nodebox/__init__.py

```
__version__='1.9.23'

def get_version():
    return __version__
```

nodebox/console.py

```
import AppKit
NSApplication = AppKit.NSApplication

try:
    import nodebox
except ImportError:
    import sys, os
    nodebox_dir = os.path.dirname(os.path.abspath(__file__))
    sys.path.append(os.path.dirname(nodebox_dir))

import nodebox.graphics
graphics = nodebox.graphics

import nodebox.util
util = nodebox.util

#from nodebox import graphics
#from nodebox import util

class NodeBoxRunner(object):

    def __init__(self):
        # Force NSApp initialisation.
        NSApplication.sharedApplication().activateIgnoringOtherApps_(0)
        self.namespace = {}
        self.canvas = graphics.Canvas()
        self.context = graphics.Context(self.canvas, self.namespace)
        self.__doc__ = {}
        self._pageNumber = 1
        self.frame = 1

    def _check_animation(self):
        """Returns False if this is not an animation, True otherwise.
        Throws an expection if the animation is not correct (missing a draw method)."""
        if self.canvas.speed is not None:
            if not self.namespace.has_key('draw'):
                raise graphics.NodeBoxError('Not a correct animation: No draw() method.')
            return True
        return False

    def run(self, source_or_code):
        self._initNamespace()
        if isinstance(source_or_code, basestring):
            source_or_code = compile(source_or_code + "\n\n", "<Untitled>", "exec")
        exec source_or_code in self.namespace
        if self._check_animation():
            if self.namespace.has_key('setup'):
                self.namespace['setup']()
            self.namespace['draw']()

    def run_multiple(self, source_or_code, frames):
        if isinstance(source_or_code, basestring):
            source_or_code = compile(source_or_code + "\n\n", "<Untitled>", "exec")
```

```
        # First frame is special:
        self.run(source_or_code)
        yield 1
        animation = self._check_animation()

        for i in range(frames-1):
            self.canvas.clear()
            self.frame = i + 2
            self.namespace["PAGENUM"] = self.namespace["FRAME"] = self.frame
            if animation:
                self.namespace['draw']()
            else:
                exec source_or_code in self.namespace
            yield self.frame

    def _initNamespace(self, frame=1):
        self.canvas.clear()
        self.namespace.clear()
        # Add everything from the namespace
        for name in graphics.__all__:
            self.namespace[name] = getattr(graphics, name)
        for name in util.__all__:
            self.namespace[name] = getattr(util, name)
        # Add everything from the context object
        self.namespace["_ctx"] = self.context
        for attrName in dir(self.context):
            self.namespace[attrName] = getattr(self.context, attrName)
        # Add the document global
        self.namespace["__doc__"] = self.__doc__
        # Add the frame
        self.frame = frame
        self.namespace["PAGENUM"] = self.namespace["FRAME"] = self.frame

def make_image(source_or_code, outputfile):

    """Given a source string or code object, executes the scripts and saves the result as
    an image.  Supported image extensions: pdf, tiff, png, jpg, gif"""

    runner = NodeBoxRunner()
    runner.run(source_or_code)
    runner.canvas.save(outputfile)

def make_movie(source_or_code, outputfile, frames, fps=30):

    """Given a source string or code object, executes the scripts and saves the result as
    a movie.

    You also have to specify the number of frames to render.
    Supported movie extension: mov"""

    from nodebox.util import QTSupport
    runner = NodeBoxRunner()
    movie = QTSupport.Movie(outputfile, fps)
    for frame in runner.run_multiple(source_or_code, frames):
        movie.add(runner.canvas)
    movie.save()

def usage(err=""):
    if len(err) > 0:
        err = '\n\nError: ' + str(err)
    print """NodeBox console runner
Usage: console.py sourcefile imagefile
   or: console.py sourcefile moviefile number_of_frames [fps]
Supported image extensions: pdf, tiff, png, jpg, gif
```

```
    Supported movie extension:  mov""" + err
120
    def main():
        import sys, os
        if len(sys.argv) < 2:
            usage()
125     elif len(sys.argv) == 3: # Should be an image
            basename, ext = os.path.splitext(sys.argv[2])
            if ext not in ('.pdf', '.gif', '.jpg', '.jpeg', '.png', '.tiff'):
                return usage('This is not a supported image format.')
            make_image(open(sys.argv[1]).read(), sys.argv[2])
130     elif len(sys.argv) == 4 or len(sys.argv) == 5: # Should be a movie
            basename, ext = os.path.splitext(sys.argv[2])
            if ext != '.mov':
                return usage('This is not a supported movie format.')
            if len(sys.argv) == 5:
135             try:
                    fps = int(sys.argv[4])
                except ValueError:
                    return usage()
            else:
140             fps = 30
            make_movie(open(sys.argv[1]).read(), sys.argv[2], int(sys.argv[3]), fps)

    def test():
        # Creating the NodeBoxRunner class directly:
145     runner = NodeBoxRunner()
        testscript = ('size(500,500)\n'
                      'for i in range(400):\n'
                      '  oval(random(WIDTH),random(HEIGHT),50,50, '
                      'fill=(random(), 0,0,random()))')
150     runner.run(testscript)
        runner.canvas.save('console-test.pdf')
        runner.canvas.save('console-test.png')

        # Using the runner for animations:
155     runner = NodeBoxRunner()
        for frame in runner.run_multiple('size(300, 300)\ntext(FRAME, 100, 100)', 10):
            runner.canvas.save('console-test-frame%02i.png' % frame)

        # Using the shortcut functions:
160     make_image('size(200,200)\ntext(FRAME, 100, 100)', 'console-test.gif')
        make_movie('size(200,200)\ntext(FRAME, 100, 100)', 'console-test.mov', 10)

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

nodebox/PyFontify.py

```
    """Module to analyze Python source code; for syntax coloring tools.

    Interface:
        for tag, start, end, sublist in fontify(pytext, searchfrom, searchto):
 5          ...

    The 'pytext' argument is a string containing Python source code.
    The (optional) arguments 'searchfrom' and 'searchto' may contain a slice in pytext.
    The returned value is a list of tuples, formatted like this:
 10     [('keyword', 0, 6, None), ('keyword', 11, 17, None), ('comment', 23, 53, None), etc. ]
    The tuple contents are always like this:
        (tag, startindex, endindex, sublist)
    tag is one of 'keyword', 'string', 'comment' or 'identifier'
    sublist is not used, hence always None.
```

```
"""

# Based on FontText.py by Mitchell S. Chapman,
# which was modified by Zachary Roadhouse,
# then un-Tk'd by Just van Rossum.
# Many thanks for regular expression debugging & authoring are due to:
#   Tim (the-incredib-ly y'rs) Peters and Cristian Tismer
# So, who owns the copyright? ;-) How about this:
# Copyright 1996-2003:
#   Mitchell S. Chapman,
#   Zachary Roadhouse,
#   Tim Peters,
#   Just van Rossum

# from __future__ import generators

__version__ = "0.5"

import re
import graphics
import util

from keyword import kwlist as keywordsList
keywordsList = keywordsList[:]
keywordsList += ["None", "True", "False"]
keywordsList += graphics.__all__
keywordsList += util.__all__
keywordsList += dir(graphics.Context)

# Build up a regular expression which will match anything
# interesting, including multi-line triple-quoted strings.
commentPat = r"#[^\n]*"

pat = r"[uU]?[rR]?q[^\\q\n]*(\\[\000-\377][^\\q\n]*)*q?"
quotePat = pat.replace("q", "'") + "|" + pat.replace('q', '"')

# Way to go, Tim!
pat = r"""
    [uU]?[rR]?
    qqq
    [^\\q]*
    (
        (   \\[\000-\377]
        |   q
            (   \\[\000-\377]
            |   [^\q]
            |   q
                (   \\[\000-\377]
                |   [^\\q]
                )
            )
        )
        [^\\q]*
    )*
    (qqq)?
"""
pat = "".join(pat.split())  # get rid of whitespace
tripleQuotePat = pat.replace("q", "'") + "|" + pat.replace('q', '"')

# Build up a regular expression which matches all and only
# Python keywords. This will let us skip the uninteresting
# identifier references.
keyPat = r"\b(" + "|".join(keywordsList) + r")\b"
```

```
    matchPat = commentPat + "|" + keyPat + "|(" + tripleQuotePat + "|" + quotePat + ")"
 80 matchRE = re.compile(matchPat)

    idKeyPat = "[ \t]*([A-Za-z_][A-Za-z_0-9.]*)"    # Ident w. leading whitespace.
    idRE = re.compile(idKeyPat)
    asRE = re.compile(r".*?\b(as)\b")
 85
    def fontify(pytext, searchfrom=0, searchto=None):
        if searchto is None:
            searchto = len(pytext)
        # Cache a few attributes for quicker reference.
 90     search = matchRE.search
        idMatch = idRE.match
        asMatch = asRE.match

        commentTag = 'comment'
 95     stringTag = 'string'
        keywordTag = 'keyword'
        identifierTag = 'identifier'

        start = 0
100     end = searchfrom
        while 1:
            m = search(pytext, end)
            if m is None:
                break   # EXIT LOOP
105         if start >= searchto:
                break   # EXIT LOOP
            keyword = m.group(1)
            if keyword is not None:
                # matched a keyword
110             start, end = m.span(1)
                yield keywordTag, start, end, None
                if keyword in ["def", "class"]:
                    # If this was a defining keyword, color the
                    # following identifier.
115                 m = idMatch(pytext, end)
                    if m is not None:
                        start, end = m.span(1)
                        yield identifierTag, start, end, None
                elif keyword == "import":
120                 # color all the "as" words on same line;
                    # cheap approximation to the truth
                    while 1:
                        m = asMatch(pytext, end)
                        if not m:
125                         break
                        start, end = m.span(1)
                        yield keywordTag, start, end, None
            elif m.group(0)[0] == "#":
                start, end = m.span()
130             yield commentTag, start, end, None
            else:
                start, end = m.span()
                yield stringTag, start, end, None

135 def test(path):
        f = open(path)
        text = f.read()
        f.close()
        for tag, start, end, sublist in fontify(text):
140         print tag, repr(text[start:end])

    if __name__ == "__main__":
```

```
    import sys
    test(sys.argv[1])
```

nodebox/geo/__init__.py

```
# Geometric functionality

import math

try:
    # Faster C versions.
    import cGeo
    isqrt = inverse_sqrt = cGeo.fast_inverse_sqrt
    angle = cGeo.angle
    distance = cGeo.distance
    coordinates = cGeo.coordinates

except ImportError:
    def inverse_sqrt(x):
        return 1.0 / math.sqrt(x)

    isqrt = inverse_sqrt

    def angle(x0, y0, x1, y1):
        return math.degrees( math.atan2(y1-y0, x1-x0) )

    def distance(x0, y0, x1, y1):
        return math.sqrt(math.pow(x1-x0, 2) + math.pow(y1-y0, 2))

    def coordinates(x0, y0, distance, angle):
        x1 = x0 + math.cos(math.radians(angle)) * distance
        y1 = y0 + math.sin(math.radians(angle)) * distance
        return x1, y1

def reflect(x0, y0, x1, y1, d=1.0, a=180):
    d *= distance(x0, y0, x1, y1)
    a += angle(x0, y0, x1, y1)
    x, y = coordinates(x0, y0, d, a)
    return x, y
```

nodebox/geo/pathmatics.py

```
from math import sqrt, pow

# from nodebox.geo import distance

def linepoint(t, x0, y0, x1, y1):

    """Returns coordinates for point at t on the line.

    Calculates the coordinates of x and y for a point
    at t on a straight line.

    The t parameter is a number between 0.0 and 1.0,
    x0 and y0 define the starting point of the line,
    x1 and y1 the ending point of the line,

    """

    out_x = x0 + t * (x1-x0)
    out_y = y0 + t * (y1-y0)
    return (out_x, out_y)
```

```
   def linelength(x0, y0, x1, y1):

       """Returns the length of the line."""
       #return distance(x0,y0, x1,y1)
       a = pow(abs(x0 - x1), 2)
       b = pow(abs(y0 - y1), 2)
       return sqrt(a+b)

   def curvepoint(t, x0, y0, x1, y1, x2, y2, x3, y3, handles=False):

       """Returns coordinates for point at t on the spline.

       Calculates the coordinates of x and y for a point
       at t on the cubic bezier spline, and its control points,
       based on the de Casteljau interpolation algorithm.

       The t parameter is a number between 0.0 and 1.0,
       x0 and y0 define the starting point of the spline,
       x1 and y1 its control point,
       x3 and y3 the ending point of the spline,
       x2 and y2 its control point.

       If the handles parameter is set,
       returns not only the point at t,
       but the modified control points of p0 and p3
       should this point split the path as well.
       """

       mint = 1 - t

       x01   = x0 * mint + x1 * t
       y01   = y0 * mint + y1 * t
       x12   = x1 * mint + x2 * t
       y12   = y1 * mint + y2 * t
       x23   = x2 * mint + x3 * t
       y23   = y2 * mint + y3 * t

       out_c1x = x01 * mint + x12 * t
       out_c1y = y01 * mint + y12 * t
       out_c2x = x12 * mint + x23 * t
       out_c2y = y12 * mint + y23 * t
       out_x = out_c1x * mint + out_c2x * t
       out_y = out_c1y * mint + out_c2y * t

       if not handles:
           return (out_x, out_y, out_c1x, out_c1y, out_c2x, out_c2y)
       else:
           return (out_x, out_y, out_c1x, out_c1y, out_c2x, out_c2y, x01, y01, x23, y23)

   def curvelength(x0, y0, x1, y1, x2, y2, x3, y3, n=20):

       """Returns the length of the spline.

       Integrates the estimated length of the cubic bezier spline
       defined by x0, y0, ... x3, y3, by adding the lengths of
       lineair lines between points at t.

       The number of points is defined by n
       (n=10 would add the lengths of lines between 0.0 and 0.1,
       between 0.1 and 0.2, and so on).

       The default n=20 is fine for most cases, usually
       resulting in a deviation of less than 0.01.
```

```
    """

    length = 0
    xi = x0
    yi = y0

    for i in range(n):
        t = 1.0 * (i+1) / n
        pt_x, pt_y, pt_c1x, pt_c1y, pt_c2x, pt_c2y = curvepoint(t, x0, y0,
                                                                x1, y1,
                                                                x2, y2,
                                                                x3, y3)
        c = sqrt(pow(abs(xi-pt_x),2) + pow(abs(yi-pt_y),2))
        length += c
        xi = pt_x
        yi = pt_y

    return length
```

nodebox/graphics/__init__.py

```
import cocoa
graphics_impl = cocoa

import AppKit

# I really dont like it but cocoa.py has an __all__...
from cocoa import *

# from nodebox.util import _copy_attr, _copy_attrs
import nodebox.util
_copy_attr = nodebox.util._copy_attr
_copy_attrs = nodebox.util._copy_attrs

import nodebox.geo

# add graphics commands from cocoa
__all__ = list(graphics_impl.__all__)
__all__.extend(['Context'])

class Context(object):

    KEY_UP = graphics_impl.KEY_UP
    KEY_DOWN = graphics_impl.KEY_DOWN
    KEY_LEFT = graphics_impl.KEY_LEFT
    KEY_RIGHT = graphics_impl.KEY_RIGHT
    KEY_BACKSPACE = graphics_impl.KEY_BACKSPACE
    KEY_TAB = graphics_impl.KEY_TAB
    KEY_ESC = graphics_impl.KEY_ESC

    NORMAL = graphics_impl.NORMAL
    FORTYFIVE = graphics_impl.FORTYFIVE

    def __init__(self, canvas=None, ns=None):

        """Initializes the context.

        Note that we have to give the namespace of the executing script,
        which is a hack to keep the WIDTH and HEIGHT properties updated.
        Python's getattr only looks up property values once: at assign time."""

        if canvas is None:
            canvas = Canvas()
```

```
            if ns is None:
                ns = {}
        self.canvas = canvas
        self._ns = ns
        self._imagecache = {}
        self._vars = []
        self._resetContext()

    def _resetContext(self):
        self._outputmode = RGB
        self._colormode = RGB
        self._colorrange = 1.0
        self._fillcolor = self.Color()
        self._strokecolor = None
        self._strokewidth = 1.0
        self._capstyle = BUTT
        self._joinstyle = MITER
        self.canvas.background = self.Color(1.0)
        self._path = None
        self._autoclosepath = True
        self._transform = Transform()
        self._transformmode = CENTER
        self._transformstack = []
        self._fontname = "Helvetica"
        self._fontsize = 24
        self._lineheight = 1.2
        self._align = LEFT
        self._noImagesHint = False
        self._oldvars = self._vars
        self._vars = []

    def ximport(self, libName):

        lib = __import__(libName)
        self._ns[libName] = lib
        lib._ctx = self
        return lib

    ### Setup methods ###

    def size(self, width, height):
        if width == 0 and height == 0:
            # set to main screen size
            allsc = AppKit.NSScreen.screens()
            mainscreen = allsc[0]
            mainframe = mainscreen.frame()
            width = mainframe.size.width
            height = mainframe.size.height

        self.canvas.width = width
        self.canvas.height = height
        self._ns["WIDTH"] = width
        self._ns["HEIGHT"] = height

    def _get_width(self):
        return self.canvas.width

    WIDTH = property(_get_width)

    def _get_height(self):
        return self.canvas.height

    HEIGHT = property(_get_height)
```

```
    def speed(self, speed):
        self.canvas.speed = speed

    def background(self, *args):
        if len(args) > 0:
            if len(args) == 1 and args[0] is None:
                self.canvas.background = None
            else:
                self.canvas.background = self.Color(args)
        return self.canvas.background

    def outputmode(self, mode=None):
        if mode is not None:
            self._outputmode = mode
        return self._outputmode

    ### Variables ###

    def var(self, name, type, default=None, min=0, max=100, value=None):
        v = Variable(name, type, default, min, max, value)
        v = self.addvar(v)

    def addvar(self, v):
        oldvar = self.findvar(v.name)
        if oldvar is not None:
            if oldvar.compliesTo(v):
                v.value = oldvar.value
        self._vars.append(v)
        self._ns[v.name] = v.value

    def findvar(self, name):
        for v in self._oldvars:
            if v.name == name:
                return v
        return None

    ### Objects ####

    def _makeInstance(self, clazz, args, kwargs):
        """Creates an instance of a class defined in this document.
           This method sets the context of the object to the current context."""
        inst = clazz(self, *args, **kwargs)
        return inst

    def BezierPath(self, *args, **kwargs):
        return self._makeInstance(BezierPath, args, kwargs)

    def ClippingPath(self, *args, **kwargs):
        return self._makeInstance(ClippingPath, args, kwargs)

    def Rect(self, *args, **kwargs):
        return self._makeInstance(Rect, args, kwargs)

    def Oval(self, *args, **kwargs):
        return self._makeInstance(Oval, args, kwargs)

    def Color(self, *args, **kwargs):
        return self._makeInstance(Color, args, kwargs)

    def Image(self, *args, **kwargs):
        return self._makeInstance(Image, args, kwargs)

    def Text(self, *args, **kwargs):
        return self._makeInstance(Text, args, kwargs)
```

```
        ### Primitives ###

        def rect(self, x, y, width, height, roundness=0.0, draw=True, **kwargs):
175         BezierPath.checkKwargs(kwargs)
            p = self.BezierPath(**kwargs)
            if roundness == 0:
                p.rect(x, y, width, height)
            else:
180             curve = min(width*roundness, height*roundness)
                p.moveto(x, y+curve)
                p.curveto(x, y, x, y, x+curve, y)
                p.lineto(x+width-curve, y)
                p.curveto(x+width, y, x+width, y, x+width, y+curve)
185             p.lineto(x+width, y+height-curve)
                p.curveto(x+width, y+height, x+width, y+height, x+width-curve, y+height)
                p.lineto(x+curve, y+height)
                p.curveto(x, y+height, x, y+height, x, y+height-curve)
                p.closepath()
190         p.inheritFromContext(kwargs.keys())

            if draw:
                p.draw()
            return p
195
        def oval(self, x, y, width, height, draw=True, **kwargs):
            BezierPath.checkKwargs(kwargs)
            path = self.BezierPath(**kwargs)
            path.oval(x, y, width, height)
200         path.inheritFromContext(kwargs.keys())

            if draw:
                path.draw()
            return path
205
        ellipse = oval

        def arc(self, x, y, r, startAngle, endAngle, draw=True, **kwargs):
            BezierPath.checkKwargs(kwargs)
210         path = self.BezierPath(**kwargs)
            path.arc(x, y, r, startAngle, endAngle)
            path.inheritFromContext(kwargs.keys())
            if draw:
                path.draw()
215         return path

        def line(self, x1, y1, x2, y2, draw=True, **kwargs):
            BezierPath.checkKwargs(kwargs)
            p = self.BezierPath(**kwargs)
220         p.line(x1, y1, x2, y2)
            p.inheritFromContext(kwargs.keys())
            if draw:
                p.draw()
            return p
225
        def star(self, startx, starty, points=20, outer= 100, inner = 50, draw=True, **kwargs):
            BezierPath.checkKwargs(kwargs)
            from math import sin, cos, pi

230         p = self.BezierPath(**kwargs)
            p.moveto(startx, starty + outer)

            for i in range(1, int(2 * points)):
                angle = i * pi / points
```

```
            x = sin(angle)
            y = cos(angle)
            if i % 2:
                radius = inner
            else:
                radius = outer
            x = startx + radius * x
            y = starty + radius * y
            p.lineto(x,y)

        p.closepath()
        p.inheritFromContext(kwargs.keys())
        if draw:
            p.draw()
        return p

    def arrow(self, x, y, width=100, type=NORMAL, draw=True, **kwargs):

        """Draws an arrow.

        Draws an arrow at position x, y, with a default width of 100.
        There are two different types of arrows: NORMAL and trendy FORTYFIVE degrees arrows.
        When draw=False then the arrow's path is not ended, similar to endpath(draw=False)."""

        BezierPath.checkKwargs(kwargs)
        if type==NORMAL:
            return self._arrow(x, y, width, draw, **kwargs)
        elif type==FORTYFIVE:
            return self._arrow45(x, y, width, draw, **kwargs)
        else:
            raise NodeBoxError("arrow: available types for arrow() are NORMAL and FORTYFIVE\n")

    def _arrow(self, x, y, width, draw, **kwargs):

        head = width * .4
        tail = width * .2

        p = self.BezierPath(**kwargs)
        p.moveto(x, y)
        p.lineto(x-head, y+head)
        p.lineto(x-head, y+tail)
        p.lineto(x-width, y+tail)
        p.lineto(x-width, y-tail)
        p.lineto(x-head, y-tail)
        p.lineto(x-head, y-head)
        p.lineto(x, y)
        p.closepath()
        p.inheritFromContext(kwargs.keys())
        if draw:
            p.draw()
        return p

    def _arrow45(self, x, y, width, draw, **kwargs):

        head = .3
        tail = 1 + head

        p = self.BezierPath(**kwargs)
        p.moveto(x, y)
        p.lineto(x, y+width*(1-head))
        p.lineto(x-width*head, y+width)
        p.lineto(x-width*head, y+width*tail*.4)
        p.lineto(x-width*tail*.6, y+width)
        p.lineto(x-width, y+width*tail*.6)
```

```
            p.lineto(x-width*tail*.4, y+width*head)
        p.lineto(x-width, y+width*head)
            p.lineto(x-width*(1-head), y)
            p.lineto(x, y)
            p.inheritFromContext(kwargs.keys())
            if draw:
                p.draw()
            return p

        ### Path Commands ###

        def beginpath(self, x=None, y=None):
            self._path = self.BezierPath()
            self._pathclosed = False
            if x != None and y != None:
                self._path.moveto(x,y)

        def moveto(self, x, y):
            if self._path is None:
                raise NodeBoxError, "No current path. Use beginpath() first."
            self._path.moveto(x,y)

        def lineto(self, x, y):
            if self._path is None:
                raise NodeBoxError, "No current path. Use beginpath() first."
            self._path.lineto(x, y)

        def curveto(self, x1, y1, x2, y2, x3, y3):
            if self._path is None:
                raise NodeBoxError, "No current path. Use beginpath() first."
            self._path.curveto(x1, y1, x2, y2, x3, y3)

        def closepath(self):
            if self._path is None:
                raise NodeBoxError, "No current path. Use beginpath() first."
            if not self._pathclosed:
                self._path.closepath()

        def endpath(self, draw=True):
            if self._path is None:
                raise NodeBoxError, "No current path. Use beginpath() first."
            if self._autoclosepath:
                self.closepath()
            p = self._path
            p.inheritFromContext()
            if draw:
                p.draw()
            self._path = None
            self._pathclosed = False
            return p

        def drawpath(self, path, **kwargs):
            BezierPath.checkKwargs(kwargs)
            if isinstance(path, (list, tuple)):
                path = self.BezierPath(path, **kwargs)
            else: # Set the values in the current bezier path with the kwargs
                for arg_key, arg_val in kwargs.items():
                    setattr(path, arg_key, _copy_attr(arg_val))
            path.inheritFromContext(kwargs.keys())
            path.draw()

        def autoclosepath(self, close=True):
            self._autoclosepath = close
```

```
    def findpath(self, points, curvature=1.0):
        import bezier
        path = bezier.findpath(points, curvature=curvature)
        path._ctx = self
        path.inheritFromContext()
        return path

    ### Clipping Commands ###

    def beginclip(self, path):
        cp = self.ClippingPath(path)
        self.canvas.push(cp)
        return cp

    def endclip(self):
        self.canvas.pop()

    ### Transformation Commands ###

    def push(self): #, all=False):
        top = (self._transform.matrix,)
        if False: # all:
            top = (self._align, self._autoclosepath, self._capstyle, self._colormode,
                   self._fillcolor, self._fontname, self._fontsize, self._joinstyle,
                   self._lineheight, self._outputmode, self._strokecolor,
                   self._strokewidth, self._transformmode, self._transform.matrix)
        self._transformstack.append(top)

    def pop(self):
        try:
            top = self._transformstack.pop()
        except IndexError, e:
            raise NodeBoxError, "pop: too many pops!"
        if len(top) > 1:
            self._align, self._autoclosepath, self._capstyle, self._colormode, self._fillcolor, self._f
        else:
            self._transform.matrix = top[0]

    def transform(self, mode=None):
        if mode is not None:
            self._transformmode = mode
        return self._transformmode

    def translate(self, x, y):
        self._transform.translate(x, y)

    def reset(self):
        self._transform = Transform()

    def rotate(self, degrees=0, radians=0):
        self._transform.rotate(-degrees,-radians)

    def translate(self, x=0, y=0):
        self._transform.translate(x,y)

    def scale(self, x=1, y=None):
        self._transform.scale(x,y)

    def skew(self, x=0, y=0):
        self._transform.skew(x,y)

    ### Color Commands ###

    color = Color
```

```
    def colormode(self, mode=None, range=None):
        if mode is not None:
            self._colormode = mode
        if range is not None:
            self._colorrange = float(range)
        return self._colormode

    def colorrange(self, range=None):
        if range is not None:
            self._colorrange = float(range)
        return self._colorrange

    def nofill(self):
        self._fillcolor = None

    def fill(self, *args):
        if len(args) > 0:
            self._fillcolor = self.Color(*args)
        return self._fillcolor

    def nostroke(self):
        self._strokecolor = None

    def stroke(self, *args):
        if len(args) > 0:
            self._strokecolor = self.Color(*args)
        return self._strokecolor

    def strokewidth(self, width=None):
        if width is not None:
            self._strokewidth = max(width, 0.0001)
        return self._strokewidth

    def capstyle(self, style=None):
        if style is not None:
            if style not in (BUTT, ROUND, SQUARE):
                raise NodeBoxError, 'Line cap style should be BUTT, ROUND or SQUARE.'
            self._capstyle = style
        return self._capstyle

    def joinstyle(self, style=None):
        if style is not None:
            if style not in (MITER, ROUND, BEVEL):
                raise NodeBoxError, 'Line join style should be MITER, ROUND or BEVEL.'
            self._joinstyle = style
        return self._joinstyle

    ### Font Commands ###

    def font(self, fontname=None, fontsize = None):
        if fontname is not None:
            if not Text.font_exists(fontname):
                raise NodeBoxError, 'Font "%s" not found.' % fontname
            else:
                self._fontname = fontname
        if fontsize is not None:
            self._fontsize = fontsize
        return self._fontname

    def fontsize(self, fontsize=None):
        if fontsize is not None:
            self._fontsize = fontsize
        return self._fontsize
```

```
        def lineheight(self, lineheight=None):
            if lineheight is not None:
                self._lineheight = max(lineheight, 0.01)
495         return self._lineheight

        def align(self, align=None):
            if align is not None:
                self._align = align
500         return self._align

        def textwidth(self, txt, width=None, **kwargs):
            """Calculates the width of a single-line string."""
            return self.textmetrics(txt, width, **kwargs)[0]
505
        def textheight(self, txt, width=None, **kwargs):
            """Calculates the height of a (probably) multi-line string."""
            return self.textmetrics(txt, width, **kwargs)[1]

510     def text(self, txt, x, y, width=None, height=None, outline=False, draw=True, **kwargs):
            Text.checkKwargs(kwargs)
            txt = self.Text(txt, x, y, width, height, **kwargs)
            txt.inheritFromContext(kwargs.keys())
            if outline:
515             path = txt.path
                if draw:
                    path.draw()
                return path
            else:
520             if draw:
                    txt.draw()
                return txt

        def textpath(self, txt, x, y, width=None, height=None, **kwargs):
525         Text.checkKwargs(kwargs)
            txt = self.Text(txt, x, y, width, height, **kwargs)
            txt.inheritFromContext(kwargs.keys())
            return txt.path

530     def textmetrics(self, txt, width=None, height=None, **kwargs):
            txt = self.Text(txt, 0, 0, width, height, **kwargs)
            txt.inheritFromContext(kwargs.keys())
            return txt.metrics

535     def alltextmetrics(self, txt, width=None, height=None, **kwargs):
            txt = self.Text(txt, 0, 0, width, height, **kwargs)
            txt.inheritFromContext(kwargs.keys())
            return txt.allmetrics

540     ### Image commands ###

        def image(self, path, x, y, width=None, height=None, alpha=1.0, data=None, draw=True, **kwargs):
            img = self.Image(path, x, y, width, height, alpha, data=data, **kwargs)
            img.inheritFromContext(kwargs.keys())
545         if draw:
                img.draw()
            return img

        def imagesize(self, path, data=None):
550         img = self.Image(path, data=data)
            return img.size

        ### Canvas proxy ###
```

```
555     def save(self, fname, format=None):
            self.canvas.save(fname, format)

        ## cGeo

560     def isqrt( self, v):
            return nodebox.geo.isqrt( v )

        def angle(self, x0, y0, x1, y1):
            return nodebox.geo.angle( x0, y0, x1, y1)
565
        def distance(self, x0, y0, x1, y1):
            return nodebox.geo.distance( x0, y0, x1, y1)

        def coordinates(self, x0, y0, distance, angle):
570         return nodebox.geo.coordinates(x0, y0, distance, angle)

        def reflect(self, x0, y0, x1, y1, d=1.0, a=180):
            return nodebox.geo.reflect(x0, y0, x1, y1, d=1.0, a=180)
```

nodebox/graphics/bezier.py

```
    # Bezier - last updated for NodeBox 1.8.3
    # Author: Tom De Smedt <tomdesmedt@trapdoor.be>
    # Manual: http://nodebox.net/code/index.php/Bezier
    # Copyright (c) 2007 by Tom De Smedt.
  5 # Refer to the "Use" section on http://nodebox.net/code
    # Thanks to Dr. Florimond De Smedt at the Free University of Brussels for the math routines.

    from nodebox.graphics import BezierPath, PathElement, NodeBoxError, Point
    from nodebox.graphics import MOVETO, LINETO, CURVETO, CLOSE
 10
    try:
        import cPathmatics
        linepoint = cPathmatics.linepoint
        linelength = cPathmatics.linelength
 15     curvepoint = cPathmatics.curvepoint
        curvelength = cPathmatics.curvelength
    except:
        import nodebox.geo.pathmatics
        linepoint = nodebox.geo.pathmatics.linepoint
 20     linelength = nodebox.geo.pathmatics.linelength
        curvepoint = nodebox.geo.pathmatics.curvepoint
        curvelength = nodebox.geo.pathmatics.curvelength

    def segment_lengths(path, relative=False, n=20):
 25     """Returns a list with the lengths of each segment in the path.

        >>> path = BezierPath(None)
        >>> segment_lengths(path)
        []
 30     >>> path.moveto(0, 0)
        >>> segment_lengths(path)
        []
        >>> path.lineto(100, 0)
        >>> segment_lengths(path)
 35     [100.0]
        >>> path.lineto(100, 300)
        >>> segment_lengths(path)
        [100.0, 300.0]
        >>> segment_lengths(path, relative=True)
 40     [0.25, 0.75]
        >>> path = BezierPath(None)
```

```
        >>> path.moveto(1, 2)
        >>> path.curveto(3, 4, 5, 6, 7, 8)
        >>> segment_lengths(path)
 45     [8.48528137423857]
        """

        lengths = []
        first = True
 50
        for el in path:
            if first == True:
                close_x, close_y = el.x, el.y
                first = False
 55         elif el.cmd == MOVETO:
                close_x, close_y = el.x, el.y
                lengths.append(0.0)
            elif el.cmd == CLOSE:
                lengths.append(linelength(x0, y0, close_x, close_y))
 60         elif el.cmd == LINETO:
                lengths.append(linelength(x0, y0, el.x, el.y))
            elif el.cmd == CURVETO:
                x3, y3, x1, y1, x2, y2 = (el.x, el.y, el.ctrl1.x, el.ctrl1.y,
                                          el.ctrl2.x, el.ctrl2.y)
 65             lengths.append(curvelength(x0, y0, x1, y1, x2, y2, x3, y3, n))

            if el.cmd != CLOSE:
                x0 = el.x
                y0 = el.y
 70
        if relative:
            length = sum(lengths)
            try:
                return map(lambda l: l / length, lengths)
 75         except ZeroDivisionError:
                # If the length is zero, just return zero for all segments
                return [0.0] * len(lengths)
        else:
            return lengths
 80
    def length(path, segmented=False, n=20):

        """Returns the length of the path.

 85     Calculates the length of each spline in the path,
        using n as a number of points to measure.

        When segmented is True, returns a list
        containing the individual length of each spline
 90     as values between 0.0 and 1.0,
        defining the relative length of each spline
        in relation to the total path length.

        The length of an empty path is zero:
 95     >>> path = BezierPath(None)
        >>> length(path)
        0.0

        >>> path.moveto(0, 0)
100     >>> path.lineto(100, 0)
        >>> length(path)
        100.0

        >>> path.lineto(100, 100)
105     >>> length(path)
```

```
        200.0

        # Segmented returns a list of each segment
        >>> length(path, segmented=True)
110     [0.5, 0.5]
        """

        if not segmented:
            return sum(segment_lengths(path, n=n), 0.0)
115     else:
            return segment_lengths(path, relative=True, n=n)

    def _locate(path, t, segments=None):

120     """Locates t on a specific segment in the path.

        Returns (index, t, PathElement)

        A path is a combination of lines and curves (segments).
125     The returned index indicates the start of the segment
        that contains point t.

        The returned t is the absolute time on that segment,
        in contrast to the relative t on the whole of the path.
130     The returned point is the last MOVETO,
        any subsequent CLOSETO after i closes to that point.

        When you supply the list of segment lengths yourself,
        as returned from length(path, segmented=True),
135     point() works about thirty times faster in a for-loop,
        since it doesn't need to recalculate the length
        during each iteration. Note that this has been deprecated:
        the BezierPath now caches the segment lengths the moment you use
        them.
140
        >>> path = BezierPath(None)
        >>> _locate(path, 0.0)
        Traceback (most recent call last):
            ...
145     NodeBoxError: The given path is empty
        >>> path.moveto(0,0)
        >>> _locate(path, 0.0)
        Traceback (most recent call last):
            ...
150     NodeBoxError: The given path is empty
        >>> path.lineto(100, 100)
        >>> _locate(path, 0.0)
        (0, 0.0, Point(x=0.000, y=0.000))
        >>> _locate(path, 1.0)
155     (0, 1.0, Point(x=0.000, y=0.000))
        """

        if segments == None:
            segments = path.segmentlengths(relative=True)
160
        if len(segments) == 0:
            raise NodeBoxError, "The given path is empty"

        for i, el in enumerate(path):
165         if i == 0 or el.cmd == MOVETO:
                closeto = Point(el.x, el.y)
            if t <= segments[i] or i == len(segments)-1: break
            else: t -= segments[i]
```

```
    try: t /= segments[i]
    except ZeroDivisionError: pass
    if i == len(segments)-1 and segments[i] == 0: i -= 1

    return (i, t, closeto)

def point(path, t, segments=None):

    """Returns coordinates for point at t on the path.

    Gets the length of the path, based on the length
    of each curve and line in the path.
    Determines in what segment t falls.
    Gets the point on that segment.

    When you supply the list of segment lengths yourself,
    as returned from length(path, segmented=True),
    point() works about thirty times faster in a for-loop,
    since it doesn't need to recalculate the length
    during each iteration. Note that this has been deprecated:
    the BezierPath now caches the segment lengths the moment you use
    them.

    >>> path = BezierPath(None)
    >>> point(path, 0.0)
    Traceback (most recent call last):
        ...
    NodeBoxError: The given path is empty
    >>> path.moveto(0, 0)
    >>> point(path, 0.0)
    Traceback (most recent call last):
        ...
    NodeBoxError: The given path is empty
    >>> path.lineto(100, 0)
    >>> point(path, 0.0)
    PathElement(LINETO, ((0.000, 0.000),))
    >>> point(path, 0.1)
    PathElement(LINETO, ((10.000, 0.000),))
    """

    if len(path) == 0:
        raise NodeBoxError, "The given path is empty"

    i, t, closeto = _locate(path, t, segments=segments)

    x0, y0 = path[i].x, path[i].y
    p1 = path[i+1]

    if p1.cmd == CLOSE:
        x, y = linepoint(t, x0, y0, closeto.x, closeto.y)
        return PathElement(LINETO, ((x, y),))
    elif p1.cmd == LINETO:
        x1, y1 = p1.x, p1.y
        x, y = linepoint(t, x0, y0, x1, y1)
        return PathElement(LINETO, ((x, y),))
    elif p1.cmd == CURVETO:
        x3, y3, x1, y1, x2, y2 = (p1.x, p1.y,
                                  p1.ctrl1.x, p1.ctrl1.y,
                                  p1.ctrl2.x, p1.ctrl2.y)
        x, y, c1x, c1y, c2x, c2y = curvepoint(t, x0, y0, x1, y1, x2, y2, x3, y3)
        return PathElement(CURVETO, ((c1x, c1y), (c2x, c2y), (x, y)))
    else:
        raise NodeBoxError, "Unknown cmd for p1 %s" % p1
```

```
    def points(path, amount=100):
235     """Returns an iterator with a list of calculated points for the path.
        This method calls the point method <amount> times, increasing t,
        distributing point spacing linearly.

        >>> path = BezierPath(None)
240     >>> list(points(path))
        Traceback (most recent call last):
            ...
        NodeBoxError: The given path is empty
        >>> path.moveto(0, 0)
245     >>> list(points(path))
        Traceback (most recent call last):
            ...
        NodeBoxError: The given path is empty
        >>> path.lineto(100, 0)
250     >>> list(points(path, amount=4))
        [PathElement(LINETO, ((0.000, 0.000),)), PathElement(LINETO, ((33.333, 0.000),)), PathElement(LINET
        """

        if len(path) == 0:
255         raise NodeBoxError, "The given path is empty"

        # The delta value is divided by amount - 1, because we also want the last point (t=1.0)
        # If I wouldn't use amount - 1, I fall one point short of the end.
        # E.g. if amount = 4, I want point at t 0.0, 0.33, 0.66 and 1.0,
260     # if amount = 2, I want point at t 0.0 and t 1.0
        try:
            delta = 1.0/(amount-1)
        except ZeroDivisionError:
            delta = 1.0
265
        for i in xrange(amount):
            yield point(path, delta*i)

    def contours(path):
270     """Returns a list of contours in the path.

        A contour is a sequence of lines and curves
        separated from the next contour by a MOVETO.

275     For example, the glyph "o" has two contours:
        the inner circle and the outer circle.

        >>> path = BezierPath(None)
        >>> path.moveto(0, 0)
280     >>> path.lineto(100, 100)
        >>> len(contours(path))
        1

        A new contour is defined as something that starts with a moveto:
285     >>> path.moveto(50, 50)
        >>> path.curveto(150, 150, 50, 250, 80, 95)
        >>> len(contours(path))
        2

290     Empty moveto's don't do anything:
        >>> path.moveto(50, 50)
        >>> path.moveto(50, 50)
        >>> len(contours(path))
        2
295
        It doesn't matter if the path is closed or open:
        >>> path.closepath()
```

```
        >>> len(contours(path))
        2
    """
        contours = []
        current_contour = None
        empty = True
        for i, el in enumerate(path):
            if el.cmd == MOVETO:
                if not empty:
                    contours.append(current_contour)
                current_contour = BezierPath(path._ctx)
                current_contour.moveto(el.x, el.y)
                empty = True
            elif el.cmd == LINETO:
                empty = False
                current_contour.lineto(el.x, el.y)
            elif el.cmd == CURVETO:
                empty = False
                current_contour.curveto(el.ctrl1.x, el.ctrl1.y,
                    el.ctrl2.x, el.ctrl2.y, el.x, el.y)
            elif el.cmd == CLOSE:
                current_contour.closepath()
        if not empty:
            contours.append(current_contour)
        return contours

    def findpath(points, curvature=1.0):

        """Constructs a path between the given list of points.

        Interpolates the list of points and determines
        a smooth bezier path betweem them.

        The curvature parameter offers some control on
        how separate segments are stitched together:
        from straight angles to smooth curves.
        Curvature is only useful if the path has more than  three points.
        """

        # The list of points consists of Point objects,
        # but it shouldn't crash on something straightforward
        # as someone supplying a list of (x,y)-tuples.

        from types import TupleType
        for i, pt in enumerate(points):
            if type(pt) == TupleType:
                points[i] = Point(pt[0], pt[1])

        if len(points) == 0: return None
        if len(points) == 1:
            path = BezierPath(None)
            path.moveto(points[0].x, points[0].y)
            return path
        if len(points) == 2:
            path = BezierPath(None)
            path.moveto(points[0].x, points[0].y)
            path.lineto(points[1].x, points[1].y)
            return path

        # Zero curvature means straight lines.

        curvature = max(0, min(1, curvature))
        if curvature == 0:
            path = BezierPath(None)
```

```
            path.moveto(points[0].x, points[0].y)
            for i in range(len(points)):
                path.lineto(points[i].x, points[i].y)
        return path

    curvature = 4 + (1.0-curvature)*40

    dx = {0: 0, len(points)-1: 0}
    dy = {0: 0, len(points)-1: 0}
    bi = {1: -0.25}
    ax = {1: (points[2].x-points[0].x-dx[0]) / 4}
    ay = {1: (points[2].y-points[0].y-dy[0]) / 4}

    for i in range(2, len(points)-1):
        bi[i] = -1 / (curvature + bi[i-1])
        ax[i] = -(points[i+1].x-points[i-1].x-ax[i-1]) * bi[i]
        ay[i] = -(points[i+1].y-points[i-1].y-ay[i-1]) * bi[i]

    r = range(1, len(points)-1)
    r.reverse()
    for i in r:
        dx[i] = ax[i] + dx[i+1] * bi[i]
        dy[i] = ay[i] + dy[i+1] * bi[i]

    path = BezierPath(None)
    path.moveto(points[0].x, points[0].y)
    for i in range(len(points)-1):
        path.curveto(points[i].x + dx[i],
                     points[i].y + dy[i],
                     points[i+1].x - dx[i+1],
                     points[i+1].y - dy[i+1],
                     points[i+1].x,
                     points[i+1].y)

    return path

def insert_point(path, t):

    """Returns a path copy with an extra point at t.
    >>> path = BezierPath(None)
    >>> path.moveto(0, 0)
    >>> insert_point(path, 0.1)
    Traceback (most recent call last):
        ...
    NodeBoxError: The given path is empty
    >>> path.moveto(0, 0)
    >>> insert_point(path, 0.2)
    Traceback (most recent call last):
        ...
    NodeBoxError: The given path is empty
    >>> path.lineto(100, 50)
    >>> len(path)
    2
    >>> path = insert_point(path, 0.5)
    >>> len(path)
    3
    >>> path[1]
    PathElement(LINETO, ((50.000, 25.000),))
    >>> path = BezierPath(None)
    >>> path.moveto(0, 100)
    >>> path.curveto(0, 50, 100, 50, 100, 100)
    >>> path = insert_point(path, 0.5)
    >>> path[1]
    PathElement(CURVETO, ((0.000, 75.000), (25.000, 62.5), (50.000, 62.500))
```

```
        """

        i, t, closeto = _locate(path, t)

    x0 = path[i].x
        y0 = path[i].y
        p1 = path[i+1]
        p1cmd, x3, y3, x1, y1, x2, y2 = (p1.cmd, p1.x, p1.y,
                                                 p1.ctrl1.x, p1.ctrl1.y,
                                                 p1.ctrl2.x, p1.ctrl2.y)

        if p1cmd == CLOSE:
            pt_cmd = LINETO
            pt_x, pt_y = linepoint(t, x0, y0, closeto.x, closeto.y)
    elif p1cmd == LINETO:
            pt_cmd = LINETO
            pt_x, pt_y = linepoint(t, x0, y0, x3, y3)
        elif p1cmd == CURVETO:
            pt_cmd = CURVETO
            s = curvepoint(t, x0, y0, x1, y1, x2, y2, x3, y3, True)
            pt_x, pt_y, pt_c1x, pt_c1y, pt_c2x, pt_c2y, pt_h1x, pt_h1y, pt_h2x, pt_h2y = s
        else:
            raise NodeBoxError, "Locate should not return a MOVETO"

    new_path = BezierPath(None)
        new_path.moveto(path[0].x, path[0].y)
        for j in range(1, len(path)):
            if j == i+1:
                if pt_cmd == CURVETO:
                    new_path.curveto(pt_h1x, pt_h1y,
                                     pt_c1x, pt_c1y,
                                     pt_x, pt_y)
                    new_path.curveto(pt_c2x, pt_c2y,
                                     pt_h2x, pt_h2y,
                                     path[j].x, path[j].y)
                elif pt_cmd == LINETO:
                    new_path.lineto(pt_x, pt_y)
                    if path[j].cmd != CLOSE:
                        new_path.lineto(path[j].x, path[j].y)
                    else:
                        new_path.closepath()
                else:
                    raise NodeBoxError, "Didn't expect pt_cmd %s here" % pt_cmd

            else:
                if path[j].cmd == MOVETO:
                    new_path.moveto(path[j].x, path[j].y)
                if path[j].cmd == LINETO:
                    new_path.lineto(path[j].x, path[j].y)
                if path[j].cmd == CURVETO:
                    new_path.curveto(path[j].ctrl1.x, path[j].ctrl1.y,
                                     path[j].ctrl2.x, path[j].ctrl2.y,
                                     path[j].x, path[j].y)
                if path[j].cmd == CLOSE:
                    new_path.closepath()
        return new_path

    def _test():
        import doctest, bezier
        return doctest.testmod(bezier)

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

nodebox/graphics/cocoa.py

```
import os
import warnings

import pdb

# from random import choice, shuffle
import random
choice = random.choice
shuffle = random.shuffle

import objc
super = objc.super

# from AppKit import *
import AppKit
NSBezierPath = AppKit.NSBezierPath
NSColor = AppKit.NSColor
NSGraphicsContext = AppKit.NSGraphicsContext

NSView = AppKit.NSView

NSDeviceCMYKColorSpace = AppKit.NSDeviceCMYKColorSpace
NSDeviceRGBColorSpace = AppKit.NSDeviceRGBColorSpace
NSAffineTransform = AppKit.NSAffineTransform
NSImage = AppKit.NSImage
NSImageCacheNever = AppKit.NSImageCacheNever
NSCompositeSourceOver = AppKit.NSCompositeSourceOver
NSLeftTextAlignment = AppKit.NSLeftTextAlignment
NSFont = AppKit.NSFont
NSMutableParagraphStyle = AppKit.NSMutableParagraphStyle
NSLineBreakByWordWrapping = AppKit.NSLineBreakByWordWrapping
NSParagraphStyleAttributeName = AppKit.NSParagraphStyleAttributeName
NSForegroundColorAttributeName = AppKit.NSForegroundColorAttributeName
NSFontAttributeName = AppKit.NSFontAttributeName
NSTextStorage = AppKit.NSTextStorage
NSLayoutManager = AppKit.NSLayoutManager
NSTextContainer = AppKit.NSTextContainer
NSRectFillUsingOperation = AppKit.NSRectFillUsingOperation
NSGIFFileType = AppKit.NSGIFFileType
NSJPEGFileType = AppKit.NSJPEGFileType
NSJPEGFileType = AppKit.NSJPEGFileType
NSPNGFileType = AppKit.NSPNGFileType
NSTIFFFileType = AppKit.NSTIFFFileType
NSBitmapImageRep = AppKit.NSBitmapImageRep
NSString = AppKit.NSString
NSData = AppKit.NSData
NSAffineTransformStruct = AppKit.NSAffineTransformStruct

from nodebox.util import _copy_attr, _copy_attrs, makeunicode

try:
    import cPolymagic
except ImportError, e:
    warnings.warn('Could not load cPolymagic: %s' % e)

__all__ = [
        "DEFAULT_WIDTH", "DEFAULT_HEIGHT",
        "inch", "cm", "mm",
        "RGB", "HSB", "CMYK",
        "CENTER", "CORNER",
        "MOVETO", "LINETO", "CURVETO", "CLOSE",
        "MITER", "ROUND", "BEVEL", "BUTT", "SQUARE",
```

```
            "LEFT", "RIGHT", "CENTER", "JUSTIFY",
            "NORMAL","FORTYFIVE",
            "NUMBER", "TEXT", "BOOLEAN","BUTTON", "MENU",
            "NodeBoxError",
            "Point", "Grob", "BezierPath", "PathElement", "ClippingPath", "Rect", "Oval",
            "Color", "Transform", "Image", "Text",
            "Variable", "Canvas",
            ]

    DEFAULT_WIDTH, DEFAULT_HEIGHT = 1000, 1000

    # unused
    inch = 72.0
    cm = inch / 2.54
    mm = cm * 10.0

    RGB = "rgb"
    HSB = "hsb"
    CMYK = "cmyk"

    CENTER = "center"
    CORNER = "corner"

    MOVETO = AppKit.NSMoveToBezierPathElement
    LINETO = AppKit.NSLineToBezierPathElement
    CURVETO = AppKit.NSCurveToBezierPathElement
    CLOSE = AppKit.NSClosePathBezierPathElement

    MITER = AppKit.NSMiterLineJoinStyle
    ROUND = AppKit.NSRoundLineJoinStyle # Also used for NSRoundLineCapStyle, same value.
    BEVEL = AppKit.NSBevelLineJoinStyle
    BUTT = AppKit.NSButtLineCapStyle
    SQUARE = AppKit.NSSquareLineCapStyle

    LEFT = AppKit.NSLeftTextAlignment
    RIGHT = AppKit.NSRightTextAlignment
    CENTER = AppKit.NSCenterTextAlignment
    JUSTIFY = AppKit.NSJustifiedTextAlignment

    NORMAL=1
    FORTYFIVE=2

    NUMBER = 1
    TEXT = 2
    BOOLEAN = 3
    BUTTON = 4
    MENU = 5

    # unused
    KEY_UP = 126
    KEY_DOWN = 125
    KEY_LEFT = 123
    KEY_RIGHT = 124
    KEY_BACKSPACE = 51
    KEY_TAB = 48
    KEY_ESC = 53

    _STATE_NAMES = {
        '_outputmode':    'outputmode',
        '_colorrange':    'colorrange',
        '_fillcolor':     'fill',
        '_strokecolor':   'stroke',
        '_strokewidth':   'strokewidth',
        '_capstyle':      'capstyle',
```

```
        '_joinstyle':     'joinstyle',
        '_transform':     'transform',
        '_transformmode': 'transformmode',
130     '_fontname':      'font',
        '_fontsize':      'fontsize',
        '_align':         'align',
        '_lineheight':    'lineheight',
    }
135
    def _save():
        NSGraphicsContext.currentContext().saveGraphicsState()

    def _restore():
140     NSGraphicsContext.currentContext().restoreGraphicsState()

    class NodeBoxError(Exception): pass

    class Point(object):
145
        def __init__(self, *args):
            if len(args) == 2:
                self.x, self.y = args
            elif len(args) == 1:
150             self.x, self.y = args[0]
            elif len(args) == 0:
                self.x = self.y = 0.0
            else:
                raise NodeBoxError, "Wrong initializer for Point object"
155
        def __repr__(self):
            return "Point(x=%.3f, y=%.3f)" % (self.x, self.y)

        def __eq__(self, other):
160         if other is None: return False
            return self.x == other.x and self.y == other.y

        def __ne__(self, other):
            return not self.__eq__(other)
165
    class Grob(object):
        """A GRaphic OBject is the base class for all DrawingPrimitives."""

        def __init__(self, ctx):
170         """Initializes this object with the current context."""
            self._ctx = ctx

        def draw(self):
            """Appends the grob to the canvas.
175            This will result in a draw later on, when the scene graph is rendered."""
            self._ctx.canvas.append(self)

        def copy(self):
            """Returns a deep copy of this grob."""
180         raise NotImplementedError, "Copy is not implemented on this Grob class."

        def inheritFromContext(self, ignore=()):
            attrs_to_copy = list(self.__class__.stateAttributes)
            [attrs_to_copy.remove(k) for k, v in _STATE_NAMES.items() if v in ignore]
185         _copy_attrs(self._ctx, self, attrs_to_copy)

        def checkKwargs(self, kwargs):
            remaining = [arg for arg in kwargs.keys() if arg not in self.kwargs]
            if remaining:
190             raise NodeBoxError, "Unknown argument(s) '%s'" % ", ".join(remaining)
```

```
        checkKwargs = classmethod(checkKwargs)

    class TransformMixin(object):

195     """Mixin class for transformation support.
        Adds the _transform and _transformmode attributes to the class."""

        def __init__(self):
            self._reset()
200
        def _reset(self):
            self._transform = Transform()
            self._transformmode = CENTER

205     def _get_transform(self):
            return self._transform
        def _set_transform(self, transform):
            self._transform = Transform(transform)
        transform = property(_get_transform, _set_transform)
210
        def _get_transformmode(self):
            return self._transformmode
        def _set_transformmode(self, mode):
            self._transformmode = mode
215     transformmode = property(_get_transformmode, _set_transformmode)

        def translate(self, x, y):
            self._transform.translate(x, y)

220     def reset(self):
            self._transform = Transform()

        def rotate(self, degrees=0, radians=0):
            self._transform.rotate(-degrees,-radians)
225
        def translate(self, x=0, y=0):
            self._transform.translate(x,y)

        def scale(self, x=1, y=None):
230         self._transform.scale(x,y)

        def skew(self, x=0, y=0):
            self._transform.skew(x,y)

235 class ColorMixin(object):

        """Mixin class for color support.
        Adds the _fillcolor, _strokecolor and _strokewidth attributes to the class."""

240     def __init__(self, **kwargs):
            try:
                self._fillcolor = Color(self._ctx, kwargs['fill'])
            except KeyError:
                self._fillcolor = Color(self._ctx)
245         try:
                self._strokecolor = Color(self._ctx, kwargs['stroke'])
            except KeyError:
                self._strokecolor = None
            self._strokewidth = kwargs.get('strokewidth', 1.0)
250
        def _get_fill(self):
            return self._fillcolor
        def _set_fill(self, *args):
            self._fillcolor = Color(self._ctx, *args)
```

```
    fill = property(_get_fill, _set_fill)

    def _get_stroke(self):
        return self._strokecolor
    def _set_stroke(self, *args):
        self._strokecolor = Color(self._ctx, *args)
    stroke = property(_get_stroke, _set_stroke)

    def _get_strokewidth(self):
        return self._strokewidth
    def _set_strokewidth(self, strokewidth):
        self._strokewidth = max(strokewidth, 0.0001)
    strokewidth = property(_get_strokewidth, _set_strokewidth)

class BezierPath(Grob, TransformMixin, ColorMixin):
    """A BezierPath provides a wrapper around NSBezierPath."""

    stateAttributes = ('_fillcolor', '_strokecolor', '_strokewidth', '_capstyle',
                       '_joinstyle', '_transform', '_transformmode')
    kwargs = ('fill', 'stroke', 'strokewidth', 'capstyle', 'joinstyle')

    def __init__(self, ctx, path=None, **kwargs):
        super(BezierPath, self).__init__(ctx)
        TransformMixin.__init__(self)
        ColorMixin.__init__(self, **kwargs)
        self.capstyle = kwargs.get('capstyle', BUTT)
        self.joinstyle = kwargs.get('joinstyle', MITER)
        self._segment_cache = None
        if path is None:
            self._nsBezierPath = NSBezierPath.bezierPath()
        elif isinstance(path, (list,tuple)):
            self._nsBezierPath = NSBezierPath.bezierPath()
            self.extend(path)
        elif isinstance(path, BezierPath):
            self._nsBezierPath = path._nsBezierPath.copy()
            _copy_attrs(path, self, self.stateAttributes)
        elif isinstance(path, NSBezierPath):
            self._nsBezierPath = path
        else:
            raise NodeBoxError, "Don't know what to do with %s." % path

    def _get_path(self):
        s = "The 'path' attribute is deprecated. Please use _nsBezierPath instead."
        warnings.warn(s, DeprecationWarning, stacklevel=2)
        return self._nsBezierPath
    path = property(_get_path)

    def copy(self):
        return self.__class__(self._ctx, self)

    ### Cap and Join style ###

    def _get_capstyle(self):
        return self._capstyle
    def _set_capstyle(self, style):
        if style not in (BUTT, ROUND, SQUARE):
            raise NodeBoxError, 'Line cap style should be BUTT, ROUND or SQUARE.'
        self._capstyle = style
    capstyle = property(_get_capstyle, _set_capstyle)

    def _get_joinstyle(self):
        return self._joinstyle
    def _set_joinstyle(self, style):
        if style not in (MITER, ROUND, BEVEL):
```

```
            raise NodeBoxError, 'Line join style should be MITER, ROUND or BEVEL.'
        self._joinstyle = style
    joinstyle = property(_get_joinstyle, _set_joinstyle)

    ### Path methods ###

    def moveto(self, x, y):
        self._segment_cache = None
        self._nsBezierPath.moveToPoint_( (x, y) )

    def lineto(self, x, y):
        self._segment_cache = None
        self._nsBezierPath.lineToPoint_( (x, y) )

    def curveto(self, x1, y1, x2, y2, x3, y3):
        self._segment_cache = None
        self._nsBezierPath.curveToPoint_controlPoint1_controlPoint2_(
                                            (x3, y3), (x1, y1), (x2, y2) )

    # relativeMoveToPoint_( NSPoint )
    # relativeLineToPoint_( NSPoint )
    # relativeCurveToPoint:(NSPoint)aPoint controlPoint1:(NSPoint)controlPoint1 controlPoint2:(NSPoint)
    # appendBezierPathWithOvalInRect_
    # appendBezierPathWithArcFromPoint_(NSPoint)fromPoint toPoint_(NSPoint)toPoint radius:(CGFloat)
    # appendBezierPathWithArcWithCenter:(NSPoint)center radius:(CGFloat)radius startAngle:(CGFloat)star
    # appendBezierPathWithArcWithCenter:(NSPoint)center radius:(CGFloat)radius startAngle:(CGFloat)star

    def closepath(self):
        self._segment_cache = None
        self._nsBezierPath.closePath()

    def setlinewidth(self, width):
        self.linewidth = width

    def _get_bounds(self):
        try:
            return self._nsBezierPath.bounds()
        except:
            # Path is empty -- no bounds
            return (0,0) , (0,0)

    bounds = property(_get_bounds)

    def contains(self, x, y):
        return self._nsBezierPath.containsPoint_((x,y))

    ### Basic shapes ###

    def rect(self, x, y, width, height):
        self._segment_cache = None
        self._nsBezierPath.appendBezierPathWithRect_( ((x, y),
                                                       (width, height)) )

    def oval(self, x, y, width, height):
        self._segment_cache = None
        self._nsBezierPath.appendBezierPathWithOvalInRect_( ((x, y),
                                                             (width, height)) )
    ellipse = oval

    def arc(self, x, y, r, startAngle, endAngle):
        self._segment_cache = None
        self._nsBezierPath.appendBezierPathWithArcWithCenter_radius_startAngle_endAngle_(
                                        (x,y), r, startAngle, endAngle)
```

```
        def line(self, x1, y1, x2, y2):
            self._segment_cache = None
            self._nsBezierPath.moveToPoint_( (x1, y1) )
            self._nsBezierPath.lineToPoint_( (x2, y2) )

        ### List methods ###

        def __getitem__(self, index):
            cmd, el = self._nsBezierPath.elementAtIndex_associatedPoints_(index)
            return PathElement(cmd, el)

        def __iter__(self):
            for i in range(len(self)):
                yield self[i]

        def __len__(self):
            return self._nsBezierPath.elementCount()

        def extend(self, pathElements):
            self._segment_cache = None
            for el in pathElements:
                if isinstance(el, (list, tuple)):
                    x, y = el
                    if len(self) == 0:
                        cmd = MOVETO
                    else:
                        cmd = LINETO
                    self.append(PathElement(cmd, ((x, y),)))
                elif isinstance(el, PathElement):
                    self.append(el)
                else:
                    raise NodeBoxError, "Don't know how to handle %s" % el

        def append(self, el):
            self._segment_cache = None
            if el.cmd == MOVETO:
                self.moveto(el.x, el.y)
            elif el.cmd == LINETO:
                self.lineto(el.x, el.y)
            elif el.cmd == CURVETO:
                self.curveto(el.ctrl1.x, el.ctrl1.y, el.ctrl2.x, el.ctrl2.y, el.x, el.y)
            elif el.cmd == CLOSE:
                self.closepath()

        def _get_contours(self):
            from nodebox.graphics import bezier
            return bezier.contours(self)
        contours = property(_get_contours)

        ### Drawing methods ###

        def _get_transform(self):
            trans = self._transform.copy()
            if (self._transformmode == CENTER):
                (x, y), (w, h) = self.bounds
                deltax = x + w / 2
                deltay = y + h / 2
                t = Transform()
                t.translate(-deltax,-deltay)
                trans.prepend(t)
                t = Transform()
                t.translate(deltax,deltay)
                trans.append(t)
            return trans
```

```
    transform = property(_get_transform)

    def _draw(self):
        _save()
        self.transform.concat()
        if (self._fillcolor):
            self._fillcolor.set()
            self._nsBezierPath.fill()
        if (self._strokecolor):
            self._strokecolor.set()
            self._nsBezierPath.setLineWidth_(self._strokewidth)
            self._nsBezierPath.setLineCapStyle_(self._capstyle)
            self._nsBezierPath.setLineJoinStyle_(self._joinstyle)
            self._nsBezierPath.stroke()
        _restore()

    ### Geometry ###

    def fit(self, x=None, y=None, width=None, height=None, stretch=False):

        """Fits this path to the specified bounds.

        All parameters are optional; if no parameters are specified,
        nothing will happen. Specifying a parameter will constrain its value:

        - x: The path will be positioned at the specified x value
        - y: The path will be positioned at the specified y value
        - width: The path will be of the specified width
        - height: The path will be of the specified height
        - stretch: If both width and height are defined, either stretch the path or
                   keep the aspect ratio.
        """

        (px, py), (pw, ph) = self.bounds
        t = Transform()
        if x is not None and y is None:
            t.translate(x, py)
        elif x is None and y is not None:
            t.translate(px, y)
        elif x is not None and y is not None:
            t.translate(x, y)
        else:
            t.translate(px, py)
        if width is not None and height is None:
            t.scale(width / pw)
        elif width is None and height is not None:
            t.scale(height / ph)
        elif width is not None and height is not None:
            if stretch:
                t.scale(width /pw, height / ph)
            else:
                t.scale(min(width /pw, height / ph))
        t.translate(-px, -py)
        self._nsBezierPath = t.transformBezierPath(self)._nsBezierPath

    ### Mathematics ###

    def segmentlengths(self, relative=False, n=10):
        import bezier
        if relative: # Use the opportunity to store the segment cache.
            if self._segment_cache is None:
                self._segment_cache = bezier.segment_lengths(self,
                                                             relative=True, n=n)
            return self._segment_cache
```

```
            else:
                return bezier.segment_lengths(self, relative=False, n=n)

        def _get_length(self, segmented=False, n=10):
            import bezier
            return bezier.length(self, segmented=segmented, n=n)
        length = property(_get_length)

        def point(self, t):
            import bezier
            return bezier.point(self, t)

        def points(self, amount=100):
            import bezier
            if len(self) == 0:
                raise NodeBoxError, "The given path is empty"

            # The delta value is divided by amount - 1, because we also want the
            # last point (t=1.0)
            # If I wouldn't use amount - 1, I fall one point short of the end.
            # E.g. if amount = 4, I want point at t 0.0, 0.33, 0.66 and 1.0,
            # if amount = 2, I want point at t 0.0 and t 1.0
            try:
                delta = 1.0/(amount-1)
            except ZeroDivisionError:
                delta = 1.0

            for i in xrange(amount):
                yield self.point(delta*i)

        def addpoint(self, t):
            import bezier
            self._nsBezierPath = bezier.insert_point(self, t)._nsBezierPath
            self._segment_cache = None

        ### Clipping operations ###

        def intersects(self, other):
            return cPolymagic.intersects(self._nsBezierPath, other._nsBezierPath)

        def union(self, other, flatness=0.6):
            return BezierPath(self._ctx, cPolymagic.union(self._nsBezierPath,
                                                          other._nsBezierPath, flatness))

        def intersect(self, other, flatness=0.6):
            return BezierPath(self._ctx, cPolymagic.intersect(self._nsBezierPath,
                                                          other._nsBezierPath, flatness))

        def difference(self, other, flatness=0.6):
            return BezierPath(self._ctx, cPolymagic.difference(self._nsBezierPath,
                                                          other._nsBezierPath, flatness))

        def xor(self, other, flatness=0.6):
            return BezierPath(self._ctx, cPolymagic.xor(self._nsBezierPath,
                                                          other._nsBezierPath, flatness))

    class PathElement(object):

        def __init__(self, cmd=None, pts=None):
            self.cmd = cmd
            if cmd == MOVETO:
                assert len(pts) == 1
                self.x, self.y = pts[0]
                self.ctrl1 = Point(pts[0])
```

```
575             self.ctrl2 = Point(pts[0])
            elif cmd == LINETO:
                assert len(pts) == 1
                self.x, self.y = pts[0]
                self.ctrl1 = Point(pts[0])
580             self.ctrl2 = Point(pts[0])
            elif cmd == CURVETO:
                assert len(pts) == 3
                self.ctrl1 = Point(pts[0])
                self.ctrl2 = Point(pts[1])
585             self.x, self.y = pts[2]
            elif cmd == CLOSE:
                assert pts is None or len(pts) == 0
                self.x = self.y = 0.0
                self.ctrl1 = Point(0.0, 0.0)
590             self.ctrl2 = Point(0.0, 0.0)
            else:
                self.x = self.y = 0.0
                self.ctrl1 = Point()
                self.ctrl2 = Point()
595
        def __repr__(self):
            if self.cmd == MOVETO:
                return "PathElement(MOVETO, ((%.3f, %.3f),))" % (self.x, self.y)
            elif self.cmd == LINETO:
600             return "PathElement(LINETO, ((%.3f, %.3f),))" % (self.x, self.y)
            elif self.cmd == CURVETO:
                s = "PathElement(CURVETO, ((%.3f, %.3f), (%.3f, %s), (%.3f, %.3f))"
                return s % (self.ctrl1.x, self.ctrl1.y,
                            self.ctrl2.x, self.ctrl2.y,
605                         self.x, self.y)
            elif self.cmd == CLOSE:
                return "PathElement(CLOSE)"

        def __eq__(self, other):
610         if other is None: return False
            if self.cmd != other.cmd: return False
            return self.x == other.x and self.y == other.y \
                and self.ctrl1 == other.ctrl1 and self.ctrl2 == other.ctrl2

615     def __ne__(self, other):
            return not self.__eq__(other)

    class ClippingPath(Grob):

620     def __init__(self, ctx, path):
            self._ctx = ctx
            self.path = path
            self._grobs = []

625     def append(self, grob):
            self._grobs.append(grob)

        def _draw(self):
            _save()
630         cp = self.path.transform.transformBezierPath(self.path)
            cp._nsBezierPath.addClip()
            for grob in self._grobs:
                grob._draw()
            _restore()
635
    class Rect(BezierPath):

        def __init__(self, ctx, x, y, width, height, **kwargs):
```

```
            warnings.warn("Rect is deprecated. Use BezierPath's rect method.",
640                                             DeprecationWarning, stacklevel=2)
            r = (x,y), (width,height)
            super(Rect, self).__init__(ctx, NSBezierPath.bezierPathWithRect_(r),
                                       **kwargs)

645     def copy(self):
            raise NotImplementedError, "Please don't use Rect anymore"

    class Oval(BezierPath):

650     def __init__(self, ctx, x, y, width, height, **kwargs):
            warnings.warn("Oval is deprecated. Use BezierPath's oval method.",
                          DeprecationWarning, stacklevel=2)
            r = (x,y), (width,height)
            super(Oval, self).__init__(ctx, NSBezierPath.bezierPathWithOvalInRect_(r),
655                                        **kwargs)

        def copy(self):
            raise NotImplementedError, "Please don't use Oval anymore"

660 class Color(object):

        def __init__(self, ctx, *args):
            self._ctx = ctx
            params = len(args)
665
            # Decompose the arguments into tuples.
            if params == 1 and isinstance(args[0], tuple):
                args = args[0]
                params = len(args)
670
            if params == 1 and args[0] is None:
                clr = NSColor.colorWithDeviceWhite_alpha_(0.0, 0.0)
            elif params == 1 and isinstance(args[0], Color):
                if self._ctx._outputmode == RGB:
675                 clr = args[0]._rgb
                else:
                    clr = args[0]._cmyk
            elif params == 1 and isinstance(args[0], NSColor):
                clr = args[0]
680         elif (    params == 1
                  and isinstance(args[0], (str,unicode))
                  and len(args[0]) in (3,4,5,6,7,8,9)):
                # hex param
                try:
685                 a = args[0]
                    # kill hash char
                    if a[0] == '#':
                        a = a[1:]
                    alpha = 1.0
690                 n = len(a)
                    if n in (3,4):
                        div = 15.0
                        if n == 3:
                            r, g, b = a[:]
695                     else:
                            r, g, b, alpha = a[:]
                    else:
                        div = 255.0
                        if n == 6:
700                         r, g, b = a[:2], a[2:4], a[4:6]
                        else:
                            r, g, b, alpha = a[:2], a[2:4], a[4:6], a[6:8]
```

```
                r = int(r, 16) / div
                g = int(g, 16) / div
                b = int(b, 16) / div
                if n in (4,8):
                    alpha = int(alpha, 16) / div
                clr = NSColor.colorWithDeviceRed_green_blue_alpha_(r, g, b, alpha)
            except Exception, err:
                print "Color parsing error:", err
                clr = NSColor.colorWithDeviceWhite_alpha_(0, 1)

        elif params == 1: # Gray, no alpha
            args = self._normalizeList(args)
            g, = args
            clr = NSColor.colorWithDeviceWhite_alpha_(g, 1)
        elif params == 2: # Gray and alpha
            args = self._normalizeList(args)
            g, a = args
            clr = NSColor.colorWithDeviceWhite_alpha_(g, a)
        elif params == 3 and self._ctx._colormode == RGB: # RGB, no alpha
            args = self._normalizeList(args)
            r,g,b = args
            clr = NSColor.colorWithDeviceRed_green_blue_alpha_(r, g, b, 1)
        elif params == 3 and self._ctx._colormode == HSB: # HSB, no alpha
            args = self._normalizeList(args)
            h, s, b = args
            clr = NSColor.colorWithDeviceHue_saturation_brightness_alpha_(h, s, b, 1)
        elif params == 4 and self._ctx._colormode == RGB: # RGB and alpha
            args = self._normalizeList(args)
            r,g,b, a = args
            clr = NSColor.colorWithDeviceRed_green_blue_alpha_(r, g, b, a)
        elif params == 4 and self._ctx._colormode == HSB: # HSB and alpha
            args = self._normalizeList(args)
            h, s, b, a = args
            clr = NSColor.colorWithDeviceHue_saturation_brightness_alpha_(h, s, b, a)
        elif params == 4 and self._ctx._colormode == CMYK: # CMYK, no alpha
            args = self._normalizeList(args)
            c, m, y, k  = args
            clr = NSColor.colorWithDeviceCyan_magenta_yellow_black_alpha_(c, m, y, k, 1)
        elif params == 5 and self._ctx._colormode == CMYK: # CMYK and alpha
            args = self._normalizeList(args)
            c, m, y, k, a  = args
            clr = NSColor.colorWithDeviceCyan_magenta_yellow_black_alpha_(c, m, y, k, a)
        else:
            clr = NSColor.colorWithDeviceWhite_alpha_(0, 1)

        self._cmyk = clr.colorUsingColorSpaceName_(NSDeviceCMYKColorSpace)
        self._rgb = clr.colorUsingColorSpaceName_(NSDeviceRGBColorSpace)

    def __repr__(self):
        return "%s(%.3f, %.3f, %.3f, %.3f)" % (self.__class__.__name__, self.red,
                self.green, self.blue, self.alpha)

    def set(self):
        self.nsColor.set()

    def _get_nsColor(self):
        if self._ctx._outputmode == RGB:
            return self._rgb
        else:
            return self._cmyk
    nsColor = property(_get_nsColor)

    def copy(self):
        new = self.__class__(self._ctx)
```

```
            new._rgb = self._rgb.copy()
            new._updateCmyk()
            return new

    def _updateCmyk(self):
        self._cmyk = self._rgb.colorUsingColorSpaceName_(NSDeviceCMYKColorSpace)

    def _updateRgb(self):
        self._rgb = self._cmyk.colorUsingColorSpaceName_(NSDeviceRGBColorSpace)

    def _get_hue(self):
        return self._rgb.hueComponent()

    def _set_hue(self, val):
        val = self._normalize(val)
        h, s, b, a = self._rgb.getHue_saturation_brightness_alpha_(None, None, None, None)
        self._rgb = NSColor.colorWithDeviceHue_saturation_brightness_alpha_(val, s, b, a)
        self._updateCmyk()
    h = hue = property(_get_hue, _set_hue, doc="the hue of the color")

    def _get_saturation(self):
        return self._rgb.saturationComponent()
    def _set_saturation(self, val):
        val = self._normalize(val)
        h, s, b, a = self._rgb.getHue_saturation_brightness_alpha_(None, None, None, None)
        self._rgb = NSColor.colorWithDeviceHue_saturation_brightness_alpha_(h, val, b, a)
        self._updateCmyk()
    s = saturation = property(_get_saturation,
                              _set_saturation,
                              doc="the saturation of the color")

    def _get_brightness(self):
        return self._rgb.brightnessComponent()

    def _set_brightness(self, val):
        val = self._normalize(val)
        h, s, b, a = self._rgb.getHue_saturation_brightness_alpha_(None, None, None, None)
        self._rgb = NSColor.colorWithDeviceHue_saturation_brightness_alpha_(h, s, val, a)
        self._updateCmyk()
    v = brightness = property(_get_brightness,
                              _set_brightness,
                              doc="the brightness of the color")

    def _get_hsba(self):
        return self._rgb.getHue_saturation_brightness_alpha_(None, None, None, None)

    def _set_hsba(self, values):
        val = self._normalize(val)
        h, s, b, a = values
        self._rgb = NSColor.colorWithDeviceHue_saturation_brightness_alpha_(h, s, b, a)
        self._updateCmyk()
    hsba = property(_get_hsba,
                    _set_hsba,
                    doc="the hue, saturation, brightness and alpha of the color")

    def _get_red(self):
        return self._rgb.redComponent()

    def _set_red(self, val):
        val = self._normalize(val)
        r, g, b, a = self._rgb.getRed_green_blue_alpha_(None, None, None, None)
        self._rgb = NSColor.colorWithDeviceRed_green_blue_alpha_(val, g, b, a)
        self._updateCmyk()
    r = red = property(_get_red, _set_red, doc="the red component of the color")
```

```
        def _get_green(self):
            return self._rgb.greenComponent()

    835 def _set_green(self, val):
            val = self._normalize(val)
            r, g, b, a = self._rgb.getRed_green_blue_alpha_(None, None, None, None)
            self._rgb = NSColor.colorWithDeviceRed_green_blue_alpha_(r, val, b, a)
            self._updateCmyk()
    840 g = green = property(_get_green, _set_green, doc="the green component of the color")

        def _get_blue(self):
            return self._rgb.blueComponent()
        def _set_blue(self, val):
    845         val = self._normalize(val)
            r, g, b, a = self._rgb.getRed_green_blue_alpha_(None, None, None, None)
            self._rgb = NSColor.colorWithDeviceRed_green_blue_alpha_(r, g, val, a)
            self._updateCmyk()
        b = blue = property(_get_blue, _set_blue, doc="the blue component of the color")
    850
        def _get_alpha(self):
            return self._rgb.alphaComponent()
        def _set_alpha(self, val):
            val = self._normalize(val)
    855         r, g, b, a = self._rgb.getRed_green_blue_alpha_(None, None, None, None)
            self._rgb = NSColor.colorWithDeviceRed_green_blue_alpha_(r, g, b, val)
            self._updateCmyk()
        a = alpha = property(_get_alpha, _set_alpha, doc="the alpha component of the color")

    860 def _get_rgba(self):
            return self._rgb.getRed_green_blue_alpha_(None, None, None, None)

        def _set_rgba(self, val):
            val = self._normalizeList(val)
    865         r, g, b, a = val
            self._rgb = NSColor.colorWithDeviceRed_green_blue_alpha_(r, g, b, a)
            self._updateCmyk()
        rgba = property(_get_rgba,
                        _set_rgba,
    870                 doc="the red, green, blue and alpha values of the color")

        def _get_cyan(self):
            return self._cmyk.cyanComponent()

    875 def _set_cyan(self, val):
            val = self._normalize(val)
            c, m, y, k, a = self.cmyka
            self._cmyk = NSColor.colorWithDeviceCyan_magenta_yellow_black_alpha_(val, m, y, k, a)
            self._updateRgb()
    880 c = cyan = property(_get_cyan, _set_cyan, doc="the cyan component of the color")

        def _get_magenta(self):
            return self._cmyk.magentaComponent()

    885 def _set_magenta(self, val):
            val = self._normalize(val)
            c, m, y, k, a = self.cmyka
            self._cmyk = NSColor.colorWithDeviceCyan_magenta_yellow_black_alpha_(c, val, y, k, a)
            self._updateRgb()
    890 m = magenta = property(_get_magenta,
                              _set_magenta,
                              doc="the magenta component of the color")

        def _get_yellow(self):
```

```
        return self._cmyk.yellowComponent()

    def _set_yellow(self, val):
        val = self._normalize(val)
        c, m, y, k, a = self.cmyka
        self._cmyk = NSColor.colorWithDeviceCyan_magenta_yellow_black_alpha_(
                                                                c, m, val, k, a)
        self._updateRgb()
    y = yellow = property(_get_yellow,
                          _set_yellow,
                          doc="the yellow component of the color")

    def _get_black(self):
        return self._cmyk.blackComponent()

    def _set_black(self, val):
        val = self._normalize(val)
        c, m, y, k, a = self.cmyka
        self._cmyk = NSColor.colorWithDeviceCyan_magenta_yellow_black_alpha_(
                                                                c, m, y, val, a)
        self._updateRgb()
    k = black = property(_get_black,
                         _set_black,
                         doc="the black component of the color")

    def _get_cmyka(self):
        return (self._cmyk.cyanComponent(),
                self._cmyk.magentaComponent(),
                self._cmyk.yellowComponent(),
                self._cmyk.blackComponent(),
                self._cmyk.alphaComponent())
    cmyka = property(_get_cmyka, doc="a tuple containing the CMYKA values for this color")

    def blend(self, otherColor, factor):
        """Blend the color with otherColor with a factor; return the new color. Factor
        is a float between 0.0 and 1.0.
        """
        if hasattr(otherColor, "color"):
            otherColor = otherColor._rgb
        return self.__class__(color=self._rgb.blendedColorWithFraction_ofColor_(
                factor, otherColor))

    def _normalize(self, v):
        """Bring the color into the 0-1 scale for the current colorrange"""
        if self._ctx._colorrange == 1.0:
            return v
        return v / self._ctx._colorrange

    def _normalizeList(self, lst):
        """Bring the color into the 0-1 scale for the current colorrange"""
        r = self._ctx._colorrange
        if r == 1.0:
            return lst
        return [v / r for v in lst]

color = Color

class Transform(object):

    def __init__(self, transform=None):
        if transform is None:
            transform = NSAffineTransform.transform()
        elif isinstance(transform, Transform):
            matrix = transform._nsAffineTransform.transformStruct()
```

```
            transform = NSAffineTransform.transform()
            transform.setTransformStruct_(matrix)
        elif isinstance(transform, (list, tuple, NSAffineTransformStruct)):
            matrix = tuple(transform)
            transform = NSAffineTransform.transform()
            transform.setTransformStruct_(matrix)
        elif isinstance(transform, NSAffineTransform):
            pass
        else:
            raise NodeBoxError, "Don't know how to handle transform %s." % transform
        self._nsAffineTransform = transform

    def _get_transform(self):
        s = ("The 'transform' attribute is deprecated. "
             "Please use _nsAffineTransform instead.")
        warnings.warn(s, DeprecationWarning, stacklevel=2)
        return self._nsAffineTransform
    transform = property(_get_transform)

    def set(self):
        self._nsAffineTransform.set()

    def concat(self):
        self._nsAffineTransform.concat()

    def copy(self):
        return self.__class__(self._nsAffineTransform.copy())

    def __repr__(self):
        return "<%s [%.3f %.3f %.3f %.3f %.3f %.3f]>" % ((self.__class__.__name__,)
                                                         + tuple(self))

    def __iter__(self):
        for value in self._nsAffineTransform.transformStruct():
            yield value

    def _get_matrix(self):
        return self._nsAffineTransform.transformStruct()

    def _set_matrix(self, value):
        self._nsAffineTransform.setTransformStruct_(value)
    matrix = property(_get_matrix, _set_matrix)

    def rotate(self, degrees=0, radians=0):
        if degrees:
            self._nsAffineTransform.rotateByDegrees_(degrees)
        else:
            self._nsAffineTransform.rotateByRadians_(radians)

    def translate(self, x=0, y=0):
        self._nsAffineTransform.translateXBy_yBy_(x, y)

    def scale(self, x=1, y=None):
        if y is None:
            y = x
        self._nsAffineTransform.scaleXBy_yBy_(x, y)

    def skew(self, x=0, y=0):
        import math
        x = math.pi * x / 180
        y = math.pi * y / 180
        t = Transform()
        t.matrix = 1, math.tan(y), -math.tan(x), 1, 0, 0
        self.prepend(t)
```

```
        def invert(self):
            self._nsAffineTransform.invert()

        def append(self, other):
            if isinstance(other, Transform):
                other = other._nsAffineTransform
            self._nsAffineTransform.appendTransform_(other)

        def prepend(self, other):
            if isinstance(other, Transform):
                other = other._nsAffineTransform
            self._nsAffineTransform.prependTransform_(other)

        def transformPoint(self, point):
            return self._nsAffineTransform.transformPoint_(point)

        def transformBezierPath(self, path):
            if isinstance(path, BezierPath):
                path = BezierPath(path._ctx, path)
            else:
                raise NodeBoxError, "Can only transform BezierPaths"
            path._nsBezierPath = self._nsAffineTransform.transformBezierPath_(path._nsBezierPath)
            return path

    class Image(Grob, TransformMixin):

        stateAttributes = ('_transform', '_transformmode')
        kwargs = ()

        def __init__(self, ctx, path=None, x=0, y=0,
                         width=None, height=None, alpha=1.0, image=None, data=None):
            """
            Parameters:
             - path: A path to a certain image on the local filesystem.
             - x: Horizontal position.
             - y: Vertical position.
             - width: Maximum width. Images get scaled according to this factor.
             - height: Maximum height. Images get scaled according to this factor.
                 If a width and height are both given, the smallest
                 of the two is chosen.
             - alpha: transparency factor
             - image: optionally, an Image or NSImage object.
             - data: a stream of bytes of image data.
            """
            super(Image, self).__init__(ctx)
            TransformMixin.__init__(self)
            if data is not None:
                if not isinstance(data, NSData):
                    data = NSData.dataWithBytes_length_(data, len(data))
                self._nsImage = NSImage.alloc().initWithData_(data)
                if self._nsImage is None:
                    raise NodeBoxError, "can't read image %r" % path
                self._nsImage.setFlipped_(True)
                self._nsImage.setCacheMode_(NSImageCacheNever)
            elif image is not None:
                if isinstance(image, NSImage):
                    self._nsImage = image
                    self._nsImage.setFlipped_(True)
                else:
                    raise NodeBoxError, "Don't know what to do with %s." % image
            elif path is not None:
                if not os.path.exists(path):
                    raise NodeBoxError, 'Image "%s" not found.' % path
```

```
                curtime = os.path.getmtime(path)
                try:
                    image, lasttime = self._ctx._imagecache[path]
1090                if lasttime != curtime:
                        image = None
                except KeyError:
                    pass
                if image is None:
1095                image = NSImage.alloc().initWithContentsOfFile_(path)
                    if image is None:
                        raise NodeBoxError, "Can't read image %r" % path
                    image.setFlipped_(True)
                    image.setCacheMode_(NSImageCacheNever)
1100                self._ctx._imagecache[path] = (image, curtime)
                self._nsImage = image
            self.x = x
            self.y = y
            self.width = width
1105        self.height = height
            self.alpha = alpha
            self.debugImage = False

        def _get_image(self):
1110        w = "The 'image' attribute is deprecated. Please use _nsImage instead."
            warnings.warn(w, DeprecationWarning, stacklevel=2)
            return self._nsImage
        image = property(_get_image)

1115    def copy(self):
            new = self.__class__(self._ctx)
            _copy_attrs(self, new, ('image', 'x', 'y', 'width', 'height',
                                    '_transform', '_transformmode', 'alpha', 'debugImage'))
            return new
1120
        def getSize(self):
            return self._nsImage.size()

        size = property(getSize)
1125
        def _draw(self):
            """Draw an image on the given coordinates."""

            srcW, srcH = self._nsImage.size()
1130        srcRect = ((0, 0), (srcW, srcH))

            # Width or height given
            if self.width is not None or self.height is not None:
                if self.width is not None and self.height is not None:
1135                factor = min(self.width / srcW, self.height / srcH)
                elif self.width is not None:
                    factor = self.width / srcW
                elif self.height is not None:
                    factor = self.height / srcH
1140            _save()

                # Center-mode transforms: translate to image center
                if self._transformmode == CENTER:
                    # This is the hardest case: center-mode transformations with given
1145                # width or height.
                    # Order is very important in this code.

                    # Set the position first, before any of the scaling or transformations
                    # are done.
1150                # Context transformations might change the translation, and we don't
```

```
                # want that.
                t = Transform()
                t.translate(self.x, self.y)
                t.concat()

                # Set new width and height factors. Note that no scaling is done yet:
                # they're just here to set the new center of the image according to
                # the scaling factors.
                srcW = srcW * factor
                srcH = srcH * factor

                # Move image to newly calculated center.
                dX = srcW / 2
                dY = srcH / 2
                t = Transform()
                t.translate(dX, dY)
                t.concat()

                # Do current transformation.
                self._transform.concat()

                # Move back to the previous position.
                t = Transform()
                t.translate(-dX, -dY)
                t.concat()

                # Finally, scale the image according to the factors.
                t = Transform()
                t.scale(factor)
                t.concat()
            else:
                # Do current transformation
                self._transform.concat()
                # Scale according to width or height factor
                t = Transform()
                t.translate(self.x, self.y) # Here we add the positioning of the image.
                t.scale(factor)
                t.concat()

            # A debugImage draws a black rectangle instead of an image.
            if self.debugImage:
                Color(self._ctx).set()
                pt = BezierPath()
                pt.rect(0, 0, srcW / factor, srcH / factor)
                pt.fill()
            else:
                self._nsImage.drawAtPoint_fromRect_operation_fraction_((0, 0),
                                        srcRect, NSCompositeSourceOver, self.alpha)
            _restore()
        # No width or height given
        else:
            _save()
            x,y = self.x, self.y
            # Center-mode transforms: translate to image center
            if self._transformmode == CENTER:
                deltaX = srcW / 2
                deltaY = srcH / 2
                t = Transform()
                t.translate(x+deltaX, y+deltaY)
                t.concat()
                x = -deltaX
                y = -deltaY
            # Do current transformation
            self._transform.concat()
```

```
            # A debugImage draws a black rectangle instead of an image.
            if self.debugImage:
                Color(self._ctx).set()
                pt = BezierPath()
                pt.rect(x, y, srcW, srcH)
                pt.fill()
            else:
                # The following code avoids a nasty bug in Cocoa/PyObjC.
                # Apparently, EPS files are put on a different position when drawn
                # with a certain position.
                # However, this only happens when the alpha value is set to 1.0: set
                # it to something lower and the positioning is the same as a bitmap
                # file.
                # I could of course make every EPS image have an alpha value of
                # 0.9999, but this solution is better: always use zero coordinates for
                # drawAtPoint and use a transform to set the final position.
                t = Transform()
                t.translate(x,y)
                t.concat()
                self._nsImage.drawAtPoint_fromRect_operation_fraction_(
                                (0,0), srcRect, NSCompositeSourceOver, self.alpha)
            _restore()

    class Text(Grob, TransformMixin, ColorMixin):

        stateAttributes = ('_transform', '_transformmode', '_fillcolor', '_fontname',
                           '_fontsize', '_align', '_lineheight')
        kwargs = ('fill', 'font', 'fontsize', 'align', 'lineheight')

        __dummy_color = NSColor.blackColor()

        def __init__(self, ctx, text, x=0, y=0, width=None, height=None, **kwargs):
            super(Text, self).__init__(ctx)
            TransformMixin.__init__(self)
            ColorMixin.__init__(self, **kwargs)
            self.text = makeunicode(text)
            self.x = x
            self.y = y
            self.width = width
            self.height = height
            self._fontname = kwargs.get('font', "Helvetica")
            self._fontsize = kwargs.get('fontsize', 24)
            self._lineheight = max(kwargs.get('lineheight', 1.2), 0.01)
            self._align = kwargs.get('align', NSLeftTextAlignment)

        def copy(self):
            new = self.__class__(self._ctx, self.text)
            _copy_attrs(self, new,
                ('x', 'y', 'width', 'height', '_transform', '_transformmode',
                '_fillcolor', '_fontname', '_fontsize', '_align', '_lineheight'))
            return new

        def font_exists(cls, fontname):
            # Check if the font exists.
            f = NSFont.fontWithName_size_(fontname, 12)
            return f is not None
        font_exists = classmethod(font_exists)

        def _get_font(self):
            return NSFont.fontWithName_size_(self._fontname, self._fontsize)
        font = property(_get_font)

        def _getLayoutManagerTextContainerTextStorage(self, clr=__dummy_color):
            paraStyle = NSMutableParagraphStyle.alloc().init()
```

```
            paraStyle.setAlignment_(self._align)
            paraStyle.setLineBreakMode_(NSLineBreakByWordWrapping)
            paraStyle.setLineHeightMultiple_(self._lineheight)

            d = {
                NSParagraphStyleAttributeName:  paraStyle,
                NSForegroundColorAttributeName: clr,
                NSFontAttributeName:            self.font
            }

            t = unicode(self.text)
            textStorage = NSTextStorage.alloc().initWithString_attributes_(t, d)
            try:
                textStorage.setFont_(self.font)
            except ValueError:
                raise NodeBoxError("Text.draw(): font '%s' not available.\n" % self._fontname)
                return

            layoutManager = NSLayoutManager.alloc().init()
            textContainer = NSTextContainer.alloc().init()
            if self.width != None:
                textContainer.setContainerSize_((self.width,1000000))
                textContainer.setWidthTracksTextView_(False)
                textContainer.setHeightTracksTextView_(False)
            layoutManager.addTextContainer_(textContainer)
            textStorage.addLayoutManager_(layoutManager)
            return layoutManager, textContainer, textStorage

        def _draw(self):
            if self._fillcolor is None:
                return

            s = self._getLayoutManagerTextContainerTextStorage(self._fillcolor.nsColor)
            layoutManager, textContainer, textStorage = s

            x,y = self.x, self.y
            glyphRange = layoutManager.glyphRangeForTextContainer_(textContainer)
            s = layoutManager.boundingRectForGlyphRange_inTextContainer_(glyphRange,
                                                                         textContainer)
            (dx, dy), (w, h) = s
            preferredWidth, preferredHeight = textContainer.containerSize()
            if self.width is not None:
                if self._align == RIGHT:
                    x += preferredWidth - w
                elif self._align == CENTER:
                    x += preferredWidth/2 - w/2

            _save()
            # Center-mode transforms: translate to image center
            if self._transformmode == CENTER:
                deltaX = w / 2
                deltaY = h / 2
                t = Transform()
                t.translate(x+deltaX, y-self.font.defaultLineHeightForFont()+deltaY)
                t.concat()
                self._transform.concat()
                layoutManager.drawGlyphsForGlyphRange_atPoint_(glyphRange, (-deltaX-dx,-deltaY-dy))
            else:
                self._transform.concat()
                layoutManager.drawGlyphsForGlyphRange_atPoint_(glyphRange,
                                        (x-dx, y-dy-self.font.defaultLineHeightForFont()))
            _restore()
            return (w, h)
```

```
        def _get_allmetrics(self):
            items = self._getLayoutManagerTextContainerTextStorage()
            layoutManager, textContainer, textStorage = items
            glyphRange = layoutManager.glyphRangeForTextContainer_(textContainer)
            (dx, dy), (w, h) = layoutManager.boundingRectForGlyphRange_inTextContainer_(
                                                                glyphRange, textContainer)
            # print "metrics (dx,dy):", (dx,dy)
            return dx,dy,w,h
        allmetrics = property(_get_allmetrics)

        def _get_metrics(self):
            dx,dy,w,h = self._get_allmetrics()
            return w,h
        metrics = property(_get_metrics)

        def _get_path(self):
            items = self._getLayoutManagerTextContainerTextStorage()
            layoutManager, textContainer, textStorage = items
            x, y = self.x, self.y
            glyphRange = layoutManager.glyphRangeForTextContainer_(textContainer)
            (dx, dy), (w, h) = layoutManager.boundingRectForGlyphRange_inTextContainer_(
                                                                glyphRange, textContainer)
            preferredWidth, preferredHeight = textContainer.containerSize()
            if self.width is not None:
               if self._align == RIGHT:
                   x += preferredWidth - w
               elif self._align == CENTER:
                   x += preferredWidth/2 - w/2
            length = layoutManager.numberOfGlyphs()
            path = NSBezierPath.bezierPath()
            for glyphIndex in range(length):
                lineFragmentRect = layoutManager.lineFragmentRectForGlyphAtIndex_effectiveRange_(
                                                                    glyphIndex, None)
                # HACK: PyObjc 2.0 and 2.2 are subtly different:
                #  - 2.0 (bundled with OS X 10.5) returns one argument: the rectangle.
                #  - 2.2 (bundled with OS X 10.6) returns two arguments: the rectangle and the range.
                # So we check if we got one or two arguments back (in a tuple) and unpack them.
                if isinstance(lineFragmentRect, tuple):
                    lineFragmentRect = lineFragmentRect[0]
                layoutPoint = layoutManager.locationForGlyphAtIndex_(glyphIndex)

                # Here layoutLocation is the location (in container coordinates)
                # where the glyph was laid out.
                finalPoint = [lineFragmentRect[0][0],lineFragmentRect[0][1]]
                finalPoint[0] += layoutPoint[0] - dx
                finalPoint[1] += layoutPoint[1] - dy
                g = layoutManager.glyphAtIndex_(glyphIndex)
                if g == 0:
                    continue
                path.moveToPoint_((finalPoint[0], -finalPoint[1]))
                path.appendBezierPathWithGlyph_inFont_(g, self.font)
                path.closePath()
            path = BezierPath(self._ctx, path)
            trans = Transform()
            trans.translate(x,y-self.font.defaultLineHeightForFont())
            trans.scale(1.0,-1.0)
            path = trans.transformBezierPath(path)
            path.inheritFromContext()
            return path
        path = property(_get_path)

    class Variable(object):
        def __init__(self, name, typ, default=None, minV=0, maxV=100, value=None):
            self.name = makeunicode(name)
```

```
            self.type = typ or NUMBER
            self.default = default
            self.min = minV
            self.max = maxV
            if self.type == NUMBER:
                if default is None:
                    self.default = 50
                else:
                    self.default = default
                self.min = minV
                self.max = maxV
            elif self.type == TEXT:
                if default is None:
                    self.default = "hello"
                else:
                    self.default = makeunicode(default)
            elif self.type == BOOLEAN:
                if default is None:
                    self.default = True
                else:
                    self.default = default
            elif self.type == BUTTON:
                self.default = makeunicode(self.name)
            elif self.type == MENU:
                # value is list of menuitems
                # default is name of function to call with selected menu item name
                if default is not None:
                    self.dispatchfunction = default
                    self.default = None
                if value is not None:
                    self.menuitems = [makeunicode(i) for i in value]
                    # set value to first entry
                    value = self.menuitems[0]
            self.value = value or self.default

        def sanitize(self, val):
            """Given a Variable and a value, cleans it out"""
            if self.type == NUMBER:
                try:
                    return float(val)
                except ValueError:
                    return 0.0
            elif self.type == TEXT:
                return unicode(str(val), "utf_8", "replace")
                try:
                    return unicode(str(val), "utf_8", "replace")
                except:
                    return ""
            elif self.type == BOOLEAN:
                if unicode(val).lower() in ("true", "1", "yes"):
                    return True
                else:
                    return False

        def compliesTo(self, v):
            """Return whether I am compatible with the given var:
                 - Type should be the same
                 - My value should be inside the given vars' min/max range.
            """
            if self.type == v.type:
                if self.type == NUMBER:
                    if self.value < self.min or self.value > self.max:
                        return False
                return True
```

```
            return False

        def __repr__(self):
            s = "Variable(name=%s, typ=%s, default=%s, min=%s, max=%s, value=%s)"
            return s % (self.name, self.type, self.default, self.min, self.max, self.value)

    class _PDFRenderView(NSView):

        # This view was created to provide PDF data.
        # Strangely enough, the only way to get PDF data from Cocoa is by asking
        # dataWithPDFInsideRect_ from a NSView. So, we create one just to get to
        # the PDF data.

        def initWithCanvas_(self, canvas):

            # for some unknown reason the following line stopped working
            # Solution: use objc.super -- see import
            super(_PDFRenderView, self).initWithFrame_( ((0, 0), (canvas.width, canvas.height)) )
            # for some unknown reason this is the solution for the preceding problem
            # self.initWithFrame_( ((0, 0), (canvas.width, canvas.height)) )
            # it is the only super in this file, having a NS* superclass

            self.canvas = canvas
            return self

        def drawRect_(self, rect):
            self.canvas.draw()

        def isOpaque(self):
            return False

        def isFlipped(self):
            return True

    class Canvas(Grob):

        def __init__(self, width=DEFAULT_WIDTH, height=DEFAULT_HEIGHT):
            self.width = width
            self.height = height
            self.speed = None
            self.mousedown = False
            self.clear()

        def clear(self):
            self._grobs = self._container = []
            self._grobstack = [self._grobs]

        def _get_size(self):
            return self.width, self.height
        size = property(_get_size)

        def append(self, el):
            self._container.append(el)

        def __iter__(self):
            for grob in self._grobs:
                yield grob

        def __len__(self):
            return len(self._grobs)

        def __getitem__(self, index):
            return self._grobs[index]
```

```
    def push(self, containerGrob):
        self._grobstack.insert(0, containerGrob)
        self._container.append(containerGrob)
        self._container = containerGrob

    def pop(self):
        try:
            del self._grobstack[0]
            self._container = self._grobstack[0]
        except IndexError, e:
            raise NodeBoxError, "pop: too many canvas pops!"

    def draw(self):
        if self.background is not None:
            self.background.set()
            NSRectFillUsingOperation(((0,0), (self.width, self.height)),
                                     NSCompositeSourceOver)
        for grob in self._grobs:
            grob._draw()

    def _get_nsImage(self):
        img = NSImage.alloc().initWithSize_((self.width, self.height))
        img.setFlipped_(True)
        img.lockFocus()
        self.draw()
        img.unlockFocus()
        return img
    _nsImage = property(_get_nsImage)

    def _getImageData(self, format):
        if format == 'pdf':
            view = _PDFRenderView.alloc().initWithCanvas_(self)
            return view.dataWithPDFInsideRect_(view.bounds())
        elif format == 'eps':
            view = _PDFRenderView.alloc().initWithCanvas_(self)
            return view.dataWithEPSInsideRect_(view.bounds())
        else:
            imgTypes = {"gif":  NSGIFFileType,
                        "jpg":  NSJPEGFileType,
                        "jpeg": NSJPEGFileType,
                        "png":  NSPNGFileType,
                        "tiff": NSTIFFFileType}
            if format not in imgTypes:
                e = "Filename should end in .pdf, .eps, .tiff, .gif, .jpg or .png"
                raise NodeBoxError, e
            data = self._nsImage.TIFFRepresentation()
            if format != 'tiff':
                imgType = imgTypes[format]
                rep = NSBitmapImageRep.imageRepWithData_(data)
                return rep.representationUsingType_properties_(imgType, None)
            else:
                return data

    def save(self, fname, format=None):
        if format is None:
            basename, ext = os.path.splitext(fname)
            format = ext[1:].lower() # Skip the dot
        data = self._getImageData(format)
        fname = NSString.stringByExpandingTildeInPath(fname)
        data.writeToFile_atomically_(fname, False)

def _test():
    import doctest, cocoa
    return doctest.testmod(cocoa)
```

```
if __name__==’__main__’:
    _test()
```

nodebox/gui/__init__.py

nodebox/gui/mac/__init__.py

```
import sys
import os
import traceback, linecache
import re
import objc
import time
import random
import signal
import atexit

import pprint
pp = pprint.pprint

import pdb

# set to true to have stdio on the terminal
kwdbg = False

# if true print out some debug info on stdout
kwlog = False

import Foundation
import AppKit
NSObject = AppKit.NSObject
NSColor = AppKit.NSColor
NSScriptCommand = AppKit.NSScriptCommand

NSDocument = AppKit.NSDocument
NSDocumentController = AppKit.NSDocumentController

NSNotificationCenter = AppKit.NSNotificationCenter

NSFontAttributeName = AppKit.NSFontAttributeName
NSScreen = AppKit.NSScreen
NSMenu = AppKit.NSMenu
NSCursor = AppKit.NSCursor
NSTimer = AppKit.NSTimer
NSForegroundColorAttributeName = AppKit.NSForegroundColorAttributeName

NSPasteboard = AppKit.NSPasteboard
NSPDFPboardType = AppKit.NSPDFPboardType
NSPostScriptPboardType = AppKit.NSPostScriptPboardType
NSTIFFPboardType = AppKit.NSTIFFPboardType

NSBundle = AppKit.NSBundle
NSSavePanel = AppKit.NSSavePanel
NSLog = AppKit.NSLog
NSApp = AppKit.NSApp
NSPrintOperation = AppKit.NSPrintOperation
NSWindow = AppKit.NSWindow
NSBorderlessWindowMask = AppKit.NSBorderlessWindowMask
NSBackingStoreBuffered = AppKit.NSBackingStoreBuffered
NSView = AppKit.NSView
```

```
    NSGraphicsContext = AppKit.NSGraphicsContext
 55 NSRectFill = AppKit.NSRectFill
    NSAffineTransform = AppKit.NSAffineTransform
    NSFocusRingTypeExterior = AppKit.NSFocusRingTypeExterior
    NSResponder = AppKit.NSResponder

 60 NSURL = AppKit.NSURL
    NSWorkspace = AppKit.NSWorkspace
    NSBezierPath = AppKit.NSBezierPath

    import threading
 65 Thread = threading.Thread

    import ValueLadder
    MAGICVAR = ValueLadder.MAGICVAR

 70 import PyDETextView

    import preferences
    NodeBoxPreferencesController = preferences.NodeBoxPreferencesController
    LibraryFolder = preferences.LibraryFolder
 75
    import util
    errorAlert = util.errorAlert

    # from nodebox import util
 80 import nodebox.util
    util = nodebox.util
    makeunicode = nodebox.util.makeunicode

    import nodebox.util.ottobot
 85 genProgram = nodebox.util.ottobot.genProgram

    import nodebox.util.QTSupport
    QTSupport = nodebox.util.QTSupport

 90 # from nodebox import graphics
    import nodebox.graphics
    graphics = nodebox.graphics

    # AppleScript enumerator codes for PDF and Quicktime export
 95 PDF = 0x70646678 # 'pdfx'
    QUICKTIME = 0x71747878 # 'qt  '

    black = NSColor.blackColor()
    VERY_LIGHT_GRAY = black.blendedColorWithFraction_ofColor_(0.95,
100                                                           NSColor.whiteColor())
    DARKER_GRAY = black.blendedColorWithFraction_ofColor_(0.8,
                                                          NSColor.whiteColor())

    # from nodebox.gui.mac.dashboard import *
105 # from nodebox.gui.mac.progressbar import ProgressBarController
    import dashboard
    DashboardController = dashboard.DashboardController

    import progressbar
110 ProgressBarController = progressbar.ProgressBarController

    class ExportCommand(NSScriptCommand):
        pass

115 class OutputFile(object):

        def __init__(self, data, isErr=False):
```

```
            self.data = data
            self.isErr = isErr

        def write(self, data):
            if isinstance(data, str):
                try:
                    data = unicode(data, "utf_8", "replace")
                except UnicodeDecodeError:
                    data = "XXX " + repr(data)
            self.data.append((self.isErr, data))

    # class defined in NodeBoxDocument.xib
    class NodeBoxDocument(NSDocument):

        graphicsView = objc.IBOutlet()
        outputView = objc.IBOutlet()
        textView = objc.IBOutlet()
        window = objc.IBOutlet()
        variablesController = objc.IBOutlet()
        dashboardController = objc.IBOutlet()
        animationSpinner = objc.IBOutlet()

        # The ExportImageAccessory adds:
        exportImageAccessory = objc.IBOutlet()
        exportImageFormat = objc.IBOutlet()
        exportImagePageCount = objc.IBOutlet()

        # The ExportMovieAccessory adds:
        exportMovieAccessory = objc.IBOutlet()
        exportMovieFrames = objc.IBOutlet()
        exportMovieFps = objc.IBOutlet()

        # When the PageCount accessory is loaded, we also add:
        pageCount = objc.IBOutlet()
        pageCountAccessory = objc.IBOutlet()

        # When the ExportSheet is loaded, we also add:
        exportSheet = objc.IBOutlet()
        exportSheetIndicator = objc.IBOutlet()

        path = None
        exportDir = None
        magicvar = None # Used for value ladders.
        _code = None
        vars = []
        movie = None

        def windowNibName(self):
            return "NodeBoxDocument"

        def init(self):
            self = super(NodeBoxDocument, self).init()
            nc = NSNotificationCenter.defaultCenter()
            nc.addObserver_selector_name_object_(self,
                                                 "textFontChanged:",
                                                 "PyDETextFontChanged",
                                                 None)
            self.namespace = {}
            self.canvas = graphics.Canvas()
            self.context = graphics.Context(self.canvas, self.namespace)
            self.animationTimer = None
            self.__doc__ = {}
            self._pageNumber = 1
            self._frame = 150
```

```
            self.fullScreen = None
            self._seed = time.time()

        # another debugging not completed
        #if not self.graphicsView:
        #    pdb.set_trace()
        #    print

        # this is None
        self.currentView = self.graphicsView
        return self

    def autosavesInPlace(self):
        return True

    def close(self):
        self.stopScript()
        super(NodeBoxDocument, self).close()

    def __del__(self):
        nc = NSNotificationCenter.defaultCenter()
        nc.removeObserver_name_object_(self, "PyDETextFontChanged", None)
        # text view has a couple of circular refs, it can let go of them now
        self.textView._cleanup()

    def textFontChanged_(self, notification):
        font = PyDETextView.getBasicTextAttributes()[NSFontAttributeName]
        self.outputView.setFont_(font)

    def readFromFile_ofType_(self, path, tp):
        if self.textView is None:
            # we're not yet fully loaded
            self.path = path
        else:
            # "revert"
            self.readFromUTF8_(path)
        return True

    def writeToFile_ofType_(self, path, tp):
        f = file(path, "w")
        text = self.textView.string()
        f.write(text.encode("utf8"))
        f.close()
        return True

    def windowControllerDidLoadNib_(self, controller):
        if self.path:
            self.readFromUTF8_(self.path)
        font = PyDETextView.getBasicTextAttributes()[NSFontAttributeName]
        self.outputView.setFont_(font)
        self.textView.window().makeFirstResponder_(self.textView)
        self.windowControllers()[0].setWindowFrameAutosaveName_("NodeBoxDocumentWindow")

    def readFromUTF8_(self, path):
        f = file(path)
        text = unicode(f.read(), "utf_8")
        f.close()
        self.textView.setString_(text)
        self.textView.usesTabs = "\t" in text

    def cleanRun_newSeed_buildInterface_(self, fn, newSeed, buildInterface):
        self.animationSpinner.startAnimation_(None)

        # Prepare everything for running the script
```

```
            self.prepareRun()

            # Run the actual script
            success = self.fastRun_newSeed_(fn, newSeed)
            self.animationSpinner.stopAnimation_(None)

            if success and buildInterface:

                # Build the interface
                self.vars = self.namespace["_ctx"]._vars
                if len(self.vars) > 0:
                    self.buildInterface_(None)

            return success

        def prepareRun(self):

            # Compile the script
            success, output = self.boxedRun_args_(self._compileScript, [])
            self.flushOutput_(output)
            if not success:
                return False

            # Initialize the namespace
            self._initNamespace()

            # Reset the pagenum
            self._pageNum = 1

            # Reset the frame
            self._frame = 1

            self.speed = self.canvas.speed = None

        def fastRun_newSeed_(self, fn, newSeed = False):
            """This is the old signature. Dispatching to the new with args"""
            return self.fastRun_newSeed_args_( fn, newSeed, [])
        def fastRun_newSeed_args_(self, fn, newSeed = False, args=[]):
            # Check if there is code to run
            if self._code is None:
                return False

            # Clear the canvas
            self.canvas.clear()

            # Generate a new seed, if needed
            if newSeed:
                self._seed = time.time()
            random.seed(self._seed)

            # Set the mouse position

            # kw fix
            if not self.currentView:
                self.currentView = self.graphicsView
            window = self.currentView.window()
            pt = window.mouseLocationOutsideOfEventStream()
            mx, my = window.contentView().convertPoint_toView_(pt, self.currentView)
            # Hack: mouse coordinates are flipped vertically in FullscreenView.
            # This flips them back.
            if isinstance(self.currentView, FullscreenView):
                my = self.currentView.bounds()[1][1] - my
            if self.fullScreen is None:
                mx /= self.currentView.zoom
```

```
            my /= self.currentView.zoom
        self.namespace["MOUSEX"], self.namespace["MOUSEY"] = mx, my
        self.namespace["mousedown"] = self.currentView.mousedown
        self.namespace["keydown"] = self.currentView.keydown
        self.namespace["key"] = self.currentView.key
        self.namespace["keycode"] = self.currentView.keycode
        self.namespace["scrollwheel"] = self.currentView.scrollwheel
        self.namespace["wheeldelta"] = self.currentView.wheeldelta

        # Reset the context
        self.context._resetContext()

        # Initalize the magicvar
        self.namespace[MAGICVAR] = self.magicvar

        # Set the pagenum
        self.namespace['PAGENUM'] = self._pageNumber

        # Set the frame
        self.namespace['FRAME'] = self._frame

        # Run the script
        success, output = self.boxedRun_args_(fn, args)
        self.flushOutput_(output)
        if not success:
            return False

        # Display the output of the script
        self.currentView.setCanvas_(self.canvas)

        return True

    @objc.IBAction
    def runFullscreen_(self, sender):
        if self.fullScreen is not None: return
        self.stopScript()
        self.currentView = FullscreenView.alloc().init()
        self.currentView.canvas = None
        fullRect = NSScreen.mainScreen().frame()
        self.fullScreen = FullscreenWindow.alloc().initWithRect_(fullRect)
        self.fullScreen.setContentView_(self.currentView)
        self.fullScreen.makeKeyAndOrderFront_(self)
        self.fullScreen.makeFirstResponder_(self.currentView)
        NSMenu.setMenuBarVisible_(False)
        NSCursor.hide()
        self._runScript()

    @objc.IBAction
    def runScript_(self, sender):
        self.runScript()

    def runScript(self, compile=True, newSeed=True):
        if self.fullScreen is not None: return
        self.currentView = self.graphicsView
        self._runScript(compile, newSeed)

    def _runScript(self, compile=True, newSeed=True):
        if not self.cleanRun_newSeed_buildInterface_(self._execScript, True, True):
            pass

        # Check whether we are dealing with animation
        if self.canvas.speed is not None:
            if not self.namespace.has_key("draw"):
                errorAlert("Not a proper NodeBox animation",
```

```
                    "NodeBox animations should have at least a draw() method.")
                return

            # Check if animationTimer is already running
            if self.animationTimer is not None:
                self.stopScript()

            self.speed = self.canvas.speed

            # Run setup routine
            if self.namespace.has_key("setup"):
                self.fastRun_newSeed_(self.namespace["setup"], False)
            window = self.currentView.window()
            window.makeFirstResponder_(self.currentView)

            # Start the timer
            timer = NSTimer.scheduledTimerWithTimeInterval_target_selector_userInfo_repeats_
            self.animationTimer = timer(1.0 / self.speed,
                                        self,
                                        objc.selector(self.doFrame, signature="v@:@"),
                                        None,
                                        True)

            # Start the spinner
            self.animationSpinner.startAnimation_(None)

    def runScriptFast(self):
        if self.animationTimer is None:
            self.fastRun_newSeed_(self._execScript, False)
        else:
            # XXX: This can be sped up. We just run _execScript to get the
            # method with __MAGICVAR__ into the namespace, and execute
            # that, so it should only be called once for animations.
            self.fastRun_newSeed_(self._execScript, False)
            self.fastRun_newSeed_(self.namespace["draw"], False)

    def doFrame(self):
        self.fastRun_newSeed_(self.namespace["draw"], True)
        self._frame += 1

    def source(self):
        return self.textView.string()

    def setSource_(self, source):
        self.textView.setString_(source)

    @objc.IBAction
    def stopScript_(self, sender=None):
        self.stopScript()

    def stopScript(self):
        if self.namespace.has_key("stop"):
            success, output = self.boxedRun_args_(self.namespace["stop"], [])
            self.flushOutput_(output)
        self.animationSpinner.stopAnimation_(None)
        if self.animationTimer is not None:
            self.animationTimer.invalidate()
            self.animationTimer = None
        if self.fullScreen is not None:
            self.currentView = self.graphicsView
            self.fullScreen = None
            NSMenu.setMenuBarVisible_(True)
        NSCursor.unhide()
        self.textView.hideValueLadder()
```

```
            window = self.textView.window()
            window.makeFirstResponder_(self.textView)

        def _compileScript(self, source=None):
            if source is None:
                source = self.textView.string()
            self._code = None
            self._code = compile(source + "\n\n",
                                 self.scriptName.encode('ascii', 'ignore'),
                                 "exec")

        def _initNamespace(self):

            self.namespace.clear()
            # Add everything from the namespace
            for name in graphics.__all__:
                self.namespace[name] = getattr(graphics, name)
            for name in util.__all__:
                self.namespace[name] = getattr(util, name)

            # debug print all collected keywords
            if kwlog:
                print "util.__all__:"
                pp(util.__all__)
                print "graphics.__all__:"
                pp(graphics.__all__)

            # Add everything from the context object
            self.namespace["_ctx"] = self.context
            for attrName in dir(self.context):
                self.namespace[attrName] = getattr(self.context, attrName)
            # Add the document global
            self.namespace["__doc__"] = self.__doc__
            # Add the page number
            self.namespace["PAGENUM"] = self._pageNumber
            # Add the frame number
            self.namespace["FRAME"] = self._frame
            # Add the magic var
            self.namespace[MAGICVAR] = self.magicvar
            # XXX: will be empty after reset.
            #for var in self.vars:
            #    self.namespace[var.name] = var.value

        def _execScript(self):
            exec self._code in self.namespace
            self.__doc__ = self.namespace.get("__doc__", self.__doc__)

        def boxedRun_args_(self, method, args):
            """
            Runs the given method in a boxed environment.
            Boxed environments:
             - Have their current directory set to the directory of the file
             - Have their argument set to the filename
             - Have their outputs redirect to an output stream.
            Returns:
               A tuple containing:
                 - A boolean indicating whether the run was successful
                 - The OutputFile
            """

            self.scriptName = self.fileName()
            libpath = LibraryFolder()
            libDir = libpath.libDir
```

```
            if not self.scriptName:
                curDir = os.getenv("HOME")
                self.scriptName = "<untitled>"
505         else:
                curDir = os.path.dirname(self.scriptName)

            save = sys.stdout, sys.stderr
            saveDir = os.getcwd()
510         saveArgv = sys.argv
            sys.argv = [self.scriptName]
            if os.path.exists(libDir):
                sys.path.insert(0, libDir)
            os.chdir(curDir)
515         sys.path.insert(0, curDir)
            output = []

            # for pdb debugging in terminal this needs to be switched off
            if not kwdbg:
520             sys.stdout = OutputFile(output, False)
                sys.stderr = OutputFile(output, True)
            self._scriptDone = False
            try:
                if self.animationTimer is None:
525                 pass
                    # Creating a thread is a heavy operation,
                    # don't install it when animating, where speed is crucial
                    #t = Thread(target=self._userCancelledMonitor,
                    #           name="UserCancelledMonitor")
530                 #t.start()
                try:
                    method(*args)
                except KeyboardInterrupt:
                    self.stopScript()
535             except:
                    etype, value, tb = sys.exc_info()
                    if tb.tb_next is not None:
                        tb = tb.tb_next  # skip the frame doing the exec
                    traceback.print_exception(etype, value, tb)
540                 etype = value = tb = None
                    return False, output
            finally:
                self._scriptDone = True
                sys.stdout, sys.stderr = save
545             os.chdir(saveDir)
                sys.path.remove(curDir)
                try:
                    sys.path.remove(libDir)
                except ValueError:
550                 pass
                sys.argv = saveArgv
                #self.flushOutput_()
            return True, output

555     # from Mac/Tools/IDE/PyEdit.py
        def _userCancelledMonitor(self):
            from Carbon import Evt
            while not self._scriptDone:
                if Evt.CheckEventQueueForUserCancel():
560                 # Send a SIGINT signal to ourselves.
                    # This gets delivered to the main thread,
                    # cancelling the running script.
                    os.kill(os.getpid(), signal.SIGINT)
                    break
565             time.sleep(0.25)
```

```
        def flushOutput_(self, output):
            outAttrs = PyDETextView.getBasicTextAttributes()
            errAttrs = outAttrs.copy()
            # XXX err color from user defaults...
            errAttrs[NSForegroundColorAttributeName] = NSColor.redColor()

            outputView = self.outputView
            outputView.setSelectedRange_((outputView.textStorage().length(), 0))
            lastErr = None
            for isErr, data in output:
                if isErr != lastErr:
                    attrs = [outAttrs, errAttrs][isErr]
                    outputView.setTypingAttributes_(attrs)
                    lastErr = isErr
                outputView.insertText_(data)
            # del self.output

        @objc.IBAction
        def copyImageAsPDF_(self, sender):
            pboard = NSPasteboard.generalPasteboard()
            # graphicsView implements the pboard delegate method to provide the data
            pboard.declareTypes_owner_( [NSPDFPboardType,
                                         NSPostScriptPboardType,
                                         NSTIFFPboardType],
                                        self.graphicsView)

        @objc.IBAction
        def exportAsImage_(self, sender):
            exportPanel = NSSavePanel.savePanel()
            exportPanel.setRequiredFileType_("pdf")
            exportPanel.setNameFieldLabel_("Export To:")
            exportPanel.setPrompt_("Export")
            exportPanel.setCanSelectHiddenExtension_(True)
            if not NSBundle.loadNibNamed_owner_("ExportImageAccessory", self):
                NSLog("Error -- could not load ExportImageAccessory.")
            self.exportImagePageCount.setIntValue_(1)
            exportPanel.setAccessoryView_(self.exportImageAccessory)
            path = self.fileName()
            if path:
                dirName, fileName = os.path.split(path)
                fileName, ext = os.path.splitext(fileName)
                fileName += ".pdf"
            else:
                dirName, fileName = None, "Untitled.pdf"
            # If a file was already exported, use that folder as the default.
            if self.exportDir is not None:
                dirName = self.exportDir
            exportPanel.beginSheetForDirectory_file_modalForWindow_modalDelegate_didEndSelector_contextInfo
                dirName,
                fileName,
                NSApp().mainWindow(),
                self,
                "exportPanelDidEnd:returnCode:contextInfo:", 0)

        def exportPanelDidEnd_returnCode_contextInfo_(self, panel, returnCode, context):
            if returnCode:
                fname = panel.filename()
                self.exportDir = os.path.split(fname)[0] # Save the directory we exported to.
                pages = self.exportImagePageCount.intValue()
                format = panel.requiredFileType()
                panel.close()
                self.doExportAsImage_fmt_pages_(fname, format, pages)
        exportPanelDidEnd_returnCode_contextInfo_ = objc.selector(
```

```
        exportPanelDidEnd_returnCode_contextInfo_,
        signature="v@:@ii")

    @objc.IBAction
    def exportImageFormatChanged_(self, sender):
        image_formats = ('pdf', 'eps', 'png', 'tiff', 'jpg', 'gif')
        panel = sender.window()
        panel.setRequiredFileType_(image_formats[sender.indexOfSelectedItem()])

    def doExportAsImage_fmt_pages_(self, fname, format, pages):
        basename, ext = os.path.splitext(fname)
        # When saving one page (the default), just save the current graphics
        # context. When generating multiple pages, we run the script again
        # (so we don't use the current displayed view) for the first page,
        # and then for every next page.
        if pages == 1:
            if self.graphicsView.canvas is None:
                self.runScript()
            self.canvas.save(fname, format)
        elif pages > 1:
            pb = ProgressBarController.alloc().init()
            pb.begin_maxval_("Generating %s images..." % pages, pages)
            try:
                if not self.cleanRun_newSeed_buildInterface_(self._execScript, True, True):
                    return
                self._pageNumber = 1
                self._frame = 1

                # If the speed is set, we are dealing with animation
                if self.canvas.speed is None:
                    for i in range(pages):
                        if i > 0: # Run has already happened first time
                            self.fastRun_newSeed_(self._execScript, True)
                        counterAsString = "-%5d" % self._pageNumber
                        counterAsString = counterAsString.replace(' ', '0')
                        exportName = basename + counterAsString + ext

                        self.canvas.save(exportName, format)
                        self.graphicsView.setNeedsDisplay_(True)
                        self._pageNumber += 1
                        self._frame += 1
                        pb.inc()
                else:
                    if self.namespace.has_key("setup"):
                        self.fastRun_newSeed_(self.namespace["setup"], False)
                    for i in range(pages):
                        self.fastRun_newSeed_(self.namespace["draw"], True)
                        counterAsString = "-%5d" % self._pageNumber
                        counterAsString = counterAsString.replace(' ', '0')
                        exportName = basename + counterAsString + ext
                        self.canvas.save(exportName, format)
                        self.graphicsView.setNeedsDisplay_(True)
                        self._pageNumber += 1
                        self._frame += 1
                        pb.inc()
                    if self.namespace.has_key("stop"):
                        success, output = self.boxedRun_args_(self.namespace["stop"], [])
                        self.flushOutput_(output)
            except KeyboardInterrupt:
                pass
            pb.end()
            del pb
        self._pageNumber = 1
        self._frame = 1
```

```
    @objc.IBAction
    def exportAsMovie_(self, sender):
        exportPanel = NSSavePanel.savePanel()
        exportPanel.setRequiredFileType_("pdf")
        exportPanel.setNameFieldLabel_("Export To:")
        exportPanel.setPrompt_("Export")
        exportPanel.setCanSelectHiddenExtension_(True)
        exportPanel.setAllowedFileTypes_(["mov"])
        if not NSBundle.loadNibNamed_owner_("ExportMovieAccessory", self):
            NSLog("Error -- could not load ExportMovieAccessory.")
        self.exportMovieFrames.setIntValue_(150)
        self.exportMovieFps.setIntValue_(30)
        exportPanel.setAccessoryView_(self.exportMovieAccessory)
        path = self.fileName()
        if path:
            dirName, fileName = os.path.split(path)
            fileName, ext = os.path.splitext(fileName)
            fileName += ".mov"
        else:
            dirName, fileName = None, "Untitled.mov"
        # If a file was already exported, use that folder as the default.
        if self.exportDir is not None:
            dirName = self.exportDir
        exportPanel.beginSheetForDirectory_file_modalForWindow_modalDelegate_didEndSelector_contextInfo
            dirName,
            fileName,
            NSApp().mainWindow(),
            self,
            "qtPanelDidEnd:returnCode:contextInfo:", 0)

    def qtPanelDidEnd_returnCode_contextInfo_(self, panel, returnCode, context):
        if returnCode:
            fname = panel.filename()
            self.exportDir = os.path.split(fname)[0] # Save the directory we exported to.
            frames = self.exportMovieFrames.intValue()
            fps = self.exportMovieFps.floatValue()
            panel.close()

            if frames <= 0 or fps <= 0: return
            self.doExportAsMovie_frames_fps_(fname, frames, fps)
    qtPanelDidEnd_returnCode_contextInfo_ = objc.selector(qtPanelDidEnd_returnCode_contextInfo_,
                                                          signature="v@:@ii")

    def doExportAsMovie_frames_fps_(self, fname, frames, fps):
        # Only load QTSupport when necessary.
        # QTSupport loads QTKit, which wants to establish a connection to the window
        # server.
        # If we load QTSupport before something is on screen, the connection to the
        # window server cannot be established.

        try:
            os.unlink(fname)
        except:
            pass
        try:
            fp = open(fname, 'w')
            fp.close()
        except:
            errorAlert("File Error", ("Could not create file '%s'. "
                                      "Perhaps it is locked or busy.") % fname)
            return

        movie = None
```

```
            pb = ProgressBarController.alloc().init()
            pb.begin_maxval_("Generating %s frames..." % frames, frames)
            try:
                if not self.cleanRun_newSeed_buildInterface_(self._execScript, True, True):
                    return
                self._pageNumber = 1
                self._frame = 1

                movie = QTSupport.Movie(fname, fps)
                # If the speed is set, we are dealing with animation
                if self.canvas.speed is None:
                    for i in range(frames):
                        if i > 0: # Run has already happened first time
                            self.fastRun_newSeed_(self._execScript, True)
                        movie.add(self.canvas)
                        self.graphicsView.setNeedsDisplay_(True)
                        pb.inc()
                        self._pageNumber += 1
                        self._frame += 1
                else:
                    if self.namespace.has_key("setup"):
                        self.fastRun_newSeed_(self.namespace["setup"], False)
                    for i in range(frames):
                        self.fastRun_newSeed_(self.namespace["draw"], True)
                        movie.add(self.canvas)
                        self.graphicsView.setNeedsDisplay_(True)
                        pb.inc()
                        self._pageNumber += 1
                        self._frame += 1
                    if self.namespace.has_key("stop"):
                        success, output = self.boxedRun_args_(self.namespace["stop"], [])
                        self.flushOutput_(output)
            except KeyboardInterrupt:
                pass
            pb.end()
            del pb
            movie.save()
            self._pageNumber = 1
            self._frame = 1

        @objc.IBAction
        def printDocument_(self, sender):
            op = NSPrintOperation.printOperationWithView_printInfo_(self.graphicsView,
                                                                    self.printInfo())
            op.runOperationModalForWindow_delegate_didRunSelector_contextInfo_(
                NSApp().mainWindow(), self, "printOperationDidRun:success:contextInfo:",
                0)

        def printOperationDidRun_success_contextInfo_(self, op, success, info):
            if success:
                self.setPrintInfo_(op.printInfo())

        printOperationDidRun_success_contextInfo_ = objc.selector(
                                                printOperationDidRun_success_contextInfo_,
                                                signature="v@:@ci")

        @objc.IBAction
        def buildInterface_(self, sender):
            self.dashboardController.buildInterface_(self.vars)

        def validateMenuItem_(self, menuItem):
            if menuItem.action() in ("exportAsImage:", "exportAsMovie:"):
                return self.canvas is not None
```

```
            return True

        # Zoom commands, forwarding to the graphics view.

        @objc.IBAction
        def zoomIn_(self, sender):
            if self.fullScreen is not None: return
            self.graphicsView.zoomIn_(sender)

        @objc.IBAction
        def zoomOut_(self, sender):
            if self.fullScreen is not None: return
            self.graphicsView.zoomOut_(sender)

        @objc.IBAction
        def zoomToTag_(self, sender):
            if self.fullScreen is not None: return
            self.graphicsView.zoomTo_(sender.tag() / 100.0)

        @objc.IBAction
        def zoomToFit_(self, sender):
            if self.fullScreen is not None: return
            self.graphicsView.zoomToFit_(sender)

    class FullscreenWindow(NSWindow):
        def initWithRect_(self, fullRect):
            objc.super(FullscreenWindow,
                       self).initWithContentRect_styleMask_backing_defer_(
                                            fullRect,
                                            NSBorderlessWindowMask,
                                            NSBackingStoreBuffered,
                                            True)
            return self

        def canBecomeKeyWindow(self):
            return True

    class FullscreenView(NSView):

        def init(self):
            super(FullscreenView, self).init()
            self.mousedown = False
            self.keydown = False
            self.key = None
            self.keycode = None
            self.scrollwheel = False
            self.wheeldelta = 0.0
            return self

        def setCanvas_(self, canvas):
            self.canvas = canvas
            self.setNeedsDisplay_(True)
            if not hasattr(self, "screenRect"):
                self.screenRect = NSScreen.mainScreen().frame()
                cw, ch = self.canvas.size
                sw, sh = self.screenRect[1]
                self.scalingFactor = calc_scaling_factor(cw, ch, sw, sh)
                nw, nh = cw * self.scalingFactor, ch * self.scalingFactor
                self.scaledSize = nw, nh
                self.dx = (sw - nw) / 2.0
                self.dy = (sh - nh) / 2.0

        def drawRect_(self, rect):
            NSGraphicsContext.currentContext().saveGraphicsState()
```

```
            NSColor.blackColor().set()
            NSRectFill(rect)
            if self.canvas is not None:
                t = NSAffineTransform.transform()
890             t.translateXBy_yBy_(self.dx, self.dy)
                t.scaleBy_(self.scalingFactor)
                t.concat()
                clip = NSBezierPath.bezierPathWithRect_(
                                    ((0, 0), (self.canvas.width, self.canvas.height)) )
895             clip.addClip()
                self.canvas.draw()
            NSGraphicsContext.currentContext().restoreGraphicsState()

        def isFlipped(self):
900         return True

        def mouseDown_(self, event):
            self.mousedown = True

905     def mouseUp_(self, event):
            self.mousedown = False

        def keyDown_(self, event):
            self.keydown = True
910         self.key = event.characters()
            self.keycode = event.keyCode()

        def keyUp_(self, event):
            self.keydown = False
915         self.key = event.characters()
            self.keycode = event.keyCode()

        def scrollWheel_(self, event):
            self.scrollwheel = True
920         self.wheeldelta = event.deltaY()

        def canBecomeKeyView(self):
            return True

925     def acceptsFirstResponder(self):
            return True

    def calc_scaling_factor(width, height, maxwidth, maxheight):
        return min(float(maxwidth) / width, float(maxheight) / height)
930
    class ZoomPanel(NSView):
        pass

    # class defined in NodeBoxGraphicsView.xib
935 class NodeBoxGraphicsView(NSView):
        document = objc.IBOutlet()
        zoomLevel = objc.IBOutlet()
        zoomField = objc.IBOutlet()
        zoomSlider = objc.IBOutlet()
940
        # The zoom levels are 10%, 25%, 50%, 75%, 100%, 200% and so on up to 2000%.
        zoomLevels = [0.1, 0.25, 0.5, 0.75]
        zoom = 1.0
        while zoom <= 20.0:
945         zoomLevels.append(zoom)
            zoom += 1.0

        def awakeFromNib(self):
            self.canvas = None
```

```
        self._dirty = False
        self.mousedown = False
        self.keydown = False
        self.key = None
        self.keycode = None
        self.scrollwheel = False
        self.wheeldelta = 0.0
        self._zoom = 1.0
        self.setFrameSize_( (graphics.DEFAULT_WIDTH, graphics.DEFAULT_HEIGHT) )
        self.setFocusRingType_(NSFocusRingTypeExterior)
        if self.superview() is not None:
            self.superview().setBackgroundColor_(VERY_LIGHT_GRAY)

    def setCanvas_(self, canvas):
        self.canvas = canvas
        if canvas is not None:
            w, h = self.canvas.size
            self.setFrameSize_([w*self._zoom, h*self._zoom])
        self.markDirty()

    def getZoom(self):
        return self._zoom
    def setZoom_(self, zoom):
        self._zoom = zoom
        self.zoomLevel.setTitle_("%i%%" % (self._zoom * 100.0))
        self.zoomSlider.setFloatValue_(self._zoom * 100.0)
        self.setCanvas_(self.canvas)
    zoom = property(getZoom, setZoom_)

    @objc.IBAction
    def dragZoom_(self, sender):
        self.zoom = self.zoomSlider.floatValue() / 100.0
        self.setCanvas_(self.canvas)

    def findNearestZoomIndex_(self, zoom):
        """Returns the nearest zoom level, and whether we found a direct, exact
        match or a fuzzy match."""
        try: # Search for a direct hit first.
            idx = self.zoomLevels.index(zoom)
            return idx, True
        except ValueError: # Can't find the zoom level, try looking at the indexes.
            idx = 0
            try:
                while self.zoomLevels[idx] < zoom:
                    idx += 1
            except KeyError: # End of the list
                idx = len(self.zoomLevels) - 1 # Just return the last index.
            return idx, False

    @objc.IBAction
    def zoomIn_(self, sender):
        idx, direct = self.findNearestZoomIndex_(self.zoom)
        # Direct hits are perfect, but indirect hits require a bit of help.
        # Because of the way indirect hits are calculated, they are already
        # rounded up to the upper zoom level; this means we don't need to add 1.
        if direct:
            idx += 1
        idx = max(min(idx, len(self.zoomLevels)-1), 0)
        self.zoom = self.zoomLevels[idx]

    @objc.IBAction
    def zoomOut_(self, sender):
        idx, direct = self.findNearestZoomIndex_(self.zoom)
        idx -= 1
```

```
            idx = max(min(idx, len(self.zoomLevels)-1), 0)
            self.zoom = self.zoomLevels[idx]

        @objc.IBAction
        def resetZoom_(self, sender):
            self.zoom = 1.0

        def zoomTo_(self, zoom):
            self.zoom = zoom

        @objc.IBAction
        def zoomToFit_(self, sender):
            w, h = self.canvas.size
            fw, fh = self.superview().frame()[1]
            factor = min(fw / w, fh / h)
            self.zoom = factor

        def markDirty(self, redraw=True):
            self._dirty = True
            if redraw:
                self.setNeedsDisplay_(True)

        def setFrameSize_(self, size):
            self._image = None
            NSView.setFrameSize_(self, size)

        def isOpaque(self):
            return False

        def isFlipped(self):
            return True

        def drawRect_(self, rect):
            if self.canvas is not None:
                NSGraphicsContext.currentContext().saveGraphicsState()
                try:
                    if self.zoom != 1.0:
                        t = NSAffineTransform.transform()
                        t.scaleBy_(self.zoom)
                        t.concat()
                        clip = NSBezierPath.bezierPathWithRect_( ( (0, 0),
                                                                   (self.canvas.width,
                                                                    self.canvas.height)) )
                        clip.addClip()
                    self.canvas.draw()
                except:
                    # A lot of code just to display the error in the output view.
                    etype, value, tb = sys.exc_info()
                    if tb.tb_next is not None:
                        tb = tb.tb_next  # skip the frame doing the exec
                    traceback.print_exception(etype, value, tb)
                    data = "".join(traceback.format_exception(etype, value, tb))
                    attrs = PyDETextView.getBasicTextAttributes()
                    attrs[NSForegroundColorAttributeName] = NSColor.redColor()
                    outputView = self.document.outputView
                    outputView.setSelectedRange_((outputView.textStorage().length(), 0))
                    outputView.setTypingAttributes_(attrs)
                    outputView.insertText_(data)
                NSGraphicsContext.currentContext().restoreGraphicsState()

        def _updateImage(self):
            if self._dirty:
                self._image = self.canvas._nsImage
                self._dirty = False
```

```
        # pasteboard delegate method
    def pasteboard_provideDataForType_(self, pboard, type):
        if NSPDFPboardType:
            pboard.setData_forType_(self.pdfData, NSPDFPboardType)
        elif NSPostScriptPboardType:
            pboard.setData_forType_(self.epsData, NSPostScriptPboardType)
        elif NSTIFFPboardType:
            pboard.setData_forType_(self.tiffData, NSTIFFPboardType)

    def _get_pdfData(self):
        if self.canvas:
            return self.canvas._getImageData('pdf')
    pdfData = property(_get_pdfData)

    def _get_epsData(self):
        if self.canvas:
            return self.canvas._getImageData('eps')
    epsData = property(_get_epsData)

    def _get_tiffData(self):
        return self.canvas._getImageData('tiff')
    tiffData = property(_get_tiffData)

    def _get_pngData(self):
        return self.canvas._getImageData('png')
    pngData = property(_get_pngData)

    def _get_gifData(self):
        return self.canvas._getImageData('gif')
    gifData = property(_get_gifData)

    def _get_jpegData(self):
        return self.canvas._getImageData('jpeg')
    jpegData = property(_get_jpegData)

    def mouseDown_(self, event):
        self.mousedown = True

    def mouseUp_(self, event):
        self.mousedown = False

    def keyDown_(self, event):
        self.keydown = True
        self.key = event.characters()
        self.keycode = event.keyCode()

    def keyUp_(self, event):
        self.keydown = False
        self.key = event.characters()
        self.keycode = event.keyCode()

    def scrollWheel_(self, event):
        NSResponder.scrollWheel_(self, event)
        self.scrollwheel = True
        self.wheeldelta = event.deltaY()

    def canBecomeKeyView(self):
        return True

    def acceptsFirstResponder(self):
        return True

class NodeBoxAppDelegate(NSObject):
```

```
        def awakeFromNib(self):
            self._prefsController = None
            libpath = LibraryFolder()

        @objc.IBAction
        def showPreferencesPanel_(self, sender):
            if self._prefsController is None:
                self._prefsController = NodeBoxPreferencesController.alloc().init()
            self._prefsController.showWindow_(sender)

        @objc.IBAction
        def generateCode_(self, sender):
            """Generate a piece of NodeBox code using OttoBot"""
            # from nodebox.util.ottobot import genProgram
            controller = NSDocumentController.sharedDocumentController()
            doc = controller.newDocument_(sender)
            doc = controller.currentDocument()
            doc.textView.setString_(genProgram())
            doc.runScript()

        @objc.IBAction
        def showHelp_(self, sender):
            url = NSURL.URLWithString_("http://nodebox.net/code/index.php/Reference")
            NSWorkspace.sharedWorkspace().openURL_(url)

        @objc.IBAction
        def showSite_(self, sender):
            url = NSURL.URLWithString_("http://nodebox.net/")
            NSWorkspace.sharedWorkspace().openURL_(url)

        @objc.IBAction
        def showLibrary_(self, sender):
            libpath = LibraryFolder()
            url = NSURL.fileURLWithPath_( makeunicode(libpath.libDir) )
            NSWorkspace.sharedWorkspace().openURL_(url)

        def applicationWillTerminate_(self, note):
            # import atexit
            atexit._run_exitfuncs()
```

nodebox/gui/mac/AskString.py

```
    __all__ = ["AskString"]

    import objc

    import Foundation

    import AppKit
    NSApp = AppKit.NSApp
    # class defined in AskString.xib
    class AskStringWindowController(AppKit.NSWindowController):
        questionLabel = objc.IBOutlet()
        textField = objc.IBOutlet()

        def __new__(cls, question, resultCallback, default="", parentWindow=None):
            self = cls.alloc().initWithWindowNibName_("AskString")
            self.question = question
            self.resultCallback = resultCallback
            self.default = default
            self.parentWindow = parentWindow
            if self.parentWindow is None:
```

```
            self.window().setFrameUsingName_("AskStringPanel")
            self.setWindowFrameAutosaveName_("AskStringPanel")
            self.showWindow_(self)
        else:
            NSApp().beginSheet_modalForWindow_modalDelegate_didEndSelector_contextInfo_(
                self.window(), self.parentWindow, None, None, 0)
        self.retain()
        return self

    def windowWillClose_(self, notification):
        self.autorelease()

    def awakeFromNib(self):
        self.questionLabel.setStringValue_(self.question)
        self.textField.setStringValue_(self.default)

    def done(self):
        if self.parentWindow is None:
            self.close()
        else:
            sheet = self.window()
            NSApp().endSheet_(sheet)
            sheet.orderOut_(self)

    def ok_(self, sender):
        value = self.textField.stringValue()
        self.done()
        self.resultCallback(value)

    def cancel_(self, sender):
        self.done()
        self.resultCallback(None)

def AskString(question, resultCallback, default="", parentWindow=None):
    AskStringWindowController(question, resultCallback, default, parentWindow)
```

nodebox/gui/mac/dashboard.py

```
import AppKit

NSObject = AppKit.NSObject
NSFont = AppKit.NSFont
NSMiniControlSize = AppKit.NSMiniControlSize
NSOnState = AppKit.NSOnState
NSOffState = AppKit.NSOffState
NSTextField = AppKit.NSTextField
NSRightTextAlignment = AppKit.NSRightTextAlignment
NSSlider = AppKit.NSSlider
NSMiniControlSize = AppKit.NSMiniControlSize
NSGraphiteControlTint = AppKit.NSGraphiteControlTint
NSButton = AppKit.NSButton
NSSwitchButton = AppKit.NSSwitchButton
NSSmallControlSize = AppKit.NSSmallControlSize
NSPopUpButton = AppKit.NSPopUpButton

import objc

from nodebox import graphics

SMALL_FONT = NSFont.systemFontOfSize_(NSFont.smallSystemFontSize())
MINI_FONT = NSFont.systemFontOfSize_(NSFont.systemFontSizeForControlSize_(NSMiniControlSize))

```

```
    # class defined in NodeBoxDocument.xib
    class DashboardController(NSObject):
        document = objc.IBOutlet()
        documentWindow = objc.IBOutlet()
30      panel = objc.IBOutlet()

        def clearInterface(self):
            for s in list(self.panel.contentView().subviews()):
                s.removeFromSuperview()
35
        def numberChanged_(self, sender):
            var = self.document.vars[sender.tag()]
            var.value = sender.floatValue()
            self.document.runScript(compile=False, newSeed=False)
40
        def textChanged_(self, sender):
            var = self.document.vars[sender.tag()]
            var.value = sender.stringValue()
            self.document.runScript(compile=False, newSeed=False)
45
        def booleanChanged_(self, sender):
            var = self.document.vars[sender.tag()]
            if sender.state() == NSOnState:
                var.value = True
50          else:
                var.value = False
            self.document.runScript(compile=False, newSeed=False)

        def buttonClicked_(self, sender):
55          var = self.document.vars[sender.tag()]
            self.document.fastRun_newSeed_(self.document.namespace[var.name], True)
            #self.document.runFunction_(var.name)

        def menuSelected_(self, sender):
60          var = self.document.vars[sender.tag()]
            sel = sender.titleOfSelectedItem()
            var.value = sel
            fn = var.dispatchfunction
            self.document.fastRun_newSeed_args_(fn, True, [sel,])
65          #self.document.runFunction_(var.name)

        def buildInterface_(self, variables):
            self.vars = variables
            self.clearInterface()
70          if len(self.vars) > 0:
                self.panel.orderFront_(None)
            else:
                self.panel.orderOut_(None)
                return
75
            # Set the title of the parameter panel to the title of the window
            self.panel.setTitle_(self.documentWindow.title())

            (px,py),(pw,ph) = self.panel.frame()
80          # Height of the window. Each element has a height of 21.
            # The extra "fluff" is 38 pixels.
            ph = len(self.vars) * 21 + 54
            # Start of first element
            # First element is the height minus the fluff.
85          y = ph - 49
            cnt = 0
            for v in self.vars:
                if v.type == graphics.NUMBER:
                    self.addLabel_y_c_(v, y, cnt)
```

```
                self.addSlider_y_c_(v, y, cnt)
            elif v.type == graphics.TEXT:
                self.addLabel_y_c_(v, y, cnt)
                self.addTextField_y_c_(v, y, cnt)
            elif v.type == graphics.BOOLEAN:
                self.addSwitch_y_c_(v, y, cnt)
            elif v.type == graphics.BUTTON:
                self.addButton_y_c_(v, y, cnt)
            elif v.type == graphics.MENU:
                self.addLabel_y_c_(v, y, cnt)
                self.addMenu_y_c_(v, y, cnt)
            y -= 21
            cnt += 1
        self.panel.setFrame_display_animate_( ((px,py),(pw,ph)), True, True )

    def addLabel_y_c_(self, v, y, cnt):
        control = NSTextField.alloc().init()
        control.setFrame_(((0,y),(100,13)))
        control.setStringValue_(v.name + ":")
        control.setAlignment_(NSRightTextAlignment)
        control.setEditable_(False)
        control.setBordered_(False)
        control.setDrawsBackground_(False)
        control.setFont_(SMALL_FONT)
        self.panel.contentView().addSubview_(control)

    def addSlider_y_c_(self, v, y, cnt):
        control = NSSlider.alloc().init()
        control.setMaxValue_(v.max)
        control.setMinValue_(v.min)
        control.setFloatValue_(v.value)
        control.setFrame_(((108,y-1),(172,13)))
        control.cell().setControlSize_(NSMiniControlSize)
        control.cell().setControlTint_(NSGraphiteControlTint)
        control.setContinuous_(True)
        control.setTarget_(self)
        control.setTag_(cnt)
        control.setAction_(objc.selector(self.numberChanged_, signature="v@:@@"))
        self.panel.contentView().addSubview_(control)

    def addTextField_y_c_(self, v, y, cnt):
        control = NSTextField.alloc().init()
        control.setStringValue_(v.value)
        control.setFrame_(((108,y-2),(172,15)))
        control.cell().setControlSize_(NSMiniControlSize)
        control.cell().setControlTint_(NSGraphiteControlTint)
        control.setFont_(MINI_FONT)
        control.setTarget_(self)
        control.setTag_(cnt)
        control.setAction_(objc.selector(self.textChanged_, signature="v@:@@"))
        self.panel.contentView().addSubview_(control)

    def addSwitch_y_c_(self, v, y, cnt):
        control = NSButton.alloc().init()
        control.setButtonType_(NSSwitchButton)
        if v.value:
            control.setState_(NSOnState)
        else:
            control.setState_(NSOffState)
        control.setFrame_(((108,y-2),(172,16)))
        control.setTitle_(v.name)
        control.setFont_(SMALL_FONT)
        control.cell().setControlSize_(NSSmallControlSize)
        control.cell().setControlTint_(NSGraphiteControlTint)
```

```
            control.setTarget_(self)
            control.setTag_(cnt)
            control.setAction_(objc.selector(self.booleanChanged_, signature="v@:@@"))
            self.panel.contentView().addSubview_(control)

        def addButton_y_c_(self, v, y, cnt):
            control = NSButton.alloc().init()
            control.setFrame_(((108, y-2),(172,16)))
            control.setTitle_(v.name)
            control.setBezelStyle_(1)
            control.setFont_(SMALL_FONT)
            control.cell().setControlSize_(NSMiniControlSize)
            control.cell().setControlTint_(NSGraphiteControlTint)
            control.setTarget_(self)
            control.setTag_(cnt)
            control.setAction_(objc.selector(self.buttonClicked_, signature="v@:@@"))
            self.panel.contentView().addSubview_(control)
        def addMenu_y_c_(self, v, y, cnt):
            control = NSPopUpButton.alloc().init()
            control.setFrame_( ((108, y-2),(172,16)) )
            control.setPullsDown_( False )
            control.removeAllItems()
            for title in v.menuitems:
                control.addItemWithTitle_( title )
            control.setTitle_(v.value)
            control.synchronizeTitleAndSelectedItem()
            control.setBezelStyle_(1)
            control.setFont_(SMALL_FONT)
            control.cell().setControlSize_(NSMiniControlSize)
            control.cell().setControlTint_(NSGraphiteControlTint)
            control.setTarget_(self)
            control.setTag_(cnt)
            control.setAction_(objc.selector(self.menuSelected_, signature="v@:@@"))
            self.panel.contentView().addSubview_(control)
```

nodebox/gui/mac/preferences.py

```
import sys
import os
# import pdb

import objc

import AppKit
NSWindowController = AppKit.NSWindowController
NSForegroundColorAttributeName = AppKit.NSForegroundColorAttributeName
NSNotificationCenter = AppKit.NSNotificationCenter
NSFontManager = AppKit.NSFontManager
NSFontAttributeName = AppKit.NSFontAttributeName
NSUserDefaults = AppKit.NSUserDefaults
NSOpenPanel = AppKit.NSOpenPanel

from PyDETextView import getBasicTextAttributes, getSyntaxTextAttributes
from PyDETextView import setTextFont, setBasicTextAttributes, setSyntaxTextAttributes

class LibraryFolder(object):
    def __init__(self):
        prefpath = ""
        try:
            prefpath = NSUserDefaults.standardUserDefaults().objectForKey_("libraryPath")
        except Exception, err:
            print "LibraryFolder: prefpath:", repr(prefpath)
            prefpath = ""
```

```
            stdpath = os.path.join(os.getenv("HOME"), "Library", "Application Support",
                                   "NodeBox")

        if prefpath and os.path.exists( prefpath ):
            self.libDir = prefpath
            NSUserDefaults.standardUserDefaults().setObject_forKey_( self.libDir,
                                                                     "libraryPath")
        else:
            self.libDir = stdpath
            try:
                if not os.path.exists(self.libDir):
                    os.mkdir(libDir)
            except OSError:
                pass
            except IOError:
                pass

    # class defined in NodeBoxPreferences.xib
    class NodeBoxPreferencesController(NSWindowController):
        commentsColorWell = objc.IBOutlet()
        fontPreview = objc.IBOutlet()
        libraryPath = objc.IBOutlet()
        funcClassColorWell = objc.IBOutlet()
        keywordsColorWell = objc.IBOutlet()
        stringsColorWell = objc.IBOutlet()

        def init(self):
            self = self.initWithWindowNibName_("NodeBoxPreferences")
            self.setWindowFrameAutosaveName_("NodeBoxPreferencesPanel")
            self.timer = None
            return self

        def awakeFromNib(self):
            self.textFontChanged_(None)
            syntaxAttrs = syntaxAttrs = getSyntaxTextAttributes()
            self.stringsColorWell.setColor_(syntaxAttrs["string"][NSForegroundColorAttributeName])
            self.keywordsColorWell.setColor_(syntaxAttrs["keyword"][NSForegroundColorAttributeName])
            self.funcClassColorWell.setColor_(syntaxAttrs["identifier"][NSForegroundColorAttributeName])
            self.commentsColorWell.setColor_(syntaxAttrs["comment"][NSForegroundColorAttributeName])
            libpath = LibraryFolder()
            self.libraryPath.setStringValue_( libpath.libDir )

            nc = NSNotificationCenter.defaultCenter()
            nc.addObserver_selector_name_object_(self, "textFontChanged:", "PyDETextFontChanged", None)

        def windowWillClose_(self, notification):
            fm = NSFontManager.sharedFontManager()
            fp = fm.fontPanel_(False)
            if fp is not None:
                fp.setDelegate_(None)
                fp.close()

        @objc.IBAction
        def updateColors_(self, sender):
            if self.timer is not None:
                self.timer.invalidate()
            self.timer = NSTimer.scheduledTimerWithTimeInterval_target_selector_userInfo_repeats_(
                    1.0, self, "timeToUpdateTheColors:", None, False)

        def timeToUpdateTheColors_(self, sender):
            syntaxAttrs = getSyntaxTextAttributes()
            syntaxAttrs["string"][NSForegroundColorAttributeName] = self.stringsColorWell.color()
            syntaxAttrs["keyword"][NSForegroundColorAttributeName] = self.keywordsColorWell.color()
            syntaxAttrs["identifier"][NSForegroundColorAttributeName] = self.funcClassColorWell.color()
```

```
            syntaxAttrs["comment"][NSForegroundColorAttributeName] = self.commentsColorWell.color()
            setSyntaxTextAttributes(syntaxAttrs)

        @objc.IBAction
 95     def chooseFont_(self, sender):
            fm = NSFontManager.sharedFontManager()
            basicAttrs = getBasicTextAttributes()
            fm.setSelectedFont_isMultiple_(basicAttrs[NSFontAttributeName], False)
            fm.orderFrontFontPanel_(sender)
100         fp = fm.fontPanel_(False)
            fp.setDelegate_(self)

        @objc.IBAction
        def chooseLibrary_(self, sender):
105         panel = NSOpenPanel.openPanel()
            panel.setCanChooseFiles_(False)
            panel.setCanChooseDirectories_(True)
            panel.setAllowsMultipleSelection_(False)
            rval = panel.runModalForTypes_([])
110         if rval:
                s = [t for t in panel.filenames()]
                s = s[0]
                NSUserDefaults.standardUserDefaults().setObject_forKey_( s,
                                                                         "libraryPath")
115             libpath = LibraryFolder()
                self.libraryPath.setStringValue_( libpath.libDir )

        @objc.IBAction
        def changeFont_(self, sender):
120         oldFont = getBasicTextAttributes()[NSFontAttributeName]
            newFont = sender.convertFont_(oldFont)
            if oldFont != newFont:
                setTextFont(newFont)

125     def textFontChanged_(self, notification):
            basicAttrs = getBasicTextAttributes()
            font = basicAttrs[NSFontAttributeName]
            self.fontPreview.setFont_(font)
            size = font.pointSize()
130         if size == int(size):
                size = int(size)
            s = u"%s %s" % (font.displayName(), size)
            self.fontPreview.setStringValue_(s)
```

nodebox/gui/mac/progressbar.py

```
    import objc
    import AppKit
    NSDefaultRunLoopMode = AppKit.NSDefaultRunLoopMode

  5 class ProgressBarController(AppKit.NSWindowController):
        messageField = objc.IBOutlet()
        progressBar = objc.IBOutlet()

        def init(self):
 10         AppKit.NSBundle.loadNibNamed_owner_("ProgressBarSheet", self)
            return self

        def begin_maxval_(self, message, maxval):
            self.value = 0
 15         self.message = message
            self.maxval = maxval
            self.progressBar.setMaxValue_(self.maxval)
```

```
            self.messageField.cell().setTitle_(self.message)
            parentWindow = AppKit.NSApp().keyWindow()
            AppKit.NSApp().beginSheet_modalForWindow_modalDelegate_didEndSelector_contextInfo_(self.window(
        def inc(self):
            self.value += 1
            self.progressBar.setDoubleValue_(self.value)
            date = AppKit.NSDate.dateWithTimeIntervalSinceNow_(0.01)
            AppKit.NSRunLoop.currentRunLoop().acceptInputForMode_beforeDate_(NSDefaultRunLoopMode, date)
        def end(self):
            AppKit.NSApp().endSheet_(self.window())
            self.window().orderOut_(self)
```

nodebox/gui/mac/PyDETextView.py

```
from bisect import bisect
import re
import objc
super = objc.super

import AppKit

NSBackgroundColorAttributeName = AppKit.NSBackgroundColorAttributeName
NSBeep = AppKit.NSBeep
NSColor = AppKit.NSColor
NSCommandKeyMask = AppKit.NSCommandKeyMask
NSDictionary = AppKit.NSDictionary
NSEvent = AppKit.NSEvent
NSFont = AppKit.NSFont
NSFontAttributeName = AppKit.NSFontAttributeName
NSForegroundColorAttributeName = AppKit.NSForegroundColorAttributeName
NSLigatureAttributeName = AppKit.NSLigatureAttributeName
NSLiteralSearch = AppKit.NSLiteralSearch
NSNotificationCenter = AppKit.NSNotificationCenter
NSObject = AppKit.NSObject
NSStringPboardType = AppKit.NSStringPboardType
NSTextStorage = AppKit.NSTextStorage
NSTextStorageEditedCharacters = AppKit.NSTextStorageEditedCharacters
NSTextView = AppKit.NSTextView
NSURL = AppKit.NSURL
NSURLPboardType = AppKit.NSURLPboardType
NSViewWidthSizable = AppKit.NSViewWidthSizable

NSCalibratedRGBColorSpace = AppKit.NSCalibratedRGBColorSpace
NSUserDefaults = AppKit.NSUserDefaults

import nodebox.PyFontify
fontify = nodebox.PyFontify.fontify

from nodebox.gui.mac.ValueLadder import ValueLadder

from nodebox.util import _copy_attr, _copy_attrs, makeunicode

whiteRE = re.compile(r"[ \t]+")
commentRE = re.compile(r"[ \t]*(#)")

def findWhitespace(s, pos=0):
    m = whiteRE.match(s, pos)
    if m is None:
        return pos
    return m.end()
```

```
    stringPat = r"q[^\\q\n]*(\\[\000-\377][^\\q\n]*)*q"
    stringOrCommentPat = stringPat.replace("q", "'") + "|" + stringPat.replace('q', '"') + "|#.*"
stringOrCommentRE = re.compile(stringOrCommentPat)

    def removeStringsAndComments(s):
        items = []
        while 1:
            m = stringOrCommentRE.search(s)
            if m:
                start = m.start()
                end = m.end()
                items.append(s[:start])
                if s[start] != "#":
                    items.append("X" * (end - start))  # X-out strings
                s = s[end:]
            else:
                items.append(s)
                break
        return "".join(items)

    class PyDETextView(NSTextView):

        document = objc.IBOutlet()

        def awakeFromNib(self):
            # Can't use a subclass of NSTextView as an NSTextView in IB,
            # so we need to set some attributes programmatically
            scrollView = self.superview().superview()
            self.setFrame_(((0, 0), scrollView.contentSize()))
            self.setAutoresizingMask_(NSViewWidthSizable)
            self.textContainer().setWidthTracksTextView_(True)
            self.setAllowsUndo_(True)
            self.setRichText_(False)
            self.setTypingAttributes_(getBasicTextAttributes())
            self.setUsesFindPanel_(True)
            self.usesTabs = 0
            self.indentSize = 4
            self._string = self.textStorage().mutableString().nsstring()
            self._storageDelegate = PyDETextStorageDelegate(self.textStorage())

            # FDB: no wrapping
            # Thanks to http://cocoa.mamasam.com/COCOADEV/2003/12/2/80304.php
            scrollView = self.enclosingScrollView()
            scrollView.setHasHorizontalScroller_(True)
            self.setHorizontallyResizable_(True)
            layoutSize = self.maxSize()
            layoutSize = (layoutSize[1], layoutSize[1])
            self.setMaxSize_(layoutSize)
            self.textContainer().setWidthTracksTextView_(False)
            self.textContainer().setContainerSize_(layoutSize)

            # FDB: value ladder
            self.valueLadder = None

            nc = NSNotificationCenter.defaultCenter()
            nc.addObserver_selector_name_object_(self, "textFontChanged:",
                                                 "PyDETextFontChanged", None)

        def drawRect_(self, rect):
            NSTextView.drawRect_(self, rect)
            if self.valueLadder is not None and self.valueLadder.visible:
                self.valueLadder.draw()

        def hideValueLadder(self):
```

```
            if self.valueLadder is not None:
                self.valueLadder.hide()
                if self.valueLadder.dirty:
                    self.document.updateChangeCount_(True)
            self.valueLadder = None

        def mouseUp_(self, event):
            self.hideValueLadder()
            NSTextView.mouseUp_(self, event)

        def mouseDragged_(self,event):
            if self.valueLadder is not None:
                self.valueLadder.mouseDragged_(event)
            else:
                NSTextView.mouseDragged_(self, event)

        def mouseDown_(self, event):
            if event.modifierFlags() & NSCommandKeyMask:
                screenPoint = NSEvent.mouseLocation()
                viewPoint =   self.superview().convertPoint_fromView_(event.locationInWindow(),
                                                                      self.window().contentView())

                c = self.characterIndexForPoint_(screenPoint)

                txt = self.string()
                # XXX move code into ValueLadder
                try:
                    if txt[c] in "1234567890.":
                        # Find full number
                        begin = c
                        end = c
                        try:
                            while txt[begin-1] in "1234567890.":
                                begin-=1
                        except IndexError:
                            pass
                        try:
                            while txt[end+1] in "1234567890.":
                                end+=1
                        except IndexError:
                            pass
                        end+=1
                        self.valueLadder = ValueLadder(self,
                                                       eval(txt[begin:end]),
                                                       (begin,end),
                                                       screenPoint, viewPoint)
                except IndexError:
                    pass
            else:
                NSTextView.mouseDown_(self,event)

        def acceptableDragTypes(self):
            return list(super(PyDETextView, self).acceptableDragTypes()) + [NSURLPboardType]

        def draggingEntered_(self, dragInfo):
            pboard = dragInfo.draggingPasteboard()
            types = pboard.types()
            if NSURLPboardType in pboard.types():
                # Convert URL to string, replace pboard entry, let NSTextView
                # handle the drop as if it were a plain text drop.
                url = NSURL.URLFromPasteboard_(pboard)
                if url.isFileURL():
                    s = url.path()
                else:
```

```
                s = url.absoluteString()
            s = 'u"%s"' % s.replace('"', '\\"')
            pboard.declareTypes_owner_([NSStringPboardType], self)
            pboard.setString_forType_(s, NSStringPboardType)
        return super(PyDETextView, self).draggingEntered_(dragInfo)

    def _cleanup(self):
        # delete two circular references
        del self._string
        del self._storageDelegate

    def __del__(self):
        nc = NSNotificationCenter.defaultCenter()
        nc.removeObserver_name_object_(self, "PyDETextFontChanged", None)

    @objc.IBAction
    def jumpToLine_(self, sender):
        from nodebox.gui.mac.AskString import AskString
        AskString("Jump to line number:", self.jumpToLineCallback_,
                  parentWindow=self.window())

    def jumpToLineCallback_(self, value):
        if value is None:
            return  # user cancelled
        try:
            lineNo = int(value.strip())
        except ValueError:
            NSBeep()
        else:
            self.jumpToLineNr_(lineNo)

    def jumpToLineNr_(self, lineNo):
        lines = self.textStorage().string().splitlines()
        lineNo = min(max(0, lineNo - 1), len(lines))
        length_of_prevs = sum([len(line)+1 for line in lines[:lineNo]])
        curlen = len(lines[lineNo])
        rng = (length_of_prevs, curlen)
        self.setSelectedRange_(rng)
        self.scrollRangeToVisible_(rng)
        self.setNeedsDisplay_(True)

    def textFontChanged_(self, notification):
        basicAttrs = getBasicTextAttributes()
        self.setTypingAttributes_(basicAttrs)
        # Somehow the next line is needed, we crash otherwise :(
        self.layoutManager().invalidateDisplayForCharacterRange_(
                                                        (0, self._string.length()))
        self._storageDelegate.textFontChanged_(notification)

    def setTextStorage_str_tabs_(self, storage, string, usesTabs):
        storage.addLayoutManager_(self.layoutManager())
        self._string = string
        self.usesTabs = usesTabs

    @objc.IBAction
    def changeFont_(self, sender):
        # Change the font through the user prefs API, we'll get notified
        # through textFontChanged_
        font = getBasicTextAttributes()[NSFontAttributeName]
        font = sender.convertFont_(font)
        setTextFont(font)

    def getLinesForRange_(self, rng):
        rng = self._string.lineRangeForRange_(rng)
```

```
        return self._string.substringWithRange_(rng), rng

    def getIndent(self):
        if self.usesTabs:
            return "\t"
        else:
            return self.indentSize * " "

    def drawInsertionPointInRect_color_turnedOn_(self, pt, color, on):
        self.insertionPoint = pt
        super(PyDETextView, self).drawInsertionPointInRect_color_turnedOn_(pt, color, on)

    def keyDown_(self, event):
        super(PyDETextView, self).keyDown_(event)
        char = event.characters()[:1]
        if char in ")]}":
            selRng = self.selectedRange()
            line, lineRng, pos = self.findMatchingIndex_paren_(selRng[0] - 1, char)
            if pos is not None:
                self.balanceParens_(lineRng[0] + pos)

    def balanceParens_(self, index):
        rng = (index, 1)
        oldAttrs, effRng = self.textStorage().attributesAtIndex_effectiveRange_(index,
                                                                                 None)
        balancingAttrs = {
            NSBackgroundColorAttributeName: NSColor.selectedTextBackgroundColor()
        }
        # Must use temp attrs otherwise the attrs get reset right away due to colorizing.
        self.layoutManager().setTemporaryAttributes_forCharacterRange_(balancingAttrs,
                                                                       rng)
        self.performSelector_withObject_afterDelay_("resetBalanceParens:",
                (oldAttrs, effRng), 0.2)

    def resetBalanceParens_(self, (attrs, rng)):
        self.layoutManager().setTemporaryAttributes_forCharacterRange_(attrs, rng)

    def iterLinesBackwards_maxChars_(self, end, maxChars):
        begin = max(0, end - maxChars)
        if end > 0:
            prevChar = self._string.characterAtIndex_(end - 1)
            if prevChar == "\n":
                end += 1
        lines, linesRng = self.getLinesForRange_((begin, end - begin))
        lines = lines[:end - linesRng[0]]
        linesRng = (linesRng[0], len(lines))
        lines = lines.splitlines(True)
        lines.reverse()
        for line in lines:
            nChars = len(line)
            yield line, (end - nChars, nChars)
            end -= nChars
        assert end == linesRng[0]

    def findMatchingIndex_paren_(self, index, paren):
        openToCloseMap = {"(": ")", "[": "]", "{": "}"}
        if paren:
            stack = [paren]
        else:
            stack = []
        line, lineRng, pos = None, None, None
        for line, lineRng in self.iterLinesBackwards_maxChars_(index, 8192):
            line = removeStringsAndComments(line)
            pos = None
```

```
            for i in range(len(line)-1, -1, -1):
                c = line[i]
                if c in ")]}":
                    stack.append(c)
                elif c in "([{":
                    if not stack:
                        if not paren:
                            pos = i
                        break
                    elif stack[-1] != openToCloseMap[c]:
                        # mismatch
                        stack = []
                        break
                    else:
                        stack.pop()
                        if paren and not stack:
                            pos = i
                            break
            if not stack:
                break
        return line, lineRng, pos

    def insertNewline_(self, sender):
        selRng = self.selectedRange()
        super(PyDETextView, self).insertNewline_(sender)
        line, lineRng, pos = self.findMatchingIndex_paren_(selRng[0], None)
        if line is None:
            return
        leadingSpace = ""
        if pos is None:
            m = whiteRE.match(line)
            if m is not None:
                leadingSpace = m.group()
        else:
            leadingSpace = re.sub(r"[^\t]", " ", line[:pos + 1])
        line, lineRng = self.getLinesForRange_((selRng[0], 0))
        line = removeStringsAndComments(line).strip()
        if line and line[-1] == ":":
            leadingSpace += self.getIndent()

        if leadingSpace:
            self.insertText_(leadingSpace)

    def insertTab_(self, sender):
        if self.usesTabs:
            return super(PyDETextView, self).insertTab_(sender)
        self.insertText_("")
        selRng = self.selectedRange()
        assert selRng[1] == 0
        lines, linesRng = self.getLinesForRange_(selRng)
        sel = selRng[0] - linesRng[0]
        whiteEnd = findWhitespace(lines, sel)
        nSpaces = self.indentSize - (whiteEnd % self.indentSize)
        self.insertText_(nSpaces * " ")
        sel += nSpaces
        whiteEnd += nSpaces
        sel = min(whiteEnd, sel + (sel % self.indentSize))
        self.setSelectedRange_((sel + linesRng[0], 0))

    def deleteBackward_(self, sender):
        self.delete_fwd_superf_(sender, False, super(PyDETextView, self).deleteBackward_)

    def deleteForward_(self, sender):
        self.delete_fwd_superf_(sender, True, super(PyDETextView, self).deleteForward_)
```

```
        def delete_fwd_superf_(self, sender, isForward, superFunc):
370         selRng = self.selectedRange()
            if self.usesTabs or selRng[1]:
                return superFunc(sender)
            lines, linesRng = self.getLinesForRange_(selRng)
            sel = selRng[0] - linesRng[0]
375         whiteEnd = findWhitespace(lines, sel)
            whiteBegin = sel
            while whiteBegin and lines[whiteBegin-1] == " ":
                whiteBegin -= 1
            if not isForward:
380             white = whiteBegin
            else:
                white = whiteEnd
            if white == sel or (whiteEnd - whiteBegin) <= 1:
                return superFunc(sender)
385         nSpaces = (whiteEnd % self.indentSize)
            if nSpaces == 0:
                nSpaces = self.indentSize
            offset = sel % self.indentSize
            if not isForward and offset == 0:
390             offset = nSpaces
            delBegin = sel - offset
            delEnd = delBegin + nSpaces
            delBegin = max(delBegin, whiteBegin)
            delEnd = min(delEnd, whiteEnd)
395         self.setSelectedRange_((linesRng[0] + delBegin, delEnd - delBegin))
            self.insertText_("")

        @objc.IBAction
        def indent_(self, sender):
400         def indentFilter(lines):
                indent = self.getIndent()
                indentedLines = []
                for line in lines:
                    if line.strip():
405                     indentedLines.append(indent + line)
                    else:
                        indentedLines.append(line)
                [indent + line for line in lines[:-1]]
                return indentedLines
410         self.filterLines_(indentFilter)

        @objc.IBAction
        def dedent_(self, sender):
            def dedentFilter(lines):
415             indent = self.getIndent()
                dedentedLines = []
                indentSize = len(indent)
                for line in lines:
                    if line.startswith(indent):
420                     line = line[indentSize:]
                    dedentedLines.append(line)
                return dedentedLines
            self.filterLines_(dedentFilter)

425     @objc.IBAction
        def comment_(self, sender):
            def commentFilter(lines):
                commentedLines = []
                indent = self.getIndent()
430             pos = 100
                for line in lines:
```

```
                if not line.strip():
                    continue
                pos = min(pos, findWhitespace(line))
            for line in lines:
                if line.strip():
                    commentedLines.append(line[:pos] + "#" + line[pos:])
                else:
                    commentedLines.append(line)
            return commentedLines
        self.filterLines_(commentFilter)

    @objc.IBAction
    def uncomment_(self, sender):
        def uncommentFilter(lines):
            commentedLines = []
            commentMatch = commentRE.match
            for line in lines:
                m = commentMatch(line)
                if m is not None:
                    pos = m.start(1)
                    line = line[:pos] + line[pos+1:]
                commentedLines.append(line)
            return commentedLines
        self.filterLines_(uncommentFilter)

    def filterLines_(self, filterFunc):
        selRng = self.selectedRange()
        lines, linesRng = self.getLinesForRange_(selRng)

        filteredLines = filterFunc(lines.splitlines(True))

        filteredLines = "".join(filteredLines)
        if lines == filteredLines:
            return
        self.setSelectedRange_(linesRng)
        self.insertText_(filteredLines)
        newSelRng = linesRng[0], len(filteredLines)
        self.setSelectedRange_(newSelRng)

class PyDETextStorageDelegate(NSObject):

    def __new__(cls, *args, **kwargs):
        return cls.alloc().init()

    def __init__(self, textStorage=None):
        self._syntaxColors = getSyntaxTextAttributes()
        self._haveScheduledColorize = False
        self._source = None  # XXX
        self._dirty = []
        if textStorage is None:
            textStorage = NSTextStorage.alloc().init()
        self._storage = textStorage
        self._storage.setAttributes_range_(getBasicTextAttributes(),
                (0, textStorage.length()))
        self._string = self._storage.mutableString().nsstring()
        self._lineTracker = LineTracker(self._string)
        self._storage.setDelegate_(self)

    def textFontChanged_(self, notification):
        self._storage.setAttributes_range_(getBasicTextAttributes(),
                (0, self._storage.length()))
        self._syntaxColors = getSyntaxTextAttributes()
        self._dirty = [0]
        self.scheduleColorize()
```

```
        def textStorage(self):
            return self._storage

500     def string(self):
            return self._string

        def lineIndexFromCharIndex_(self, charIndex):
            return self._lineTracker.lineIndexFromCharIndex_(charIndex)
505
        def charIndexFromLineIndex_(self, lineIndex):
            return self._lineTracker.charIndexFromLineIndex_(lineIndex)

        def numberOfLines(self):
510         return self._lineTracker.numberOfLines()

        def getSource(self):
            if self._source is None:
                self._source = unicode(self._string)
515         return self._source

        def textStorageWillProcessEditing_(self, notification):
            if not self._storage.editedMask() & NSTextStorageEditedCharacters:
                return
520         rng = self._storage.editedRange()
            # make darn sure we don't get infected with return chars
            s = self._string
            s.replaceOccurrencesOfString_withString_options_range_("\r", "\n",
                                                                   NSLiteralSearch , rng)
525
        def textStorageDidProcessEditing_(self, notification):
            if not self._storage.editedMask() & NSTextStorageEditedCharacters:
                return
            self._source = None
530         rng = self._storage.editedRange()
            try:
                self._lineTracker._update(rng, self._storage.changeInLength())
            except:
                import traceback
535             traceback.print_exc()
            start = rng[0]
            rng = (0, 0)
            count = 0
            while start > 0:
540             # find the last colorized token and start from there.
                start -= 1
                attrs, rng = self._storage.attributesAtIndex_effectiveRange_(start, None)
                value = attrs.objectForKey_(NSForegroundColorAttributeName)
                if value != None:
545                 count += 1
                    if count > 1:
                        break
                # uncolorized section, track back
                start = rng[0] - 1
550         rng = self._string.lineRangeForRange_((rng[0], 0))
            self._dirty.append(rng[0])
            self.scheduleColorize()

        def scheduleColorize(self):
555         if not self._haveScheduledColorize:
                self.performSelector_withObject_afterDelay_("colorize", None, 0.0)
                self._haveScheduledColorize = True

        def colorize(self):
```

```
        self._haveScheduledColorize = False
        self._storage.beginEditing()
        try:
            try:
                self._colorize()
            except:
                import traceback
                traceback.print_exc()
        finally:
            self._storage.endEditing()

    def _colorize(self):
        if not self._dirty:
            return
        storage = self._storage
        source = self.getSource()
        sourceLen = len(source)
        dirtyStart = self._dirty.pop()

        getColor = self._syntaxColors.get
        setAttrs = storage.setAttributes_range_
        getAttrs = storage.attributesAtIndex_effectiveRange_
        basicAttrs = getBasicTextAttributes()

        lastEnd = end = dirtyStart
        count = 0
        sameCount = 0
        for tag, start, end, sublist in fontify(source, dirtyStart):
            end = min(end, sourceLen)
            rng = (start, end - start)
            attrs = getColor(tag)
            oldAttrs, oldRng = getAttrs(rng[0], None)
            if attrs is not None:
                clearRng = (lastEnd, start - lastEnd)
                if clearRng[1]:
                    setAttrs(basicAttrs, clearRng)
                setAttrs(attrs, rng)
                if rng == oldRng and attrs == oldAttrs:
                    sameCount += 1
                    if sameCount > 4:
                        # due to backtracking we have to account for a few more
                        # tokens, but if we've seen a few tokens that were already
                        # colorized the way we want, we're done
                        return
                else:
                    sameCount = 0
            else:
                rng = (lastEnd, end - lastEnd)
                if rng[1]:
                    setAttrs(basicAttrs, rng)
            count += 1
            if count > 200:
                # enough for now, schedule a new chunk
                self._dirty.append(end)
                self.scheduleColorize()
                break
            lastEnd = end
        else:
            # reset coloring at the end
            end = min(sourceLen, end)
            rng = (end, sourceLen - end)
            if rng[1]:
                setAttrs(basicAttrs, rng)
```

```
    class LineTracker(object):

        def __init__(self, string):
            self.string = string
            self.lines, self.lineStarts, self.lineLengths = self._makeLines()

        def _makeLines(self, start=0, end=None):
            lines = []
            lineStarts = []
            lineLengths = []
            string = self.string
            if end is None:
                end = string.length()
            else:
                end = min(end, string.length())
            rng = string.lineRangeForRange_((start, end - start))
            pos = rng[0]
            end = pos + rng[1]
            while pos < end:
                lineRng = string.lineRangeForRange_((pos, 0))
                line = makeunicode(string.substringWithRange_(lineRng))
                assert len(line) == lineRng[1]
                lines.append(line)
                lineStarts.append(lineRng[0])
                lineLengths.append(lineRng[1])
                if not lineRng[1]:
                    break
                pos += lineRng[1]
            return lines, lineStarts, lineLengths

        def _update(self, editedRange, changeInLength):
            oldRange = editedRange[0], editedRange[1] - changeInLength
            start = self.lineIndexFromCharIndex_(oldRange[0])
            if oldRange[1]:
                end = self.lineIndexFromCharIndex_(oldRange[0] + oldRange[1])
            else:
                end = start

            lines, lineStarts, lineLengths = self._makeLines(
                editedRange[0], editedRange[0] + editedRange[1] + 1)
            self.lines[start:end + 1] = lines
            self.lineStarts[start:] = lineStarts  # drop invalid tail
            self.lineLengths[start:end + 1] = lineLengths
            # XXX: This assertion doesn't actually assert
            # assert "".join(self.lines) == unicode(self.string)

        def lineIndexFromCharIndex_(self, charIndex):
            lineIndex = bisect(self.lineStarts, charIndex)
            if lineIndex == 0:
                return 0
            nLines = len(self.lines)
            nLineStarts = len(self.lineStarts)
            if lineIndex == nLineStarts and nLineStarts != nLines:
                # update line starts
                i = nLineStarts - 1
                assert i >= 0
                pos = self.lineStarts[i]
                while pos <= charIndex and i < nLines:
                    pos = pos + self.lineLengths[i]
                    self.lineStarts.append(pos)
                    i += 1
                lineIndex = i

            lineIndex -= 1
```

```
            start = self.lineStarts[lineIndex]
            line = self.lines[lineIndex]
690         if (    line[-1:] == "\n"
                and not (start <= charIndex < start + self.lineLengths[lineIndex])):
                lineIndex += 1
            return lineIndex

695     def charIndexFromLineIndex_(self, lineIndex):
            if not self.lines:
                return 0
            if lineIndex == len(self.lines):
                return self.lineStarts[-1] + self.lineLengths[-1]
700         try:
                return self.lineStarts[lineIndex]
            except IndexError:
                # update lineStarts
                for i in range(min(len(self.lines), lineIndex + 1) - len(self.lineStarts)):
705                 self.lineStarts.append(self.lineStarts[-1] + self.lineLengths[-1])
                # XXX: Assertion doesn't actually assert.
                #assert len(self.lineStarts) == len(self.lineLengths) == len(self.lines)
                if lineIndex == len(self.lineStarts):
                    return self.lineStarts[-1] + self.lineLengths[-1]
710             return self.lineStarts[lineIndex]

        def numberOfLines(self):
            return len(self.lines)

715 _basicFont = NSFont.userFixedPitchFontOfSize_(11)

    _BASICATTRS = {NSFontAttributeName: _basicFont,
                   NSLigatureAttributeName: 0}
    _SYNTAXCOLORS = {
720     "keyword": {NSForegroundColorAttributeName: NSColor.blueColor()},
        "identifier": {
            NSForegroundColorAttributeName: NSColor.redColor().shadowWithLevel_(0.2)},
        "string": {NSForegroundColorAttributeName: NSColor.magentaColor()},
        "comment": {NSForegroundColorAttributeName: NSColor.grayColor()},
725 }
    for key, value in _SYNTAXCOLORS.items():
        newVal = _BASICATTRS.copy()
        newVal.update(value)
        _SYNTAXCOLORS[key] = NSDictionary.dictionaryWithDictionary_(newVal)
730 _BASICATTRS = NSDictionary.dictionaryWithDictionary_(_BASICATTRS)

    def unpackAttrs(d):
        unpacked = {}
        for key, value in d.items():
735         if key == NSFontAttributeName:
                name = value["name"]
                size = value["size"]
                value = NSFont.fontWithName_size_(name, size)
            elif key in (NSForegroundColorAttributeName, NSBackgroundColorAttributeName):
740             r, g, b, a = map(float, value.split())
                value = NSColor.colorWithCalibratedRed_green_blue_alpha_(r, g, b, a)
            elif isinstance(value, (dict, NSDictionary)):
                value = unpackAttrs(value)
            unpacked[key] = value
745     return unpacked

    def packAttrs(d):
        packed = {}
        for key, value in d.items():
750         if key == NSFontAttributeName:
                value = {"name": value.fontName(), "size": value.pointSize()}
```

```
            elif key in (NSForegroundColorAttributeName, NSBackgroundColorAttributeName):
                col = value.colorUsingColorSpaceName_(NSCalibratedRGBColorSpace)
                channels = col.getRed_green_blue_alpha_(None, None, None, None)
                value = " ".join(map(str, channels))
            elif isinstance(value, (dict, NSDictionary)):
                value = packAttrs(value)
            packed[key] = value
        return packed

    def getBasicTextAttributes():
        attrs = NSUserDefaults.standardUserDefaults().objectForKey_(
                "PyDEDefaultTextAttributes")
        return unpackAttrs(attrs)

    def getSyntaxTextAttributes():
        attrs = NSUserDefaults.standardUserDefaults().objectForKey_(
                "PyDESyntaxTextAttributes")
        return unpackAttrs(attrs)

    def setBasicTextAttributes(basicAttrs):
        if basicAttrs != getBasicTextAttributes():
            NSUserDefaults.standardUserDefaults().setObject_forKey_(
                    packAttrs(basicAttrs), "PyDEDefaultTextAttributes")
            nc = NSNotificationCenter.defaultCenter()
            nc.postNotificationName_object_("PyDETextFontChanged", None)

    def setSyntaxTextAttributes(syntaxAttrs):
        if syntaxAttrs != getSyntaxTextAttributes():
            NSUserDefaults.standardUserDefaults().setObject_forKey_(
                    packAttrs(syntaxAttrs), "PyDESyntaxTextAttributes")
            nc = NSNotificationCenter.defaultCenter()
            nc.postNotificationName_object_("PyDETextFontChanged", None)

    def setTextFont(font):
        basicAttrs = getBasicTextAttributes()
        syntaxAttrs = getSyntaxTextAttributes()
        basicAttrs[NSFontAttributeName] = font
        for v in syntaxAttrs.values():
            v[NSFontAttributeName] = font
        setBasicTextAttributes(basicAttrs)
        setSyntaxTextAttributes(syntaxAttrs)

    _defaultUserDefaults = {
        "PyDEDefaultTextAttributes": packAttrs(_BASICATTRS),
        "PyDESyntaxTextAttributes": packAttrs(_SYNTAXCOLORS),
    }

    NSUserDefaults.standardUserDefaults().registerDefaults_(_defaultUserDefaults)
```

nodebox/gui/mac/util.py

```
    import AppKit

    def errorAlert(msgText, infoText):
        # Force NSApp initialisation.
        AppKit.NSApplication.sharedApplication().activateIgnoringOtherApps_(0)
        alert = AppKit.NSAlert.alloc().init()
        alert.setMessageText_(msgText)
        alert.setInformativeText_(infoText)
        alert.setAlertStyle_(AppKit.NSCriticalAlertStyle)
        btn = alert.addButtonWithTitle_("OK")
        return alert.runModal()
```

nodebox/gui/mac/ValueLadder.py

```
#from Foundation import *
#from AppKit import *

import compiler
parse = compiler.parse

import compiler.ast
Sub = compiler.ast.Sub
UnarySub = compiler.ast.UnarySub
Add = compiler.ast.Add

import Foundation
import AppKit

NSObject = AppKit.NSObject
NSColor = AppKit.NSColor
NSMutableParagraphStyle = AppKit.NSMutableParagraphStyle
NSCenterTextAlignment = AppKit.NSCenterTextAlignment
NSFont = AppKit.NSFont
NSForegroundColorAttributeName = AppKit.NSForegroundColorAttributeName
NSCursor = AppKit.NSCursor
NSGraphicsContext = AppKit.NSGraphicsContext
NSBezierPath = AppKit.NSBezierPath
NSString = AppKit.NSString
NSEvent = AppKit.NSEvent
NSAlternateKeyMask = AppKit.NSAlternateKeyMask
NSShiftKeyMask = AppKit.NSShiftKeyMask
NSParagraphStyleAttributeName = AppKit.NSParagraphStyleAttributeName
NSFontAttributeName = AppKit.NSFontAttributeName

MAGICVAR = "__magic_var__"

class ValueLadder:

    view = None
    visible = False
    value = None
    origValue = None
    dirty = False
    type = None
    negative = False
    unary = False
    add = False

    def __init__(self, textView, value, clickPos, screenPoint, viewPoint):
        self.textView = textView
        self.value = value
        self.origValue = value
        self.type = type(value)
        self.clickPos = clickPos
        self.origX, self.origY = screenPoint
        self.x, self.y = screenPoint
        self.viewPoint = viewPoint
        (x,y),(self.width,self.height) = self.textView.bounds()
        self.originalString = self.textView.string()
        self.backgroundColor = NSColor.colorWithCalibratedRed_green_blue_alpha_(
                                                                0.4,0.4,0.4, 1.0)
        self.strokeColor = NSColor.colorWithCalibratedRed_green_blue_alpha_(
                                                                0.1,0.1,0.1, 1.0)
        self.textColor = NSColor.colorWithCalibratedRed_green_blue_alpha_(
                                                                1.0,1.0,1.0, 1.0)
        paraStyle = NSMutableParagraphStyle.alloc().init()
```

```
            paraStyle.setAlignment_(NSCenterTextAlignment)
            font = NSFont.fontWithName_size_("Monaco", 10)
 65         self.textAttributes = {
                NSForegroundColorAttributeName: self.textColor,
                NSParagraphStyleAttributeName:  paraStyle,NSFontAttributeName:font}

            # To speed things up, the code is compiled only once.
 70         # The number is replaced with a magic variable, that is set in the
            # namespace when executing the code.
            begin,end = self.clickPos
            self.patchedSource = (self.originalString[:begin]
                                  + MAGICVAR
 75                               + self.originalString[end:])

            #ast = parse(self.patchedSource + "\n\n")
            #self._checkSigns(ast)
            success, output = self.textView.document.boxedRun_args_(self._parseAndCompile, [])
 80         if success:
                self.show()
            else:
                self.textView.document._flushOutput(output)

 85     def _parseAndCompile(self):
            ast = parse(self.patchedSource.encode('ascii', 'replace') + "\n\n")
            self._checkSigns(ast)
            self.textView.document._compileScript(self.patchedSource)

 90     def _checkSigns(self, node):
            """Recursively check for special sign cases.

            The following cases are special:
            - Substraction. When you select the last part of a substraction
 95           (e.g. the 5 of "10-5"), it might happen that you drag the number to
              a positive value. In that case, the result should be "10+5".
            - Unary substraction. Values like "-5" should have their sign removed
              when you drag them to a positive value.
            - Addition. When you select the last part of an addition
100           (e.g. the 5 of "10+5"), and drag the number to a negative value,
              the result should be "10-5".

            This algorithm checks for these cases. It tries to find the magic var,
            and then checks the parent node to see if it is one of these cases,
105         then sets the appropriate state variables in the object.

            This algorithm is recursive. Because we have to differ between a
            "direct hit" (meaning the current child was the right one) and a
            "problem resolved" (meaning the algorithm found the node, did its
110         work and now needs to bail out), we have three return codes:
            - -1: nothing was found in this node and its child nodes.
            -  1: direct hit. The child you just searched contains the magicvar.
                  check the current node to see if it is one of the special cases.
            -  0: bail out. Somewhere, a child contained the magicvar, and we
115               acted upon it. Now leave this algorithm as soon as possible.
            """

            # Check whether I am the correct node
            try:
120             if node.name == MAGICVAR:
                    return 1 # If i am, return the "direct hit" code.
            except AttributeError:
                pass

125         # We keep an index to see what child we are checking. This
            # is important for binary operations, were we are only interested
```

```
            # in the second part. ("a-10" has to change to "a+10",
            # but "10-a" shouldn't change to "+10-a")
            index = 0
            # Recursively check my children
            for child in node.getChildNodes():
                retVal = self._checkSigns(child)
                # Direct hit. The child I just searched contains the magicvar.
                # Check whether this node is one of the special cases.
                if retVal == 1:
                    # Unary substitution.
                    if isinstance(node, UnarySub):
                        self.negative = True
                        self.unary = True
                    # Binary substitution. Only the second child is of importance.
                    elif isinstance(node, Sub) and index == 1:
                        self.negative = True
                    # Binary addition. Only the second child is of importance.
                    elif isinstance(node, Add) and index == 1:
                        self.add = True
                    # Return the "bail out" code, whether we found some
                    # special case or not. There can only be one magicvar in the
                    # code, so once that is found we can stop looking.
                    return 0
                # If the child returns a bail out code, we leave this routine
                # without checking the other children, passing along the
                # bail out code.
                elif retVal == 0:
                    return 0 # Nothing more needs to be done.

                # Next child.
                index += 1

            # We searched all children, but couldn't find any magicvars.
            return -1

        def show(self):
            self.visible = True
            self.textView.setNeedsDisplay_(True)
            NSCursor.hide()

        def hide(self):
            """Hide the ValueLadder and update the code.

            Updating the code means we have to replace the current value with
            the new value, and account for any special cases."""

            self.visible = False
            begin,end = self.clickPos

            # Potentionally change the sign on the number.
            # The following cases are valid:
            # - A subtraction where the value turned positive "random(5-8)" --> "random(5+8)"
            # - A unary subtraction where the value turned positive "random(-5)" --> "random(5)"
            #   Note that the sign dissapears here.
            # - An addition where the second part turns negative "random(5+8)" --> "random(5-8)"
            # Note that the code replaces the sign on the place where it was, leaving the code intact.

            # Case 1: Negative numbers where the new value is negative as well.
            # This means the numbers turn positive.
            if self.negative and self.value < 0:
                # Find the minus sign.
                i = begin - 1
                notFound = True
                while True:
```

```
                if self.originalString[i] == '-':
                    if self.unary: # Unary subtractions will have the sign removed.
                        # Re-create the string: the spaces between the value and the '-' + the value
                        value = self.originalString[i+1:begin] + str(abs(self.value))
                    else: # Binary subtractions get a '+'
                        value = '+' + self.originalString[i+1:begin] + str(abs(self.value))
                    range = (i,end-i)
                    break
                i -= 1
        # Case 2: Additions (only additions where we are the second part
        # interests us, this is checked already on startup)
        elif self.add and self.value < 0:
            # Find the plus sign.
            i = begin - 1
            notFound = True
            while True:
                if self.originalString[i] == '+':
                    # Re-create the string:
                    # - a '+' (instead of the minus)
                    # - the spaces between the '-' and the constant
                    # - the constant itself
                    value = '-' + self.originalString[i+1:begin] + str(abs(self.value))
                    range = (i,end-i)
                    break
                i -= 1
        # Otherwise, it's a normal case. Note that here also, positive numbers
        # can turn negative, but no existing signs have to be changed.
        else:
            value = str(self.value)
            range = (begin, end-begin)

        # The following textView methods make sure that an undo operation
        # is registered, so users can undo their drag.
        self.textView.shouldChangeTextInRange_replacementString_(range, value)
        self.textView.textStorage().replaceCharactersInRange_withString_(range, value)
        self.textView.didChangeText()
        self.textView.setNeedsDisplay_(True)
        self.textView.document.currentView.direct = False
        NSCursor.unhide()

    def draw(self):
        mx,my=self.viewPoint

        x = mx-20
        w = 80
        h = 20
        h2 = h*2

        context = NSGraphicsContext.currentContext()
        aa = context.shouldAntialias()
        context.setShouldAntialias_(False)
        r = ((mx-w/2,my+12),(w,h))
        NSBezierPath.setDefaultLineWidth_(0)
        self.backgroundColor.set()
        NSBezierPath.fillRect_(r)
        self.strokeColor.set()
        NSBezierPath.strokeRect_(r)

        # A standard value just displays the value that you have been dragging.
        if not self.negative:
            v = str(self.value)
        # When the value is negative, we don't display a double negative,
        # but a positive.
        elif self.value < 0:
```

```
            v = str(abs(self.value))
        # When the value is positive, we have to add a minus sign.
        else:
            v = "-" + str(self.value)

        NSString.drawInRect_withAttributes_(v, ((mx-w/2,my+14),(w,h2)), self.textAttributes)
        context.setShouldAntialias_(aa)

    def mouseDragged_(self, event):
        mod = event.modifierFlags()
        newX, newY = NSEvent.mouseLocation()
        deltaX = newX-self.x
        delta = deltaX
        if self.negative:
            delta = -delta
        if mod & NSAlternateKeyMask:
            delta /= 100.0
        elif mod & NSShiftKeyMask:
            delta *= 10.0
        self.value = self.type(self.value + delta)
        self.x, self.y = newX, newY
        self.dirty = True
        self.textView.setNeedsDisplay_(True)
        self.textView.document.magicvar = self.value
        self.textView.document.currentView.direct = True
        self.textView.document.runScriptFast()
```

nodebox/util/__init__.py

```
import os
import datetime
import glob

import random as librandom
choice = librandom.choice

import unicodedata
import objc

import Foundation
import AppKit

import kgp

__all__ = ('grid', 'random', 'choice', 'files', 'autotext', '_copy_attr', '_copy_attrs',
           'datestring','makeunicode', 'filelist', 'imagefiles',
           'fontnames', 'fontfamilies')

### Utilities ###

def makeunicode(s, srcencoding="utf-8", normalizer="NFC"):
    typ = type(s)
    # convert to str first; for number types etc.
    if typ not in (str, unicode, Foundation.NSMutableAttributedString,
        objc.pyobjc_unicode, Foundation.NSMutableStringProxyForMutableAttributedString,
        Foundation.NSString):
        # print "makeunicode() convert:", typ
        s = str(s)
    if typ not in (unicode, Foundation.NSMutableAttributedString, objc.pyobjc_unicode,
                   Foundation.NSMutableStringProxyForMutableAttributedString):
        try:
            s = unicode(s, srcencoding)
        except TypeError, err:
```

```
            print
            print "makeunicode():", err
            print repr(s)
            print type(s)
            print
    if typ in (unicode,):
        s = unicodedata.normalize(normalizer, s)
    return s

def datestring(dt = None, dateonly=False, nospaces=True, nocolons=True):
    """Make an ISO datestring. The defaults are good for using the result of
    'datestring()' in a filename.
    """
    if not dt:
        now = str(datetime.datetime.now())
    else:
        now = str(dt)
    if not dateonly:
        now = now[:19]
    else:
        now = now[:10]
    if nospaces:
        now = now.replace(" ", "_")
    if nocolons:
        now = now.replace(":", "")
    return now

def grid(cols, rows, colSize=1, rowSize=1, shuffled=False):
    """Returns an iterator that contains coordinate tuples.

    The grid can be used to quickly create grid-like structures.
    A common way to use them is:
        for x, y in grid(10,10,12,12):
            rect(x,y, 10,10)
    """
    # Prefer using generators.
    rowRange = xrange(int(rows))
    colRange = xrange(int(cols))
    # Shuffled needs a real list, though.
    if (shuffled):
        rowRange = list(rowRange)
        colRange = list(colRange)
        shuffle(rowRange)
        shuffle(colRange)
    for y in rowRange:
        for x in colRange:
            yield (x*colSize,y*rowSize)

def random(v1=None, v2=None):
    """Returns a random value.

    This function does a lot of things depending on the parameters:
    - If one or more floats is given, the random value will be a float.
    - If all values are ints, the random value will be an integer.

    - If one value is given, random returns a value from 0 to the given value.
      This value is not inclusive.
    - If two values are given, random returns a value between the two; if two
      integers are given, the two boundaries are inclusive.
    """
    if v1 != None and v2 == None: # One value means 0 -> v1
        if isinstance(v1, float):
            return librandom.random() * v1
        else:
```

```
                return int(librandom.random() * v1)
    elif v1 != None and v2 != None: # v1 -> v2
        if isinstance(v1, float) or isinstance(v2, float):
            start = min(v1, v2)
            end = max(v1, v2)
            return start + librandom.random() * (end-start)
        else:
            start = min(v1, v2)
            end = max(v1, v2) + 1
            return int(start + librandom.random() * (end-start))
    else: # No values means 0.0 -> 1.0
        return librandom.random()

def files(path="*"):
    """Returns a list of files.

    You can use wildcards to specify which files to pick, e.g.
        f = files('*.gif')
    """
    f = glob.glob(path)
    f = [makeunicode(t) for t in f]
    return f

def filelist( folderpathorlist, pathonly=True ):
    """Walk a folder or a list of folders and return
    paths or ((filepath, size, lastmodified, mode) tuples..
    """

    folders = folderpathorlist
    if type(folderpathorlist) in (str, unicode):
        folders = [folderpathorlist]
    result = []
    for folder in folders:
        folder = os.path.expanduser( folder )
        folder = os.path.abspath( folder )
        for root, dirs, files in os.walk( folder ):
            root = makeunicode( root )

            # skip if dir starts with '.'
            _, parentfolder = os.path.split(root)
            if parentfolder[0] == u".":
                continue

            for thefile in files:
                thefile = makeunicode( thefile )
                basename, ext = os.path.splitext(thefile)

                # exclude dotfiles
                if thefile.startswith('.'):
                    continue

                # exclude the specials
                for item in (u'\r', u'\n', u'\t'):
                    if item in thefile:
                        continue

                filepath = os.path.join( root, thefile )

                record = filepath
                if not pathonly:
                    islink = os.path.islink( filepath )
                    if islink:
                        info = os.lstat( filepath )
                    else:
```

```
                        info = os.stat( filepath )
                    lastmodified = datetime.datetime.fromtimestamp( info.st_mtime )
                    record = (filepath, info.st_size, lastmodified,
                              oct(info.st_mode), islink )
                yield record

    def imagefiles( folderpathorlist, pathonly=True ):
        """Use filelist to extract all imagefiles"""
        result = []
        filetuples = filelist( folderpathorlist, pathonly=pathonly )

        # 2017-06-23 - kw .eps dismissed
        extensions = tuple(".pdf .tif .tiff .gif .jpg .jpeg .png".split())
        for filetuple in filetuples:
            path = filetuple
            if not pathonly:
                path = filetuple[0]
            _, ext = os.path.splitext( path )
            if ext.lower() not in extensions:
                continue
            if pathonly:
                yield path
            else:
                yield filetuple

    def fontnames():
        fm = AppKit.NSFontManager.sharedFontManager()
        l = fm.availableFonts()
        result = []
        for i in l:
            # filter out the weird fontnames
            if i.startswith(u'.'):
                continue
            result.append( makeunicode(i) )
        return result

    class FontRecord:
        def __init__(self, psname, familyname, style, weight, traits, traitnames):
            self.psname = psname
            self.familyname = familyname
            self.style = style
            self.weight = weight
            self.traits = traits
            self.traitnames = traitnames
        def __repr__(self):
            return (u'FontRecord( psname="%s", familyname="%s", style="%s", '
                    u'weight=%.2f, traits="%s", traitnames=%s)') % (
                                self.psname, self.familyname, self.style,
                                self.weight, self.traits, self.traitnames)

    def fontfamilies(flat=False):
        fm = AppKit.NSFontManager.sharedFontManager()
        l = fm.availableFontFamilies()

        def makeTraitsList( traits ):
            appleTraits = {
                0x00000001: u"italic",
                0x00000002: u"bold",
                0x00000004: u"unbold",
                0x00000008: u"nonstandardcharacterset",
                0x00000010: u"narrow",
                0x00000020: u"expanded",
                0x00000040: u"condensed",
                0x00000080: u"smallcaps",
```

```
                0x00000100: u"poster",
                0x00000200: u"compressed",
                0x00000400: u"fixedpitch",
230             0x01000000: u"unitalic"}
            result = []
            keys = appleTraits.keys()
            for key in keys:
                if traits & key == key:
235                 result.append( appleTraits[key])
            return result

        def makeFontRecord(fnt):
            psname, styl, weight, traits = fnt
240         psname = makeunicode(psname)
            styl = makeunicode(styl)
            weight = float( weight )
            traits = int(traits)
            traitNames = makeTraitsList( traits )
245         return FontRecord(psname, familyName, styl, weight, traits, traitNames)

        if flat:
            result = []
        else:
250         result = {}
        for fn in l:
            familyName = makeunicode( fn )
            if not flat:
                result[familyName] = famfonts = {}
255
            subs = fm.availableMembersOfFontFamily_( familyName )
            for fnt in subs:
                fontRec = makeFontRecord( fnt )
                if not flat:
260                 result[familyName][fontRec.style] = fontRec
                else:
                    result.append( fontRec )
        return result

265 def autotext(sourceFile):
        k = kgp.KantGenerator(sourceFile)
        return k.output()

    def _copy_attr(v):
270     if v is None:
            return None
        elif hasattr(v, "copy"):
            return v.copy()
        elif isinstance(v, list):
275         return list(v)
        elif isinstance(v, tuple):
            return tuple(v)
        elif isinstance(v, (int, str, unicode, float, bool, long)):
            return v
280     else:
            raise NodeBoxError, "Don't know how to copy '%s'." % v

    def _copy_attrs(source, target, attrs):
        for attr in attrs:
285         setattr(target, attr, _copy_attr(getattr(source, attr)))
```

nodebox/util/kgp/__init__.py

```
#!/usr/bin/env python2
```

```
"""Kant Generator for Python

Generates mock philosophy based on a context-free grammar

Usage: python kgp.py [options] [source]

Options:
  -g ..., --grammar=...   use specified grammar file or URL
  -h, --help              show this help
  -d                      show debugging information while parsing

Examples:
  kgp.py                  generates several paragraphs of Kantian philosophy
  kgp.py -g husserl.xml   generates several paragraphs of Husserl
  kpg.py "<xref id='paragraph'/>"  generates a paragraph of Kant
  kgp.py template.xml     reads from template.xml to decide what to generate

This program is part of "Dive Into Python", a free Python book for
experienced programmers.  Visit http://diveintopython.org/ for the
latest version.
"""

__author__ = "Mark Pilgrim (f8dy@diveintopython.org)"
__version__ = "$Revision: 1.3 $"
__date__ = "$Date: 2002/05/28 17:05:23 $"
__copyright__ = "Copyright (c) 2001 Mark Pilgrim"
__license__ = "Python"

from xml.dom import minidom
import random
import sys
import getopt

_debug = 0

def openAnything(source):
    """URI, filename, or string --> stream

    This function lets you define parsers that take any input source
    (URL, pathname to local or network file, or actual data as a string)
    and deal with it in a uniform manner.  Returned object is guaranteed
    to have all the basic stdio read methods (read, readline, readlines).
    Just .close() the object when you're done with it.

    Examples:
    >>> from xml.dom import minidom
    >>> sock = openAnything("http://localhost/kant.xml")
    >>> doc = minidom.parse(sock)
    >>> sock.close()
    >>> sock = openAnything("c:\\inetpub\\wwwroot\\kant.xml")
    >>> doc = minidom.parse(sock)
    >>> sock.close()
    >>> sock = openAnything("<ref id='conjunction'><text>and</text><text>or</text></ref>")
    >>> doc = minidom.parse(sock)
    >>> sock.close()
    """

    if hasattr(source, "read"):
        return source

    if source == "-":
        import sys
        return sys.stdin

```

```
        # try to open with urllib (if source is http, ftp, or file URL)
        import urllib
        try:
            return urllib.urlopen(source)
    70  except (IOError, OSError):
            pass

        # try to open with native open function (if source is pathname)
        try:
    75      return open(source)
        except (IOError, OSError):
            pass

        # treat source as string
    80  import StringIO
        return StringIO.StringIO(str(source))

    class NoSourceError(Exception): pass

    85 class KantGenerator:
        """generates mock philosophy based on a context-free grammar"""

        def __init__(self, grammar, source=None):
            self.loadGrammar(grammar)
    90      self.loadSource(source and source or self.getDefaultSource())
            self.refresh()

        def _load(self, source):
            """load XML input source, return parsed XML document
    95
            - a URL of a remote XML file ("http://diveintopython.org/kant.xml")
            - a filename of a local XML file ("~/diveintopython/common/py/kant.xml")
            - standard input ("-")
            - the actual XML document, as a string
    100     """
            sock = openAnything(source)
            xmldoc = minidom.parse(sock).documentElement
            sock.close()
            return xmldoc
    105
        def loadGrammar(self, grammar):
            """load context-free grammar"""
            self.grammar = self._load(grammar)
            self.refs = {}
    110     for ref in self.grammar.getElementsByTagName("ref"):
                self.refs[ref.attributes["id"].value] = ref

        def loadSource(self, source):
            """load source"""
    115     self.source = self._load(source)

        def getDefaultSource(self):
            """guess default source of the current grammar

    120     The default source will be one of the <ref>s that is not
            cross-referenced.  This sounds complicated but it's not.
            Example: The default source for kant.xml is
            "<xref id='section'/>", because 'section' is the one <ref>
            that is not <xref>'d anywhere in the grammar.
    125     In most grammars, the default source will produce the
            longest (and most interesting) output.
            """
            xrefs = {}
            for xref in self.grammar.getElementsByTagName("xref"):
```

```
            xrefs[xref.attributes["id"].value] = 1
        xrefs = xrefs.keys()
        standaloneXrefs = [e for e in self.refs.keys() if e not in xrefs]
        if not standaloneXrefs:
            raise NoSourceError, "can't guess source, and no source specified"
        return '<xref id="%s"/>' % random.choice(standaloneXrefs)

    def reset(self):
        """reset parser"""
        self.pieces = []
        self.capitalizeNextWord = 0

    def refresh(self):
        """reset output buffer, re-parse entire source file, and return output

        Since parsing involves a good deal of randomness, this is an
        easy way to get new output without having to reload a grammar file
        each time.
        """
        self.reset()
        self.parse(self.source)
        return self.output()

    def output(self):
        """output generated text"""
        return "".join(self.pieces)

    def randomChildElement(self, node):
        """choose a random child element of a node

        This is a utility method used by do_xref and do_choice.
        """
        choices = [e for e in node.childNodes
                   if e.nodeType == e.ELEMENT_NODE]
        chosen = random.choice(choices)
        if _debug:
            sys.stderr.write('%s available choices: %s\n' % \
                (len(choices), [e.toxml() for e in choices]))
            sys.stderr.write('Chosen: %s\n' % chosen.toxml())
        return chosen

    def parse(self, node):
        """parse a single XML node

        A parsed XML document (from minidom.parse) is a tree of nodes
        of various types.  Each node is represented by an instance of the
        corresponding Python class (Element for a tag, Text for
        text data, Document for the top-level document).  The following
        statement constructs the name of a class method based on the type
        of node we're parsing ("parse_Element" for an Element node,
        "parse_Text" for a Text node, etc.) and then calls the method.
        """
        parseMethod = getattr(self, "parse_%s" % node.__class__.__name__)
        parseMethod(node)

    def parse_Document(self, node):
        """parse the document node

        The document node by itself isn't interesting (to us), but
        its only child, node.documentElement, is: it's the root node
        of the grammar.
        """
        self.parse(node.documentElement)
```

```
        def parse_Text(self, node):
195         """parse a text node

            The text of a text node is usually added to the output buffer
            verbatim.  The one exception is that <p class='sentence'> sets
            a flag to capitalize the first letter of the next word.  If
200         that flag is set, we capitalize the text and reset the flag.
            """
            text = node.data
            if self.capitalizeNextWord:
                self.pieces.append(text[0].upper())
205             self.pieces.append(text[1:])
                self.capitalizeNextWord = 0
            else:
                self.pieces.append(text)

210     def parse_Element(self, node):
            """parse an element

            An XML element corresponds to an actual tag in the source:
            <xref id='...'>, <p chance='...'>, <choice>, etc.
215         Each element type is handled in its own method.  Like we did in
            parse(), we construct a method name based on the name of the
            element ("do_xref" for an <xref> tag, etc.) and
            call the method.
            """
220         handlerMethod = getattr(self, "do_%s" % node.tagName)
            handlerMethod(node)

        def parse_Comment(self, node):
            """parse a comment
225
            The grammar can contain XML comments, but we ignore them
            """
            pass

230     def do_xref(self, node):
            """handle <xref id='...'> tag

            An <xref id='...'> tag is a cross-reference to a <ref id='...'>
            tag.  <xref id='sentence'/> evaluates to a randomly chosen child of
235         <ref id='sentence'>.
            """
            id = node.attributes["id"].value
            self.parse(self.randomChildElement(self.refs[id]))

240     def do_p(self, node):
            """handle <p> tag

            The <p> tag is the core of the grammar.  It can contain almost
            anything: freeform text, <choice> tags, <xref> tags, even other
245         <p> tags.  If a "class='sentence'" attribute is found, a flag
            is set and the next word will be capitalized.  If a "chance='X'"
            attribute is found, there is an X% chance that the tag will be
            evaluated (and therefore a (100-X)% chance that it will be
            completely ignored)
250         """
            keys = node.attributes.keys()
            if "class" in keys:
                if node.attributes["class"].value == "sentence":
                    self.capitalizeNextWord = 1
255         if "chance" in keys:
                chance = int(node.attributes["chance"].value)
                doit = (chance > random.randrange(100))
```

```
            else:
                doit = 1
        if doit:
            for child in node.childNodes: self.parse(child)

    def do_choice(self, node):
        """handle <choice> tag

        A <choice> tag contains one or more <p> tags.  One <p> tag
        is chosen at random and evaluated; the rest are ignored.
        """
        self.parse(self.randomChildElement(node))

def usage():
    print __doc__

def main(argv):
    grammar = "kant.xml"
    try:
        opts, args = getopt.getopt(argv, "hg:d", ["help", "grammar="])
    except getopt.GetoptError:
        usage()
        sys.exit(2)
    for opt, arg in opts:
        if opt in ("-h", "--help"):
            usage()
            sys.exit()
        elif opt == '-d':
            global _debug
            _debug = 1
        elif opt in ("-g", "--grammar"):
            grammar = arg

    source = "".join(args)
    k = KantGenerator(grammar, source)
    print k.output()

if __name__ == "__main__":
    main(sys.argv[1:])
```

nodebox/util/ottobot/__init__.py

```
from AppKit import NSFontManager

from nodebox.util import random, choice

COMP_WIDTH = 500
COMP_HEIGHT = 500

XCOORD = 1
YCOORD = 2
XSIZE = 3
YSIZE = 4
ROTATION = 5
SCALE = 6
CONTROLPOINT = 7
COLOR = 8
STROKEWIDTH = 9
LOOP = 10
GRIDDELTA = 12
GRIDCOUNT = 13
GRIDWIDTH = 14
GRIDHEIGHT = 15
```

```
   SKEW = 16
   STARPOINTS = 17

class Context:
    def __init__(self):
        self._indent = 0
        self._grid = False

    def indent(self):
        self._indent += 1

    def dedent(self):
        self._indent -= 1

    def spaces(self):
        return "    " * self._indent

    def inGrid(self):
        return self._grid

def nrReally(ctx, numberclass):
    if numberclass == XCOORD:
        if ctx.inGrid():
            #return "x"
            return "x + %s" % nr(ctx,GRIDDELTA)
        else:
            return random(-COMP_WIDTH/2,COMP_WIDTH/2)
    elif numberclass == YCOORD:
        if ctx.inGrid():
            #return "y"
            return "y + %s" % nr(ctx,GRIDDELTA)
        else:
            return random(-COMP_HEIGHT/2,COMP_HEIGHT/2)
    elif numberclass == XSIZE:
        return random(0,COMP_WIDTH)
    elif numberclass == YSIZE:
        return random(0,COMP_HEIGHT)
    elif numberclass == ROTATION:
        return random(0,360)
    elif numberclass == SCALE:
        return random(0.5,1.5)
    elif numberclass == CONTROLPOINT:
        return random(-100,100)
    elif numberclass == COLOR:
        return random()
    elif numberclass == STROKEWIDTH:
        return random(1,20)
    elif numberclass == LOOP:
        return random(2, 20)
    elif numberclass == GRIDDELTA:
        return random(-100,100)
    elif numberclass == GRIDCOUNT:
        return random(2, 10)
    elif numberclass == GRIDWIDTH:
        return 20
        return random(1,100)
    elif numberclass == GRIDHEIGHT:
        return 20
        return random(1, 100)
    elif numberclass == SKEW:
        return random(1,80)
    elif numberclass == STARPOINTS:
        return random(2,100)

```

```
    def nr(ctx, numberclass):
        if not ctx.inGrid() and random() > 0.5:
            return "random(%s)" % nrReally(ctx, numberclass)
        else:
 90         return "%s" % nrReally(ctx, numberclass)

    ### DRAWING COMMANDS ###

    def genDraw(ctx):
 95     fn = choice((genRect,genOval,genArrow,genStar,genPath))
        return fn(ctx)

    def genRect(ctx):
        return ctx.spaces() + """rect(%s,%s,%s,%s)\n"""  % (
100         nr(ctx,XCOORD),nr(ctx,YCOORD),nr(ctx,XSIZE),nr(ctx,YSIZE))

    def genOval(ctx):
        return ctx.spaces() + """oval(%s,%s,%s,%s)\n"""  % (
            nr(ctx,XCOORD),nr(ctx,YCOORD),nr(ctx,XSIZE),nr(ctx,YSIZE))
105
    def genArrow(ctx):
        return ctx.spaces() + """arrow(%s,%s,%s)\n""" % (
            nr(ctx,XCOORD),nr(ctx,YCOORD),nr(ctx,XSIZE))

110 def genStar(ctx):
        return ctx.spaces() + """star(%s,%s,%s,%s,%s)\n""" % (
            nr(ctx,XCOORD),nr(ctx,YCOORD),nr(ctx,STARPOINTS),nr(ctx,XSIZE),nr(ctx,XSIZE))

    def genPath(ctx):
115     s = ctx.spaces() + """beginpath(%s,%s)\n""" % (
            nr(ctx,XCOORD),nr(ctx,YCOORD))
        for i in range(random(1,10)):
            s += genPathDraw(ctx)
        s += ctx.spaces() + """endpath()\n"""
120     return s

    def genPathDraw(ctx):
        fn = choice((genLineto, genCurveto))
        return fn(ctx)
125
    def genLineto(ctx):
        return ctx.spaces() + """lineto(%s,%s)\n""" % (nr(ctx,XCOORD),nr(ctx,YCOORD))

    def genCurveto(ctx):
130     return ctx.spaces() + """curveto(%s,%s,%s,%s,%s,%s)\n""" % (
            nr(ctx,XCOORD),nr(ctx,YCOORD),nr(ctx,CONTROLPOINT),nr(ctx,CONTROLPOINT),nr(ctx,CONTROLPOINT),nr

    ### TRANSFORM ###

135 def genTransform(ctx):
        fn = choice((genRotate, genTranslate, genScale, genSkew, genReset))
        return fn(ctx)

    def genRotate(ctx):
140     return ctx.spaces() + """rotate(%s)\n""" % nr(ctx,ROTATION)

    def genTranslate(ctx):
        return ctx.spaces() + """translate(%s,%s)\n""" % (nr(ctx,XCOORD), nr(ctx,YCOORD))

145 def genScale(ctx):
        return ctx.spaces() + """scale(%s)\n""" % (nr(ctx,SCALE))

    def genSkew(ctx):
        return ctx.spaces() + """skew(%s)\n""" % (nr(ctx,SKEW))
```

```
150
    def genReset(ctx):
        return ctx.spaces() + """reset()\n"""

    ### COLOR ###
155
    def genColor(ctx):
        fn = choice((genFill,genFill,genFill,genFill,genFill,genFill,genStroke,genStroke,genStroke,genNofil
        return fn(ctx)

160 def genFill(ctx):
        return ctx.spaces() + """fill(%s,%s,%s,%s)\n""" % (nr(ctx,COLOR),nr(ctx,COLOR), nr(ctx,COLOR), nr(c

    def genStroke(ctx):
        return ctx.spaces() + """stroke(%s,%s,%s,%s)\n""" % (nr(ctx,COLOR), nr(ctx,COLOR), nr(ctx,COLOR), n
165
    def genNofill(ctx):
        return ctx.spaces() + """nofill()\n"""

    def genNostroke(ctx):
170     return ctx.spaces() + """nostroke()\n"""

    def genStrokewidth(ctx):
        return ctx.spaces() + """strokewidth(%s)\n""" % nr(ctx,STROKEWIDTH)

175 ### LOOP ###
    def genLoop(ctx):
        fn = choice((genFor, genGrid))
        return fn(ctx)

180 def genFor(ctx):
        if ctx._indent >= 2: return ""
        s = ctx.spaces() + """for i in range(%s):\n""" % nr(ctx,LOOP)
        ctx.indent()
        for i in range(random(5)):
185         s += genStatement(ctx)
        s += genVisual(ctx)
        ctx.dedent()
        return s

190 def genGrid(ctx):
        if ctx.inGrid(): return ""
        s = ctx.spaces() + """for x, y in grid(%s,%s,%s,%s):\n""" % (nr(ctx,GRIDCOUNT), nr(ctx,GRIDCOUNT),
        ctx.indent()
        ctx._grid = True
195     for i in range(random(5)):
            s += genStatement(ctx)
        s += genVisual(ctx)
        ctx.dedent()
        ctx._grid = False
200     return s

    ### MAIN ###

    def genVisual(ctx):
205     fn = choice((genDraw,))
        return fn(ctx)

    def genStatement(ctx):
        fn = choice((genVisual,genLoop,genColor,genTransform))
210     return fn(ctx)

    def genProgram():
        s = """# This code is generated with OTTOBOT,
```

```
    # the automatic NodeBox code generator.
215 size(%s, %s)
    translate(%s, %s)
    colormode(HSB)
    """ % (COMP_WIDTH, COMP_HEIGHT, COMP_WIDTH/2, COMP_HEIGHT/2)
        ctx = Context()
220     for i in range(random(10,20)):
            s += genStatement(ctx)
        return s

    if __name__ == '__main__':
225     print genProgram()
```

nodebox/util/QTSupport/__init__.py

```
    import os
    import tempfile
    import Foundation
    NSNumber = Foundation.NSNumber
  5
    import AppKit
    NSImage = AppKit.NSImage
    NSApplication = AppKit.NSApplication
    NSColor = AppKit.NSColor
 10 NSData = AppKit.NSData
    NSBitmapImageRep = AppKit.NSBitmapImageRep
    NSJPEGFileType = AppKit.NSJPEGFileType

    import QTKit
 15 QTMovie = QTKit.QTMovie
    QTDataReference = QTKit.QTDataReference
    QTMovieFileNameAttribute = QTKit.QTMovieFileNameAttribute
    QTMakeTimeRange = QTKit.QTMakeTimeRange
    QTMakeTime = QTKit.QTMakeTime
 20 QTMovieEditableAttribute = QTKit.QTMovieEditableAttribute
    QTAddImageCodecType = QTKit.QTAddImageCodecType
    QTMovieFlatten = QTKit.QTMovieFlatten

    class Movie(object):
 25
        def __init__(self, fname, fps=30):
            if os.path.exists(fname):
                os.remove(fname)
            self.frame = 1
 30         self.fname = fname
            self.tmpfname = None
            self.firstFrame = True
            self.movie = None
            self.fps = fps
 35         self._time = QTMakeTime(int(600/self.fps), 600)

        def add(self, canvas_or_context):
            if self.movie is None:
                # The first frame will be written to a temporary png file,
 40             # then opened as a movie file, then saved again as a movie.
                handle, self.tmpfname = tempfile.mkstemp('.tiff')
                canvas_or_context.save(self.tmpfname)
                try:
                    movie, err = QTMovie.movieWithFile_error_(self.tmpfname, None)
 45                 movie.setAttribute_forKey_(NSNumber.numberWithBool_(True), QTMovieEditableAttribute)
                    range = QTMakeTimeRange(QTMakeTime(0,600), movie.duration())
                    movie.scaleSegment_newDuration_(range, self._time)
                    if err is not None:
```

```
                    raise str(err)
                movie.writeToFile_withAttributes_(self.fname, {QTMovieFlatten:True})
                self.movie, err = QTMovie.movieWithFile_error_(self.fname, None)
                self.movie.setAttribute_forKey_(NSNumber.numberWithBool_(True), QTMovieEditableAttribut
                if err is not None:
                    raise str(err)
                self.imageTrack = self.movie.tracks()[0]
            finally:
                os.remove(self.tmpfname)
        else:
            try:
                canvas_or_context.save(self.tmpfname)
                img = NSImage.alloc().initByReferencingFile_(self.tmpfname)
                self.imageTrack.addImage_forDuration_withAttributes_(img, self._time, {QTAddImageCodecT
            finally:
                try:
                    os.remove(self.tmpfname)
                except OSError, err:
                    print err
                    # pass
        self.frame += 1

    def save(self):
        self.movie.updateMovieFile()

def test():
    import sys
    sys.path.insert(0, '../..')
    sys.path.insert(0, '../../..')
    from nodebox.graphics import Canvas, Context
    from math import sin

    NSApplication.sharedApplication().activateIgnoringOtherApps_(0)
    w, h = 500, 300
    m = Movie("xx3.mov")
    for i in range(200):
        print "Frame", i
        ctx = Context()
        ctx.size(w, h)
        ctx.rect(100.0+sin(i/10.0)*100.0,i/2.0,100,100)
        ctx.text(str(i), i*2, 200)
        m.add(ctx)
    m.save()

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

nodebox/util/vdiff.py

```
import os
import PIL.Image as Image

HTML_HEADER = r'''
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/html4/strict.dtd">
<head>
<meta http-equiv="content-type" content="text/html; charset=utf-8">
<title>Vdiff Test Results</title>
<style type="text/css" media="all">
body { margin: 20px 0 20px 150px; }
body, td, th { font: 11px/1.5em "Lucida Grande", sans-serif; }
h1 { font-size: 160%; padding: 0; margin: 0em 0 -2em 0; }
h2 { font-size: 130%; padding: 0; margin: 4em 0 0.2em 0; clear:both; }
img { float: left; border: 1px solid #000; margin: 2px; }
```

```
.different table { background: red; }
table.statistics { margin:2px; width:16em; border:1px solid #666; }
table.statistics td { font-weight: bold; text-align: right; padding: 2px 5px; }
table.statistics td + td { font-weight: normal; text-align: left; }
tr.even { background: #eee; }
tr.odd { background: #ddd; }
</style>
</head>
<body>
<h1>vdiff tests</h1>
'''

HTML_FOOTER = r'''
</body>
</html>
'''

def format_stats(stats):
    if stats.number_of_differences > 0:
        clz = " different"
    else:
        clz = ""

    html  = """<h2>%s</h2>\n""" % stats.name
    html += """<div class="stats%s">""" % clz
    html += """<a href="%s" target="_blank"><img src="%s" width="150" height="150"></a>\n""" % (stats.f
    html += """<a href="%s" target="_blank"><img src="%s" width="150" height="150"></a>\n""" % (stats.f
    if stats.comparison_image_fname is not None:
        html += """<a href="%s" target="_blank"><img class="compare" src="%s" width="150" height="150">
    html += """<table class="statistics" height="152">\n"""
    html += """<tr class="odd"><td>Differences:</td><td>%i</td></tr>\n""" % len(stats.differences)
    html += """<tr class="even"><td>Total delta:</td><td>%i</td></tr>\n""" % stats.total_delta
    html += """<tr class="odd"><td>Max delta:</td><td>%i</td></tr>\n""" % stats.max_delta
    html += """<tr class="even"><td>Mean:</td><td>%.4f</td></tr>\n""" % stats.mean
    html += """<tr class="odd"><td>Stdev:</td><td>%.4f</td></tr>\n""" % stats.stdev
    html += """</table>\n"""
    html += """</div>"""
    return html

def format_stats_list(stats_list):
    html = HTML_HEADER
    for stats in stats_list:
        html += format_stats(stats)
    html += HTML_FOOTER
    return html

def compare_pixel(px1, px2):
    if px1 == px2:
        return 0
    r1, g1, b1, a1 = px1
    r2, g2, b2, a2 = px2
    return abs(r1-r2) + abs(g1-g2) + abs(b1-b2) + abs(a1-a2)

def visual_diff(img1, img2, threshold=0, stop_on_diff=False):
    if isinstance(img1, str) or isinstance(img1, unicode):
        img1 = Image.open(img1)
        img1 = img1.convert("RGBA")
    if isinstance(img2, str) or isinstance(img2, unicode):
        img2 = Image.open(img2)
        img2 = img2.convert("RGBA")
    assert img1.size == img2.size
    w, h = img1.size
    data1 = img1.getdata()
    data2 = img2.getdata()
```

```
        size = len(data1)
 80     differences = []
        for i in xrange(size):
            delta = compare_pixel(data1[i], data2[i])
            if delta > threshold:
                x = i % w
 85             y = i / w
                differences.append( ( (x, y), data1[i], data2[i], delta ) )
                if stop_on_diff:
                    # print data1[i], data2[i]
                    break
 90     return differences

    def make_comparison_image(size, differences):
        img = Image.new("L", size, color=255)
        for pos, d1, d2, delta in differences:
 95         img.putpixel(pos, 255-delta)
        return img

    def isEqual(fname1, fname2, threshold=0):
        diff = visual_diff(fname1, fname2, threshold, stop_on_diff=True)
100     if len(diff) == 0:
            return True
        return False

    class Statistics(object):
105     def __init__(self, fname1, fname2, differences=None, name=""):
            self.fname1 = fname1
            self.fname2 = fname2
            if differences is None:
                differences = visual_diff(fname1, fname2)
110         self.differences = differences
            self.name = name

            img1 = Image.open(fname1)
            self.width, self.height = img1.size
115
            self._comparison_image = None
            self.comparison_image_fname = None
            self.calculate()

120     def calculate(self):
            diff = self.differences

            total_delta = 0
            max_delta = 0
125         for pos, d1, d2, delta in diff:
                total_delta += delta
                max_delta = max(max_delta, delta)
            self.total_delta = total_delta
            self.max_delta = max_delta
130         self.mean = mean = total_delta / float(self.width * self.height)

            stdev = 0
            for pos, d1, d2, delta in diff:
                stdev += pow(delta-mean, 2)
135         stdev /= float(self.width * self.height)
            self.stdev = stdev

        def _get_size(self):
            return self.width, self.height
140     size = property(_get_size)

        def _get_number_of_differences(self):
```

```
            return len(self.differences)
        number_of_differences = property(_get_number_of_differences)

        def _get_comparison_image(self):
            if self._comparison_image is None:
                self._comparison_image = make_comparison_image(self.size, self.differences)
            return self._comparison_image
        comparison_image = property(_get_comparison_image)

        def save_comparison_image(self, fname):
            self.comparison_image.save(fname)
            self.comparison_image_fname = fname

        def __str__(self):
            return "<Statistics diff:%s total_delta:%s max_delta:%s mean:%.4f stdev:%.4f>" % (
                len(self.differences), self.total_delta, self.max_delta, self.mean, self.stdev)

    def statistics(fname1, fname2, threshold=0):
        diff = visual_diff(fname1, fname2)
        stats = Statistics(fname1, fname2, diff)

        print "Differences:", len(stats.differences)
        print "Total delta:", stats.total_delta
        print "Max delta:", stats.max_delta
        print "Mean:", stats.mean
        print "Stdev:", stats.stdev

        stats.comparison_image.save('cmp.png')

    def test_vdiff(self):
        #fname1 = 'vdiff-tests/001-added-square/original.png'
        #fname2 = 'vdiff-tests/001-added-square/bluesquare.png'

        #fname1 = 'vdiff-tests/002-antialiased-text/preview.png'
        #fname2 = 'vdiff-tests/002-antialiased-text/photoshop.png'

        #fname1 = 'vdiff-tests/003-movement/original.png'
        #fname2 = 'vdiff-tests/003-movement/moved.png'

        #fname1 = 'vdiff-tests/004-color/original.png'
        #fname2 = 'vdiff-tests/004-color/darker.png'

        #fname1 = 'vdiff-tests/005-antialiased-text/none.png'
        #fname2 = 'vdiff-tests/005-antialiased-text/smooth.png'

        #fname1 = 'vdiff-tests/006-totally-different/ant.png'
        #fname2 = 'vdiff-tests/006-totally-different/people.png'

        fname1 = 'vdiff-tests/007-black-white/black.png'
        fname2 = 'vdiff-tests/007-black-white/white.png'

        statistics(fname1, fname2)

    def usage():
        print """vdiff -- visually compare images
    Usage: vdiff <image1> <image2> [threshold]"""

    if __name__=='__main__':
        import sys
        if len(sys.argv) < 3:
            usage()
        else:
            fname1 = sys.argv[1]
            fname2 = sys.argv[2]
```

```
        try:
            threshold = int(sys.argv[3])
        except:
            threshold = 0
        statistics(fname1, fname2, threshold)
```