```
# graphics.py
"""Simple object oriented graphics library

The library is designed to make it very easy for novice programmers to
experiment with computer graphics in an object oriented fashion. It is
written by John Zelle for use with the book "Python Programming: An
Introduction to Computer Science" (Franklin, Beedle & Associates).

LICENSE: This is open-source software released under the terms of the
GPL (http://www.gnu.org/licenses/gpl.html).

PLATFORMS: The package is a wrapper around Tkinter and should run on
any platform where Tkinter is available.

INSTALLATION: Put this file somewhere where Python can see it.

OVERVIEW: There are two kinds of objects in the library. The GraphWin
class implements a window where drawing can be done and various
GraphicsObjects are provided that can be drawn into a GraphWin. As a
simple example, here is a complete program to draw a circle of radius
10 centered in a 100x100 window:

--------------------------------------------------------------------
from graphics import *

def main():
    win = GraphWin("My Circle", 100, 100)
    c = Circle(Point(50,50), 10)
    c.draw(win)
    win.getMouse() # Pause to view result
    win.close()    # Close window when done

main()
--------------------------------------------------------------------
GraphWin objects support coordinate transformation through the
setCoords method and pointer-based input through getMouse.

The library provides the following graphical objects:
    Point
    Line
    Circle
    Oval
    Rectangle
    Polygon
    Text
    Entry (for text-based input)
    Image

Various attributes of graphical objects can be set such as
outline-color, fill-color and line-width. Graphical objects also
support moving and hiding for animation effects.

The library also provides a very simple class for pixel-based image
manipulation, Pixmap. A pixmap can be loaded from a file and displayed
using an Image object. Both getPixel and setPixel methods are provided
for manipulating the image.

DOCUMENTATION: For complete documentation, see Chapter 4 of "Python
Programming: An Introduction to Computer Science" by John Zelle,
published by Franklin, Beedle & Associates.  Also see
http://mcsp.wartburg.edu/zelle/python for a quick reference"""

# Version 4.2 5/26/2011
#     * Modified Image to allow multiple undraws like other GraphicsObjects
# Version 4.1 12/29/2009
#     * Merged Pixmap and Image class. Old Pixmap removed, use Image.
# Version 4.0.1 10/08/2009
#     * Modified the autoflush on GraphWin to default to True
#     * Autoflush check on close, setBackground
#     * Fixed getMouse to flush pending clicks at entry
# Version 4.0 08/2009
#     * Reverted to non-threaded version. The advantages (robustness,
#         efficiency, ability to use with other Tk code, etc.) outweigh
#         the disadvantage that interactive use with IDLE is slightly more
#         cumbersome.
#     * Modified to run in either Python 2.x or 3.x (same file).
#     * Added Image.getPixmap()
#     * Added update() -- stand alone function to cause any pending
#           graphics changes to display.
#
# Version 3.4 10/16/07
#     Fixed GraphicsError to avoid "exploded" error messages.
# Version 3.3 8/8/06
#     Added checkMouse method to GraphWin
# Version 3.2.3
#     Fixed error in Polygon init spotted by Andrew Harrington
```

```
#     Fixed improper threading in Image constructor
# Version 3.2.2 5/30/05
#     Cleaned up handling of exceptions in Tk thread. The graphics package
#     now raises an exception if attempt is made to communicate with
#     a dead Tk thread.
# Version 3.2.1 5/22/05
#     Added shutdown function for tk thread to eliminate race-condition
#        error "chatter" when main thread terminates
#     Renamed various private globals with _
# Version 3.2 5/4/05
#     Added Pixmap object for simple image manipulation.
# Version 3.1 4/13/05
#     Improved the Tk thread communication so that most Tk calls
#        do not have to wait for synchonization with the Tk thread.
#        (see _tkCall and _tkExec)
# Version 3.0 12/30/04
#     Implemented Tk event loop in separate thread. Should now work
#        interactively with IDLE. Undocumented autoflush feature is
#        no longer necessary. Its default is now False (off). It may
#        be removed in a future version.
#     Better handling of errors regarding operations on windows that
#       have been closed.
#     Addition of an isClosed method to GraphWindow class.

# Version 2.2 8/26/04
#     Fixed cloning bug reported by Joseph Oldham.
#     Now implements deep copy of config info.
# Version 2.1 1/15/04
#     Added autoflush option to GraphWin. When True (default) updates on
#        the window are done after each action. This makes some graphics
#        intensive programs sluggish. Turning off autoflush causes updates
#        to happen during idle periods or when flush is called.
# Version 2.0
#     Updated Documentation
#     Made Polygon accept a list of Points in constructor
#     Made all drawing functions call TK update for easier animations
#          and to make the overall package work better with
#          Python 2.3 and IDLE 1.0 under Windows (still some issues).
#     Removed vestigial turtle graphics.
#     Added ability to configure font for Entry objects (analogous to Text)
#     Added setTextColor for Text as an alias of setFill
#     Changed to class-style exceptions
#     Fixed cloning of Text objects

# Version 1.6
#     Fixed Entry so StringVar uses _root as master, solves weird
#            interaction with shell in Idle
#     Fixed bug in setCoords. X and Y coordinates can increase in
#           "non-intuitive" direction.
#     Tweaked wm_protocol so window is not resizable and kill box closes.

# Version 1.5
#     Fixed bug in Entry. Can now define entry before creating a
#     GraphWin. All GraphWins are now toplevel windows and share
#     a fixed root (called _root).

# Version 1.4
#     Fixed Garbage collection of Tkinter images bug.
#     Added ability to set text atttributes.
#     Added Entry boxes.

import time, os, sys

try:  # import as appropriate for 2.x vs. 3.x
   import tkinter as tk
except:
   import Tkinter as tk


##########################################################################
# Module Exceptions

class GraphicsError(Exception):
    """Generic error class for graphics module exceptions."""
    pass

OBJ_ALREADY_DRAWN = "Object currently drawn"
UNSUPPORTED_METHOD = "Object doesn't support operation"
BAD_OPTION = "Illegal option value"
DEAD_THREAD = "Graphics thread quit unexpectedly"

_root = tk.Tk()
_root.withdraw()

def update():
    _root.update()
```

```
############################################################################
# Graphics classes start here

class GraphWin(tk.Canvas):

    """A GraphWin is a toplevel window for displaying graphics."""

    def __init__(self, title="Graphics Window",
                 width=200, height=200, autoflush=True):
        master = tk.Toplevel(_root)
        master.protocol("WM_DELETE_WINDOW", self.close)
        tk.Canvas.__init__(self, master, width=width, height=height)
        self.master.title(title)
        self.pack()
        master.resizable(0,0)
        self.foreground = "black"
        self.items = []
        self.mouseX = None
        self.mouseY = None
        self.bind("<Button-1>", self._onClick)
        self.height = height
        self.width = width
        self.autoflush = autoflush
        self._mouseCallback = None
        self.trans = None
        self.closed = False
        master.lift()
        if autoflush: _root.update()

    def __checkOpen(self):
        if self.closed:
            raise GraphicsError("window is closed")

    def setBackground(self, color):
        """Set background color of the window"""
        self.__checkOpen()
        self.config(bg=color)
        self.__autoflush()

    def setCoords(self, x1, y1, x2, y2):
        """Set coordinates of window to run from (x1,y1) in the
        lower-left corner to (x2,y2) in the upper-right corner."""
        self.trans = Transform(self.width, self.height, x1, y1, x2, y2)

    def close(self):
        """Close the window"""

        if self.closed: return
        self.closed = True
        self.master.destroy()
        self.__autoflush()


    def isClosed(self):
        return self.closed


    def isOpen(self):
        return not self.closed


    def __autoflush(self):
        if self.autoflush:
            _root.update()


    def plot(self, x, y, color="black"):
        """Set pixel (x,y) to the given color"""
        self.__checkOpen()
        xs,ys = self.toScreen(x,y)
        self.create_line(xs,ys,xs+1,ys, fill=color)
        self.__autoflush()

    def plotPixel(self, x, y, color="black"):
        """Set pixel raw (independent of window coordinates) pixel
        (x,y) to color"""
        self.__checkOpen()
        self.create_line(x,y,x+1,y, fill=color)
        self.__autoflush()

    def flush(self):
        """Update drawing to the window"""
        self.__checkOpen()
        self.update_idletasks()

    def getMouse(self):
        """Wait for mouse click and return Point object representing
```

```
        the click"""
        self.update()      # flush any prior clicks
        self.mouseX = None
        self.mouseY = None
        while self.mouseX == None or self.mouseY == None:
            self.update()
            if self.isClosed(): raise GraphicsError("getMouse in closed window")
            time.sleep(.1) # give up thread
        x,y = self.toWorld(self.mouseX, self.mouseY)
        self.mouseX = None
        self.mouseY = None
        return Point(x,y)

    def checkMouse(self):
        """Return last mouse click or None if mouse has
        not been clicked since last call"""
        if self.isClosed():
            raise GraphicsError("checkMouse in closed window")
        self.update()
        if self.mouseX != None and self.mouseY != None:
            x,y = self.toWorld(self.mouseX, self.mouseY)
            self.mouseX = None
            self.mouseY = None
            return Point(x,y)
        else:
            return None

    def getHeight(self):
        """Return the height of the window"""
        return self.height

    def getWidth(self):
        """Return the width of the window"""
        return self.width

    def toScreen(self, x, y):
        trans = self.trans
        if trans:
            return self.trans.screen(x,y)
        else:
            return x,y

    def toWorld(self, x, y):
        trans = self.trans
        if trans:
            return self.trans.world(x,y)
        else:
            return x,y

    def setMouseHandler(self, func):
        self._mouseCallback = func

    def _onClick(self, e):
        self.mouseX = e.x
        self.mouseY = e.y
        if self._mouseCallback:
            self._mouseCallback(Point(e.x, e.y))

class Transform:

    """Internal class for 2-D coordinate transformations"""

    def __init__(self, w, h, xlow, ylow, xhigh, yhigh):
        # w, h are width and height of window
        # (xlow,ylow) coordinates of lower-left [raw (0,h-1)]
        # (xhigh,yhigh) coordinates of upper-right [raw (w-1,0)]
        xspan = (xhigh-xlow)
        yspan = (yhigh-ylow)
        self.xbase = xlow
        self.ybase = yhigh
        self.xscale = xspan/float(w-1)
        self.yscale = yspan/float(h-1)

    def screen(self,x,y):
        # Returns x,y in screen (actually window) coordinates
        xs = (x-self.xbase) / self.xscale
        ys = (self.ybase-y) / self.yscale
        return int(xs+0.5),int(ys+0.5)

    def world(self,xs,ys):
        # Returns xs,ys in world coordinates
        x = xs*self.xscale + self.xbase
        y = self.ybase - ys*self.yscale
        return x,y


# Default values for various item configuration options. Only a subset of
```

```
#   keys may be present in the configuration dictionary for a given item
DEFAULT_CONFIG = {"fill":"",
      "outline":"black",
      "width":"1",
      "arrow":"none",
      "text":"",
      "justify":"center",
                  "font": ("helvetica", 12, "normal")}

class GraphicsObject:

    """Generic base class for all of the drawable objects"""
    # A subclass of GraphicsObject should override _draw and
    #   and _move methods.

    def __init__(self, options):
        # options is a list of strings indicating which options are
        # legal for this object.

        # When an object is drawn, canvas is set to the GraphWin(canvas)
        #    object where it is drawn and id is the TK identifier of the
        #    drawn shape.
        self.canvas = None
        self.id = None

        # config is the dictionary of configuration options for the widget.
        config = {}
        for option in options:
            config[option] = DEFAULT_CONFIG[option]
        self.config = config

    def setFill(self, color):
        """Set interior color to color"""
        self._reconfig("fill", color)

    def setOutline(self, color):
        """Set outline color to color"""
        self._reconfig("outline", color)

    def setWidth(self, width):
        """Set line weight to width"""
        self._reconfig("width", width)

    def draw(self, graphwin):

        """Draw the object in graphwin, which should be a GraphWin
        object.  A GraphicsObject may only be drawn into one
        window. Raises an error if attempt made to draw an object that
        is already visible."""

        if self.canvas and not self.canvas.isClosed(): raise GraphicsError(OBJ_ALREADY_DRAWN)
        if graphwin.isClosed(): raise GraphicsError("Can't draw to closed window")
        self.canvas = graphwin
        self.id = self._draw(graphwin, self.config)
        if graphwin.autoflush:
            _root.update()


    def undraw(self):

        """Undraw the object (i.e. hide it). Returns silently if the
        object is not currently drawn."""

        if not self.canvas: return
        if not self.canvas.isClosed():
            self.canvas.delete(self.id)
            if self.canvas.autoflush:
                _root.update()
        self.canvas = None
        self.id = None


    def move(self, dx, dy):

        """move object dx units in x direction and dy units in y
        direction"""

        self._move(dx,dy)
        canvas = self.canvas
        if canvas and not canvas.isClosed():
            trans = canvas.trans
            if trans:
                x = dx/ trans.xscale
                y = -dy / trans.yscale
            else:
                x = dx
                y = dy
```

```
            self.canvas.move(self.id, x, y)
            if canvas.autoflush:
                _root.update()

    def _reconfig(self, option, setting):
        # Internal method for changing configuration of the object
        # Raises an error if the option does not exist in the config
        #    dictionary for this object
        if option not in self.config:
            raise GraphicsError(UNSUPPORTED_METHOD)
        options = self.config
        options[option] = setting
        if self.canvas and not self.canvas.isClosed():
            self.canvas.itemconfig(self.id, options)
            if self.canvas.autoflush:
                _root.update()


    def _draw(self, canvas, options):
        """draws appropriate figure on canvas with options provided
        Returns Tk id of item drawn"""
        pass # must override in subclass


    def _move(self, dx, dy):
        """updates internal state of object to move it dx,dy units"""
        pass # must override in subclass


class Point(GraphicsObject):
    def __init__(self, x, y):
        GraphicsObject.__init__(self, ["outline", "fill"])
        self.setFill = self.setOutline
        self.x = x
        self.y = y

    def _draw(self, canvas, options):
        x,y = canvas.toScreen(self.x,self.y)
        return canvas.create_rectangle(x,y,x+1,y+1,options)

    def _move(self, dx, dy):
        self.x = self.x + dx
        self.y = self.y + dy

    def clone(self):
        other = Point(self.x,self.y)
        other.config = self.config.copy()
        return other

    def getX(self): return self.x
    def getY(self): return self.y

class _BBox(GraphicsObject):
    # Internal base class for objects represented by bounding box
    # (opposite corners) Line segment is a degenerate case.

    def __init__(self, p1, p2, options=["outline","width","fill"]):
        GraphicsObject.__init__(self, options)
        self.p1 = p1.clone()
        self.p2 = p2.clone()

    def _move(self, dx, dy):
        self.p1.x = self.p1.x + dx
        self.p1.y = self.p1.y + dy
        self.p2.x = self.p2.x + dx
        self.p2.y = self.p2.y  + dy

    def getP1(self): return self.p1.clone()

    def getP2(self): return self.p2.clone()

    def getCenter(self):
        p1 = self.p1
        p2 = self.p2
        return Point((p1.x+p2.x)/2.0, (p1.y+p2.y)/2.0)

class Rectangle(_BBox):

    def __init__(self, p1, p2):
        _BBox.__init__(self, p1, p2)

    def _draw(self, canvas, options):
        p1 = self.p1
        p2 = self.p2
        x1,y1 = canvas.toScreen(p1.x,p1.y)
        x2,y2 = canvas.toScreen(p2.x,p2.y)
        return canvas.create_rectangle(x1,y1,x2,y2,options)
```

```
    def clone(self):
        other = Rectangle(self.p1, self.p2)
        other.config = self.config.copy()
        return other

class Oval(_BBox):

    def __init__(self, p1, p2):
        _BBox.__init__(self, p1, p2)

    def clone(self):
        other = Oval(self.p1, self.p2)
        other.config = self.config.copy()
        return other

    def _draw(self, canvas, options):
        p1 = self.p1
        p2 = self.p2
        x1,y1 = canvas.toScreen(p1.x,p1.y)
        x2,y2 = canvas.toScreen(p2.x,p2.y)
        return canvas.create_oval(x1,y1,x2,y2,options)

class Circle(Oval):

    def __init__(self, center, radius):
        p1 = Point(center.x-radius, center.y-radius)
        p2 = Point(center.x+radius, center.y+radius)
        Oval.__init__(self, p1, p2)
        self.radius = radius

    def clone(self):
        other = Circle(self.getCenter(), self.radius)
        other.config = self.config.copy()
        return other

    def getRadius(self):
        return self.radius

class Line(_BBox):

    def __init__(self, p1, p2):
        _BBox.__init__(self, p1, p2, ["arrow","fill","width"])
        self.setFill(DEFAULT_CONFIG['outline'])
        self.setOutline = self.setFill

    def clone(self):
        other = Line(self.p1, self.p2)
        other.config = self.config.copy()
        return other

    def _draw(self, canvas, options):
        p1 = self.p1
        p2 = self.p2
        x1,y1 = canvas.toScreen(p1.x,p1.y)
        x2,y2 = canvas.toScreen(p2.x,p2.y)
        return canvas.create_line(x1,y1,x2,y2,options)

    def setArrow(self, option):
        if not option in ["first","last","both","none"]:
            raise GraphicsError(BAD_OPTION)
        self._reconfig("arrow", option)


class Polygon(GraphicsObject):

    def __init__(self, *points):
        # if points passed as a list, extract it
        if len(points) == 1 and type(points[0]) == type([]):
            points = points[0]
        self.points = list(map(Point.clone, points))
        GraphicsObject.__init__(self, ["outline", "width", "fill"])

    def clone(self):
        other = Polygon(*self.points)
        other.config = self.config.copy()
        return other

    def getPoints(self):
        return list(map(Point.clone, self.points))

    def _move(self, dx, dy):
        for p in self.points:
            p.move(dx,dy)

    def _draw(self, canvas, options):
        args = [canvas]
```

```
        for p in self.points:
            x,y = canvas.toScreen(p.x,p.y)
            args.append(x)
            args.append(y)
        args.append(options)
        return GraphWin.create_polygon(*args)

class Text(GraphicsObject):

    def __init__(self, p, text):
        GraphicsObject.__init__(self, ["justify","fill","text","font"])
        self.setText(text)
        self.anchor = p.clone()
        self.setFill(DEFAULT_CONFIG['outline'])
        self.setOutline = self.setFill

    def _draw(self, canvas, options):
        p = self.anchor
        x,y = canvas.toScreen(p.x,p.y)
        return canvas.create_text(x,y,options)

    def _move(self, dx, dy):
        self.anchor.move(dx,dy)

    def clone(self):
        other = Text(self.anchor, self.config['text'])
        other.config = self.config.copy()
        return other

    def setText(self,text):
        self._reconfig("text", text)

    def getText(self):
        return self.config["text"]

    def getAnchor(self):
        return self.anchor.clone()

    def setFace(self, face):
        if face in ['helvetica','arial','courier','times roman']:
            f,s,b = self.config['font']
            self._reconfig("font",(face,s,b))
        else:
            raise GraphicsError(BAD_OPTION)

    def setSize(self, size):
        if 5 <= size <= 36:
            f,s,b = self.config['font']
            self._reconfig("font", (f,size,b))
        else:
            raise GraphicsError(BAD_OPTION)

    def setStyle(self, style):
        if style in ['bold','normal','italic', 'bold italic']:
            f,s,b = self.config['font']
            self._reconfig("font", (f,s,style))
        else:
            raise GraphicsError(BAD_OPTION)

    def setTextColor(self, color):
        self.setFill(color)


class Entry(GraphicsObject):

    def __init__(self, p, width):
        GraphicsObject.__init__(self, [])
        self.anchor = p.clone()
        #print self.anchor
        self.width = width
        self.text = tk.StringVar(_root)
        self.text.set("")
        self.fill = "gray"
        self.color = "black"
        self.font = DEFAULT_CONFIG['font']
        self.entry = None

    def _draw(self, canvas, options):
        p = self.anchor
        x,y = canvas.toScreen(p.x,p.y)
        frm = tk.Frame(canvas.master)
        self.entry = tk.Entry(frm,
                              width=self.width,
                              textvariable=self.text,
                              bg = self.fill,
                              fg = self.color,
                              font=self.font)
```

```
        self.entry.pack()
        #self.setFill(self.fill)
        return canvas.create_window(x,y,window=frm)

    def getText(self):
        return self.text.get()

    def _move(self, dx, dy):
        self.anchor.move(dx,dy)

    def getAnchor(self):
        return self.anchor.clone()

    def clone(self):
        other = Entry(self.anchor, self.width)
        other.config = self.config.copy()
        other.text = tk.StringVar()
        other.text.set(self.text.get())
        other.fill = self.fill
        return other

    def setText(self, t):
        self.text.set(t)


    def setFill(self, color):
        self.fill = color
        if self.entry:
            self.entry.config(bg=color)


    def _setFontComponent(self, which, value):
        font = list(self.font)
        font[which] = value
        self.font = tuple(font)
        if self.entry:
            self.entry.config(font=self.font)


    def setFace(self, face):
        if face in ['helvetica','arial','courier','times roman']:
            self._setFontComponent(0, face)
        else:
            raise GraphicsError(BAD_OPTION)

    def setSize(self, size):
        if 5 <= size <= 36:
            self._setFontComponent(1,size)
        else:
            raise GraphicsError(BAD_OPTION)

    def setStyle(self, style):
        if style in ['bold','normal','italic', 'bold italic']:
            self._setFontComponent(2,style)
        else:
            raise GraphicsError(BAD_OPTION)

    def setTextColor(self, color):
        self.color=color
        if self.entry:
            self.entry.config(fg=color)


class Image(GraphicsObject):

    idCount = 0
    imageCache = {} # tk photoimages go here to avoid GC while drawn

    def __init__(self, p, *pixmap):
        GraphicsObject.__init__(self, [])
        self.anchor = p.clone()
        self.imageId = Image.idCount
        Image.idCount = Image.idCount + 1
        if len(pixmap) == 1: # file name provided
            self.img = tk.PhotoImage(file=pixmap[0], master=_root)
        else: # width and height provided
            width, height = pixmap
            self.img = tk.PhotoImage(master=_root, width=width, height=height)

    def _draw(self, canvas, options):
        p = self.anchor
        x,y = canvas.toScreen(p.x,p.y)
        self.imageCache[self.imageId] = self.img # save a reference
        return canvas.create_image(x,y,image=self.img)

    def _move(self, dx, dy):
        self.anchor.move(dx,dy)
```

```
    def undraw(self):
        try:
            del self.imageCache[self.imageId]  # allow gc of tk photoimage
        except KeyError:
            pass
        GraphicsObject.undraw(self)

    def getAnchor(self):
        return self.anchor.clone()

    def clone(self):
        other = Image(Point(0,0), 0, 0)
        other.img = self.img.copy()
        other.anchor = self.anchor.clone()
        other.config = self.config.copy()
        return other

    def getWidth(self):
        """Returns the width of the image in pixels"""
        return self.img.width()

    def getHeight(self):
        """Returns the height of the image in pixels"""
        return self.img.height()

    def getPixel(self, x, y):
        """Returns a list [r,g,b] with the RGB color values for pixel (x,y)
        r,g,b are in range(256)

        """

        value = self.img.get(x,y)
        if type(value) ==  type(0):
            return [value, value, value]
        else:
            return list(map(int, value.split()))

    def setPixel(self, x, y, color):
        """Sets pixel (x,y) to the given color

        """
        self.img.put("{" + color +"}", (x, y))


    def save(self, filename):
        """Saves the pixmap image to filename.
        The format for the save image is determined from the filname extension.

        """

        path, name = os.path.split(filename)
        ext = name.split(".")[-1]
        self.img.write( filename, format=ext)


def color_rgb(r,g,b):
    """r,g,b are intensities of red, green, and blue in range(256)
    Returns color specifier string for the resulting color"""
    return "#%02x%02x%02x" % (r,g,b)

def test():
    win = GraphWin()
    win.setCoords(0,0,10,10)
    t = Text(Point(5,5), "Centered Text")
    t.draw(win)
    p = Polygon(Point(1,1), Point(5,3), Point(2,7))
    p.draw(win)
    e = Entry(Point(5,6), 10)
    e.draw(win)
    win.getMouse()
    p.setFill("red")
    p.setOutline("blue")
    p.setWidth(2)
    s = ""
    for pt in p.getPoints():
        s = s + "(%0.1f,%0.1f) " % (pt.getX(), pt.getY())
    t.setText(e.getText())
    e.setFill("green")
    e.setText("Spam!")
    e.move(2,0)
    win.getMouse()
    p.move(2,3)
    s = ""
    for pt in p.getPoints():
        s = s + "(%0.1f,%0.1f) " % (pt.getX(), pt.getY())
    t.setText(s)
```

```
    win.getMouse()
    p.undraw()
    e.undraw()
    t.setStyle("bold")
    win.getMouse()
    t.setStyle("normal")
    win.getMouse()
    t.setStyle("italic")
    win.getMouse()
    t.setStyle("bold italic")
    win.getMouse()
    t.setSize(14)
    win.getMouse()
    t.setFace("arial")
    t.setSize(20)
    win.getMouse()
    win.close()

if __name__ == "__main__":
    test()
```