyword. Each iteration will be written to
        it, and the deferred will fire when the iterations are
        done. Otherwise, the deferred will fire with an
        L{iteration.Deferator}.
        """
        def done(statusResult):
```

```python
            status, result = statusResult
            if status == 'e':
                return Failure(errors.ThreadError(result))
            elif status == 'i':
                if consumer:
                    ip = iteration.IterationProducer(dr, consumer)
                    return ip.run()
                return result
            return result

        consumer = kw.pop('consumer', None)
        return self.call(f, *args, **kw).addCallback(done)

    def stop(self):
        """
        @return: A C{Deferred} that fires when the task loop has ended and
          its thread has terminated.
        """
        if not self.threadRunning:
            return defer.succeed(None)
        # Tell the thread to quit with a null task
        self.callTuple = None
        self.event.set()
        # Now stop the lock
        self.dLock.addStopper(self.thread.join)
        return self.dLock.stop()


class IterationGetter(object):
    """
    Abstract base class for objects that munch data on one end and act
    like iterators to yield it on the other end.

    @see: L{Consumerator} and L{Filerator}.
    """
    class IterationStopper:
        pass

    def __init__(self):
        self.runState = 'init'
        self.d = defer.Deferred()
        # Locks for my iteration-consuming thread, the
        # blocking-iterator thread, and the next-iteration event
        self.cLock = threading.Semaphore()
        self.bLock = threading.Lock()
        self.nLock = threading.Lock()
        # Lock both of my iteration-processing loops until an
        # iteration is received
        self.cLock.acquire()
        self.bLock.acquire()
        # We leave the next-iteration lock unlocked; the
        # iteration-consuming thread will lock it to overwrite the
        # blocking-iterator thread's value of each iteration

    def start(self):
        """
```

```
        Call this when I should start listening for iterations.
        """
        self.thread = threading.Thread(name=repr(self), target=self.loop)
        self.thread.start()

    def loop(self):
        """
        @see: L{Consumerator.loop} and L{Filerator.loop}
        """
        raise NotImplementedError("You must override this in a subclass")

    def deferUntilDone(self):
        """
        Returns a C{Deferred} that fires when I am done iterating.
        """
        d = defer.Deferred()
        self.d.chainDeferred(d)
        return d

    # Iterator implementation -------------------------------------------------
    # Call in its own thread

    def __iter__(self):
        return self

    def next(self):
        # Wait for the next iteration to be produced
        self.bLock.acquire()
        # Get a local reference to the iteration value
        value = self.bIterationValue
        # Now it can be changed, so release my iteration-consuming
        # loop to do so
        self.nLock.release()
        if isinstance(value, self.IterationStopper):
            # We are done iterating. The blocking caller will
            # immediately exit its loop.
            raise StopIteration
        # This is a legit iteration value, return it. Since this
        # method runs in the blocking-iterator thread, it won't
        # get called again until the caller is ready for another
        # iteration.
        return value


class Consumerator(IterationGetter):
    """
    I act like an L{IConsumer} for your Twisted code and an iterator
    for your blocking code running via a L{ThreadWorker}. This is
    handy when you are using a conventional library that relies on an
    iterator as its input::

      def render(request):
          w = png.Writer()
          c = asynqueue.Consumerator()
          c.deferUntilDone().addCallback(lambda _: request.finish())
          p = self.producePixelRows(c)
```

```
            w.write(request, c)
            return server.NOT_DONE_YET

    I work with either an I{IPushProducer} or an I{IPullProducer}. You
    can construct me with an instance of the former and I'll get
    started right away. Otherwise, call my L{registerProducer} method
    with the producer and whether it is streaming (push) or not.

    @ivar runState: 'init', 'running', 'stopping', 'stopped'

    @ivar d: A C{Deferred} that fires when iterations are done.
    """
    implements(IConsumer)

    def __init__(self, producer=None):
        """
        @param producer: The producer for me to register, if you want to
          supply an C{IPushProducer} one on instantiation. Otherwise,
          use L{registerProducer}.
        """
        super(Consumerator, self).__init__()
        self.dLock = util.DeferredLock()
        if producer:
            self.registerProducer(producer, True)

    def loop(self):
        """
        Runs a loop in a dedicated thread that waits for new iterations to
        be produced. When I get an instance of
        L{self.IterationStopper}, the loop exits. I then call my "all
        done" C{Deferred} and delete my reference to the producer.
        """
        self.runState = 'running'
        while True:
            # Wait for an iteration
            self.cLock.acquire()
            # Get a copy of the value
            value = self.cIterationValue
            # Release the consumer interface to write another
            # iteration
            reactor.callFromThread(self.dLock.release)
            # Wait until it's safe to overwrite the blocking-iterator
            # loop's copy
            self.nLock.acquire()
            # Now do so and release it to work on the new copy
            self.bIterationValue = value
            self.bLock.release()
            if isinstance(value, self.IterationStopper):
                # This was the post-iteration signal; this loop is now
                # done.
                break
        # Wait until we know the iteration stopper was noticed and the
        # blocking iterations stopped.
        self.runState = 'stopping'
        self.nLock.acquire()
        reactor.callFromThread(self.d.callback, None)
```

```python
        self.runState = 'stopped'
        reactor.callFromThread(delattr, self, 'producer')

    def stop(self):
        """
        Good manners urge you to call this to cleanly break out of a loop
        of my iterations so that my producer doesn't keep working for
        nothing. Calling this method at the Twisted main-loop level is
        also a fine way to quit producing and iterating when you know
        you're done.

        Not part of the official iterator implementation, but
        useful for a Twisted way of iterating. You need a way of
        letting whatever is producing the iterations know that there
        won't be any more of them.
        """
        if hasattr(self, 'producer'):
            self.producer.stopProducing()
        return self.unregisterProducer()

    # --- IConsumer implementation --------------------------------------------

    def write(self, data):
        """
        The producer calls this with a chunk of I{data}. It goes through
        two stages to emerge from my blocking end as an iteration, via
        L{next}.
        """
        def handleData(null, x):
            self.cIterationValue = x
            if self.runState == 'running':
                # Release my iteration-consuming loop to work on the next
                # iteration value
                self.cLock.release()
                # The producer can and should write another iteration now
                self.producer.resumeProducing()

        if self.streaming and self.runState == 'running':
            # The producer is a IPushProducer, so tell it to hold off
            # on any more iteration values for the moment while
            # everything it's sent (and may yet send) gets processed
            self.producer.pauseProducing()
        # Handle the data in the order received
        return self.dLock.acquire().addCallback(handleData, data)

    def registerProducer(self, producer, streaming):
        """
        L{IConsumer} implementation
        """
        if hasattr(self, 'producer'):
            raise RuntimeError()
        self.producer = producer
        self.streaming = streaming
        if not streaming:
            producer.resumeProducing()
        self.start()
```

```python
    def unregisterProducer(self):
        """
        L{IConsumer} implementation
        """
        if not hasattr(self, 'producer'):
            return defer.succeed(None)
        return self.write(self.IterationStopper())


class Filerator(IterationGetter):
    """
    Stream data to me in one end and I will iterate it out the other.

    Acts like a file handle for writing in one thread (even the main
    one under the Twisted event loop) and an iterator in another
    thread. Hook me up to an L{iteration.Deferator} to stream data
    over a worker interface.

    You must call my L{close} method to stop me from iterating.
    """
    def __init__(self):
        super(Filerator, self).__init__()
        self.itemBuffer = []
        self.start()

    @property
    def closed(self):
        return self.runState == 'stopped'

    def loop(self):
        """
        Runs a loop in a dedicated thread that waits for new iterations to
        be written. When I get an instance of
        L{self.IterationStopper}, the loop exits.
        """
        self.runState = 'running'
        while True:
            # Wait for an iteration
            self.cLock.acquire()
            # Get the oldest value in the FIFO buffer
            value = self.itemBuffer.pop(0)
            # Wait until it's safe to overwrite the blocking-iterator
            # loop's copy
            self.nLock.acquire()
            # Now do so and release it to work on the new copy
            self.bIterationValue = value
            self.bLock.release()
            if isinstance(value, self.IterationStopper):
                # This was the post-iteration signal; this loop is now
                # done.
                break
        # Wait until we know the iteration stopper was noticed and the
        # blocking iterations stopped.
        self.runState = 'stopping'
        self.nLock.acquire()
```

```python
            self.runState = 'stopped'
            reactor.callFromThread(self.d.callback, None)

    def write(self, data):
        """
        This is called with a chunk of I{data}. It goes through two stages
        to emerge from my blocking end as an iteration, via L{next}.
        """
        if self.closed:
            raise ValueError("Closed, not accepting writes")
        self.itemBuffer.append(data)
        if self.runState == 'running':
            # Release my iteration-consuming loop to work on the next
            # iteration value. The cLock object is actually a
            # semaphore, so it's OK if this gets called multiple times
            # before the other loop can acquire it again.
            self.cLock.release()

    def writelines(self, lines):
        """
        Adds a list full of data chunks to my buffer.
        """
        for line in lines:
            self.write(line)

    def flush(self):
        """
        Doesn't do anything, because I am always trying to flush my buffer
        by iterating its contents.
        """

    def close(self):
        """
        Closing me as a "file" tells me that I can stop iterating once the
        buffer is flushed.
        """
        if not self.closed:
            self.write(self.IterationStopper())


class OrderedItemProducer(object):
    """
    Produces blocking iterations in the order they are requested via
    an asynchronous function call.

    I am an implementor of Twisted's C{IPushProducer} interface that
    produces an iteration to a blocking call I{fb} for every time you
    call a non-blocking item-generating function I{fb} via my
    L{produceItem} method. Significantly, the items are buffered as
    needed so that the iterations appear in the order of the calls to
    L{produceItem} that generated them, B{not} necessarily in the
    order in which they are actually generated.

    Start things off by constructing an instance of me, with an
    existing task queue if you have one you want me to use, and then
    running L{start} with your blocking f-args-kw combination. Then
```

```
    call L{produceItem} repeatedly with whatever f-args-kw combination
    results (eventually) in new items to iterate. These calls may
    return deferred results and should not block.

    When you are done having me produce iterations, call
    L{stop}. Whatever loop the blocking-iterator call is in will then
    terminate and function I{fb} should end.

    @ivar i: My L{Consumerator} instance, which acts like an iterator
      for whatever function you supply to L{start}.

    @ivar q: The L{TaskQueue} instance I use, either supplied by you
      during construction or instantiated by me. Either way, you will
      have to call L{TaskQueue.shutdown} on this eventually when
      you're done with the queue.
    """
    implements(IPushProducer)

    def __init__(self):
        self.itemBuffer = {}
        self.k1, self.k2 = 0, 0
        self.seriesID = hash(self)
        self.i = Consumerator(self)
        self.dLock = defer.DeferredLock()
        self.dt = util.DeferredTracker()
        self.dLock.acquire()
        self.produce = True

    def start(self, fb, *args, **kw):
        """
        Starts the blocking function call C{fb(i, *args, **kw)} that
        relies on my L{Consumerator} instance I{i} for iterations, in
        traditional blocking fashion. The function must accept C{i} as
        its first argument, and can also accept further arguments
        C{*args} and keywords C{**kw}, which you can specify in your
        call to L{start}.

        @return: A C{Deferred} that fires when the blocking call has
          started in a dedicated thread. Shouldn't take long at all.
        """
        def started():
            self.dLock.release()
            dStarted.callback(None)
        def runner():
            # This function runs via the queue in my dedicated thread
            reactor.callFromThread(started)
            # The actual blocking call
            result = fb(self.i, *args, **kw)
            reactor.callFromThread(finished, result)
        def finished(result):
            self.dFinished.callback(result)
            self.stopProducing()
        dStarted = defer.Deferred()
        self.dFinished = defer.Deferred()
        thread = threading.Thread(name=repr(self), target=runner)
        thread.start()
```

```
        return dStarted

    def produceItem(self, fp, *args, **kw):
        """
        Runs C{fp(*args, **kw)} to generate an item that I produce as an
        iteration to whatever blocking call was (or will be) set
        running via L{start}.

        While I am running, the returned C{Deferred} fires after the
        call to I{fp} with the item value produced by the call to
        I{f}. You don't need to do anything with these deferreds if
        you don't want to use them for concurrency limiting; they are
        accounted for in L{stop}.

        If my L{stopProducing} method has been called, I no longer
        produce iterations and calls to this method do not run
        I{fp}. The returned C{Deferred} fires immediately with C{None}.
        """
        def gotItem(item):
            # We have a result, but we need to wait our turn to
            # actually produce it
            return self.dLock.acquire().addCallback(gotLock, item)
        def gotLock(lock, item):
            if self.k2 == k1 and self.produce:
                self._writeItem(item)
            elif self.produce is not None:
                self.itemBuffer[k1] = item
                self._flushBuffer()
            self.dLock.release()
            return item
        if self.produce is None:
            return defer.succeed(None)
        k1 = self.k1
        self.k1 += 1
        d = defer.maybeDeferred(fp, *args, **kw).addCallback(gotItem)
        self.dt.put(d)
        return d

    @defer.inlineCallbacks
    def stop(self):
        """
        Call this to indicate that iterations are done. After any pending
        calls from L{produceItem} are done, my L{Consumerator} will
        raise L{StopIteration} for the blocking iteration-caller in
        I{fb} and that function should exit. Whatever value it returns
        will fire the C{Deferred} that is returned here.

        If I{fb} exited early for some reason, the C{Deferred} this
        method returns will have fired already.

        Repeated calls to this method make no sense and will be
        rewarded with deferreds immediately firing with C{None}.
        """
        yield self.dt.deferToAll()
        yield self.dLock.acquire()
        yield self.i.stop()
```

```python
        if hasattr(self, 'dFinished'):
            result = yield self.dFinished
            del self.dFinished
        else:
            result = None
        self.dLock.release()
        defer.returnValue(result)

    def _writeItem(self, item):
        self.i.write(item)
        self.k2 += 1
        self._flushBuffer()

    def _flushBuffer(self):
        if self.k2 in self.itemBuffer:
            item = self.itemBuffer.pop(self.k2)
            # This will result in another call to resumeProducing
            self._writeItem(item)

    def stopProducing(self):
        self.produce = None

    def pauseProducing(self):
        self.produce = False

    def resumeProducing(self):
        self.produce = True
```

util.py

```
# AsynQueue:
# Asynchronous task queueing based on the Twisted framework, with task
# prioritization and a powerful worker interface.
#
# Copyright (C) 2006-2007, 2015 by Edwin A. Suominen,
# http://edsuom.com/AsynQueue
#
# See edsuom.com for API documentation as well as information about
# Ed's background and other projects, software and otherwise.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the
# License. You may obtain a copy of the License at
#
#   http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing,
# software distributed under the License is distributed on an "AS
# IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either
# express or implied. See the License for the specific language
# governing permissions and limitations under the License.

"""
Miscellaneous useful stuff.

L{callAfterDeferred} is a cool little function that looks for a
C{Deferred} as an attribute of some namespace (i.e., object) and does
a call after it fires. L{DeferredTracker} lets you to track and wait
for deferreds without actually having received a reference to
them. L{DeferredLock} lets you shut things down when you get the lock.

L{CallRunner} is used by L{threads.ThreadWorker} and
L{process.ProcessWorker}. You probably won't need to use it yourself,
unless perhaps you come up with an entirely new kind of
L{interfaces.IWorker} implementation.
"""

import os, signal
from time import time
import cPickle as pickle
import cProfile as profile
from contextlib import contextmanager

from twisted.internet import defer, reactor, protocol
from twisted.python.failure import Failure

import errors, info, iteration


def o2p(obj):
    """
    Converts an object into a pickle string or a blank string if an
```

```
    empty container.
    """
    if isinstance(obj, (list, tuple, dict)) and not obj:
        return ""
    return pickle.dumps(obj) #, pickle.HIGHEST_PROTOCOL)

def p2o(pickledString, defaultObj=None):
    """
    Converts a pickle string into its represented object, or into the
    default object you specify if it's a blank string.

    Note that a reference to the default object itself will be
    returned, not a copy of it. So make sure you only supply an empty
    Python primitive, e.g., C{[]}.
    """
    if not pickledString:
        return defaultObj
    return pickle.loads(pickledString)

def callAfterDeferred(namespace, dName, f, *args, **kw):
    """
    Looks for a C{Deferred} I{dName} as an attribute of I{namespace}
    and does the f-args-kw call, chaining its call to the C{Deferred}
    if necessary.

    Note that the original deferred's value is swallowed when it calls
    the new deferred's callback; the original deferred must be for
    signalling readiness only and its return value not relied upon.
    """
    def call(discarded):
        delattr(namespace, dName)
        return defer.maybeDeferred(f, *args, **kw)

    d = getattr(namespace, dName, None)
    if d is None:
        return defer.maybeDeferred(f, *args, **kw)
    if d.called:
        delattr(namespace, dName)
        return defer.maybeDeferred(f, *args, **kw)
    d2 = defer.Deferred().addCallback(call)
    d.chainDeferred(d2)
    return d2

def killProcess(pid):
    """
    Kills the process with the supplied PID, returning a deferred that
    fires when it's no longer running. The return value is C{True} if
    the process was alive and had to be killed, C{False} if it was
    already dead.
    """
    def ready(stdout):
        pt.loseConnection()
        if pidString in stdout:
            os.kill(pid, signal.SIGTERM)
            return True
```

```python
            return False
        pidString = str(pid)
        pp = ProcessProtocol()
        args = ("/bin/ps", '-p', pidString)
        pt = reactor.spawnProcess(pp, args[0], args)
        return pp.d.addCallback(ready)


# For Testing
# --------------------------------------------------------------------------------
def testFunction(x):
    """
    I{For testing only.}
    """
    return 2*x
class TestStuff(object):
    """
    I{For testing only.}
    """
    @staticmethod
    def divide(x, y):
        return x/y
    def add(self, x, y):
        return x+y
    def accumulate(self, y):
        if not hasattr(self, 'x'):
            self.x = 0
        self.x += y
        return self.x
    def setStuff(self, N1, N2):
        self.stuff = ["x"*N1] * N2
        return self
    def stufferator(self):
        for chunk in self.stuff:
            yield chunk
    def blockingTask(self, x, delay):
        import time
        time.sleep(delay)
        return 2*x
# --------------------------------------------------------------------------------

class ProcessProtocol(object):
    """
    I am a simple protocol for spawning a subordinate process.

    @ivar d: A C{Deferred} that fires with an initial chunch of stdout
    from the process.
    """
    def __init__(self, stopper=None):
        self.stopper = lambda x: None if stopper is None else stopper
        self.d = defer.Deferred()

    def makeConnection(self, pt):
        self.pid = pt.pid
```

```
    def childDataReceived(self, childFD, data):
        data = data.strip()
        if childFD == 1:
            if data and not self.d.called:
                self.d.callback(data)
        if childFD == 2:
            print "\nERROR: {}".format(data)
            #self.stopper(self.pid)

    def childConnectionLost(self, childFD):
        self.stopper(self.pid)
    def processExited(self, reason):
        self.stopper(self.pid)
    def processEnded(self, reason):
        self.stopper(self.pid)


class DeferredTracker(object):
    """
    I allow you to track and wait for deferreds without actually having
    received a reference to them.
    """
    def __init__(self):
        self.dList = []

    def put(self, d):
        """
        Put another C{Deferred} in the tracker.
        """
        def transparentCallback(anything):
            if d in self.dList:
                self.dList.remove(d)
            return anything

        if not isinstance(d, defer.Deferred):
            raise TypeError("Object {} is not a deferred".format(repr(d)))
        d.addBoth(transparentCallback)
        self.dList.append(d)
        return d

    def _sweep(self):
        for d in self.dList:
            if d.called:
                self.dList.remove(d)

    def deferToAll(self):
        """
        Return a C{Deferred} that tracks all active deferreds that haven't
        yet fired. When the tracked deferreds fire, the returned
        deferred fires, too.
        """
        # Sweep of already called deferreds is only done when waiting
        # for all unfired ones
        self._sweep()
        return defer.DeferredList(self.dList)
```

```
    def deferToLast(self):
        """
        Return a C{Deferred} that tracks the C{Deferred} that was most
        recently put in the tracker. When the tracked deferred fires,
        the returned deferred fires, too.
        """
        def transparentCallback(anything):
            d.callback(None)
            # Any already called deferreds remaining are now removed
            self._sweep()
            return anything

        # The last-added of ALL remaining deferreds is chained to,
        # even if already called
        if self.dList:
            d = defer.Deferred()
            self.dList[-1].addBoth(transparentCallback)
            return d
        return defer.succeed(None)


class DeferredLock(defer.DeferredLock):
    """
    I am a modified form of L{defer.DeferredLock} lock that lets you
    shut things down when you get the lock.

    Raises an exception if you try to acquire the lock after a
    shutdown has been initated.
    """
    def __init__(self, allowZombies=False):
        self.N_vips = 0
        self.stoppers = []
        self.running = True
        self.allowZombies = allowZombies
        super(DeferredLock, self).__init__()

    @contextmanager
    def context(self, vip=False):
        """
        Usage example, inside a defer.inlineCallbacks function::

          with lock.context() as d:
              # "Wait" for the
              yield d
              <Do something that requires holding onto the lock>
          <Proceed with the lock released>

        """
        yield self.acquire(vip)
        self.release()

    def acquire(self, vip=False):
        """
        Like L{defer.DeferredLock.acquire} except with a I{vip}
```

```
        option. That lets you cut ahead of everyone in the regular
        waiting list and gets the next lock, after anyone else in the
        VIP line who is waiting from their own call of this method.

        If I'm stopped, calling this method results in an error unless
        I was constructed with I{allowZombies} set C{True}. Then it
        simply returns an immediate C{Deferred}.
        """
        def transparentCallback(result):
            self.N_vips -= 1
            return result

        if not self.running:
            if self.allowZombies:
                return defer.succeed(self)
            raise errors.QueueRunError(
                "Can't acquire from a stopped DeferredLock")
        d = defer.Deferred(canceller=self._cancelAcquire)
        if self.locked:
            if vip:
                d.addCallback(transparentCallback)
                self.waiting.insert(self.N_vips, d)
                self.N_vips += 1
            else:
                self.waiting.append(d)
        else:
            self.locked = True
            d.callback(self)
        return d

    def acquireAndRelease(self, vip=False):
        return self.acquire(vip).addCallback(lambda x: x.release())

    def release(self):
        """
        Acts like Twisted's regular C{defer.DeferredLock.release} unless
        I'm stopped and running with the I{allowZombies} option. Then
        calling this does nothing because the lock is acquired
        instantly in that condition.
        """
        if not self.running and self.allowZombies:
            return
        return super(DeferredLock, self).release()

    def addStopper(self, f, *args, **kw):
        """
        Add a callable (along with any args and kw) to be run when
        shutting things down. The callable may return a deferred, and
        more than one can be added. They will be called, and their
        result awaited, in the order received.

        """
        self.stoppers.append([f, args, kw])

    def stop(self):
```

```python
        """
        Shut things down, when the waiting list empties.
        """
        @defer.inlineCallbacks
        def runStoppers(me):
            while self.stoppers:
                f, args, kw = self.stoppers.pop(0)
                yield defer.maybeDeferred(f, *args, **kw)
            me.release()

        self.running = False
        return super(DeferredLock, self).acquire().addCallback(runStoppers)


class CallRunner(object):
    """
    I'm used by L{threads.ThreadLooper} and
    L{process.ProcessUniverse}.
    """
    def __init__(self, raw=False, callStats=False):
        """
        @param raw: Set C{True} to return raw iterators by default instead
          of doing L{iteration} magic.
        @param callStats: Set C{True} to accumulate a list of
          I{callTimes} for each call. B{Caution:} Can get big with
          lots of calls!
        """
        self.raw = raw
        self.info = info.Info()
        self.callStats = callStats
        if callStats:
            self.callTimes = []

    def __call__(self, callTuple):
        """
        Does the f-args-kw call in I{callTuple} to get a 2-tuple
        containing the status of the call and its result:

          - B{e}: An exception was raised; the result is a
            pretty-printed traceback string.

          - B{r}: Ran fine, the result is the return value of the
            call.

          - B{i}: Ran fine, but the result is an iterable other than a
            standard Python one.

        Honors the I{raw} option to return iterators as-is if
        desired. The called function never sees that keyword.
        """
        f, args, kw = callTuple
        raw = kw.pop('raw', None)
        if raw is None:
            raw = self.raw
        try:
```

```
        if self.callStats:
            t0 = time()
            result = f(*args, **kw)
            self.callTimes.append(time()-t0)
        else:
            result = f(*args, **kw)
        # If the task causes the thread to hang, the method
        # call will not reach this point.
    except:
        result = self.info.setCall(f, args, kw).aboutException()
        return ('e', result)
    if not raw and iteration.Deferator.isIterator(result):
        return ('i', result)
    return ('r', result)
```

wire.py

```
# AsynQueue:
# Asynchronous task queueing based on the Twisted framework, with task
# prioritization and a powerful worker interface.
#
# Copyright (C) 2006-2007, 2015 by Edwin A. Suominen,
# http://edsuom.com/AsynQueue
#
# See edsuom.com for API documentation as well as information about
# Ed's background and other projects, software and otherwise.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the
# License. You may obtain a copy of the License at
#
#   http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing,
# software distributed under the License is distributed on an "AS
# IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either
# express or implied. See the License for the specific language
# governing permissions and limitations under the License.

"""
L{WireWorker} and its support staff. For most applications, you can
use L{process} instead.

You need to start another Python interpreter somewhere using
L{WireServer} and have L{WireWorker} connect to it via Twisted AMP. My
L{ServerManager} is just the thing for that.
"""

import sys, os.path, tempfile, shutil, inspect

from zope.interface import implements
from twisted.python import reflect
from twisted.internet import reactor, defer, endpoints
from twisted.protocols import amp
from twisted.internet.protocol import Factory
from twisted.application.service import Application
from twisted.application.internet import StreamServerEndpointService

from info import Info
from util import o2p, p2o
import errors, util, iteration
from interfaces import IWorker
from threads import ThreadLooper


DEFAULT_SOCKET = b"unix:/var/run/wire"
DEFAULT_WWU_FQN = "asynqueue.wire.WireWorkerUniverse"


class RunTask(amp.Command):
```

```
    """
    Runs a task and returns the status and result.

    The I{methodName} is a string specifying the name of a method of a
    subclass of L{WireWorkerUniverse}. No callable will run that is
    not a regular, user-defined method of that object (no internal
    methods like C{__sizeof__}).

    But, see the I{Apache License}, section 8 ("Limitation of
    Liability"). There might be gaping security holes in this, and you
    should limit who you accept connections from in any event,
    preferably encrypting them.

    The I{args} and I{kw} are all pickled strings. (Be careful about
    allowing your methods to do arbitrary things with them!) The args
    and kw can be empty strings, indicating no arguments or keywords.

    The response has the following status/result structure::

      'e': An exception was raised; the result is a pretty-printed
           traceback string.

      'n': Ran fine, the result was a C{None} object.

      'r': Ran fine, the result is the pickled return value of the call.

      'i': Ran fine, but the result is an iterable other than a standard
           Python one. The result is an ID string to use for your
           calls to C{GetNext}.

      'c': Ran fine, but the result is too big for a single return
           value. So you get an ID string for calls to C{GetNext}.
    """
    arguments = [
        ('methodName', amp.String()),
        ('args', amp.String()),
        ('kw', amp.String()),
    ]
    response = [
        ('status', amp.String()),
        ('result', amp.String()),
    ]

class GetNext(amp.Command):
    """
    With a unique ID, gets the next iteration of data from an iterator
    or a task result so big that it had to be chunked.

    The response has a 'value' string with the pickled iteration value
    or a chunk of the too-big task result, and an 'isValid' bool which
    is equivalent to a L{StopIteration}.

    Strings don't need to be pickled, and if I{value} is an actual
    string, I{isRaw} will be set C{True}.
    """
    arguments = [
```

```
        ('ID', amp.String())
    ]
    response = [
        ('value', amp.String()),
        ('isValid', amp.Boolean()),
        ('isRaw', amp.Boolean()),
    ]


class WireWorkerUniverse(amp.CommandLocator):
    """
    Subclass me in code that runs on the remote interpreter, and then
    call the subclass methods via L{runTask}.

    Only methods you define in subclasses of this method, with names
    that don't start with an underscore, will be called.
    """
    @classmethod
    def check(cls, instance):
        for baseClass in inspect.getmro(instance.__class__):
            fqn = reflect.fullyQualifiedName(baseClass)
            if fqn == DEFAULT_WWU_FQN:
                return True
        raise TypeError(
            "You must provide a WireWorkerUniverse subclass instance")

    @RunTask.responder
    def runTask(self, methodName, args, kw):
        """
        This method is called to call the method specified by
        I{methodName} of my subclass running on the remote
        interpreter, with the supplied list I{args} of arguments and
        dict of keywords I{kw}, which may be empty.
        """
        if not hasattr(self, 'wr'):
            self.wr = WireRunner()
            self.info = Info()
        # The method must be a named attribute of my subclass
        # instance. No funny business with special '__foo__' type
        # methods, either.
        func = None if methodName.startswith('_') \
               else getattr(self, methodName, None)
        args = p2o(args, [])
        kw = p2o(kw, {})
        if callable(func):
            return self.wr.call(func, *args, **kw)
        # Wasn't a legit method call
        text = self.info.setCall(methodName, args, kw).aboutCall()
        return {
            'status': 'e',
            'result': "Couldn't run call '{}'".format(text)
        }

    @GetNext.responder
    def getNext(self, ID):
        """
```

```
        @see: L{GetNext}
        """
        return self.wr.getNext(ID)


class WireWorker(object):
    """
    Runs tasks "over the wire," via Twisted AMP running on an
    C{endpoint} connection.

    I implement an L{IWorker} that runs named tasks in a remote Python
    interpreter via Twisted's Asynchronous Messaging Protocol over an
    endpoint that can be a UNIX socket, TCP/IP, SSL, etc. The task
    callable must be a method of a subclass of L{WireWorkerUniverse}
    that has been imported globally, as C{UNIVERSE}, into the same
    module as the one in which your instance of me is constructed. No
    pickled callables are sent over the wire, just strings defining
    the method name of that class instance.

    For most applications, see L{process.ProcessWorker} instead.

    You can also supply a I{series} keyword containing a list of one
    or more task series that I am qualified to handle.

    When running tasks via me, don't assume that you can just call
    blocking code because it's done remotely. The AMP server on the
    other end runs under Twisted, too. (The result of the call may be
    a C{Deferred}, and that's fine.) If the call is a blocking one,
    set the I{thread} keyword C{True} for it and it will run via an
    instance of L{threads.ThreadLooper}.
    """
    implements(IWorker)
    pList = []
    tempDir = []
    cQualified = ['wire', 'remote']

    class AMP(amp.AMP):
        """
        Special disconnection-alerting AMP protocol. When my connection is
        made, I construct a C{Deferred} referenced as I{d_lcww}, which
        I will fire it if I get disconnected.
        """
        def connectionMade(self):
            self.d_lcww = defer.Deferred()
        def connectionLost(self, reason):
            if hasattr(self, 'd_lcww'):
                self.d_lcww.callback(None)
                del self.d_lcww
            return super(amp.AMP, self).connectionLost(reason)

    def __init__(self, wwu, description, series=[], raw=False):
        """
        Constructs me with a reference I{wwu} to a L{WireWorkerUniverse}
        and a client connection I{description} and immediately
        connects to a L{WireServer} running on another Python
        interpreter via the AMP protocol.
```

```
        """
        def connected(ap):
            self.ap = ap
            self.dLock.release()

        WireWorkerUniverse.check(wwu)
        self.tasks = []
        self.raw = raw
        self.iQualified = series
        # Lock that is acquired until AMP connection made
        self.dLock = util.DeferredLock(allowZombies=True)
        self.dLock.addStopper(self.stopper)
        self.dLock.acquire()
        # Make the connection
        dest = endpoints.clientFromString(reactor, description)
        endpoints.connectProtocol(
            dest, self.AMP(locator=wwu)).addCallback(connected)

    def stopper(self):
        if hasattr(self, 'ap'):
            return self.ap.transport.loseConnection()

    def _handleNext(self, ID):
        def gotResponse(response):
            if response['isValid']:
                value = response['value']
                if response['isRaw']:
                    return True, value
                return True, p2o(value)
            return False, None
        return self.ap.callRemote(GetNext, ID=ID).addCallback(gotResponse)

    @defer.inlineCallbacks
    def assembleChunkedResult(self, ID):
        pickleString = ""
        while True:
            isValid, value = yield self._handleNext(ID)
            if isValid:
                pickleString += value
            else:
                break
        defer.returnValue(p2o(pickleString))

    @defer.inlineCallbacks
    def next(self, ID):
        """
        Do a next call of the iterator held by my subordinate, over the
        wire (socket) and in Twisted fashion.
        """
        yield self.dLock.acquire(vip=True)
        isValid, value = yield self._handleNext(ID)
        self.dLock.release()
        if not isValid:
            value = Failure(StopIteration)
        defer.returnValue(value)
```

```
    def resign(self, *args):
        if hasattr(self, 'resignator'):
            self.resignator()
            del self.resignator

    # Implementation methods
    # --------------------------------------------------------------------------

    def setResignator(self, callableObject):
        """
        I resign if my underlying AMP connection is lost.
        """
        def gotLock(lock):
            self.ap.d_lcww.addCallback(self.resign)
            lock.release()
        self.resignator = callableObject
        self.dLock.acquire().addCallback(gotLock)

    @defer.inlineCallbacks
    def run(self, task):
        """
        Sends the task callable, args, kw to the process and returns a
        deferred to the eventual result.
        """
        def result(value):
            self.tasks.remove(task)
            task.callback((status, value))

        self.tasks.append(task)
        doNext = task.callTuple[2].pop('doNext', False)
        yield self.dLock.acquire(doNext)
        # Run the task via AMP, but only if it's connected
        #-----------------------------------------------------------
        if not self.ap.transport.connected:
            response = {'status':'d', 'result':None}
        else:
            kw = {}
            consumer = task.callTuple[2].pop('consumer', None)
            for k, value in enumerate(task.callTuple):
                name = RunTask.arguments[k][0]
                kw[name] = value if isinstance(value, str) else o2p(value)
            if self.raw:
                kw.setdefault('raw', True)
            # The heart of the matter
            try:
                response = yield self.ap.callRemote(RunTask, **kw)
            except:
                response = {'status':'d', 'result':None}
        #-----------------------------------------------------------
        # At this point, we can permit another remote call to get
        # going for a separate task.
        self.dLock.release()
        # Process the response. No lock problems even if that
        # involves further remote calls, i.e., GetNext
        status = response['status']
        x = response['result']
```

```
        if status == 'i':
            # What we get from the subordinate is an ID to an iterator
            # it is holding onto, but we need to give the task an
            # iterationProducer that hooks up to it.
            pf = iteration.Prefetcherator(x)
            ok = yield pf.setup(self.next, x)
            if ok:
                returnThis = iteration.Deferator(pf)
                if consumer:
                    returnThis = iteration.IterationProducer(
                        returnThis, consumer)
            else:
                # The subordinate returned an iterator, but it's not
                # one I could prefetch from. Probably empty.
                returnThis = []
            result(returnThis)
        elif status == 'c':
            # Chunked result, which will take a while
            dResult = yield self.assembleChunkedResult(x)
            result(dResult)
        elif status == 'r':
            result(p2o(x))
        elif status == 'n':
            result(None)
        elif status in ('e', 'd'):
            result(x)
            self.resign()
        else:
            raise ValueError("Unknown status {}".format(status))

    def stop(self):
        if getattr(self, '_stopped', False):
            return
        self._stopped = True
        return self.dLock.stop()

    def crash(self):
        self.stopper()
        return self.tasks


class ChunkyString(object):
    """
    I iterate chunks of a big string, deleting my internal reference
    to it when done so it can be garbage collected even if I'm not.
    """
    chunkSize = 2**15

    def __init__(self, bigString):
        self.k0 = 0
        self.N = len(bigString)
        self.bigString = bigString

    def __iter__(self):
        return self
```

```python
    def next(self):
        if not hasattr(self, 'bigString'):
            raise StopIteration
        k1 = min([self.k0 + self.chunkSize, self.N])
        thisChunk = self.bigString[self.k0:k1]
        if k1 == self.N:
            del self.bigString
        else:
            self.k0 = k1
        return thisChunk


class WireRunner(object):
    """
    An instance of me is constructed by a L{WireWorkerUniverse} on the
    server end of the AMP connection to run all tasks for its
    L{WireServer}.
    """
    def __init__(self):
        self.iterators = {}
        self.deferators = {}
        self.info = Info()
        self.dt = util.DeferredTracker()

    def shutdown(self):
        if hasattr(self, 't'):
            d = self.t.stop().addCallback(lambda _: delattr(self, 't'))
            self.dt.put(d)
        return self.dt.deferToAll()

    def _saveIterator(self, x):
        ID = str(hash(x))
        self.iterators[ID] = x
        return ID

    def call(self, f, *args, **kw):
        """
        Run the f-args-kw combination, in the regular thread or in a
        thread running if I have one.

        @return: A C{Deferred} to the status and result.
        """
        d = self._call(f, *args, **kw)
        self.dt.put(d)
        return d

    @defer.inlineCallbacks
    def _call(self, f, *args, **kw):
        def oops(failureObj, ID=None):
            if ID:
                text = self.info.aboutFailure(failureObj, ID)
                self.info.forgetID(ID)
            else:
                text = self.info.aboutFailure(failureObj)
            return ('e', text)
```

```
        response = {}
        raw = kw.pop('raw', False)
        if kw.pop('thread', False):
            if not hasattr(self, 't'):
                self.t = ThreadLooper(rawIterators=True)
            # No errback needed because L{util.CallRunner} returns an
            # 'e' status for errors
            status, result = yield self.t.call(f, *args, **kw)
        else:
            # The info object saves the call
            self.info.setCall(f, args, kw)
            ID = self.info.ID
            result = yield defer.maybeDeferred(
                f, *args, **kw).addErrback(oops, ID)
            self.info.forgetID(ID)
            if isinstance(result, tuple) and result[0] == 'e':
                status, result = result
            elif result is None:
                # A None object
                status = 'n'
                result = ""
            elif not raw and iteration.Deferator.isIterator(result):
                status = 'i'
                result = self._saveIterator(result)
            else:
                status = 'r'
                result = o2p(result)
                if len(result) > ChunkyString.chunkSize:
                    # Too big to send as a single pickled string
                    status = 'c'
                    result = self._saveIterator(ChunkyString(result))
        # At this point, we have our result (blank string for C{None},
        # an ID for an iterator, or a pickled string
        response['status'] = status
        response['result'] = result
        defer.returnValue(response)

    def getNext(self, ID):
        """
        Gets the next item for the iterator specified by I{ID}, returning
        a C{Deferred} that fires with a response containing the
        pickled item and the I{isValid} status indicating if the item
        is legit (C{False} = L{StopIteration}).
        """
        d = self._getNext(ID)
        self.dt.put(d)
        return d

    @defer.inlineCallbacks
    def _getNext(self, ID):
        def oops(failureObj, ID):
            del self.iterators[ID]
            response['isValid'] = False
            if failureObj.type == StopIteration:
                response['value'] = ""
            else:
```

```
                response['value'] = self.info.setCall(
                    "getNext", [ID]).aboutFailure(failureObj)
        def bogusResponse():
            response['value'] = ""
            response['isValid'] = False
        def handleValue(value):
            if isinstance(value, str):
                response['isRaw'] = True
                response['value'] = value
            else:
                response['value'] = o2p(value)

        response = {'isValid':True, 'isRaw':False}
        if ID in self.iterators:
            # Iterator
            if hasattr(self, 't'):
                # Get next iteration in a thread
                value = yield self.t.deferToThread(
                    self.iterators[ID].next).addErrback(oops, ID)
                if response['isValid']:
                    handleValue(value)
            else:
                # Get next iteration in main loop. No blocking!
                try:
                    value = self.iterators[ID].next()
                    handleValue(value)
                except StopIteration:
                    del self.iterators[ID]
                    bogusResponse()
        else:
            # No iterator, at least not anymore
            bogusResponse()
        defer.returnValue(response)


class WireServer(object):
    """
    An AMP server for the remote end of a L{WireWorker}.

    Construct me with either an instance or the fully qualified name
    of a L{WireWorkerUniverse} subclass. Then call my L{run} method
    with an endpoint description string to obtain a C{service} that I
    can start directly or include in the C{application} of a C{.tac}
    file, thus accepting connections to run tasks.
    """
    def __init__(self, wwu):
        if isinstance(wwu, str):
            klass = reflect.namedObject(wwu)
            wwu = klass()
        WireWorkerUniverse.check(wwu)
        self.factory = Factory()
        self.factory.protocol = lambda: amp.AMP(locator=wwu)

    def run(self, description):
        """
        Does B{no} encryption or credential checking (unless you use SSL
```

```
        endpoints).
        """
        endpoint = endpoints.serverFromString(reactor, description)
        service = StreamServerEndpointService(endpoint, self.factory)
        return service


class ServerManager(object):
    """
    I spawn one or more new Python interpreters that run a
    L{WireServer} on the local machine.
    """
    def __init__(self, wwuFQN=None):
        self.processInfo = {}
        self.wwuFQN = DEFAULT_WWU_FQN if wwuFQN is None else wwuFQN
        reactor.addSystemEventTrigger('before', 'shutdown', self.done)

    def spawn(self, description, niceness=0):
        """
        Spawns a subordinate Python interpreter.

        B{TODO:} Implement (somehow) I{niceness} keyword to accept an
        integer UNIX nice lvel for the new interpreter process.

        @param description: A server description string of the form
          used by Twisted's C{endpoints.serverFromString}. Default is
          "unix:/var/run/wire".

        @return: A C{Deferred} that fires with the PID of the new
          process if it connected OK, or C{None} if not.
        """
        def ready(response):
            self.processInfo[pt.pid] = {'pt':pt}
            if "AsynQueue WireServer listening" in response:
                return pt.pid
            self.done(pt.pid)

        # Spawn the AMP server and "wait" for it to indicate it's OK
        args = [
            sys.executable,
            "-m", "asynqueue.wire", description, self.wwuFQN]
        pp = util.ProcessProtocol(self.done)
        pt = reactor.spawnProcess(pp, sys.executable, args)
        return pp.d.addCallback(ready)

    def newSocket(self):
        """
        Assigns a unique name to a socket file in a temporary directory
        common to all processes spawned by me, which will be removed
        with all socket files after reactor shutdown. Doesn't actually
        create the socket file; the server does that.

        @return: An endpoint description using the new socket filename.
        """
        # The process name
        pName = "worker-{:03d}".format(len(self.processInfo))
```

```
        # A unique temp directory for all instances' socket files
        if not hasattr(self, 'tempDir'):
            self.tempDir = tempfile.mkdtemp()
            reactor.addSystemEventTrigger(
                'after', 'shutdown',
                shutil.rmtree, self.tempDir, ignore_errors=True)
        socketFile = os.path.join(self.tempDir, "{}.sock".format(pName))
        return b"unix:{}".format(socketFile)

    @defer.inlineCallbacks
    def done(self, pid=None):
        if pid is None:
            for pid in self.processInfo.keys():
                yield self.done(pid)
        elif pid in self.processInfo:
            thisInfo = self.processInfo[pid]
            if 'ap' in thisInfo:
                yield thisInfo['ap'].transport.loseConnection()
            if 'pt' in thisInfo:
                yield thisInfo['pt'].loseConnection()
            yield util.killProcess(pid)
            del self.processInfo[pid]


def runServer(description, wwuFQN):
    """
    Runs a L{WireServer}, listening at the specified endpoint
    I{description} without bothering with an C{application}.

    You must specify the package.module.class fully qualified name of
    a L{WireWorkerUniverse} subclass with I{wwu}.
    """
    def running():
        print "AsynQueue WireServer listening at {}".format(description)
        sys.stdout.flush()

    ws = WireServer(wwuFQN)
    service = ws.run(description)
    service.startService()
    reactor.callWhenRunning(running)
    reactor.run()


if __name__ == '__main__':
    runServer(*sys.argv[1:])
```

workers.py

```python
# AsynQueue:
# Asynchronous task queueing based on the Twisted framework, with task
# prioritization and a powerful worker interface.
#
# Copyright (C) 2006-2007, 2015 by Edwin A. Suominen,
# http://edsuom.com/AsynQueue
#
# See edsuom.com for API documentation as well as information about
# Ed's background and other projects, software and otherwise.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the
# License. You may obtain a copy of the License at
#
#   http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing,
# software distributed under the License is distributed on an "AS
# IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either
# express or implied. See the License for the specific language
# governing permissions and limitations under the License.

"""
Implementors of the L{interfaces.IWorker} interface. These objects
are what handle the tasks in your L{base.TaskQueue}.
"""
import sys, os, os.path, tempfile, shutil

from zope.interface import implements
from twisted.internet import defer

from interfaces import IWorker
import errors, info, util, iteration


# Make all our workers importable from this module
from threads import ThreadWorker
from process import ProcessWorker
from wire import WireWorker


class AsyncWorker(object):
    """
    I implement an L{IWorker} that runs tasks in the Twisted main
    loop.

    I run each L{tasks.Task} one at a time but in a well-behaved
    non-blocking manner. If the task callable doesn't return a
    C{Deferred}, it better get its work done fast. You just can't get
    away with blocking in the Twisted main loop.

    You can supply a I{series} keyword containing a list of one or
```

```
    more task series that I am qualified to handle.

    This class was mostly written for testing during development, but
    it helped keep the basic functions of a worker in mind. And who
    knows; it might be useful where you want the benefits of priority
    queueing without leaving the Twisted mindset even for a moment.
    """
    implements(IWorker)
    cQualified = ['async', 'local']

    def __init__(self, series=[], raw=False):
        """
        Constructs an instance of me with a L{util.DeferredLock}.

        @param series: A list of one or more task series that this
          particular instance of me is qualified to handle.

        @param raw: Set C{True} if you want raw iterators to be
          returned instead of L{iteration.Deferator} instances. You
          can override this in with the same keyword set C{False} in a
          call.
        """
        self.iQualified = series
        self.raw = raw
        self.info = info.Info()
        self.dLock = util.DeferredLock()

    def setResignator(self, callableObject):
        self.dLock.addStopper(callableObject)

    def run(self, task):
        """
        Implements L{IWorker.run}, running the I{task} in the main
        thread. The task callable B{must} not block.
        """
        def ready(null):
            # THOU SHALT NOT BLOCK!
            return defer.maybeDeferred(
                f, *args, **kw).addCallbacks(done, oops)

        def done(result):
            if not raw and iteration.isIterator(result):
                try:
                    result = iteration.Deferator(result)
                except:
                    result = []
                else:
                    if consumer:
                        result = iteration.IterationProducer(result, consumer)
                status = 'i'
            else:
                status = 'r'
            # Hangs if release is done after the task callback
            self.dLock.release()
            task.callback((status, result))
```

```python
        def oops(failureObj):
            text = self.info.setCall(f, args, kw).aboutFailure(failureObj)
            task.callback(('e', text))

        f, args, kw = task.callTuple
        raw = kw.pop('raw', None)
        if raw is None:
            raw = self.raw
        consumer = kw.pop('consumer', None)
        vip = (kw.pop('doNext', False) or task.priority <= -20)
        return self.dLock.acquire(vip).addCallback(ready)

    def stop(self):
        """
        Implements L{IWorker.stop}.
        """
        return self.dLock.stop()

    def crash(self):
        """
        There's no point to implementing this because the Twisted main
        loop will block along with any task you give this worker.
        """


__all__ = [
    'ThreadWorker', 'ProcessWorker', 'AsyncWorker', 'WireWorker',
    'IWorker'
]
```