# colormap.py

```
#!/usr/bin/env python
#
# mcmandelbrot
#
# An example package for AsynQueue:
# Asynchronous task queueing based on the Twisted framework, with task
# prioritization and a powerful worker interface.
#
# Copyright (C) 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.


"""
Colormapping, which is a tricky business when visualizing fractals
that you can zoom in on.
"""

import os.path
from array import array

import numpy as np


class ColorMapper(object):
    """
    I map floating-point values in the range 0.0 to 1.0 to RGB byte
    triplets.

    @cvar fileName: A file with a colormap of RGB triplets, one for
      each of many linearly increasing values to be mapped, in CSV
      format.
    """
    N_colors = 6000
    dither = 0.05 / N_colors

    def __init__(self):
```

```
        self.rgb = self.loadMap(self.N_colors)
        self.jMax = len(self.rgb) - 1

    def loadMap(self, N):
        """
        Returns an RGB colormap of dimensions C{Nx3} that transitions from
        black to red, then red to orange, then orange to white.
        """
        ranges = [
            [0.000, 2.8/3],  # Red component ranges
            [2.5/3, 3.0/3],  # Green component ranges
            [2.7/3, 1.000],  # Blue component ranges
        ]
        limits = [255, 220, 180]
        bluecycle = [20, 120]
        rgb = self._rangeMap(N, ranges, limits)
        rgb[:,2] += bluecycle[0]*np.sin(
            np.linspace(0, bluecycle[1]*2*3.141591, N)) + bluecycle[0]
        return rgb

    def _rangeMap(self, N, ranges, limits):
        rgb = np.zeros((N, 3), dtype=np.uint8)
        kt = np.rint(N*np.array(ranges)).astype(int)
        # Range #1: Increase red
        rgb[0:kt[0,1],0] = np.linspace(0, limits[0], kt[0,1])
        # Range #2: Max red, increase green
        rgb[kt[0,1]:,0] = limits[0]
        rgb[kt[1,0]:kt[1,1],1] = np.linspace(0, limits[1], kt[1,1]-kt[1,0])
        # Range #3: Max red and green, increase blue
        rgb[kt[1,1]:,1] = limits[1]
        rgb[kt[2,0]:,2] = np.linspace(0, limits[2], kt[2,1]-kt[2,0])
        return rgb

    def __call__(self, x):
        result = array('B')
        np.clip(x + (self.dither * np.random.randn(len(x))), 0, 1.0, x)
        np.rint(self.jMax * x, x)
        for j in x.astype(np.uint16):
            result.extend(self.rgb[j,:])
        return result
```

# html.py

```
#!/usr/bin/env python
#
# mcmandelbrot
#
# An example package for AsynQueue:
# Asynchronous task queueing based on the Twisted framework, with task
# prioritization and a powerful worker interface.
#
# Copyright (C) 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.


"""
A Twisted web C{Resource} that serves clickable, zoomable
Mandelbrot Set images.
"""

import sys

from twisted.application import internet, service
from twisted.internet import defer
from twisted.web import server, resource, static, util, http

from mcmandelbrot import vroot, image

MY_PORT = 8080
VERBOSE = True
HTML_FILE = "mcm.html"

HOWTO = """
Click anywhere in the image to zoom in 5x at that location. Try
exploring the edges of the black &ldquo;lakes.&rdquo;
"""

ABOUT = """
Images genera&shy;ted by the <i>mcmandelbrot</i> demo package
bun&shy;dled with my <a
href="http://edsuom.com/AsynQueue">AsynQueue</a> asyn&shy;chronous
```

```
processing pack&shy;age, which is freely available per the Apache
License. A link back to <a
href="http://mcm.edsuom.com"><b>mcm.edsuom.com</b></a> would be
apprec&shy;iated.
"""

BYLINE = " &mdash;Ed Suominen"

MORE_INFO = """
CPU and bandwidth resources for this site were con&shy;tributed by <a
href="http://tellectual.com">Tellectual Press</a>, publisher of my
book <em>Evolving out of Eden</em>.
"""

class ResourceBag(object):
    blankImage = ("blank.jpg", 'image/jpeg')
    children = {}

    def __init__(self, descriptions):
        self.children[''] = RootResource(self.blankImage[0])
        with vroot.openPackageFile(self.blankImage[0]) as fh:
            imageData = fh.read()
        self.children[self.blankImage[0]] = static.Data(
            imageData, self.blankImage[1])
        self.children['image.png'] = ImageResource(descriptions)

    def shutdown(self):
        return self.ir.shutdown()

    def putChildren(self, root):
        for path, res in self.children.iteritems():
            root.putChild(path, res)


class RootResource(resource.Resource):
    defaultParams = {
        'cr':   "-0.630",
        'ci':   "+0.000",
        'crpm': "1.40" }
    defaultTitle = \
        "Interactive Mandelbrot Set: Driven by Twisted and AsynQueue"
    formItems = (
        ("Real:", "cr"   ),
        ("Imag:", "ci"   ),
        ("+/-",   "crpm" ))
    inputSize = 10

    def __init__(self, blankImage):
        self.blankImage = blankImage
        self.vr = self.vRoot()
        resource.Resource.__init__(self)

    def render_GET(self, request):
        request.setHeader("content-type", 'text/html')
        kw = {'permalink': request.uri}
        kw.update(self.defaultParams)
```

```
        if request.args:
            for key, values in request.args.iteritems():
                kw[key] = http.unquote(values[0])
            kw['img'] = self.imageURL(kw)
            kw['onload'] = None
        else:
            kw['img'] = self.blankImage
            kw['onload'] = "updateImage()"
        return self.vr(**kw)

    def imageURL(self, params):
        """
        Returns a URL for obtaining a Mandelbrot Set image with the
        parameters in the supplied dict I{params}.
        """
        parts = []
        for name, value in params.iteritems():
            if name in self.defaultParams:
                parts.append("{}={}".format(name, value))
        return "/image.png?{}".format('&'.join(parts))

    def vRoot(self):
        """
        Populates my vroot I{vr} with an etree that renders into the HTML
        page.
        """
        def heading():
            with v.context():
                v.nc('div', 'heading')
                v.nc('p', 'bigger')
                v.textX("Interactive Mandelbrot Set")
            v.nc('div', 'subheading')
            v.nc('p', 'smaller')
            v.text("Powered by ")
            v.nc('a')
            v.text("Twisted")
            v.set('href', "http://twistedmatrix.com")
            v.tailX(" and ")
            v.ns('a')
            v.text("AsynQueue")
            v.set('href', "http://edsuom.com/AsynQueue")
            v.tail(".")

        vr = vroot.VRoot(self.defaultTitle)
        with vr as v:
            v.nc('body')
            v.addToMap('onload', 'onload')
            v.nc('div', 'container')
            v.set('id', 'container')
            v.nc('div', 'first_part')
            #-------------------------------------------------------
            with v.context():
                heading()
            v.ngc('div', 'clear').text = " "
            with v.context():
                v.nc('div')
```

```
        with v.context():
            v.nc('form')
            v.nc('div', 'form')
            v.set('name', "position")
            v.set('action', "javascript:updateImage()")
            for label, name in v.nci(
                    self.formItems, 'div', 'form_item'):
                v.nc('span', 'form_item')
                v.text(label)
                v.ns('input', 'position')
                v.addToMap(name, 'value')
                v.set('type', "text")
                v.set('size', str(self.inputSize))
                v.set('id', name)
            v.nc('div', 'form_item')
            e = v.ngc('input')
            e.set('type', "submit")
            e.set('value', "Reload")
            v.ns('div', 'form_item')
            e = v.ngc('button')
            e.set('type', "button")
            e.set('onclick', "zoomOut()")
            e.text = "Zoom Out"
        with v.context():
            v.nc('div', 'about')
            v.textX(ABOUT)
            v.nc('span', 'byline')
            v.textX(BYLINE)
        v.nc('div', 'about large_only')
        v.textX(MORE_INFO)
    v.ns('div', 'second_part')
    #-----------------------------------------------------
    with v.context():
        v.nc('div', 'image')
        v.set('id', 'image')
        with v.context():
            v.nc('img', 'mandelbrot')
            v.addToMap('img', 'src')
            v.set('id', 'mandelbrot')
            v.set('onclick', "zoomIn(event)")
            v.set('onmousemove', "hover(event)")
        v.nc('div', 'footer')
        v.nc('div', 'left')
        v.set('id', 'hover')
        v.textX(HOWTO)
        v.ns('div', 'right')
        v.nc('a', 'bold')
        v.text("Permalink")
        v.set('id', 'permalink')
        v.addToMap('permalink', 'href')
    v.ns('div', 'about small_only')
    v.set('id', 'more_info')
    v.textX(MORE_INFO)
    return vr
```

```
class ImageResource(resource.Resource):
    isLeaf = True

    def __init__(self, descriptions):
        self.imager = image.Imager(descriptions, verbose=VERBOSE)
        resource.Resource.__init__(self)

    def shutdown(self):
        return self.imager.shutdown()

    def render_GET(self, request):
        request.setHeader("content-disposition", "image.png")
        request.setHeader("content-type", 'image/png')
        self.imager.renderImage(request)
        return server.NOT_DONE_YET


class MandelbrotSite(server.Site):
    def __init__(self):
        self.rb = ResourceBag([None])
        siteResource = resource.Resource()
        self.rb.putChildren(siteResource)
        server.Site.__init__(self, siteResource)

    def stopFactory(self):
        super(MandelbrotSite, self).stopFactory()
        return self.rb.shutdown()


if '/twistd' in sys.argv[0]:
    site = MandelbrotSite()
    application = service.Application("Interactive Mandelbrot Set HTTP Server")
    internet.TCPServer(MY_PORT, site).setServiceParent(application)
```

# image.py

```
#!/usr/bin/env python
#
# mcmandelbrot
#
# An example package for AsynQueue:
# Asynchronous task queueing based on the Twisted framework, with task
# prioritization and a powerful worker interface.
#
# Copyright (C) 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.


"""
Render Mandelbrot Set images in PNG format in response to Twisted
web requests. Used by L{html}.
"""

import urlparse

from twisted.internet import defer

import wire


class RunnerToken(object):
    """
    B{TODO:} Implmement this with C{asynqueue.base.Priority} to
    dispatch requests to runners based on how fast they've run
    previous ones.
    """
    scores = {}
    def __init__(self, runnerInstance):
        self.r = runnerInstance


class Imager(object):
```

```
    """
    Call L{renderImage} with Twisted web I{request} objects as much as
    you like to write PNG images in response to them.

    Call L{shutdown} when done.
    """
    Nx = 640
    Nx_min = 240
    Nx_max = 10000 # 100 megapixels ought to be enough

    N_values = 3000
    steepness = 3

    def __init__(self, descriptions=[], verbose=False):
        self.dStart = self.setup(descriptions, verbose)

    def setup(self, descriptions, verbose=False):
        """
        Sets me up with one or more instances of L{wire.RemoteRunner}.

        B{TODO:} Implement the use of multiple
        descriptions. Currently, the list must have one and only one.

        @param descriptions: A list of one or more Twisted C{endpoint}
          descriptions for connecting a L{wire.RemoteRunner}. Include
          a C{None} object in the list to use (or also use) one that
          runs via a UNIX socket on the local machine.
        """
        dList = []
        if len(descriptions) != 1:
            raise NotImplementedError("You can use only one runner, for now")
        for description in descriptions:
            self.runner = wire.RemoteRunner(description)
            d = self.runner.setup(
                N_values=self.N_values, steepness=self.steepness)
            dList.append(d)
        return defer.DeferredList(dList)

    def shutdown(self):
        return self.runner.shutdown()

    def setImageWidth(self, N):
        if N < self.Nx_min:
            N = self.Nx_min
        elif N > self.Nx_max:
            N = self.Nx_max
        self.Nx = N

    @defer.inlineCallbacks
    def renderImage(self, request):
        """
        Call with a Twisted.web I{request} that includes a URL query map
        in C{request.args} specifying I{cr}, I{ci}, I{crpm}, and,
        optionally, I{crpi}. Writes the PNG image data to the request
```

```
as it is generated remotely. When the image is all written,
calls C{request.finish} and fires the C{Deferred} it returns.

An example query string, for the basic Mandelbrot set overview
with 1200 points::

  ?N=1200&cr=-0.8&ci=0.0&crpm=1.45&crpi=1.2

"""
x = {}
d = request.notifyFinish().addErrback(lambda _: None)
neededNames = ['cr', 'ci', 'crpm']
for name, value in request.args.iteritems():
    if name == 'N':
        self.setImageWidth(int(value[0]))
    else:
        x[name] = float(value[0])
    if name in neededNames:
        neededNames.remove(name)
if not neededNames:
    ciPM = x.get('cipm', x['crpm'])
    if hasattr(self, 'dStart'):
        yield self.dStart
        del self.dStart
    timeSpent, N = yield self.runner.run(
        request, self.Nx,
        x['cr'], x['ci'], x['crpm'], ciPM,
        dCancel=d, requester=request.getClient())
    if not d.called:
        request.finish()
```

# main.py

```
#!/usr/bin/env python
#
# mcmandelbrot
#
# An example package for AsynQueue:
# Asynchronous task queueing based on the Twisted framework, with task
# prioritization and a powerful worker interface.
#
# Copyright (C) 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 example of C{AsynQueue} in action. Can be fun to play with if you
have a multicore CPU. You will need the following packages, which you
can get via C{pip install}:

  - C{weave} (part of SciPy)
  - C{numpy} (part of SciPy)
  - C{matplotlib}
  - C{asynqueue} (Duh...)

Command line usage::

  mcmandelbrot
    [-d <description>,] [-N, <iterations>,]
    [-s, <steepness>,] [-o, <imageFile>,]
    Nx, cr, ci, crPM[, ciPM]

Writes PNG image to stdout unless B{-o} is set, then saves it to
I{imageFile}. In that case, prints some stats about the
multiprocessing computation to I{stdout}.
```

```
To see an overview displayed with ImageMagick's I{display} command::

  mcmandelbrot 2000 -0.630 0 1.4 1.2 |display

Write a detailed view to the image I{detail.png}::

  mcmandelbrot -o detail.png 3000 -0.73249 +0.21653 0.0112

"""

import sys

from twisted.internet import defer, reactor

import runner, wire


def run(*args, **kw):
    """
    Call with::

      [-d <description>,] [-N, <values>,]
      [-s, <steepness>,] [-o, <imageFile>,]
      Nx, cr, ci, crPM[, ciPM]

    Writes PNG image to stdout unless -o is set, then saves it to
    C{imageFile}. In that case, prints some stats about the
    multiprocessing computation to stdout.

    Options-as-arguments:

      - B{d}: An endpoint I{description} of a server running a
        L{wire.server}.

    @keyword N_values: Integer number of possible values for
      Mandelbrot points during iteration. Can set with the C{-N
      values} arg instead.

    @keyword ignoreReactor: Set C{True} to let somebody else start and
      stop the reactor.
    """
    @defer.inlineCallbacks
    def reallyRun():
        if description:
            myRunner = wire.RemoteRunner(description)
            yield myRunner.setup(
                N_values=N_values, steepness=steepness)
        else:
            myRunner = runner.Runner(N_values, steepness, stats)
        runInfo = yield myRunner.run(fh, Nx, cr, ci, crPM, ciPM)
        if stats:
```

```
            yield myRunner.showStats(runInfo)
        yield myRunner.shutdown()
        if not ignoreReactor:
            reactor.stop()
        defer.returnValue(runInfo)

    def getOpt(opt, default):
        optCode = "-{}".format(opt)
        if optCode in args:
            k = args.index(optCode)
            args.pop(k)
            optType = type(default)
            return optType(args.pop(k))
        return default

    ignoreReactor = kw.pop('ignoreReactor', False)
    if not args:
        args = sys.argv[1:]
    args = list(args)
    steepness = getOpt('s', 3.0)
    N_values = getOpt('N', 2000)
    fileName = getOpt('o', "")
    description = getOpt('d', "")
    if fileName:
        stats = True
        fh = open(fileName, 'w')
    else:
        stats = False
        fh = sys.stdout
    if len(args) < 4:
        print(
            "Usage: [-s steepness] [-N values] "+\
            "[-o imageFile] [-d description] " +\
            "N cr ci crPM [ciPM]")
        sys.exit(1)
    Nx = int(args[0])
    cr, ci, crPM = [float(x) for x in args[1:4]]
    ciPM = float(args[4]) if len(args) > 4 else crPM
    if ignoreReactor:
        return reallyRun()
    else:
        reactor.callWhenRunning(reallyRun)
        reactor.run()


if __name__ == '__main__':
    run()
```

# runner.py

```
#!/usr/bin/env python
#
# mcmandelbrot
#
# An example package for AsynQueue:
# Asynchronous task queueing based on the Twisted framework, with task
# prioritization and a powerful worker interface.
#
# Copyright (C) 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 example of C{AsynQueue} in action. Can be fun to play with if you
have a multicore CPU. You will need the following packages, which you
can get via C{pip install}:

  - C{weave} (part of SciPy)
  - C{numpy} (part of SciPy)
  - C{matplotlib}
  - C{asynqueue} (Duh...)


Command line usage::

  mcmandelbrot
    [-s steepness] [-N values] [-o imageFile]
    N cr ci crPM [ciPM]

Produces chunks of a PNG image, to stdout if you don't specify an
imageFile with the C{-o} option.

Example: C{mcm 2000 -0.630 0 1.4 1.2 >overview.png}

"""

import sys, time, array
```

```
import png
import numpy as np

from zope.interface import implements
from twisted.internet import defer, reactor
from twisted.internet.interfaces import IPushProducer

import asynqueue
from asynqueue.threads import OrderedItemProducer

from valuer import MandelbrotValuer


class Runner(object):
    """
    I run a multi-process Mandelbrot Set computation operation.

    @cvar N_processes: The number of processes to use, disregarded if
      I{useThread} is set C{True} in my constructor.
    """
    N_minProcesses = 2
    N_maxProcesses = 6

    msgProto = "{} ({:+16.13f} +/-{:10E}, {:+16.13f} +/-{:10E}) "+\
               "{:d} pixels in {:4.2f} sec"

    def __init__(self, N_values, steepness, stats=False, verbose=False):
        self.q = asynqueue.ProcessQueue(self.N_processes, callStats=stats)
        self.mv = MandelbrotValuer(N_values, steepness)
        self.verbose = verbose

    def shutdown(self):
        return self.q.shutdown()

    @property
    def N_processes(self):
        maxValue = min([
            self.N_maxProcesses, asynqueue.ProcessQueue.cores()-1])
        return max([self.N_minProcesses, maxValue])

    def log(self, *args):
        if self.verbose:
            print self.msgProto.format(*args)

    def run(self, fh, Nx, cr, ci, crPM, ciPM, dCancel=None, requester=None):
        """
        Runs my L{compute} method to generate a PNG image of the
        Mandelbrot Set and write it in chunks to the file handle or
        write-capable object I{fh}.

        The image is centered at location I{cr, ci} in the complex
        plane, plus or minus I{crPM} on the real axis and I{ciPM} on
        the imaginary axis.
```

```
        @return: A C{Deferred} that fires with the total elasped time
          for the computation and the number of pixels computed.
        """
        def done(N):
            timeSpent = time.time() - t0
            if requester:
                self.log(
                    requester, cr, crPM, ci, ciPM, N, timeSpent)
            return timeSpent, N

        t0 = time.time()
        xySpans = []
        for center, plusMinus in ((cr, crPM), (ci, ciPM)):
            xySpans.append([center - plusMinus, center + plusMinus])
        xySpans[0].append(Nx)
        diffs = []
        for k in (0, 1):
            diff = xySpans[k][1] - xySpans[k][0]
            if diff <= 5E-16:
                return defer.succeed((0, 0))
            diffs.append(diff)
        xySpans[1].append(int(Nx * diffs[1] / diffs[0]))
        return self.compute(
            fh, xySpans[0], xySpans[1], dCancel).addCallback(done)

    @defer.inlineCallbacks
    def compute(self, fh, xSpan, ySpan, dCancel=None):
        """
        Computes the Mandelbrot Set under C{Twisted} and generates a
        pretty image, written as a PNG image to the supplied file
        handle I{fh} one row at a time.

        @return: A C{Deferred} that fires when the image is completely
          written and you can close the file handle, with the number
          of pixels computed (may be a lower number than expected if
          the connection terminated early).
        """
        def f(rows):
            try:
                writer = png.Writer(Nx, Ny, bitdepth=8, compression=9)
                writer.write(fh, rows)
            except Exception as e:
                # Trap ValueError caused by mid-stream cancellation
                if not isinstance(e, StopIteration):
                    if "rows" not in e.message and "height" not in e.message:
                        raise e

        crMin, crMax, Nx = xSpan
        ciMin, ciMax, Ny = ySpan
        # We have at most 5 calls in the process queue for each worker
        # servicing them, to allow midstream canceling and interleave
        # parallel computation requests.
        ds = defer.DeferredSemaphore(5*self.N_processes)
        p = OrderedItemProducer()
        yield p.start(f)
```

```
        # "The pickle module keeps track of the objects it has already
        # serialized, so that later references to the same object won't be
        # serialized again." --Python docs
        for k, ci in enumerate(np.linspace(ciMax, ciMin, Ny)):
            # "Wait" for the number of pending calls to fall back to
            # the limit
            yield ds.acquire()
            # Make sure the render hasn't been canceled
            if getattr(dCancel, 'called', False):
                break
            # Call one of my processes to get each row of values,
            # starting from the top
            d = p.produceItem(
                self.q.call, self.mv, crMin, crMax, Nx, ci,
                series='process')
            d.addCallback(lambda _: ds.release())
        yield p.stop()
        defer.returnValue(Nx*(k+1))

    def showStats(self, callInfo):
        """
        Displays stats about the run on stdout
        """
        def gotStats(stats):
            x = np.asarray(stats)
            workerTime, processTime = [np.sum(x[:,k]) for k in (0,1)]
            print "Run stats, with {:d} parallel ".format(self.N_processes) +\
                "processes running {:d} calls\n{}".format(len(stats), "-"*70)
            print "Process:\t{:7.2f} seconds, {:0.1f}% of main".format(
                processTime, 100*processTime/totalTime)
            print "Worker:\t\t{:7.2f} seconds, {:0.1f}% of main".format(
                workerTime, 100*workerTime/totalTime)
            print "Total on main:\t{:7.2f} seconds".format(totalTime)
            diffs = 1000*(x[:,0] - x[:,1])
            mean = np.mean(diffs)
            print "Mean worker-to-process overhead (ms/call): {:0.7f}".format(
                mean)
        totalTime = callInfo[0]
        print "Computed {:d} pixels in {:1.1f} seconds.".format(
            callInfo[1], totalTime)
        return self.q.stats().addCallback(gotStats)
```

# valuer.py

```
#!/usr/bin/env python
#
# mcmandelbrot
#
# An example package for AsynQueue:
# Asynchronous task queueing based on the Twisted framework, with task
# prioritization and a powerful worker interface.
#
# Copyright (C) 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.


"""
Point valuer for L{mcmandelbrot}. Each CPU core has its own copy
of L{MandelbrotValuer} that is called via the
C{AsynQueue.ProcessQueue}.
"""

import sys, time, array

import png
import weave
from weave.base_info import custom_info
import numpy as np

from zope.interface import implements
from twisted.internet import defer, reactor
from twisted.internet.interfaces import IPushProducer

import asynqueue
from asynqueue.threads import Consumerator


from colormap import ColorMapper
```

```
class my_info(custom_info):
    _extra_compile_args = ['-Wcpp']


class MandelbrotValuer(object):
    """
    Returns the values (number of iterations to escape, if at all,
    inverted) of the Mandelbrot set at point cr + i*ci in the complex
    plane, for a range of real values with a constant imaginary component.

    C code adapted from Ilan Schnell's C{iterations} function at::

      https://svn.enthought.com/svn/enthought/Mayavi/
        branches/3.0.4/examples/mayavi/mandelbrot.py}

    with periodicity testing and test-interval updating adapted from
    Simpsons's code contribution at::

      http://en.wikipedia.org/wiki/User:Simpsons_contributor/
        periodicity_checking

    and period-2 bulb testing from Wikibooks::

      http://en.wikibooks.org/wiki/Fractals/
        Iterations_in_the_complex_plane/Mandelbrot_set

    The values are inverted, i.e., subtracted from the maximum value,
    so that no-escape points (technically, the only points actually in
    the Mandelbrot Set) have zero value and points that escape
    immediately have the maximum value. This allows simple mapping to
    the classic image with a black area in the middle. Then they are
    scaled to the 0.0-1.0 range, and an exponent is applied to
    emphasize changes at shorter escape times. Finally, they are
    mapped to RGB triples and returned.

    @ivar cm: A callable object that converts C{NumPy} array inputs in
      the 0.0-1.0 range to an unsigned-int8 Python array of RGB
      triples.
    """
    support_code = """
    bool region_test(double zr, double zr2, double zi2)
    {
        double q;
        // (x+1)^2 + y2 < 1/16
        if (zr2 + 2*zr + 1 + zi2 < 0.0625) return(true);
        // q = (x-1/4)^2 + y^2
        q = zr2 - 0.5*zr + 0.0625 + zi2;
        // q*(q+(x-1/4)) < 1/4*y^2
        q *= (q + zr - 0.25);
        if (q < 0.25*zi2) return(true);
        return(false);
    }

    int eval_point(int j, int km, double cr, double ci)
```

```
{
    int k = 1;
    int N = km;
    double zr = cr;
    double zi = ci;
    double zr2 = zr * zr, zi2 = zi * zi;
    // If we are in one of the two biggest "lakes," we need go no further
    if (region_test(zr, zr2, zi2)) return N;
    // Periodicity-testing variables
    double zrp = 0, zip = 0;
    int k_check = 0, N_check = 3, k_update = 0;
    while ( k < N ) {
        // Compute Z[n+1] = Z[n]^2 + C, with escape test
        if ( zr2+zi2 > 16.0 ) return k;
        zi = 2.0 * zr * zi + ci;
        zr = zr2 - zi2 + cr;
        k++;
        // Periodicity test: If same point is reached as previously,
        // there is no escape
        if ( zr == zrp )
            if ( zi == zip ) return N;
        // Check if previous-value update needed
        if ( k_check == N_check )
        {
            // Yes, do it
            zrp = zr;
            zip = zi;
            // Check again after another N_check iterations, an
            // interval that occasionally doubles
            k_check = 0;
            if ( k_update == 5 )
            {
                k_update = 0;
                N_check *= 2;
            }
            k_update++;
        }
        k_check++;
        // Compute squares for next iteration
        zr2 = zr * zr;
        zi2 = zi * zi;
    }
}
"""
code = """
#define NPY_NO_DEPRECATED_API NPY_1_7_API_VERSION
int j, zint;
int N = km;
signed char kx, ky;
double xk, yk;
for (j=0; j<Nx[0]; j++) {
    // Evaluate five points in an X arrangement including and around the
    // one specified by X1(j) and ci
    zint = eval_point(j, km, X1(j), ci);
```

```
        Z1(j) = zint;
        kx = -1;
        ky = -1;
        while ((zint < km) && (kx < 2)) {
            xk = X1(j) + kx * qd;
            while ((zint < km) && (ky < 2)) {
                yk = (double)ci + ky * qd;
                zint = eval_point(j, km, xk, yk);
                Z1(j) += zint;
                ky += 2;
            }
            kx += 2;
        }
        if (zint == km) {
            // A no-escape evaluation at one point in the X is treated
            // as if there were no escape at any point in the X
            Z1(j) = 5*N;
        }
    }
    """
    vars = ['x', 'z', 'ci', 'qd', 'km']

    def __init__(self, N_values, steepness):
        """
        Constructor:

        @param N_values: The number of iterations to try, hence the
          range of integer values, for a single call to
          L{computeValues}. Because a 5-point star around each point
          is evaluated with the values summed, the actual range of
          values for each point is 5 times greater.

        @param steepness: The amount to rescale the values after
          scaling to the -0.5 to 0.5 range and before applying the
          logistic function and color mapping.
        """
        self.N_values = N_values
        self.steepness = steepness
        self.cm = ColorMapper()
        # The maximum possible escape value is mapped to 1.0, before
        # exponent and then color mapping are applied
        self.scale = 0.2 / N_values
        self.infoObj = my_info()

    def __call__(self, crMin, crMax, N, ci):
        """
        Computes values for I{N} points along the real (horizontal) axis
        from I{crMin} to I{crMax}, with the constant imaginary
        component I{ci}.

        @return: A Python B-array I{3*N} containing RGB triples for an
          image representing the escape values.
        """
        qd = 0.25 * (crMax - crMin) / N
```

```
        x = np.linspace(crMin, crMax, N, dtype=np.float64)
        z = self.computeValues(N, x, ci, qd)
        # Invert the iteration values so that trapped points have zero
        # value, then scale to the range [-1.0, +1.0]
        z = 2*self.scale * (5*self.N_values - z) - 1.0
        # Transform to emphasize details in the middle
        z = self.transform(z, self.steepness)
        # [-1.0, +1.0] --> [0.0, 1.0]
        z = 0.5*(z + 1.0)
        # Map to my RGB colormap
        return self.cm(z)

    def computeValues(self, N, x, ci, qd):
        """
        Computes and returns a row vector of escape iterations, integer
        values.
        """
        km = self.N_values - 1
        z = np.zeros(N, dtype=np.int)
        weave.inline(
            self.code, self.vars,
            customize=self.infoObj, support_code=self.support_code)
        return z

    def transform(self, x, k):
        """
        Transforms the input vector I{x} by taking it to a power, which is
        zero (no transform) or odd-numbered.
        """
        return np.power(x, k)
```

# vroot.py

```
#!/usr/bin/env python
#
# mcmandelbrot
#
# An example package for AsynQueue:
# Asynchronous task queueing based on the Twisted framework, with task
# prioritization and a powerful worker interface.
#
# Copyright (C) 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{VRoot} combines with the L{Baton} to provide a powerful way
to generate an HTML page. Adapted from another project for use in the
C{mcmandelbrot} demo site.
"""

from contextlib import contextmanager
import os.path, sys, re, traceback, codecs
from pkg_resources import resource_stream

from xml.dom import minidom
import xml.etree.ElementTree as ET
from xml.etree.ElementTree import SubElement


def newElement(tag, parent=None):
    if parent is None:
        return ET.Element(tag)
    return SubElement(parent, tag)

def dedent(lines, padding):
    result = []
    kChar = min([len(x)-len(x.lstrip()) for x in lines])
    for line in lines:
        result.append(u" "*padding + line[kChar:])
```

```
    return result

def dedentString(text, padding):
    lines = text.split('\n')
    return '\n'.join(dedent(lines, padding))

def openPackageFile(fileName):
    filePath = os.path.join(
        os.path.dirname(__file__), fileName)
    if os.path.exists(filePath):
        return open(filePath)
    return resource_stream(__name__, fileName)


class Baton(object):
    """
    The L{VRoot} gives an instance of me a reference I{e} to a virtual
    root element and then passes my instance to a context caller. The
    caller can use my convenience methods to generate XML subelements
    of the virtual root element, with placeholders for raw
    xml/html. Then, when the caller is done, I convert the tree from
    the virtual root element into xml/html.
    """
    reInvalidXML = re.compile(r'[\<\>\&]')
    reContentPara = re.compile(r'<p>(.+)</p>\s*$')
    reLeadingSpace = re.compile(r'\n*</[^>]+>\n*(\s*?)<')

    def __init__(self, indent):
        self.cStack = []
        self.indent = indent
        self.elements = []
        self.eChild = None
        self.lastParent = None
        self.e = self.seParent = newElement('vroot')
        self.uniquePrefix = "!UPH-{:d}".format(id(self))
        self.rePlaceholder = re.compile(
            r'{}-([0-9]+)!'.format(self.uniquePrefix))
        self.eMap = {}

    def info(self, e=None):
        info = ""
        if e is None:
            e = self.eChild
        if e is None:
            return "<No child yet>"
        klasses = e.get('class', "")
        if klasses:
            info += ' class="{}"'.format(klasses)
        text = e.text if e.text else ""
        result = "<{}{}>{}</{}>".format(
            e.tag, info, text, e.tag)
        children = list(e)
        if children:
            result += ": [{}]".format(", ".join(
                    [self.info(x) for x in children]))
```

```
        return result

    def __repr__(self):
        return "Last child: {}".format(self.info())

    def parent(self, e):
        """
        Returns the parent of the supplied element.
        """
        xpath = ".//{}/..".format(e.tag)
        for possibleParent in self.e.findall(xpath):
            if e in list(possibleParent):
                return possibleParent

    def parentAndClassFromArgs(self, args):
        klass = None
        parent = self.e if self.eChild is None else self.eChild
        for arg in args:
            if ET.iselement(arg):
                # A parent was specified, don't use last child's parent
                parent = arg
            elif isinstance(arg, (str, unicode)):
                # A class was specified
                klass = arg
        return parent, klass

    def se(self, tag, klass=None):
        """
        Convenience method to return a subelement of the last child I
        generated (from a subclass entryMethod) before being offered
        to the context caller. You can use this to generate top-level
        items within the context of your call to a VROOT.

        You can specify a CSS class for the new subelement.

        B{Do not use} this method in a VROOT's entryMethod! Doing so
        will screw up the I{lastParent}. It is for context callers.
        """
        self.lastParent = self.seParent
        e = self.eChild = newElement(tag, self.seParent)
        if klass:
            e.set('class', klass)
        return e

    def nc(self, tag, *args):
        """
        Generates a new child of the last-generated child, another
        parent specified as an argument, or my root if this is the
        first child I've generated.

        Repeated calls to this method without parents specified will
        result in a nested hierarchy.

        If a class name is specified via a string argument, the new
        child gets that set. The order of the arguments doesn't
```

```
        matter, and neither or both can be specified.

        Returns a reference to the new element and saves it as the
        last-generated child.
        """
        parent, klass = self.parentAndClassFromArgs(args)
        self.lastParent = parent
        e = self.eChild = newElement(tag, parent)
        if klass:
            e.set('class', klass)
        return e

    def nci(self, iterator, tag, *args):
        """
        Iterates over the supplied iterator to generate children of the
        last-generated child, another parent specified as an argument,
        or my root if this is the first child I've generated.

        Each child generated will be considered the last-generated one
        within its iteration, and only there. Children iterated after
        the first one will be siblings to the first.

        If a class name is specified via a string argument, each new
        child gets that set. The order of the arguments doesn't
        matter, and neither or both can be specified.

        For each iteration, generates a new child of the parent and
        yields the value of the original iteration. The new child will
        be treated the last-generated child for that iteration, and
        only that iteration.

        It's expected that you will use stuff like nc to make further
        children (i.e., grandchilden of the last-generated child)
        within each iteration, but after the iterations are done, the
        last-generated child will be restored to where it was before
        the method call.

        B{Caution}: Weird things happen if you do a C{break} or
        C{continue}. Be warned!
        """
        def eNew():
            self.eChild = newElement(tag, parent)
            if klass:
                self.eChild.set('class', klass)

        parent, klass = self.parentAndClassFromArgs(args)
        self.lastParent = parent
        self._nciChild = self.eChild
        if callable(iterator):
            for x in iterator():
                eNew()
                yield x
        else:
            for x in iterator:
                eNew()
```

```
                yield x
        self.eChild = self._nciChild

    def ns(self, tag, *args):
        """
        Generates a new child of B{the parent} to the last-generated child
        or another sibling element specified. The new child will then
        be considered the last-generated one.

        Repeated calls to this method (with or without a sibling
        element specified) will result in siblings rather than a nested
        hierarchy, because the parent doesn't change even if the child
        does.

        If a class name is specified, the new child gets that set.

        Returns a reference to the new element and saves it as the
        last-generated child.
        """
        sibling, klass = self.parentAndClassFromArgs(args)
        if sibling not in (self.e, self.eChild):
            # A sibling was supplied in the args, use its parent
            self.lastParent = self.parent(sibling)
        parent = self.lastParent
        e = self.eChild = newElement(tag, parent)
        if klass:
            e.set('class', klass)
        # Note that self.lastParent does not change, unless a sibling
        # was specified. Then its parent will be the new lastParent,
        # for further calls to this method without needing to supply
        # the sibling again.
        return e

    def nu(self, tag, klass=None):
        """
        Generates a new uncle of the last-generated child. This will
        then be considered the last-generated "child." If a class name
        is specified, the new element gets that set.

        Useful for making one or more children of an element, and then
        going on to make a sibling of that element.
        """
        return self.ns(tag, self.parent(self.eChild), klass)

    def p(self, content, *args):
        """
        Generates a paragraph element as a new child, with either the
        supplied content as its text, or, if the content is wrapped in
        paragraph tags, a placeholder for substituting (without the
        <p> tags) after serialization.

        The new element is a child of the last-generated child, unless
        the last child was also a paragraph. Then the new element is a
        sibling of that paragraph. You can also specify another parent
        as an argument. The root is used if this is the first child
```

```
        I've generated. Repeated calls to this method will produce
        sibling paragraphs, not a nested hierarchy.

        If a class name is specified via a string argument, the
        paragraph element gets that set. The order of any parent/klass
        arguments doesn't matter, and neither or both can be
        specified.
        """
        parent, klass = self.parentAndClassFromArgs(args)
        if getattr(self.eChild, 'tag', None) == 'p':
            parent = self.lastParent
        else:
            self.lastParent = parent
        e = self.eChild = newElement('p', parent)
        if klass:
            e.set('class', klass)
        content = re.sub(r'\s*\n+\s*', ' ', content)
        match = self.reContentPara.match(content)
        if match:
            content = self.np(match.group(1))
        e.text = content
        return e

    def ngc(self, tag, *args):
        """
        Generates a new subelement (somebody's grandchild) of the child
        last generated with the se, nc, ns, or p method without
        changing its status as the last child generated. You can
        specify a parent and class, as usual, in either order.

        Repeated calls to this method will result in a succesion of
        grandchildren with the same ancestors, not a nested hierarchy.

        If a class name is specified, the new element gets that
        set.

        Returns a reference to the new element without saving it
        anywhere.
        """
        parent, klass = self.parentAndClassFromArgs(args)
        eGrandChild = newElement(tag, parent)
        if klass:
            eGrandChild.set('class', klass)
        return eGrandChild

    def rp(self, parent=None):
        """
        Reset parent to use for the next child I generate, to the one
        specified or my seParent. Returns a reference to me (not the
        parent) for convenience of methods called by my context caller
        and using me to build stuff.
        """
        if parent is None:
            parent = self.seParent
        self.eChild = self.lastParent = parent
```

```
        return self

    @contextmanager
    def context(self):
        """
        Isolates a context for you to generate one or more children
        without affecting my overall context. When the context code
        completes, restores the last parent/child context to what it
        was before.
        """
        self.cStack.append((self.eChild, self.lastParent))
        yield
        self.eChild, self.lastParent = self.cStack.pop()

    def np(self, content):
        """
        Assigns a unique placeholder for the supplied content string
        and stores it to be replaced back during
        serialization. Returns the placeholder.
        """
        ph = "{}-{:d}!".format(
            self.uniquePrefix, len(self.elements))
        self.elements.append(content)
        return ph

    def possiblyPlacehold(self, e, content, attrName='text'):
        """
        If the supplied xml/html content string contains a character
        that must be escaped in XML (e.g., "<"), assigns a unique
        placeholder for the entire string and stores the content to be
        replaced back during serialization.
        """
        if self.reInvalidXML.search(content):
            content = self.np(content)
        setattr(e, attrName, content)

    def text(self, content, e=None):
        """
        Sets the text value of the last-generated child (or an element
        specified) to the supplied content, with XML escaping if
        needed.
        """
        if e is None:
            e = self.eChild
        e.text = content

    def textX(self, content, e=None):
        """
        Sets the text value of the last-generated child (or an element
        specified) to the supplied content, with a placeholder if
        needed to preserve raw XML.
        """
        if e is None:
            e = self.eChild
        if e.tag in ('div'):
```

```
            content = content.strip()
            content = "\n" + content + "\n"
        self.possiblyPlacehold(e, content)

    def tail(self, content, e=None):
        """
        Sets the tail value of the last-generated child (or an element
        specified) to the supplied content, with XML escaping if
        needed.
        """
        if e is None:
            e = self.eChild
        e.tail = content

    def tailX(self, content, e=None):
        """
        Sets the tail value of the last-generated child (or an element
        specified) to the supplied content, with a placeholder if
        needed to preserve raw XML.
        """
        if e is None:
            e = self.eChild
        self.possiblyPlacehold(e, content, 'tail')

    def set(self, name, value):
        """
        Sets an attribute of my last-generated child element.
        """
        self.eChild.set(name, value)

    def meta(self, *args, **kw):
        """
        Adds a meta tag as a new child, to the last child or a parent
        specified as a single argument, using the keywords supplied.

        The meta tag becomes the new last child.
        """
        parent = args[0] if args else None
        self.nc('meta', parent)
        for name, value in kw.iteritems():
            name = name.replace('_', '-')
            self.set(name, value)

    def margin(self, side, em):
        """
        Convenience method to set a margin of my last-generated child
        element. Specify in em with a float.
        """
        style = [self.eChild.get('style', "").strip()]
        style.append("margin-{}: {:1.1f}em;".format(side, em))
        self.set('style', " ".join(style).strip())

    def addToMap(self, elementID, attrName):
        """
        Add attribute I{attrName} of the last-generated child to my
```

```
        dynamic-value map.
        """
        self.eMap[elementID] = self.eChild, attrName

    def html(self, **kw):
        """
        Gets my HTML rendering, after substituting out any placeholders
        and applying keywords to my mapped elements.
        """
        for elementID, newValue in kw.iteritems():
            if elementID in self.eMap:
                e, attrName = self.eMap[elementID]
                if newValue is None:
                    e.attrib.pop(attrName, None)
                else:
                    e.set(attrName, newValue)
        xml = ET.tostring(self.e)
        while True:
            match = self.rePlaceholder.search(xml)
            if not match:
                break
            k = int(match.group(1))
            before = xml[:match.start(0)]
            after = xml[match.end(0):]
            replacement = self.elements[k]
            xml = before + replacement + after
        return "<html>\n{}\n</html>".format(re.sub(r"</?vroot>", r"", xml))


class Meta(type):
    """
    Load class-wide lists of lines for CSS and JS
    """
    def __new__(cls, name, parents, dct):
        dct['headLines'] = {}
        for fileType, fileName in dct['headFiles'].iteritems():
            lines = []
            fh = openPackageFile(fileName)
            for line in fh:
                stripped = line.strip()
                if not stripped:
                    continue
                if stripped.startswith('/'):
                    continue
                if stripped.startswith('# '):
                    continue
                lines.append(line.rstrip())
            fh.close()
            dct['headLines'][fileType] = lines
        return super(Meta, cls).__new__(cls, name, parents, dct)


class VRoot(object):
    """
    I am a context manager for passing you a baton with a virtual root
```

```
element (my "e" attribute) to which you can add elements, e.g., an
HTML document. When you get done, I'll put an XML string in the
baton, with the XML tag stripped out and replaced with HTML tags.

The baton has tons of convenience methods for generating tags::

  with (me) as vroot:
      ncx = vroot.se('ncx')
      ...
  xml = vroot.xml [ or = vroot() ]
  ...

Call my instance to get the XML or HTML as a string.
"""
# Number of spaces to indent each XML level
indent = 2

versionXML = 1.0
charset = "utf-8"
replaceXML = "html"
headFiles = {'css': "mcm.css", 'js': "mcm.js"}

__metaclass__ = Meta

def __init__(self, title):
    self.title = title
    self.metaTags = [
        {'name':"viewport",
         'content':"width=device-width, initial-scale=1"}]

def insert(self, v, tag, name, typ=None):
    """
    Inserts the text read from the named entry in my I{headLines}
    dict.
    """
    v.ns(tag)
    v.textX(u"\n"+u"\n".join(self.headLines[name]))
    if typ:
        v.set('type', typ)

def head(self, v):
    with v.context():
        h = v.nc('head')
        v.nc('title')
        # There might be HTML formatting chars in the title string
        v.textX(self.title)
        # Meta
        v.meta(
            h, charset=self.charset,
            content="application/xhtml+xml",
            http_equiv="Content-Type")
        for kw in self.metaTags:
            v.meta(h, **kw)
        self.insert(v, 'style', 'css')
        self.insert(v, 'script', 'js', "text/javascript")
```

```
def __call__(self, **kw):
    """
    Returns my content rendered as a complete HTML bytestring, with
    keywords substituting attributes values of mapped
    elements. Each keyword's value is the ID that the element was
    assigned its value is new value of the attribute that was
    mapped for that element.
    """
    html = self.b.html(**kw)
    # Not sure this next line is necessary. Seems to work fine without it.
    #html = self.fixCloseTags(html)
    return bytes(html)

def __enter__(self):
    self.b = Baton(self.indent)
    self.b.eMap = {}
    for name in ('version', 'xmlns'):
        value = getattr(self, name, None)
        if value:
            setattr(self.b, name, value)
    # Possible custom entry method
    self.head(self.b)
    return self.b

def __exit__(self, etype, value, trace):
    if etype is not None:
        traceback.print_exception(etype, value, trace)
        sys.exit(1)
```

# wire.py

```
#!/usr/bin/env python
#
# mcmandelbrot
#
# An example package for AsynQueue:
# Asynchronous task queueing based on the Twisted framework, with task
# prioritization and a powerful worker interface.
#
# Copyright (C) 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.


"""
Uses C{asynqueue.wire} to run and communicate with a server that
generates Mandelbrot Set images.

For the server end, use L{server} to get a Twisted C{service} object
you can start or add to an C{application}.

To communicate with the server, use L{RemoteRunner} to get an
AsynQueue C{Worker} that you can add to your C{TaskQueue}.

Both ends of the connection need to be able to import this module and
reference its L{MandelbrotWorkerUniverse} class.
"""

import sys, urlparse
from collections import namedtuple

from zope.interface import implements
from twisted.application import internet, service
from twisted.internet import defer, reactor
from twisted.internet.interfaces import IConsumer

from asynqueue import iteration
from asynqueue.base import TaskQueue
from asynqueue.wire import \
```

```
    WireWorkerUniverse, WireWorker, WireServer, ServerManager

from mcmandelbrot import runner


# Server Connection defaults
PORT = 1978
INTERFACE = None
DESCRIPTION = None
VERBOSE = True

FQN = "mcmandelbrot.wire.MandelbrotWorkerUniverse"


class Writable(object):
    delay = iteration.Delay()

    def __init__(self):
        self.ID = str(hash(self))
        self.data = []

    def write(self, data):
        self.data.append(data)

    def close(self):
        self.data.append(None)

    def getNext(self):
        def haveData(really):
            if really:
                return self.data.pop(0)
        return self.delay.untilEvent(
            lambda: bool(self.data)).addCallback(haveData)


class MandelbrotWorkerUniverse(WireWorkerUniverse):
    """
    I run on the remote Python interpreter to run a runner from there,
    accepting commands from your L{RemoteRunner} via L{run},
    L{done} and L{cancel} and sending the results and iterations
    to it.
    """
    RunInfo = namedtuple('RunInfo', ['fh', 'dRun', 'dCancel'])

    def setup(self, N_values, steepness):
        def ready(null):
            # These methods are called via a one-at-a-time queue, so
            # it's not a problem to overwrite the old runner with a
            # new one, now that we've waited for the old one's runs to
            # get canceled and then for it to shut down.
            self.pendingRuns = {}
            self.runner = runner.Runner(N_values, steepness, verbose=VERBOSE)
        return self.shutdown().addCallback(ready)

    @defer.inlineCallbacks
```

```
def shutdown(self):
    if hasattr(self, 'runner'):
        dList = []
        for ri in self.pendingRuns.itervalues():
            dList.append(ri.dRun)
            if not ri.dCancel.called:
                ri.dCancel.callback(None)
        yield defer.DeferredList(dList)
        yield self.runner.q.shutdown()

def run(self, Nx, cr, ci, crPM, ciPM, requester=None):
    """
    Does an image-generation run for the specified parameters, storing
    a C{Deferred} I{dRun} to the result in a C{namedtuple} along
    with a reference I{fh} to a new L{Writable} that will have the
    image data written to it and a C{Deferred} I{dCancel} that can
    have its callback fired to cancel the run.

    @return: A unique string I{ID} identifying the run.
    """
    def doneHere(stuff):
        fh.close()
        return stuff

    fh = Writable()
    dCancel = defer.Deferred()
    dRun = self.runner.run(
        fh, Nx, cr, ci, crPM, ciPM,
        dCancel=dCancel, requester=requester).addCallback(doneHere)
    self.pendingRuns[fh.ID] = self.RunInfo(fh, dRun, dCancel)
    return fh.ID

def getNext(self, ID):
    """
    Gets the next chunk of data to have been written to the
    L{Writable} for the run identified by I{ID}.

    @return: A C{Deferred} that fires with the data chunk when it
      is received, or an immediate C{None} object if the run isn't
      pending.
    """
    if ID in self.pendingRuns:
        return self.pendingRuns[ID].fh.getNext()

def done(self, ID):
    """
    Gets the runtime and number of points done in the run and removes
    my references to it in my I{pendingRuns} dict.

    @return: A C{Deferred} that fires with the runtime and number
      of points when the run is done, or an immediate C{None}
      object if there never was any such run or this method was
      called with this I{ID} already.
    """
```

```
        if ID in self.pendingRuns:
            return self.pendingRuns.pop(ID).dRun

    def cancel(self, ID):
        """
        Cancels a pending run I{ID}. Nothing is returned, and no exception
        is raised for calling with reference to an unknown or already
        canceled/completed run.
        """
        if ID in self.pendingRuns:
            dCancel = self.pendingRuns[ID].dCancel
            if not dCancel.called:
                dCancel.callback(None)


class RemoteRunner(object):
    """
    Call L{setup} and wait for the C{Deferred} it returns, then you
    can call L{image} as much as you like to get images streamed to
    you as iterations of C{Deferred} chunks.

    Call L{shutdown} when done, unless you are using both a remote
    server and an external instance of C{TaskQueue}.
    """
    setupDefaults = {'N_values': 3000, 'steepness': 3}

    def __init__(self, description=None, q=None):
        self.description = description
        self.sv = self.setupDefaults.copy()
        if q is None:
            self.q = TaskQueue()
            self.stopper = self.q.shutdown
        else:
            self.q = q

    @defer.inlineCallbacks
    def setup(self, **kw):
        """
        Call at least once to set things up. Repeated calls with the same
        keywords, or with no keywords, do nothing. Keywords with a
        value of C{None} are ignored.

        @keyword N_values: The number of possible values for each
          iteration.

        @keyword steepness: The steepness of the exponential applied to
          the value curve.

        @keyword worker: A custom worker to use instead of
          C{asynqueue.wire.WireWorker}.

        @return: A C{Deferred} that fires when things are setup, or
          immediately if they already are as specified.
        """
        def checkSetup():
```

```
            result = False
            if 'FLAG' not in self.sv:
                self.sv['FLAG'] = True
                result = True
            for name, value in kw.iteritems():
                if value is None:
                    continue
                if self.sv.get(name, None) != value:
                    self.sv.update(kw)
                    return True
            return result

        if checkSetup():
            if 'worker' in kw:
                worker = kw.pop('worker')
            else:
                if self.description is None:
                    # Local server running on a UNIX socket. Mostly
                    # useful for testing.
                    self.mgr = ServerManager(FQN)
                    description = self.mgr.newSocket()
                    yield self.mgr.spawn(description)
                else:
                    description = self.description
                wwu = MandelbrotWorkerUniverse()
                worker = WireWorker(wwu, description, series=['mcm'])
            yield self.q.attachWorker(worker)
            yield self.q.call(
                'setup',
                self.sv['N_values'], self.sv['steepness'], series='mcm')

    @defer.inlineCallbacks
    def shutdown(self):
        if hasattr(self, 'mgr'):
            yield self.mgr.done()
        if hasattr(self, 'stopper'):
            yield self.stopper()

    @defer.inlineCallbacks
    def run(self, fh, Nx, cr, ci, crPM, ciPM, dCancel=None, requester=None):
        """
        Runs a C{compute} method on a remote Python interpreter to
        generate a PNG image of the Mandelbrot Set and write it in
        chunks, indirectly, to the write-capable object I{fh}, which
        in this case must implement C{IConsumer}. When this method is
        called by L{image.Imager.renderImage}, I{fh} will be a request
        and those do implement C{IConsumer}.

        The image is centered at location I{cr, ci} in the complex
        plane, plus or minus I{crPM} on the real axis and I{ciPM} on
        the imaginary axis.

        @see: L{runner.run}.
```

```
        This method doesn't call L{setup}; that is taken care of by
        L{image.Imager} for HTTP requests and by L{writeImage} for
        local image file generation.

        @return: A C{Deferred} that fires with the total elasped time
          for the computation and the number of pixels computed.
        """
        def canceler(null, ID):
            return self.q.call('cancel', ID, niceness=-15, series='mcm')

        ID = yield self.q.call(
            'run', Nx, cr, ci, crPM, ciPM, requester=requester, series='mcm')
        if dCancel:
            dCancel.addCallback(canceler, ID)
        while True:
            chunk = yield self.q.call('getNext', ID, series='mcm', raw=True)
            if chunk is None:
                break
            fh.write(chunk)
        runInfo = yield self.q.call('done', ID)
        defer.returnValue(runInfo)

    @defer.inlineCallbacks
    def writeImage(self, fileName, *args, **kw):
        """
        Call with the same arguments as L{run} after I{fh}, preceded by a
        writable I{fileName}. It will be opened for writing and its
        file handle supplied to L{run} as I{fh}.

        Writes the PNG image as it is generated remotely, returning a
        C{Deferred} that fires with the result of L{run} when the
        image is all written.

        @see: L{setup} and L{run}
        """
        N_values = kw.pop('N_values', None)
        steepness = kw.pop('steepness', None)
        yield self.setup(N_values=N_values, steepness=steepness)
        fh = open(fileName, 'w')
        runInfo = yield self.run(fh, *args)
        fh.close()
        defer.returnValue(runInfo)

    def showStats(self, runInfo):
        proto = "Remote server computed {:d} pixels in {:1.1f} seconds."
        print proto.format(runInfo[1], runInfo[0])
        return defer.succeed(None)


def server(description=None, port=1978, interface=None):
    """
    Creates a Twisted C{endpoint} service for Mandelbrot Set images.

    The returned C{service} responds to connections as specified by
    I{description} and accepts 'image' commands via AMP to produce PNG
```

```
    images of the Mandelbrot set in a particular region of the complex
    plane.

    If you omit the I{description}, it will be a TCP server running on
    a particular I{port}. The default is C{1978}, which is the year in
    which the first computer image of the Mandelbrot set was
    generated. You can specify an I{interface} dotted-quad address for
    the TCP server if you want to limit connections that way.

    @see: L{MandelbrotWorkerUniverse.image}
    """
    if description is None:
        description = b"tcp:{:d}".format(port)
        if interface:
            description += ":interface={}".format(interface)
    mwu = MandelbrotWorkerUniverse()
    reactor.addSystemEventTrigger('before', 'shutdown', mwu.shutdown)
    ws = WireServer(mwu)
    return ws.run(description)


if '/twistd' in sys.argv[0]:
    application = service.Application("Mandelbrot Set PNG Image Server")
    server(DESCRIPTION, PORT, INTERFACE).setServiceParent(application)
```