```
#
# turtle.py: a Tkinter based turtle graphics module for Python
# Version 1.1b - 4. 5. 2009
#
# Copyright (C) 2006 - 2010  Gregor Lingl
# email: glingl@aon.at
#
# This software is provided 'as-is', without any express or implied
# warranty.  In no event will the authors be held liable for any damages
# arising from the use of this software.
#
# Permission is granted to anyone to use this software for any purpose,
# including commercial applications, and to alter it and redistribute it
# freely, subject to the following restrictions:
#
# 1. The origin of this software must not be misrepresented; you must not
#    claim that you wrote the original software. If you use this software
#    in a product, an acknowledgment in the product documentation would be
#    appreciated but is not required.
# 2. Altered source versions must be plainly marked as such, and must not be
#    misrepresented as being the original software.
# 3. This notice may not be removed or altered from any source distribution.


"""
Turtle graphics is a popular way for introducing programming to
kids. It was part of the original Logo programming language developed
by Wally Feurzig and Seymour Papert in 1966.

Imagine a robotic turtle starting at (0, 0) in the x-y plane. After an ``import turtle``, give it
the command turtle.forward(15), and it moves (on-screen!) 15 pixels in
the direction it is facing, drawing a line as it moves. Give it the
command turtle.right(25), and it rotates in-place 25 degrees clockwise.

By combining together these and similar commands, intricate shapes and
pictures can easily be drawn.

----- turtle.py

This module is an extended reimplementation of turtle.py from the
Python standard distribution up to Python 2.5. (See: http://www.python.org)

It tries to keep the merits of turtle.py and to be (nearly) 100%
compatible with it. This means in the first place to enable the
learning programmer to use all the commands, classes and methods
interactively when using the module from within IDLE run with
the -n switch.

Roughly it has the following features added:

- Better animation of the turtle movements, especially of turning the
  turtle. So the turtles can more easily be used as a visual feedback
  instrument by the (beginning) programmer.

- Different turtle shapes, gif-images as turtle shapes, user defined
  and user controllable turtle shapes, among them compound
  (multicolored) shapes. Turtle shapes can be stretched and tilted, which
  makes turtles very versatile geometrical objects.

```

```
- Fine control over turtle movement and screen updates via delay(),
  and enhanced tracer() and speed() methods.

- Aliases for the most commonly used commands, like fd for forward etc.,
  following the early Logo traditions. This reduces the boring work of
  typing long sequences of commands, which often occur in a natural way
  when kids try to program fancy pictures on their first encounter with
  turtle graphics.

- Turtles now have an undo()-method with configurable undo-buffer.

- Some simple commands/methods for creating event driven programs
  (mouse-, key-, timer-events). Especially useful for programming games.

- A scrollable Canvas class. The default scrollable Canvas can be
  extended interactively as needed while playing around with the turtle(s).

- A TurtleScreen class with methods controlling background color or
  background image, window and canvas size and other properties of the
  TurtleScreen.

- There is a method, setworldcoordinates(), to install a user defined
  coordinate-system for the TurtleScreen.

- The implementation uses a 2-vector class named Vec2D, derived from tuple.
  This class is public, so it can be imported by the application programmer,
  which makes certain types of computations very natural and compact.

- Appearance of the TurtleScreen and the Turtles at startup/import can be
  configured by means of a turtle.cfg configuration file.
  The default configuration mimics the appearance of the old turtle module.

- If configured appropriately the module reads in docstrings from a docstring
  dictionary in some different language, supplied separately  and replaces
  the English ones by those read in. There is a utility function
  write_docstringdict() to write a dictionary with the original (English)
  docstrings to disc, so it can serve as a template for translations.

Behind the scenes there are some features included with possible
extensions in mind. These will be commented and documented elsewhere.

"""

_ver = "turtle 1.1b- - for Python 3.1   -  4. 5. 2009"

# print(_ver)

import tkinter as TK
import types
import math
import time
import inspect

from os.path import isfile, split, join
from copy import deepcopy
from tkinter import simpledialog

_tg_classes = ['ScrolledCanvas', 'TurtleScreen', 'Screen',
               'RawTurtle', 'Turtle', 'RawPen', 'Pen', 'Shape', 'Vec2D']
_tg_screen_functions = ['addshape', 'bgcolor', 'bgpic', 'bye',
        'clearscreen', 'colormode', 'delay', 'exitonclick', 'getcanvas',
        'getshapes', 'listen', 'mainloop', 'mode', 'numinput',
        'onkey', 'onkeypress', 'onkeyrelease', 'onscreenclick', 'ontimer',
        'register_shape', 'resetscreen', 'screensize', 'setup',
        'setworldcoordinates', 'textinput', 'title', 'tracer', 'turtles', 'update',
        'window_height', 'window_width']
_tg_turtle_functions = ['back', 'backward', 'begin_fill', 'begin_poly', 'bk',
        'circle', 'clear', 'clearstamp', 'clearstamps', 'clone', 'color',
        'degrees', 'distance', 'dot', 'down', 'end_fill', 'end_poly', 'fd',
        'fillcolor', 'filling', 'forward', 'get_poly', 'getpen', 'getscreen', 'get_shapepoly',
        'getturtle', 'goto', 'heading', 'hideturtle', 'home', 'ht', 'isdown',
        'isvisible', 'left', 'lt', 'onclick', 'ondrag', 'onrelease', 'pd',
        'pen', 'pencolor', 'pendown', 'pensize', 'penup', 'pos', 'position',
        'pu', 'radians', 'right', 'reset', 'resizemode', 'rt',
        'seth', 'setheading', 'setpos', 'setposition', 'settiltangle',
        'setundobuffer', 'setx', 'sety', 'shape', 'shapesize', 'shapetransform', 'shearfactor', 'showturtle',
        'speed', 'st', 'stamp', 'tilt', 'tiltangle', 'towards',
        'turtlesize', 'undo', 'undobufferentries', 'up', 'width',
        'write', 'xcor', 'ycor']
_tg_utilities = ['write_docstringdict', 'done']

__all__ = (_tg_classes + _tg_screen_functions + _tg_turtle_functions +
           _tg_utilities) # + _math_functions)

_alias_list = ['addshape', 'backward', 'bk', 'fd', 'ht', 'lt', 'pd', 'pos',
               'pu', 'rt', 'seth', 'setpos', 'setposition', 'st',
               'turtlesize', 'up', 'width']

_CFG = {"width" : 0.5,               # Screen
        "height" : 0.75,
        "canvwidth" : 400,
        "canvheight": 300,
        "leftright": None,
        "topbottom": None,
        "mode": "standard",          # TurtleScreen
        "colormode": 1.0,
        "delay": 10,
        "undobuffersize": 1000,      # RawTurtle
        "shape": "classic",
        "pencolor" : "black",
        "fillcolor" : "black",
        "resizemode" : "noresize",
        "visible" : True,
        "language": "english",        # docstrings
        "exampleturtle": "turtle",
        "examplescreen": "screen",
        "title": "Python Turtle Graphics",
        "using_IDLE": False
       }

def config_dict(filename):
    """Convert content of config-file into dictionary."""
    with open(filename, "r") as f:
        cfglines = f.readlines()
    cfgdict = {}
    for line in cfglines:
        line = line.strip()
        if not line or line.startswith("#"):
            continue
        try:
            key, value = line.split("=")
        except:
            print("Bad line in config-file %s:\n%s" % (filename,line))
            continue
        key = key.strip()
        value = value.strip()
        if value in ["True", "False", "None", "''", '""']:
            value = eval(value)
        else:
            try:
                if "." in value:
                    value = float(value)
                else:
                    value = int(value)
            except:
                pass # value need not be converted
        cfgdict[key] = value
    return cfgdict

def readconfig(cfgdict):
    """Read config-files, change configuration-dict accordingly.

    If there is a turtle.cfg file in the current working directory,
    read it from there. If this contains an importconfig-value,
    say 'myway', construct filename turtle_mayway.cfg else use
    turtle.cfg and read it from the import-directory, where
    turtle.py is located.
    Update configuration dictionary first according to config-file,
    in the import directory, then according to config-file in the
    current working directory.
    If no config-file is found, the default configuration is used.
    """
    default_cfg = "turtle.cfg"
    cfgdict1 = {}
    cfgdict2 = {}
    if isfile(default_cfg):
        cfgdict1 = config_dict(default_cfg)
    if "importconfig" in cfgdict1:
        default_cfg = "turtle_%s.cfg" % cfgdict1["importconfig"]
    try:
        head, tail = split(__file__)
        cfg_file2 = join(head, default_cfg)
    except:
        cfg_file2 = ""
    if isfile(cfg_file2):
        cfgdict2 = config_dict(cfg_file2)
    _CFG.update(cfgdict2)
    _CFG.update(cfgdict1)

try:
    readconfig(_CFG)
except:
    print ("No configfile read, reason unknown")


class Vec2D(tuple):
    """A 2 dimensional vector class, used as a helper class
    for implementing turtle graphics.
    May be useful for turtle graphics programs also.
    Derived from tuple, so a vector is a tuple!

    Provides (for a, b vectors, k number):
       a+b vector addition
       a-b vector subtraction
       a*b inner product
       k*a and a*k multiplication with scalar
       |a| absolute value of a
       a.rotate(angle) rotation
    """
    def __new__(cls, x, y):
        return tuple.__new__(cls, (x, y))
    def __add__(self, other):
        return Vec2D(self[0]+other[0], self[1]+other[1])
    def __mul__(self, other):
        if isinstance(other, Vec2D):
            return self[0]*other[0]+self[1]*other[1]
        return Vec2D(self[0]*other, self[1]*other)
    def __rmul__(self, other):
        if isinstance(other, int) or isinstance(other, float):
            return Vec2D(self[0]*other, self[1]*other)
    def __sub__(self, other):
        return Vec2D(self[0]-other[0], self[1]-other[1])
    def __neg__(self):
        return Vec2D(-self[0], -self[1])
    def __abs__(self):
        return (self[0]**2 + self[1]**2)**0.5
    def rotate(self, angle):
        """rotate self counterclockwise by angle
        """
        perp = Vec2D(-self[1], self[0])
        angle = angle * math.pi / 180.0
        c, s = math.cos(angle), math.sin(angle)
        return Vec2D(self[0]*c+perp[0]*s, self[1]*c+perp[1]*s)
    def __getnewargs__(self):
        return (self[0], self[1])
    def __repr__(self):
        return "(%.2f,%.2f)" % self


##############################################################################
### From here up to line    : Tkinter - Interface for turtle.py            ###
### May be replaced by an interface to some different graphics toolkit     ###
##############################################################################

## helper functions for Scrolled Canvas, to forward Canvas-methods
## to ScrolledCanvas class

def __methodDict(cls, _dict):
    """helper function for Scrolled Canvas"""
    baseList = list(cls.__bases__)
    baseList.reverse()
    for _super in baseList:
        __methodDict(_super, _dict)
    for key, value in cls.__dict__.items():
        if type(value) == types.FunctionType:
            _dict[key] = value

def __methods(cls):
    """helper function for Scrolled Canvas"""
    _dict = {}
    __methodDict(cls, _dict)
    return _dict.keys()

__stringBody = (
    'def %(method)s(self, *args, **kw): return ' +
    'self.%(attribute)s.%(method)s(*args, **kw)')

def __forwardmethods(fromClass, toClass, toPart, exclude = ()):
    ### MANY CHANGES ###
    _dict_1 = {}
    __methodDict(toClass, _dict_1)
    _dict = {}
    mfc = __methods(fromClass)
    for ex in _dict_1.keys():
        if ex[:1] == '_' or ex[-1:] == '_' or ex in exclude or ex in mfc:
            pass
        else:
            _dict[ex] = _dict_1[ex]

    for method, func in _dict.items():
        d = {'method': method, 'func': func}
        if isinstance(toPart, str):
            execString = \
                __stringBody % {'method' : method, 'attribute' : toPart}
        exec(execString, d)
        setattr(fromClass, method, d[method])   ### NEWU!


class ScrolledCanvas(TK.Frame):
    """Modeled after the scrolled canvas class from Grayons's Tkinter book.

    Used as the default canvas, which pops up automatically when
    using turtle graphics functions or the Turtle class.
    """
    def __init__(self, master, width=500, height=350,
                                          canvwidth=600, canvheight=500):
        TK.Frame.__init__(self, master, width=width, height=height)
        self._rootwindow = self.winfo_toplevel()
        self.width, self.height = width, height
        self.canvwidth, self.canvheight = canvwidth, canvheight
        self.bg = "white"
        self._canvas = TK.Canvas(master, width=width, height=height,
                                 bg=self.bg, relief=TK.SUNKEN, borderwidth=2)
        self.hscroll = TK.Scrollbar(master, command=self._canvas.xview,
                                    orient=TK.HORIZONTAL)
        self.vscroll = TK.Scrollbar(master, command=self._canvas.yview)
        self._canvas.configure(xscrollcommand=self.hscroll.set,
                               yscrollcommand=self.vscroll.set)
        self.rowconfigure(0, weight=1, minsize=0)
        self.columnconfigure(0, weight=1, minsize=0)
        self._canvas.grid(padx=1, in_ = self, pady=1, row=0,
                column=0, rowspan=1, columnspan=1, sticky='news')
        self.vscroll.grid(padx=1, in_ = self, pady=1, row=0,
                column=1, rowspan=1, columnspan=1, sticky='news')
        self.hscroll.grid(padx=1, in_ = self, pady=1, row=1,
                column=0, rowspan=1, columnspan=1, sticky='news')
        self.reset()
        self._rootwindow.bind('<Configure>', self.onResize)

    def reset(self, canvwidth=None, canvheight=None, bg = None):
        """Adjust canvas and scrollbars according to given canvas size."""
        if canvwidth:
            self.canvwidth = canvwidth
        if canvheight:
            self.canvheight = canvheight
        if bg:
            self.bg = bg
        self._canvas.config(bg=bg,
                        scrollregion=(-self.canvwidth//2, -self.canvheight//2,
                                       self.canvwidth//2, self.canvheight//2))
        self._canvas.xview_moveto(0.5*(self.canvwidth - self.width + 30) /
                                                               self.canvwidth)
        self._canvas.yview_moveto(0.5*(self.canvheight- self.height + 30) /
                                                              self.canvheight)
        self.adjustScrolls()


    def adjustScrolls(self):
        """ Adjust scrollbars according to window- and canvas-size.
        """
        cwidth = self._canvas.winfo_width()
        cheight = self._canvas.winfo_height()
        self._canvas.xview_moveto(0.5*(self.canvwidth-cwidth)/self.canvwidth)
        self._canvas.yview_moveto(0.5*(self.canvheight-cheight)/self.canvheight)
        if cwidth < self.canvwidth or cheight < self.canvheight:
            self.hscroll.grid(padx=1, in_ = self, pady=1, row=1,
                              column=0, rowspan=1, columnspan=1, sticky='news')
            self.vscroll.grid(padx=1, in_ = self, pady=1, row=0,
                              column=1, rowspan=1, columnspan=1, sticky='news')
        else:
            self.hscroll.grid_forget()
            self.vscroll.grid_forget()

    def onResize(self, event):
        """self-explanatory"""
        self.adjustScrolls()

    def bbox(self, *args):
        """ 'forward' method, which canvas itself has inherited...
        """
        return self._canvas.bbox(*args)

    def cget(self, *args, **kwargs):
        """ 'forward' method, which canvas itself has inherited...
        """
        return self._canvas.cget(*args, **kwargs)

    def config(self, *args, **kwargs):
        """ 'forward' method, which canvas itself has inherited...
        """
        self._canvas.config(*args, **kwargs)

    def bind(self, *args, **kwargs):
        """ 'forward' method, which canvas itself has inherited...
        """
        self._canvas.bind(*args, **kwargs)

    def unbind(self, *args, **kwargs):
        """ 'forward' method, which canvas itself has inherited...
        """
        self._canvas.unbind(*args, **kwargs)

    def focus_force(self):
        """ 'forward' method, which canvas itself has inherited...
        """
        self._canvas.focus_force()

__forwardmethods(ScrolledCanvas, TK.Canvas, '_canvas')


class _Root(TK.Tk):
    """Root class for Screen based on Tkinter."""
    def __init__(self):
        TK.Tk.__init__(self)

    def setupcanvas(self, width, height, cwidth, cheight):
        self._canvas = ScrolledCanvas(self, width, height, cwidth, cheight)
        self._canvas.pack(expand=1, fill="both")

    def _getcanvas(self):
        return self._canvas

    def set_geometry(self, width, height, startx, starty):
        self.geometry("%dx%d%+d%+d"%(width, height, startx, starty))

    def ondestroy(self, destroy):
        self.wm_protocol("WM_DELETE_WINDOW", destroy)

    def win_width(self):
        return self.winfo_screenwidth()

    def win_height(self):
        return self.winfo_screenheight()

Canvas = TK.Canvas


class TurtleScreenBase(object):
    """Provide the basic graphics functionality.
       Interface between Tkinter and turtle.py.

       To port turtle.py to some different graphics toolkit
       a corresponding TurtleScreenBase class has to be implemented.
    """

    @staticmethod
    def _blankimage():
        """return a blank image object
        """
        img = TK.PhotoImage(width=1, height=1)
        img.blank()
        return img

    @staticmethod
    def _image(filename):
        """return an image object containing the
        imagedata from a gif-file named filename.
        """
```

```python
        return TK.PhotoImage(file=filename)

    def __init__(self, cv):
        self.cv = cv
        if isinstance(cv, ScrolledCanvas):
            w = self.cv.canvwidth
            h = self.cv.canvheight
        else:  # expected: ordinary TK.Canvas
            w = int(self.cv.cget("width"))
            h = int(self.cv.cget("height"))
            self.cv.config(scrollregion = (-w//2, -h//2, w//2, h//2 ))
        self.canvwidth = w
        self.canvheight = h
        self.xscale = self.yscale = 1.0

    def _createpoly(self):
        """Create an invisible polygon item on canvas self.cv)
        """
        return self.cv.create_polygon((0, 0, 0, 0, 0, 0), fill="", outline="")

    def _drawpoly(self, polyitem, coordlist, fill=None,
                  outline=None, width=None, top=False):
        """Configure polygonitem polyitem according to provided
        arguments:
        coordlist is sequence of coordinates
        fill is filling color
        outline is outline color
        top is a boolean value, which specifies if polyitem
        will be put on top of the canvas' displaylist so it
        will not be covered by other items.
        """
        cl = []
        for x, y in coordlist:
            cl.append(x * self.xscale)
            cl.append(-y * self.yscale)
        self.cv.coords(polyitem, *cl)
        if fill is not None:
            self.cv.itemconfigure(polyitem, fill=fill)
        if outline is not None:
            self.cv.itemconfigure(polyitem, outline=outline)
        if width is not None:
            self.cv.itemconfigure(polyitem, width=width)
        if top:
            self.cv.tag_raise(polyitem)

    def _createline(self):
        """Create an invisible line item on canvas self.cv)
        """
        return self.cv.create_line(0, 0, 0, 0, fill="", width=2,
                                   capstyle = TK.ROUND)

    def _drawline(self, lineitem, coordlist=None,
                  fill=None, width=None, top=False):
        """Configure lineitem according to provided arguments:
        coordlist is sequence of coordinates
        fill is drawing color
        width is width of drawn line.
        top is a boolean value, which specifies if polyitem
        will be put on top of the canvas' displaylist so it
        will not be covered by other items.
        """
        if coordlist is not None:
            cl = []
            for x, y in coordlist:
                cl.append(x * self.xscale)
                cl.append(-y * self.yscale)
            self.cv.coords(lineitem, *cl)
        if fill is not None:
            self.cv.itemconfigure(lineitem, fill=fill)
        if width is not None:
            self.cv.itemconfigure(lineitem, width=width)
        if top:
            self.cv.tag_raise(lineitem)

    def _delete(self, item):
        """Delete graphics item from canvas.
        If item is"all" delete all graphics items.
        """
        self.cv.delete(item)

    def _update(self):
        """Redraw graphics items on canvas
        """
        self.cv.update()

    def _delay(self, delay):
        """Delay subsequent canvas actions for delay ms."""
        self.cv.after(delay)

    def _iscolorstring(self, color):
        """Check if the string color is a legal Tkinter color string.
        """
        try:
            rgb = self.cv.winfo_rgb(color)
            ok = True
        except TK.TclError:
            ok = False
        return ok

    def _bgcolor(self, color=None):
        """Set canvas' backgroundcolor if color is not None,
        else return backgroundcolor."""
        if color is not None:
            self.cv.config(bg = color)
            self._update()
        else:
            return self.cv.cget("bg")

    def _write(self, pos, txt, align, font, pencolor):
        """Write txt at pos in canvas with specified font
        and color.
        Return text item and x-coord of right bottom corner
        of text's bounding box."""
        x, y = pos
        x = x * self.xscale
        y = y * self.yscale
        anchor = {"left":"sw", "center":"s", "right":"se" }
        item = self.cv.create_text(x-1, -y, text = txt, anchor = anchor[align],
                                        fill = pencolor, font = font)
        x0, y0, x1, y1 = self.cv.bbox(item)
        self.cv.update()
        return item, x1-1

##    def _dot(self, pos, size, color):
##        """may be implemented for some other graphics toolkit"""

    def _onclick(self, item, fun, num=1, add=None):
        """Bind fun to mouse-click event on turtle.
        fun must be a function with two arguments, the coordinates
        of the clicked point on the canvas.
        num, the number of the mouse-button defaults to 1
        """
        if fun is None:
            self.cv.tag_unbind(item, "<Button-%s>" % num)
        else:
            def eventfun(event):
                x, y = (self.cv.canvasx(event.x)/self.xscale,
                        -self.cv.canvasy(event.y)/self.yscale)
                fun(x, y)
            self.cv.tag_bind(item, "<Button-%s>" % num, eventfun, add)

    def _onrelease(self, item, fun, num=1, add=None):
        """Bind fun to mouse-button-release event on turtle.
        fun must be a function with two arguments, the coordinates
        of the point on the canvas where mouse button is released.
        num, the number of the mouse-button defaults to 1

        If a turtle is clicked, first _onclick-event will be performed,
        then _onscreensclick-event.
        """
        if fun is None:
            self.cv.tag_unbind(item, "<Button%s-ButtonRelease>" % num)
        else:
            def eventfun(event):
                x, y = (self.cv.canvasx(event.x)/self.xscale,
                        -self.cv.canvasy(event.y)/self.yscale)
                fun(x, y)
            self.cv.tag_bind(item, "<Button%s-ButtonRelease>" % num,
                             eventfun, add)

    def _ondrag(self, item, fun, num=1, add=None):
        """Bind fun to mouse-move-event (with pressed mouse button) on turtle.
        fun must be a function with two arguments, the coordinates of the
        actual mouse position on the canvas.
        num, the number of the mouse-button defaults to 1

        Every sequence of mouse-move-events on a turtle is preceded by a
        mouse-click event on that turtle.
        """
        if fun is None:
            self.cv.tag_unbind(item, "<Button%s-Motion>" % num)
        else:
            def eventfun(event):
                try:
                    x, y = (self.cv.canvasx(event.x)/self.xscale,
                           -self.cv.canvasy(event.y)/self.yscale)
                    fun(x, y)
                except:
                    pass
            self.cv.tag_bind(item, "<Button%s-Motion>" % num, eventfun, add)

    def _onscreenclick(self, fun, num=1, add=None):
        """Bind fun to mouse-click event on canvas.
        fun must be a function with two arguments, the coordinates
        of the clicked point on the canvas.
        num, the number of the mouse-button defaults to 1

        If a turtle is clicked, first _onclick-event will be performed,
        then _onscreensclick-event.
        """
        if fun is None:
            self.cv.unbind("<Button-%s>" % num)
        else:
            def eventfun(event):
                x, y = (self.cv.canvasx(event.x)/self.xscale,
                        -self.cv.canvasy(event.y)/self.yscale)
                fun(x, y)
            self.cv.bind("<Button-%s>" % num, eventfun, add)

    def _onkeyrelease(self, fun, key):
        """Bind fun to key-release event of key.
        Canvas must have focus. See method listen
        """
        if fun is None:
            self.cv.unbind("<KeyRelease-%s>" % key, None)
        else:
            def eventfun(event):
                fun()
            self.cv.bind("<KeyRelease-%s>" % key, eventfun)

    def _onkeypress(self, fun, key=None):
        """If key is given, bind fun to key-press event of key.
        Otherwise bind fun to any key-press.
        Canvas must have focus. See method listen.
        """
        if fun is None:
            if key is None:
                self.cv.unbind("<KeyPress>", None)
            else:
                self.cv.unbind("<KeyPress-%s>" % key, None)
        else:
            def eventfun(event):
                fun()
            if key is None:
                self.cv.bind("<KeyPress>", eventfun)
            else:
                self.cv.bind("<KeyPress-%s>" % key, eventfun)

    def _listen(self):
        """Set focus on canvas (in order to collect key-events)
        """
        self.cv.focus_force()

    def _ontimer(self, fun, t):
        """Install a timer, which calls fun after t milliseconds.
        """
        if t == 0:
            self.cv.after_idle(fun)
        else:
            self.cv.after(t, fun)

    def _createimage(self, image):
        """Create and return image item on canvas.
        """
        return self.cv.create_image(0, 0, image=image)

    def _drawimage(self, item, pos, image):
        """Configure image item as to draw image object
        at position (x,y) on canvas)
        """
        x, y = pos
        self.cv.coords(item, (x * self.xscale, -y * self.yscale))
        self.cv.itemconfig(item, image=image)

    def _setbgpic(self, item, image):
        """Configure image item as to draw image object
        at center of canvas. Set item to the first item
        in the displaylist, so it will be drawn below
        any other item ."""
        self.cv.itemconfig(item, image=image)
        self.cv.tag_lower(item)

    def _type(self, item):
        """Return 'line' or 'polygon' or 'image' depending on
        type of item.
        """
        return self.cv.type(item)

    def _pointlist(self, item):
        """returns list of coordinate-pairs of points of item
        Example (for insiders):
        >>> from turtle import *
        >>> getscreen()._pointlist(getturtle().turtle._item)
        [(0.0, 9.9999999999999982), (0.0, -9.9999999999999982),
        (9.9999999999999982, 0.0)]
        >>> """
        cl = self.cv.coords(item)
        pl = [(cl[i], -cl[i+1]) for i in range(0, len(cl), 2)]
        return  pl

    def _setscrollregion(self, srx1, sry1, srx2, sry2):
        self.cv.config(scrollregion=(srx1, sry1, srx2, sry2))

    def _rescale(self, xscalefactor, yscalefactor):
        items = self.cv.find_all()
        for item in items:
            coordinates = list(self.cv.coords(item))
            newcoordlist = []
            while coordinates:
                x, y = coordinates[:2]
                newcoordlist.append(x * xscalefactor)
                newcoordlist.append(y * yscalefactor)
                coordinates = coordinates[2:]
            self.cv.coords(item, *newcoordlist)

    def _resize(self, canvwidth=None, canvheight=None, bg=None):
        """Resize the canvas the turtles are drawing on. Does
        not alter the drawing window.
        """
        # needs amendment
        if not isinstance(self.cv, ScrolledCanvas):
            return self.canvwidth, self.canvheight
        if canvwidth is canvheight is bg is None:
            return self.cv.canvwidth, self.cv.canvheight
        if canvwidth is not None:
            self.canvwidth = canvwidth
        if canvheight is not None:
            self.canvheight = canvheight
        self.cv.reset(canvwidth, canvheight, bg)

    def _window_size(self):
        """ Return the width and height of the turtle window.
        """
        width = self.cv.winfo_width()
        if width <= 1:  # the window isn't managed by a geometry manager
            width = self.cv['width']
        height = self.cv.winfo_height()
        if height <= 1: # the window isn't managed by a geometry manager
            height = self.cv['height']
        return width, height

    def mainloop(self):
        """Starts event loop - calling Tkinter's mainloop function.

        No argument.

        Must be last statement in a turtle graphics program.
        Must NOT be used if a script is run from within IDLE in -n mode
        (No subprocess) - for interactive use of turtle graphics.

        Example (for a TurtleScreen instance named screen):
        >>> screen.mainloop()

        """
        TK.mainloop()

    def textinput(self, title, prompt):
        """Pop up a dialog window for input of a string.

        Arguments: title is the title of the dialog window,
        prompt is a text mostly describing what information to input.

        Return the string input
        If the dialog is canceled, return None.

        Example (for a TurtleScreen instance named screen):
        >>> screen.textinput("NIM", "Name of first player:")

        """
        return simpledialog.askstring(title, prompt)

    def numinput(self, title, prompt, default=None, minval=None, maxval=None):
        """Pop up a dialog window for input of a number.

        Arguments: title is the title of the dialog window,
        prompt is a text mostly describing what numerical information to input.
        default: default value
        minval: minimum value for imput
        maxval: maximum value for input

        The number input must be in the range minval .. maxval if these are
        given. If not, a hint is issued and the dialog remains open for
        correction. Return the number input.
        If the dialog is canceled,  return None.

        Example (for a TurtleScreen instance named screen):
        >>> screen.numinput("Poker", "Your stakes:", 1000, minval=10, maxval=10000)

        """
        return simpledialog.askfloat(title, prompt, initialvalue=default,
                                     minvalue=minval, maxvalue=maxval)


##############################################################################
###                  End of Tkinter - interface                            ###
##############################################################################


class Terminator (Exception):
    """Will be raised in TurtleScreen.update, if _RUNNING becomes False.

    This stops execution of a turtle graphics script.
    Main purpose: use in the Demo-Viewer turtle.Demo.py.
    """
    pass


class TurtleGraphicsError(Exception):
    """Some TurtleGraphics Error
    """


class Shape(object):
    """Data structure modeling shapes.

    attribute _type is one of "polygon", "image", "compound"
    attribute _data is - depending on _type a poygon-tuple,
    an image or a list constructed using the addcomponent method.
    """
    def __init__(self, type_, data=None):
        self._type = type_
        if type_ == "polygon":
            if isinstance(data, list):
                data = tuple(data)
        elif type_ == "image":
            if isinstance(data, str):
                if data.lower().endswith(".gif") and isfile(data):
                    data = TurtleScreen._image(data)
                # else data assumed to be Photoimage
        elif type_ == "compound":
            data = []
        else:
            raise TurtleGraphicsError("There is no shape type %s" % type_)
        self._data = data

    def addcomponent(self, poly, fill, outline=None):
        """Add component to a shape of type compound.

        Arguments: poly is a polygon, i. e. a tuple of number pairs.
        fill is the fillcolor of the component,
        outline is the outline color of the component.

        call (for a Shapeobject namend s):
        --   s.addcomponent(((0,0), (10,10), (-10,10)), "red", "blue")

        Example:
        >>> poly = ((0,0),(10,-5),(0,10),(-10,-5))
        >>> s = Shape("compound")
        >>> s.addcomponent(poly, "red", "blue")
        >>> # .. add more components and then use register_shape()
        """
        if self._type != "compound":
            raise TurtleGraphicsError("Cannot add component to %s Shape"
                                                                % self._type)
        if outline is None:
            outline = fill
        self._data.append([poly, fill, outline])


class Tbuffer(object):
    """Ring buffer used as undobuffer for RawTurtle objects."""
    def __init__(self, bufsize=10):
        self.bufsize = bufsize
        self.buffer = [[None]] * bufsize
        self.ptr = -1
        self.cumulate = False
    def reset(self, bufsize=None):
        if bufsize is None:
            for i in range(self.bufsize):
                self.buffer[i] = [None]
        else:
            self.bufsize = bufsize
            self.buffer = [[None]] * bufsize
        self.ptr = -1
    def push(self, item):
        if self.bufsize > 0:
            if not self.cumulate:
                self.ptr = (self.ptr + 1) % self.bufsize
                self.buffer[self.ptr] = item
            else:
                self.buffer[self.ptr].append(item)
    def pop(self):
        if self.bufsize > 0:
            item = self.buffer[self.ptr]
            if item is None:
                return None
            else:
                self.buffer[self.ptr] = [None]
                self.ptr = (self.ptr - 1) % self.bufsize
                return (item)
    def nr_of_items(self):
        return self.bufsize - self.buffer.count([None])
    def __repr__(self):
        return str(self.buffer) + " " + str(self.ptr)



class TurtleScreen(TurtleScreenBase):
    """Provides screen oriented methods like setbg etc.

    Only relies upon the methods of TurtleScreenBase and NOT
    upon components of the underlying graphics toolkit -
    which is Tkinter in this case.
    """
    _RUNNING = True

    def __init__(self, cv, mode=_CFG["mode"],
                 colormode=_CFG["colormode"], delay=_CFG["delay"]):
        self._shapes = {
                   "arrow" : Shape("polygon", ((-10,0), (10,0), (0,10))),
                  "turtle" : Shape("polygon", ((0,16), (-2,14), (-1,10), (-4,7),
                              (-7,9), (-9,8), (-6,5), (-7,1), (-5,-3), (-8,-6),
                              (-6,-8), (-4,-5), (0,-7), (4,-5), (6,-8), (8,-6),
                              (5,-3), (7,1), (6,5), (9,8), (7,9), (4,7), (1,10),
                              (2,14))),
                  "circle" : Shape("polygon", ((10,0), (9.51,3.09), (8.09,5.88),
                              (5.88,8.09), (3.09,9.51), (0,10), (-3.09,9.51),
                              (-5.88,8.09), (-8.09,5.88), (-9.51,3.09), (-10,0),
                              (-9.51,-3.09), (-8.09,-5.88), (-5.88,-8.09),
                              (-3.09,-9.51), (-0.00,-10.00), (3.09,-9.51),
                              (5.88,-8.09), (8.09,-5.88), (9.51,-3.09))),
                  "square" : Shape("polygon", ((10,-10), (10,10), (-10,10),
                              (-10,-10))),
                "triangle" : Shape("polygon", ((10,-5.77), (0,11.55),
                              (-10,-5.77))),
                  "classic": Shape("polygon", ((0,0),(-5,-9),(0,-7),(5,-9))),
                   "blank" : Shape("image", self._blankimage())
                  }

        self._bgpics = {"nopic" : ""}

        TurtleScreenBase.__init__(self, cv)
        self._mode = mode
        self._delayvalue = delay
        self._colormode = _CFG["colormode"]
        self._keys = []
        self.clear()

    def clear(self):
        """Delete all drawings and all turtles from the TurtleScreen.

        No argument.

        Reset empty TurtleScreen to its initial state: white background,
        no backgroundimage, no eventbindings and tracing on.

        Example (for a TurtleScreen instance named screen):
        >>> screen.clear()

        Note: this method is not available as function.
        """
        self._delayvalue = _CFG["delay"]
        self._colormode = _CFG["colormode"]
        self._delete("all")
        self._bgpic = self._createimage("")
        self._bgpicname = "nopic"
        self._tracing = 1
        self._updatecounter = 0
        self._turtles = []
        self.bgcolor("white")
        for btn in 1, 2, 3:
            self.onclick(None, btn)
        self.onkeypress(None)
        for key in self._keys[:]:
            self.onkey(None, key)
            self.onkeypress(None, key)
        Turtle._pen = None

    def mode(self, mode=None):
        """Set turtle-mode ('standard', 'logo' or 'world') and perform reset.

        Optional argument:
        mode -- on of the strings 'standard', 'logo' or 'world'

        Mode 'standard' is compatible with turtle.py.
        Mode 'logo' is compatible with most Logo-Turtle-Graphics.
        Mode 'world' uses userdefined 'worldcoordinates'. *Attention*: in
        this mode angles appear distorted if x/y unit-ratio doesn't equal 1.
        If mode is not given, return the current mode.

             Mode      Initial turtle heading     positive angles
         ------------|-------------------------|-------------------
          'standard'    to the right (east)       counterclockwise
            'logo'        upward    (north)         clockwise

        Examples:
        >>> mode('logo')   # resets turtle heading to north
        >>> mode()
        'logo'
        """
        if mode is None:
            return self._mode
        mode = mode.lower()
        if mode not in ["standard", "logo", "world"]:
            raise TurtleGraphicsError("No turtle-graphics-mode %s" % mode)
        self._mode = mode
        if mode in ["standard", "logo"]:
            self._setscrollregion(-self.canvwidth//2, -self.canvheight//2,
                                       self.canvwidth//2, self.canvheight//2)
            self.xscale = self.yscale = 1.0
        self.reset()

    def setworldcoordinates(self, llx, lly, urx, ury):
        """Set up a user defined coordinate-system.

        Arguments:
        llx -- a number, x-coordinate of lower left corner of canvas
        lly -- a number, y-coordinate of lower left corner of canvas
        urx -- a number, x-coordinate of upper right corner of canvas
        ury -- a number, y-coordinate of upper right corner of canvas

        Set up user coodinat-system and switch to mode 'world' if necessary.
        This performs a screen.reset. If mode 'world' is already active,
        all drawings are redrawn according to the new coordinates.

        But ATTENTION: in user-defined coordinatesystems angles may appear
        distorted. (see Screen.mode())

        Example (for a TurtleScreen instance named screen):
        >>> screen.setworldcoordinates(-10,-0.5,50,1.5)
        >>> for _ in range(36):
        ...     left(10)
        ...     forward(0.5)
        """
        if self.mode() != "world":
            self.mode("world")
        xspan = float(urx - llx)
        yspan = float(ury - lly)
        wx, wy = self._window_size()
        self.screensize(wx-20, wy-20)
        oldxscale, oldyscale = self.xscale, self.yscale
        self.xscale = self.canvwidth / xspan
        self.yscale = self.canvheight / yspan
        srx1 = llx * self.xscale
        sry1 = -ury * self.yscale
        srx2 = self.canvwidth + srx1
        sry2 = self.canvheight + sry1
        self._setscrollregion(srx1, sry1, srx2, sry2)
        self._rescale(self.xscale/oldxscale, self.yscale/oldyscale)
        self.update()

    def register_shape(self, name, shape=None):
        """Adds a turtle shape to TurtleScreen's shapelist.

        Arguments:
        (1) name is the name of a gif-file and shape is None.
            Installs the corresponding image shape.
            !! Image-shapes DO NOT rotate when turning the turtle,
            !! so they do not display the heading of the turtle!
        (2) name is an arbitrary string and shape is a tuple
            of pairs of coordinates. Installs the corresponding
            polygon shape
        (3) name is an arbitrary string and shape is a
            (compound) Shape object. Installs the corresponding
            compound shape.
        To use a shape, you have to issue the command shape(shapename).

        call: register_shape("turtle.gif")
        --or: register_shape("tri", ((0,0), (10,10), (-10,10)))

        Example (for a TurtleScreen instance named screen):
        >>> screen.register_shape("triangle", ((5,-3),(0,5),(-5,-3)))

        """
        if shape is None:
            # image
            if name.lower().endswith(".gif"):
                shape = Shape("image", self._image(name))
            else:
                raise TurtleGraphicsError("Bad arguments for register_shape.\n"
                                          + "Use  help(register_shape)" )
        elif isinstance(shape, tuple):
            shape = Shape("polygon", shape)
        ## else shape assumed to be Shape-instance
        self._shapes[name] = shape

    def _colorstr(self, color):
        """Return color string corresponding to args.

        Argument may be a string or a tuple of three
        numbers corresponding to actual colormode,
        i.e. in the range 0<=n<=colormode.

        If the argument doesn't represent a color,
        an error is raised.
        """
        if len(color) == 1:
            color = color[0]
        if isinstance(color, str):
            if self._iscolorstring(color) or color == "":
                return color
            else:
                raise TurtleGraphicsError("bad color string: %s" % str(color))
        try:
            r, g, b = color
        except:
            raise TurtleGraphicsError("bad color arguments: %s" % str(color))
        if self._colormode == 1.0:
            r, g, b = [round(255.0*x) for x in (r, g, b)]
        if not ((0 <= r <= 255) and (0 <= g <= 255) and (0 <= b <= 255)):
            raise TurtleGraphicsError("bad color sequence: %s" % str(color))
        return "#%02x%02x%02x" % (r, g, b)

    def _color(self, cstr):
        if not cstr.startswith("#"):
            return cstr
        if len(cstr) == 7:
            cl = [int(cstr[i:i+2], 16) for i in (1, 3, 5)]
        elif len(cstr) == 4:
            cl = [16*int(cstr[h], 16) for h in cstr[1:]]
        else:
            raise TurtleGraphicsError("bad colorstring: %s" % cstr)
        return tuple([c * self._colormode/255 for c in cl])

    def colormode(self, cmode=None):
        """Return the colormode or set it to 1.0 or 255.

        Optional argument:
        cmode -- one of the values 1.0 or 255

        r, g, b values of colortriples have to be in range 0..cmode.

        Example (for a TurtleScreen instance named screen):
        >>> screen.colormode()
        1.0
        >>> screen.colormode(255)
        >>> pencolor(240,160,80)
        """
        if cmode is None:
            return self._colormode
        if cmode == 1.0:
            self._colormode = float(cmode)
        elif cmode == 255:
            self._colormode = int(cmode)

    def reset(self):
        """Reset all Turtles on the Screen to their initial state.

        No argument.

        Example (for a TurtleScreen instance named screen):
        >>> screen.reset()
        """
        for turtle in self._turtles:
            turtle._setmode(self._mode)
            turtle.reset()

    def turtles(self):
        """Return the list of turtles on the screen.

        Example (for a TurtleScreen instance named screen):
        >>> screen.turtles()
        [<turtle.Turtle object at 0x00E11FB0>]
        """
        return self._turtles

    def bgcolor(self, *args):
        """Set or return backgroundcolor of the TurtleScreen.

        Arguments (if given): a color string or three numbers
        in the range 0..colormode or a 3-tuple of such numbers.

        Example (for a TurtleScreen instance named screen):
        >>> screen.bgcolor("orange")
        >>> screen.bgcolor()
        'orange'
        >>> screen.bgcolor(0.5,0,0.5)
        >>> screen.bgcolor()
        '#800080'
        """
        if args:
            color = self._colorstr(args)
        else:
            color = None
        color = self._bgcolor(color)
        if color is not None:
            color = self._color(color)
        return color

    def tracer(self, n=None, delay=None):
        """Turns turtle animation on/off and set delay for update drawings.

        Optional arguments:
        n -- nonnegative  integer
        delay -- nonnegative  integer

        If n is given, only each n-th regular screen update is really performed.
        (Can be used to accelerate the drawing of complex graphics.)
        Second arguments sets delay value (see RawTurtle.delay())

        Example (for a TurtleScreen instance named screen):
        >>> screen.tracer(8, 25)
        >>> dist = 2
        >>> for i in range(200):
        ...     fd(dist)
        ...     rt(90)
        ...     dist += 2
        """
        if n is None:
            return self._tracing
```

```python
        self._tracing = int(n)
        self._updatecounter = 0
        if delay is not None:
            self._delayvalue = int(delay)
        if self._tracing:
            self.update()

    def delay(self, delay=None):
        """ Return or set the drawing delay in milliseconds.

        Optional argument:
        delay -- positive integer

        Example (for a TurtleScreen instance named screen):
        >>> screen.delay(15)
        >>> screen.delay()
        15
        """
        if delay is None:
            return self._delayvalue
        self._delayvalue = int(delay)

    def _incrementudc(self):
        """Increment update counter."""
        if not TurtleScreen._RUNNING:
            TurtleScreen._RUNNNING = True
            raise Terminator
        if self._tracing > 0:
            self._updatecounter += 1
            self._updatecounter %= self._tracing

    def update(self):
        """Perform a TurtleScreen update.
        """
        tracing = self._tracing
        self._tracing = True
        for t in self.turtles():
            t._update_data()
            t._drawturtle()
        self._tracing = tracing
        self._update()

    def window_width(self):
        """ Return the width of the turtle window.

        Example (for a TurtleScreen instance named screen):
        >>> screen.window_width()
        640
        """
        return self._window_size()[0]

    def window_height(self):
        """ Return the height of the turtle window.

        Example (for a TurtleScreen instance named screen):
        >>> screen.window_height()
        480
        """
        return self._window_size()[1]

    def getcanvas(self):
        """Return the Canvas of this TurtleScreen.

        No argument.

        Example (for a Screen instance named screen):
        >>> cv = screen.getcanvas()
        >>> cv
        <turtle.ScrolledCanvas instance at 0x010742D8>
        """
        return self.cv

    def getshapes(self):
        """Return a list of names of all currently available turtle shapes.

        No argument.

        Example (for a TurtleScreen instance named screen):
        >>> screen.getshapes()
        ['arrow', 'blank', 'circle', ... , 'turtle']
        """
        return sorted(self._shapes.keys())

    def onclick(self, fun, btn=1, add=None):
        """Bind fun to mouse-click event on canvas.

        Arguments:
        fun -- a function with two arguments, the coordinates of the
               clicked point on the canvas.
        num -- the number of the mouse-button, defaults to 1

        Example (for a TurtleScreen instance named screen)

        >>> screen.onclick(goto)
        >>> # Subsequently clicking into the TurtleScreen will
        >>> # make the turtle move to the clicked point.
        >>> screen.onclick(None)
        """
        self._onscreenclick(fun, btn, add)

    def onkey(self, fun, key):
        """Bind fun to key-release event of key.

        Arguments:
        fun -- a function with no arguments
        key -- a string: key (e.g. "a") or key-symbol (e.g. "space")

        In order to be able to register key-events, TurtleScreen
        must have focus. (See method listen.)

        Example (for a TurtleScreen instance named screen):

        >>> def f():
        ...     fd(50)
        ...     lt(60)
        ...
        >>> screen.onkey(f, "Up")
        >>> screen.listen()

        Subsequently the turtle can be moved by repeatedly pressing
        the up-arrow key, consequently drawing a hexagon

        """
        if fun is None:
            if key in self._keys:
                self._keys.remove(key)
        elif key not in self._keys:
            self._keys.append(key)
        self._onkeyrelease(fun, key)

    def onkeypress(self, fun, key=None):
        """Bind fun to key-press event of key if key is given,
        or to any key-press-event if no key is given.

        Arguments:
        fun -- a function with no arguments
        key -- a string: key (e.g. "a") or key-symbol (e.g. "space")

        In order to be able to register key-events, TurtleScreen
        must have focus. (See method listen.)

        Example (for a TurtleScreen instance named screen
        and a Turtle instance named turtle):

        >>> def f():
        ...     fd(50)
        ...     lt(60)
        ...
        >>> screen.onkeypress(f, "Up")
        >>> screen.listen()

        Subsequently the turtle can be moved by repeatedly pressing
        the up-arrow key, or by keeping pressed the up-arrow key.
        consequently drawing a hexagon.
        """
        if fun is None:
            if key in self._keys:
                self._keys.remove(key)
        elif key is not None and key not in self._keys:
            self._keys.append(key)
        self._onkeypress(fun, key)

    def listen(self, xdummy=None, ydummy=None):
        """Set focus on TurtleScreen (in order to collect key-events)

        No arguments.
        Dummy arguments are provided in order
        to be able to pass listen to the onclick method.

        Example (for a TurtleScreen instance named screen):
        >>> screen.listen()
        """
        self._listen()

    def ontimer(self, fun, t=0):
        """Install a timer, which calls fun after t milliseconds.

        Arguments:
        fun -- a function with no arguments.
        t -- a number >= 0

        Example (for a TurtleScreen instance named screen):

        >>> running = True
        >>> def f():
        ...     if running:
        ...             fd(50)
        ...             lt(60)
        ...             screen.ontimer(f, 250)
        ...
        >>> f()   # makes the turtle marching around
        >>> running = False
        """
        self._ontimer(fun, t)

    def bgpic(self, picname=None):
        """Set background image or return name of current backgroundimage.

        Optional argument:
        picname -- a string, name of a gif-file or "nopic".

        If picname is a filename, set the corresponding image as background.
        If picname is "nopic", delete backgroundimage, if present.
        If picname is None, return the filename of the current backgroundimage.

        Example (for a TurtleScreen instance named screen):
        >>> screen.bgpic()
        'nopic'
        >>> screen.bgpic("landscape.gif")
        >>> screen.bgpic()
        'landscape.gif'
        """
        if picname is None:
            return self._bgpicname
        if picname not in self._bgpics:
            self._bgpics[picname] = self._image(picname)
        self._setbgpic(self._bgpic, self._bgpics[picname])
        self._bgpicname = picname

    def screensize(self, canvwidth=None, canvheight=None, bg=None):
        """Resize the canvas the turtles are drawing on.

        Optional arguments:
        canvwidth -- positive integer, new width of canvas in pixels
        canvheight --  positive integer, new height of canvas in pixels
        bg -- colorstring or color-tuple, new backgroundcolor
        If no arguments are given, return current (canvaswidth, canvasheight)

        Do not alter the drawing window. To observe hidden parts of
        the canvas use the scrollbars. (Can make visible those parts
        of a drawing, which were outside the canvas before!)

        Example (for a Turtle instance named turtle):
        >>> turtle.screensize(2000,1500)
        >>> # e.g. to search for an erroneously escaped turtle ;-)
        """
        return self._resize(canvwidth, canvheight, bg)

    onscreenclick = onclick
    resetscreen = reset
    clearscreen = clear
    addshape = register_shape
    onkeyrelease = onkey

class TNavigator(object):
    """Navigation part of the RawTurtle.
    Implements methods for turtle movement.
    """
    START_ORIENTATION = {
        "standard": Vec2D(1.0, 0.0),
        "world"   : Vec2D(1.0, 0.0),
        "logo"    : Vec2D(0.0, 1.0)  }
    DEFAULT_MODE = "standard"
    DEFAULT_ANGLEOFFSET = 0
    DEFAULT_ANGLEORIENT = 1

    def __init__(self, mode=DEFAULT_MODE):
        self._angleOffset = self.DEFAULT_ANGLEOFFSET
        self._angleOrient = self.DEFAULT_ANGLEORIENT
        self._mode = mode
        self.undobuffer = None
        self.degrees()
        self._mode = None
        self._setmode(mode)
        TNavigator.reset(self)

    def reset(self):
        """reset turtle to its initial values

        Will be overwritten by parent class
        """
        self._position = Vec2D(0.0, 0.0)
        self._orient =  TNavigator.START_ORIENTATION[self._mode]

    def _setmode(self, mode=None):
        """Set turtle-mode to 'standard', 'world' or 'logo'.
        """
        if mode is None:
            return self._mode
        if mode not in ["standard", "logo", "world"]:
            return
        self._mode = mode
        if mode in ["standard", "world"]:
            self._angleOffset = 0
            self._angleOrient = 1
        else: # mode == "logo":
            self._angleOffset = self._fullcircle/4.
            self._angleOrient = -1

    def _setDegreesPerAU(self, fullcircle):
        """Helper function for degrees() and radians()"""
        self._fullcircle = fullcircle
        self._degreesPerAU = 360/fullcircle
        if self._mode == "standard":
            self._angleOffset = 0
        else:
            self._angleOffset = fullcircle/4.

    def degrees(self, fullcircle=360.0):
        """ Set angle measurement units to degrees.

        Optional argument:
        fullcircle -  a number

        Set angle measurement units, i. e. set number
        of 'degrees' for a full circle. Dafault value is
        360 degrees.

        Example (for a Turtle instance named turtle):
        >>> turtle.left(90)
        >>> turtle.heading()
        90

        Change angle measurement unit to grad (also known as gon,
        grade, or gradian and equals 1/100-th of the right angle.)
        >>> turtle.degrees(400.0)
        >>> turtle.heading()
        100

        """
        self._setDegreesPerAU(fullcircle)

    def radians(self):
        """ Set the angle measurement units to radians.

        No arguments.

        Example (for a Turtle instance named turtle):
        >>> turtle.heading()
        90
        >>> turtle.radians()
        >>> turtle.heading()
        1.5707963267948966
        """
        self._setDegreesPerAU(2*math.pi)

    def _go(self, distance):
        """move turtle forward by specified distance"""
        ende = self._position + self._orient * distance
        self._goto(ende)

    def _rotate(self, angle):
        """Turn turtle counterclockwise by specified angle if angle > 0."""
        angle *= self._degreesPerAU
        self._orient = self._orient.rotate(angle)

    def _goto(self, end):
        """move turtle to position end."""
        self._position = end

    def forward(self, distance):
        """Move the turtle forward by the specified distance.

        Aliases: forward | fd

        Argument:
        distance -- a number (integer or float)

        Move the turtle forward by the specified distance, in the direction
        the turtle is headed.

        Example (for a Turtle instance named turtle):
        >>> turtle.position()
        (0.00, 0.00)
        >>> turtle.forward(25)
        >>> turtle.position()
        (25.00,0.00)
        >>> turtle.forward(-75)
        >>> turtle.position()
        (-50.00,0.00)
        """
        self._go(distance)

    def back(self, distance):
        """Move the turtle backward by distance.

        Aliases: back | backward | bk

        Argument:
        distance -- a number

        Move the turtle backward by distance ,opposite to the direction the
        turtle is headed. Do not change the turtle's heading.

        Example (for a Turtle instance named turtle):
        >>> turtle.position()
        (0.00, 0.00)
        >>> turtle.backward(30)
        >>> turtle.position()
        (-30.00, 0.00)
        """
        self._go(-distance)

    def right(self, angle):
        """Turn turtle right by angle units.

        Aliases: right | rt

        Argument:
        angle -- a number (integer or float)

        Turn turtle right by angle units. (Units are by default degrees,
        but can be set via the degrees() and radians() functions.)
        Angle orientation depends on mode. (See this.)

        Example (for a Turtle instance named turtle):
        >>> turtle.heading()
        22.0
        >>> turtle.right(45)
        >>> turtle.heading()
        337.0
        """
        self._rotate(-angle)

    def left(self, angle):
        """Turn turtle left by angle units.

        Aliases: left | lt

        Argument:
```

```
        angle -- a number (integer or float)

        Turn turtle left by angle units. (Units are by default degrees,
        but can be set via the degrees() and radians() functions.)
        Angle orientation depends on mode. (See this.)

        Example (for a Turtle instance named turtle):
        >>> turtle.heading()
        22.0
        >>> turtle.left(45)
        >>> turtle.heading()
        67.0
        """
        self._rotate(angle)

    def pos(self):
        """Return the turtle's current location (x,y), as a Vec2D-vector.

        Aliases: pos | position

        No arguments.

        Example (for a Turtle instance named turtle):
        >>> turtle.pos()
        (0.00, 240.00)
        """
        return self._position

    def xcor(self):
        """ Return the turtle's x coordinate.

        No arguments.

        Example (for a Turtle instance named turtle):
        >>> reset()
        >>> turtle.left(60)
        >>> turtle.forward(100)
        >>> print turtle.xcor()
        50.0
        """
        return self._position[0]

    def ycor(self):
        """ Return the turtle's y coordinate
        ---
        No arguments.

        Example (for a Turtle instance named turtle):
        >>> reset()
        >>> turtle.left(60)
        >>> turtle.forward(100)
        >>> print turtle.ycor()
        86.6025403784
        """
        return self._position[1]


    def goto(self, x, y=None):
        """Move turtle to an absolute position.

        Aliases: setpos | setposition | goto:

        Arguments:
        x -- a number      or     a pair/vector of numbers
        y -- a number             None

        call: goto(x, y)         # two coordinates
        --or: goto((x, y))       # a pair (tuple) of coordinates
        --or: goto(vec)          # e.g. as returned by pos()

        Move turtle to an absolute position. If the pen is down,
        a line will be drawn. The turtle's orientation does not change.

        Example (for a Turtle instance named turtle):
        >>> tp = turtle.pos()
        >>> tp
        (0.00, 0.00)
        >>> turtle.setpos(60,30)
        >>> turtle.pos()
        (60.00,30.00)
        >>> turtle.setpos((20,80))
        >>> turtle.pos()
        (20.00,80.00)
        >>> turtle.setpos(tp)
        >>> turtle.pos()
        (0.00,0.00)
        """
        if y is None:
            self._goto(Vec2D(*x))
        else:
            self._goto(Vec2D(x, y))

    def home(self):
        """Move turtle to the origin - coordinates (0,0).

        No arguments.

        Move turtle to the origin - coordinates (0,0) and set its
        heading to its start-orientation (which depends on mode).

        Example (for a Turtle instance named turtle):
        >>> turtle.home()
        """
        self.goto(0, 0)
        self.setheading(0)

    def setx(self, x):
        """Set the turtle's first coordinate to x

        Argument:
        x -- a number (integer or float)

        Set the turtle's first coordinate to x, leave second coordinate
        unchanged.

        Example (for a Turtle instance named turtle):
        >>> turtle.position()
        (0.00, 240.00)
        >>> turtle.setx(10)
        >>> turtle.position()
        (10.00, 240.00)
        """
        self._goto(Vec2D(x, self._position[1]))

    def sety(self, y):
        """Set the turtle's second coordinate to y

        Argument:
        y -- a number (integer or float)

        Set the turtle's first coordinate to x, second coordinate remains
        unchanged.

        Example (for a Turtle instance named turtle):
        >>> turtle.position()
        (0.00, 40.00)
        >>> turtle.sety(-10)
        >>> turtle.position()
        (0.00, -10.00)
        """
        self._goto(Vec2D(self._position[0], y))

    def distance(self, x, y=None):
        """Return the distance from the turtle to (x,y) in turtle step units.

        Arguments:
        x -- a number   or  a pair/vector of numbers   or   a turtle instance
        y -- a number       None                            None

        call: distance(x, y)         # two coordinates
        --or: distance((x, y))       # a pair (tuple) of coordinates
        --or: distance(vec)          # e.g. as returned by pos()
        --or: distance(mypen)        # where mypen is another turtle

        Example (for a Turtle instance named turtle):
        >>> turtle.pos()
        (0.00, 0.00)
        >>> turtle.distance(30,40)
        50.0
        >>> pen = Turtle()
        >>> pen.forward(77)
        >>> turtle.distance(pen)
        77.0
        """
        if y is not None:
            pos = Vec2D(x, y)
        if isinstance(x, Vec2D):
            pos = x
        elif isinstance(x, tuple):
            pos = Vec2D(*x)
        elif isinstance(x, TNavigator):
            pos = x._position
        return abs(pos - self._position)

    def towards(self, x, y=None):
        """Return the angle of the line from the turtle's position to (x, y).

        Arguments:
        x -- a number   or  a pair/vector of numbers   or   a turtle instance
        y -- a number       None                            None

        call: distance(x, y)         # two coordinates
        --or: distance((x, y))       # a pair (tuple) of coordinates
        --or: distance(vec)          # e.g. as returned by pos()
        --or: distance(mypen)        # where mypen is another turtle

        Return the angle, between the line from turtle-position to position
        specified by x, y and the turtle's start orientation. (Depends on
        modes - "standard" or "logo")

        Example (for a Turtle instance named turtle):
        >>> turtle.pos()
        (10.00, 10.00)
        >>> turtle.towards(0,0)
        225.0
        """
        if y is not None:
            pos = Vec2D(x, y)
        if isinstance(x, Vec2D):
            pos = x
        elif isinstance(x, tuple):
            pos = Vec2D(*x)
        elif isinstance(x, TNavigator):
            pos = x._position
        x, y = pos - self._position
        result = round(math.atan2(y, x)*180.0/math.pi, 10) % 360.0
        result /= self._degreesPerAU
        return (self._angleOffset + self._angleOrient*result) % self._fullcircle

    def heading(self):
        """ Return the turtle's current heading.

        No arguments.

        Example (for a Turtle instance named turtle):
        >>> turtle.left(67)
        >>> turtle.heading()
        67.0
        """
        x, y = self._orient
        result = round(math.atan2(y, x)*180.0/math.pi, 10) % 360.0
        result /= self._degreesPerAU
        return (self._angleOffset + self._angleOrient*result) % self._fullcircle

    def setheading(self, to_angle):
        """Set the orientation of the turtle to to_angle.

        Aliases:  setheading | seth

        Argument:
        to_angle -- a number (integer or float)

        Set the orientation of the turtle to to_angle.
        Here are some common directions in degrees:

         standard - mode:          logo-mode:
        -------------------|--------------------
           0 - east                0 - north
          90 - north              90 - east
         180 - west              180 - south
         270 - south             270 - west

        Example (for a Turtle instance named turtle):
        >>> turtle.setheading(90)
        >>> turtle.heading()
        90
        """
        angle = (to_angle - self.heading())*self._angleOrient
        full = self._fullcircle
        angle = (angle+full/2.)%full - full/2.
        self._rotate(angle)

    def circle(self, radius, extent = None, steps = None):
        """ Draw a circle with given radius.

        Arguments:
        radius -- a number
        extent (optional) -- a number
        steps (optional) -- an integer

        Draw a circle with given radius. The center is radius units left
        of the turtle; extent - an angle - determines which part of the
        circle is drawn. If extent is not given, draw the entire circle.
        If extent is not a full circle, one endpoint of the arc is the
        current pen position. Draw the arc in counterclockwise direction
        if radius is positive, otherwise in clockwise direction. Finally
        the direction of the turtle is changed by the amount of extent.

        As the circle is approximated by an inscribed regular polygon,
        steps determines the number of steps to use. If not given,
        it will be calculated automatically. Maybe used to draw regular
        polygons.

        call: circle(radius)                  # full circle
        --or: circle(radius, extent)          # arc
        --or: circle(radius, extent, steps)
        --or: circle(radius, steps=6)         # 6-sided polygon

        Example (for a Turtle instance named turtle):
        >>> turtle.circle(50)
        >>> turtle.circle(120, 180)  # semicircle
        """
        if self.undobuffer:
            self.undobuffer.push(["seq"])
            self.undobuffer.cumulate = True
        speed = self.speed()
        if extent is None:
            extent = self._fullcircle
        if steps is None:
            frac = abs(extent)/self._fullcircle
            steps = 1+int(min(11+abs(radius)/6.0, 59.0)*frac)
        w = 1.0 * extent / steps
        w2 = 0.5 * w
        l = 2.0 * radius * math.sin(w2*math.pi/180.0*self._degreesPerAU)
        if radius < 0:
            l, w, w2 = -l, -w, -w2
        tr = self._tracer()
        dl = self._delay()
        if speed == 0:
            self._tracer(0, 0)
        else:
            self.speed(0)
        self._rotate(w2)
        for i in range(steps):
            self.speed(speed)
            self._go(l)
            self.speed(0)
            self._rotate(w)
        self._rotate(-w2)
        if speed == 0:
            self._tracer(tr, dl)
        self.speed(speed)
        if self.undobuffer:
            self.undobuffer.cumulate = False

## three dummy methods to be implemented by child class:

    def speed(self, s=0):
        """dummy method - to be overwritten by child class"""
    def _tracer(self, a=None, b=None):
        """dummy method - to be overwritten by child class"""
    def _delay(self, n=None):
        """dummy method - to be overwritten by child class"""

    fd = forward
    bk = back
    backward = back
    rt = right
    lt = left
    position = pos
    setpos = goto
    setposition = goto
    seth = setheading


class TPen(object):
    """Drawing part of the RawTurtle.
    Implements drawing properties.
    """
    def __init__(self, resizemode=_CFG["resizemode"]):
        self._resizemode = resizemode # or "user" or "noresize"
        self.undobuffer = None
        TPen._reset(self)

    def _reset(self, pencolor=_CFG["pencolor"],
                     fillcolor=_CFG["fillcolor"]):
        self._pensize = 1
        self._shown = True
        self._pencolor = pencolor
        self._fillcolor = fillcolor
        self._drawing = True
        self._speed = 3
        self._stretchfactor = (1., 1.)
        self._shearfactor = 0.
        self._tilt = 0.
        self._shapetrafo = (1., 0., 0., 1.)
        self._outlinewidth = 1

    def resizemode(self, rmode=None):
        """Set resizemode to one of the values: "auto", "user", "noresize".

        (Optional) Argument:
        rmode -- one of the strings "auto", "user", "noresize"

        Different resizemodes have the following effects:
          - "auto" adapts the appearance of the turtle
                   corresponding to the value of pensize.
          - "user" adapts the appearance of the turtle according to the
                   values of stretchfactor and outlinewidth (outline),
                   which are set by shapesize()
          - "noresize" no adaption of the turtle's appearance takes place.
        If no argument is given, return current resizemode.
        resizemode("user") is called by a call of shapesize with arguments.


        Examples (for a Turtle instance named turtle):
        >>> turtle.resizemode("noresize")
        >>> turtle.resizemode()
        'noresize'
        """
        if rmode is None:
            return self._resizemode
        rmode = rmode.lower()
        if rmode in ["auto", "user", "noresize"]:
            self.pen(resizemode=rmode)

    def pensize(self, width=None):
        """Set or return the line thickness.

        Aliases:  pensize | width

        Argument:
        width -- positive number

        Set the line thickness to width or return it. If resizemode is set
        to "auto" and turtleshape is a polygon, that polygon is drawn with
        the same line thickness. If no argument is given, current pensize
        is returned.

        Example (for a Turtle instance named turtle):
        >>> turtle.pensize()
        1
        >>> turtle.pensize(10)   # from here on lines of width 10 are drawn
        """
        if width is None:
            return self._pensize
        self.pen(pensize=width)


    def penup(self):
        """Pull the pen up -- no drawing when moving.

        Aliases: penup | pu | up

        No argument

        Example (for a Turtle instance named turtle):
        >>> turtle.penup()
        """
        if not self._drawing:
            return
        self.pen(pendown=False)

    def pendown(self):
        """Pull the pen down -- drawing when moving.

        Aliases: pendown | pd | down

        No argument.

        Example (for a Turtle instance named turtle):
        >>> turtle.pendown()
        """
        if self._drawing:
            return
        self.pen(pendown=True)

    def isdown(self):
        """Return True if pen is down, False if it's up.

        No argument.

        Example (for a Turtle instance named turtle):
        >>> turtle.penup()
        >>> turtle.isdown()
        False
        >>> turtle.pendown()
        >>> turtle.isdown()
        True
        """
        return self._drawing

    def speed(self, speed=None):
        """ Return or set the turtle's speed.

        Optional argument:
        speed -- an integer in the range 0..10 or a speedstring (see below)

        Set the turtle's speed to an integer value in the range 0 .. 10.
        If no argument is given: return current speed.

        If input is a number greater than 10 or smaller than 0.5,
        speed is set to 0.
        Speedstrings  are mapped to speedvalues in the following way:
            'fastest' :  0
            'fast'    :  10
            'normal'  :  6
            'slow'    :  3
            'slowest' :  1
        speeds from 1 to 10 enforce increasingly faster animation of
        line drawing and turtle turning.

        Attention:
        speed = 0 : *no* animation takes place. forward/back makes turtle jump
        and likewise left/right make the turtle turn instantly.

        Example (for a Turtle instance named turtle):
        >>> turtle.speed(3)
        """
        speeds = {'fastest':0, 'fast':10, 'normal':6, 'slow':3, 'slowest':1 }
        if speed is None:
            return self._speed
        if speed in speeds:
            speed = speeds[speed]
        elif 0.5 < speed < 10.5:
            speed = int(round(speed))
        else:
            speed = 0
        self.pen(speed=speed)

    def color(self, *args):
        """Return or set the pencolor and fillcolor.

        Arguments:
        Several input formats are allowed.
        They use 0, 1, 2, or 3 arguments as follows:

        color()
            Return the current pencolor and the current fillcolor
            as a pair of color specification strings as are returned
            by pencolor and fillcolor.
        color(colorstring), color((r,g,b)), color(r,g,b)
            inputs as in pencolor, set both, fillcolor and pencolor,
            to the given value.
        color(colorstring1, colorstring2),
        color((r1,g1,b1), (r2,g2,b2))
            equivalent to pencolor(colorstring1) and fillcolor(colorstring2)
            and analogously, if the other input format is used.

        If turtleshape is a polygon, outline and interior of that polygon
        is drawn with the newly set colors.
        For mor info see: pencolor, fillcolor

        Example (for a Turtle instance named turtle):
        >>> turtle.color('red', 'green')
        >>> turtle.color()
        ('red', 'green')
        >>> colormode(255)
        >>> color((40, 80, 120), (160, 200, 240))
        >>> color()
        ('#285078', '#a0c8f0')
        """
        if args:
            l = len(args)
            if l == 1:
                pcolor = fcolor = args[0]
            elif l == 2:
                pcolor, fcolor = args
            elif l == 3:
                pcolor = fcolor = args
            pcolor = self._colorstr(pcolor)
            fcolor = self._colorstr(fcolor)
            self.pen(pencolor=pcolor, fillcolor=fcolor)
        else:
            return self._color(self._pencolor), self._color(self._fillcolor)

    def pencolor(self, *args):
        """ Return or set the pencolor.

        Arguments:
```

```
        Four input formats are allowed:
          - pencolor()
            Return the current pencolor as color specification string,
            possibly in hex-number format (see example).
            May be used as input to another color/pencolor/fillcolor call.
          - pencolor(colorstring)
            s is a Tk color specification string, such as "red" or "yellow"
          - pencolor((r, g, b))
            *a tuple* of r, g, and b, which represent, an RGB color,
            and each of r, g, and b are in the range 0..colormode,
            where colormode is either 1.0 or 255
          - pencolor(r, g, b)
            r, g, and b represent an RGB color, and each of r, g, and b
            are in the range 0..colormode

        If turtleshape is a polygon, the outline of that polygon is drawn
        with the newly set pencolor.

        Example (for a Turtle instance named turtle):
        >>> turtle.pencolor('brown')
        >>> tup = (0.2, 0.8, 0.55)
        >>> turtle.pencolor(tup)
        >>> turtle.pencolor()
        '#33cc8c'
        """
        if args:
            color = self._colorstr(args)
            if color == self._pencolor:
                return
            self.pen(pencolor=color)
        else:
            return self._color(self._pencolor)

    def fillcolor(self, *args):
        """ Return or set the fillcolor.

        Arguments:
        Four input formats are allowed:
          - fillcolor()
            Return the current fillcolor as color specification string,
            possibly in hex-number format (see example).
            May be used as input to another color/pencolor/fillcolor call.
          - fillcolor(colorstring)
            s is a Tk color specification string, such as "red" or "yellow"
          - fillcolor((r, g, b))
            *a tuple* of r, g, and b, which represent, an RGB color,
            and each of r, g, and b are in the range 0..colormode,
            where colormode is either 1.0 or 255
          - fillcolor(r, g, b)
            r, g, and b represent an RGB color, and each of r, g, and b
            are in the range 0..colormode

        If turtleshape is a polygon, the interior of that polygon is drawn
        with the newly set fillcolor.

        Example (for a Turtle instance named turtle):
        >>> turtle.fillcolor('violet')
        >>> col = turtle.pencolor()
        >>> turtle.fillcolor(col)
        >>> turtle.fillcolor(0, .5, 0)
        """
        if args:
            color = self._colorstr(args)
            if color == self._fillcolor:
                return
            self.pen(fillcolor=color)
        else:
            return self._color(self._fillcolor)

    def showturtle(self):
        """Makes the turtle visible.

        Aliases: showturtle | st

        No argument.

        Example (for a Turtle instance named turtle):
        >>> turtle.hideturtle()
        >>> turtle.showturtle()
        """
        self.pen(shown=True)

    def hideturtle(self):
        """Makes the turtle invisible.

        Aliases: hideturtle | ht

        No argument.

        It's a good idea to do this while you're in the
        middle of a complicated drawing, because hiding
        the turtle speeds up the drawing observably.

        Example (for a Turtle instance named turtle):
        >>> turtle.hideturtle()
        """
        self.pen(shown=False)

    def isvisible(self):
        """Return True if the Turtle is shown, False if it's hidden.

        No argument.

        Example (for a Turtle instance named turtle):
        >>> turtle.hideturtle()
        >>> print turtle.isvisible():
        False
        """
        return self._shown

    def pen(self, pen=None, **pendict):
        """Return or set the pen's attributes.

        Arguments:
            pen -- a dictionary with some or all of the below listed keys.
            **pendict -- one or more keyword-arguments with the below
                         listed keys as keywords.

        Return or set the pen's attributes in a 'pen-dictionary'
        with the following key/value pairs:
           "shown"      :   True/False
           "pendown"    :   True/False
           "pencolor"   :   color-string or color-tuple
           "fillcolor"  :   color-string or color-tuple
           "pensize"    :   positive number
           "speed"      :   number in range 0..10
           "resizemode" :   "auto" or "user" or "noresize"
           "stretchfactor": (positive number, positive number)
           "shearfactor":   number
           "outline"    :   positive number
           "tilt"       :   number

        This dictionary can be used as argument for a subsequent
        pen()-call to restore the former pen-state. Moreover one
        or more of these attributes can be provided as keyword-arguments.
        This can be used to set several pen attributes in one statement.


        Examples (for a Turtle instance named turtle):
        >>> turtle.pen(fillcolor="black", pencolor="red", pensize=10)
        >>> turtle.pen()
        {'pensize': 10, 'shown': True, 'resizemode': 'auto', 'outline': 1,
        'pencolor': 'red', 'pendown': True, 'fillcolor': 'black',
        'stretchfactor': (1,1), 'speed': 3, 'shearfactor': 0.0}
        >>> penstate=turtle.pen()
        >>> turtle.color("yellow","")
        >>> turtle.penup()
        >>> turtle.pen()
        {'pensize': 10, 'shown': True, 'resizemode': 'auto', 'outline': 1,
        'pencolor': 'yellow', 'pendown': False, 'fillcolor': '',
        'stretchfactor': (1,1), 'speed': 3, 'shearfactor': 0.0}
        >>> p.pen(penstate, fillcolor="green")
        >>> p.pen()
        {'pensize': 10, 'shown': True, 'resizemode': 'auto', 'outline': 1,
        'pencolor': 'red', 'pendown': True, 'fillcolor': 'green',
        'stretchfactor': (1,1), 'speed': 3, 'shearfactor': 0.0}
        """
        _pd =  {"shown"         : self._shown,
                "pendown"       : self._drawing,
                "pencolor"      : self._pencolor,
                "fillcolor"     : self._fillcolor,
                "pensize"       : self._pensize,
                "speed"         : self._speed,
                "resizemode"    : self._resizemode,
                "stretchfactor" : self._stretchfactor,
                "shearfactor"   : self._shearfactor,
                "outline"       : self._outlinewidth,
                "tilt"          : self._tilt
               }

        if not (pen or pendict):
            return _pd

        if isinstance(pen, dict):
            p = pen
        else:
            p = {}
        p.update(pendict)

        _p_buf = {}
        for key in p:
            _p_buf[key] = _pd[key]

        if self.undobuffer:
            self.undobuffer.push(("pen", _p_buf))

        newLine = False
        if "pendown" in p:
            if self._drawing != p["pendown"]:
                newLine = True
        if "pencolor" in p:
            if isinstance(p["pencolor"], tuple):
                p["pencolor"] = self._colorstr((p["pencolor"],))
            if self._pencolor != p["pencolor"]:
                newLine = True
        if "pensize" in p:
            if self._pensize != p["pensize"]:
                newLine = True
        if newLine:
            self._newLine()
        if "pendown" in p:
            self._drawing = p["pendown"]
        if "pencolor" in p:
            self._pencolor = p["pencolor"]
        if "pensize" in p:
            self._pensize = p["pensize"]
        if "fillcolor" in p:
            if isinstance(p["fillcolor"], tuple):
                p["fillcolor"] = self._colorstr((p["fillcolor"],))
            self._fillcolor = p["fillcolor"]
        if "speed" in p:
            self._speed = p["speed"]
        if "resizemode" in p:
            self._resizemode = p["resizemode"]
        if "stretchfactor" in p:
            sf = p["stretchfactor"]
            if isinstance(sf, (int, float)):
                sf = (sf, sf)
            self._stretchfactor = sf
        if "shearfactor" in p:
            self._shearfactor = p["shearfactor"]
        if "outline" in p:
            self._outlinewidth = p["outline"]
        if "shown" in p:
            self._shown = p["shown"]
        if "tilt" in p:
            self._tilt = p["tilt"]
        if "stretchfactor" in p or "tilt" in p or "shearfactor" in p:
            scx, scy = self._stretchfactor
            shf = self._shearfactor
            sa, ca = math.sin(self._tilt), math.cos(self._tilt)
            self._shapetrafo = ( scx*ca, scy*(shf*ca + sa),
                                -scx*sa, scy*(ca - shf*sa))
        self._update()

## three dummy methods to be implemented by child class:

    def _newLine(self, usePos = True):
        """dummy method - to be overwritten by child class"""
    def _update(self, count=True, forced=False):
        """dummy method - to be overwritten by child class"""
    def _color(self, args):
        """dummy method - to be overwritten by child class"""
    def _colorstr(self, args):
        """dummy method - to be overwritten by child class"""

    width = pensize
    up = penup
    pu = penup
    pd = pendown
    down = pendown
    st = showturtle
    ht = hideturtle


class _TurtleImage(object):
    """Helper class: Datatype to store Turtle attributes
    """

    def __init__(self, screen, shapeIndex):
        self.screen = screen
        self._type = None
        self._setshape(shapeIndex)

    def _setshape(self, shapeIndex):
        screen = self.screen
        self.shapeIndex = shapeIndex
        if self._type == "polygon" == screen._shapes[shapeIndex]._type:
            return
        if self._type == "image" == screen._shapes[shapeIndex]._type:
            return
        if self._type in ["image", "polygon"]:
            screen._delete(self._item)
        elif self._type == "compound":
            for item in self._item:
                screen._delete(item)
        self._type = screen._shapes[shapeIndex]._type
        if self._type == "polygon":
            self._item = screen._createpoly()
        elif self._type == "image":
            self._item = screen._createimage(screen._shapes["blank"]._data)
        elif self._type == "compound":
            self._item = [screen._createpoly() for item in
                                            screen._shapes[shapeIndex]._data]


class RawTurtle(TPen, TNavigator):
    """Animation part of the RawTurtle.
    Puts RawTurtle upon a TurtleScreen and provides tools for
    its animation.
    """
    screens = []

    def __init__(self, canvas=None,
                 shape=_CFG["shape"],
                 undobuffersize=_CFG["undobuffersize"],
                 visible=_CFG["visible"]):
        if isinstance(canvas, _Screen):
            self.screen = canvas
        elif isinstance(canvas, TurtleScreen):
```

```python
            if canvas not in RawTurtle.screens:
                RawTurtle.screens.append(canvas)
            self.screen = canvas
        elif isinstance(canvas, (ScrolledCanvas, Canvas)):
            for screen in RawTurtle.screens:
                if screen.cv == canvas:
                    self.screen = screen
                    break
            else:
                self.screen = TurtleScreen(canvas)
                RawTurtle.screens.append(self.screen)
        else:
            raise TurtleGraphicsError("bad canvas argument %s" % canvas)

        screen = self.screen
        TNavigator.__init__(self, screen.mode())
        TPen.__init__(self)
        screen._turtles.append(self)
        self.drawingLineItem = screen._createline()
        self.turtle = _TurtleImage(screen, shape)
        self._poly = None
        self._creatingPoly = False
        self._fillitem = self._fillpath = None
        self._shown = visible
        self._hidden_from_screen = False
        self.currentLineItem = screen._createline()
        self.currentLine = [self._position]
        self.items = [self.currentLineItem]
        self.stampItems = []
        self._undobuffersize = undobuffersize
        self.undobuffer = Tbuffer(undobuffersize)
        self._update()

    def reset(self):
        """Delete the turtle's drawings and restore its default values.

        No argument.

        Delete the turtle's drawings from the screen, re-center the turtle
        and set variables to the default values.

        Example (for a Turtle instance named turtle):
        >>> turtle.position()
        (0.00,-22.00)
        >>> turtle.heading()
        100.0
        >>> turtle.reset()
        >>> turtle.position()
        (0.00,0.00)
        >>> turtle.heading()
        0.0
        """
        TNavigator.reset(self)
        TPen._reset(self)
        self._clear()
        self._drawturtle()
        self._update()

    def setundobuffer(self, size):
        """Set or disable undobuffer.

        Argument:
        size -- an integer or None

        If size is an integer an empty undobuffer of given size is installed.
        Size gives the maximum number of turtle-actions that can be undone
        by the undo() function.
        If size is None, no undobuffer is present.

        Example (for a Turtle instance named turtle):
        >>> turtle.setundobuffer(42)
        """
        if size is None:
            self.undobuffer = None
        else:
            self.undobuffer = Tbuffer(size)

    def undobufferentries(self):
        """Return count of entries in the undobuffer.

        No argument.

        Example (for a Turtle instance named turtle):
        >>> while undobufferentries():
        ...     undo()
        """
        if self.undobuffer is None:
            return 0
        return self.undobuffer.nr_of_items()

    def _clear(self):
        """Delete all of pen's drawings"""
        self._fillitem = self._fillpath = None
        for item in self.items:
            self.screen._delete(item)
        self.currentLineItem = self.screen._createline()
        self.currentLine = []
        if self._drawing:
            self.currentLine.append(self._position)
        self.items = [self.currentLineItem]
        self.clearstamps()
        self.setundobuffer(self._undobuffersize)


    def clear(self):
        """Delete the turtle's drawings from the screen. Do not move turtle.

        No arguments.

        Delete the turtle's drawings from the screen. Do not move turtle.
        State and position of the turtle as well as drawings of other
        turtles are not affected.

        Examples (for a Turtle instance named turtle):
        >>> turtle.clear()
        """
        self._clear()
        self._update()

    def _update_data(self):
        self.screen._incrementudc()
        if self.screen._updatecounter != 0:
            return
        if len(self.currentLine)>1:
            self.screen._drawline(self.currentLineItem, self.currentLine,
                                  self._pencolor, self._pensize)

    def _update(self):
        """Perform a Turtle-data update.
        """
        screen = self.screen
        if screen._tracing == 0:
            return
        elif screen._tracing == 1:
            self._update_data()
            self._drawturtle()
            screen._update()                  # TurtleScreenBase
            screen._delay(screen._delayvalue) # TurtleScreenBase
        else:
            self._update_data()
            if screen._updatecounter == 0:
                for t in screen.turtles():
                    t._drawturtle()
                screen._update()

    def _tracer(self, flag=None, delay=None):
        """Turns turtle animation on/off and set delay for update drawings.

        Optional arguments:
        n -- nonnegative  integer
        delay -- nonnegative  integer

        If n is given, only each n-th regular screen update is really performed.
        (Can be used to accelerate the drawing of complex graphics.)
        Second arguments sets delay value (see RawTurtle.delay())

        Example (for a Turtle instance named turtle):
        >>> turtle.tracer(8, 25)
        >>> dist = 2
        >>> for i in range(200):
        ...     turtle.fd(dist)
        ...     turtle.rt(90)
        ...     dist += 2
        """
        return self.screen.tracer(flag, delay)

    def _color(self, args):
        return self.screen._color(args)

    def _colorstr(self, args):
        return self.screen._colorstr(args)

    def _cc(self, args):
        """Convert colortriples to hexstrings.
        """
        if isinstance(args, str):
            return args
        try:
            r, g, b = args
        except:
            raise TurtleGraphicsError("bad color arguments: %s" % str(args))
        if self.screen._colormode == 1.0:
            r, g, b = [round(255.0*x) for x in (r, g, b)]
        if not ((0 <= r <= 255) and (0 <= g <= 255) and (0 <= b <= 255)):
            raise TurtleGraphicsError("bad color sequence: %s" % str(args))
        return "#%02x%02x%02x" % (r, g, b)

    def clone(self):
        """Create and return a clone of the turtle.

        No argument.

        Create and return a clone of the turtle with same position, heading
        and turtle properties.

        Example (for a Turtle instance named mick):
        mick = Turtle()
        joe = mick.clone()
        """
        screen = self.screen
        self._newLine(self._drawing)

        turtle = self.turtle
        self.screen = None
        self.turtle = None  # too make self deepcopy-able

        q = deepcopy(self)

        self.screen = screen
        self.turtle = turtle

        q.screen = screen
        q.turtle = _TurtleImage(screen, self.turtle.shapeIndex)

        screen._turtles.append(q)
        ttype = screen._shapes[self.turtle.shapeIndex]._type
        if ttype == "polygon":
            q.turtle._item = screen._createpoly()
        elif ttype == "image":
            q.turtle._item = screen._createimage(screen._shapes["blank"]._data)
        elif ttype == "compound":
            q.turtle._item = [screen._createpoly() for item in
                              screen._shapes[self.turtle.shapeIndex]._data]
        q.currentLineItem = screen._createline()
        q._update()
        return q

    def shape(self, name=None):
        """Set turtle shape to shape with given name / return current shapename.

        Optional argument:
        name -- a string, which is a valid shapename

        Set turtle shape to shape with given name or, if name is not given,
        return name of current shape.
        Shape with name must exist in the TurtleScreen's shape dictionary.
        Initially there are the following polygon shapes:
        'arrow', 'turtle', 'circle', 'square', 'triangle', 'classic'.
        To learn about how to deal with shapes see Screen-method register_shape.

        Example (for a Turtle instance named turtle):
        >>> turtle.shape()
        'arrow'
        >>> turtle.shape("turtle")
        >>> turtle.shape()
        'turtle'
        """
        if name is None:
            return self.turtle.shapeIndex
        if not name in self.screen.getshapes():
            raise TurtleGraphicsError("There is no shape named %s" % name)
        self.turtle._setshape(name)
        self._update()

    def shapesize(self, stretch_wid=None, stretch_len=None, outline=None):
        """Set/return turtle's stretchfactors/outline. Set resizemode to "user".

        Optional arguments:
           stretch_wid : positive number
           stretch_len : positive number
           outline  : positive number

        Return or set the pen's attributes x/y-stretchfactors and/or outline.
        Set resizemode to "user".
        If and only if resizemode is set to "user", the turtle will be displayed
        stretched according to its stretchfactors:
        stretch_wid is stretchfactor perpendicular to orientation
        stretch_len is stretchfactor in direction of turtles orientation.
        outline determines the width of the shapes's outline.

        Examples (for a Turtle instance named turtle):
        >>> turtle.resizemode("user")
        >>> turtle.shapesize(5, 5, 12)
        >>> turtle.shapesize(outline=8)
        """
        if stretch_wid is stretch_len is outline is None:
            stretch_wid, stretch_len = self._stretchfactor
            return stretch_wid, stretch_len, self._outlinewidth
        if stretch_wid == 0 or stretch_len == 0:
            raise TurtleGraphicsError("stretch_wid/stretch_len must not be zero")
        if stretch_wid is not None:
            if stretch_len is None:
                stretchfactor = stretch_wid, stretch_wid
            else:
                stretchfactor = stretch_wid, stretch_len
        elif stretch_len is not None:
            stretchfactor = self._stretchfactor[0], stretch_len
        else:
            stretchfactor = self._stretchfactor
        if outline is None:
            outline = self._outlinewidth
        self.pen(resizemode="user",
                 stretchfactor=stretchfactor, outline=outline)

    def shearfactor(self, shear=None):
        """Set or return the current shearfactor.

        Optional argument: shear -- number, tangent of the shear angle

        Shear the turtleshape according to the given shearfactor shear,
        which is the tangent of the shear angle. DO NOT change the
        turtle's heading (direction of movement).
        If shear is not given: return the current shearfactor, i. e. the
        tangent of the shear angle, by which lines parallel to the
        heading of the turtle are sheared.

        Examples (for a Turtle instance named turtle):
        >>> turtle.shape("circle")
        >>> turtle.shapesize(5,2)
        >>> turtle.shearfactor(0.5)
        >>> turtle.shearfactor()
        >>> 0.5
        """
        if shear is None:
            return self._shearfactor
        self.pen(resizemode="user", shearfactor=shear)

    def settiltangle(self, angle):
        """Rotate the turtleshape to point in the specified direction

        Argument: angle -- number

        Rotate the turtleshape to point in the direction specified by angle,
        regardless of its current tilt-angle. DO NOT change the turtle's
        heading (direction of movement).


        Examples (for a Turtle instance named turtle):
        >>> turtle.shape("circle")
        >>> turtle.shapesize(5,2)
        >>> turtle.settiltangle(45)
        >>> stamp()
        >>> turtle.fd(50)
        >>> turtle.settiltangle(-45)
        >>> stamp()
        >>> turtle.fd(50)
        """
        tilt = -angle * self._degreesPerAU * self._angleOrient
        tilt = (tilt * math.pi / 180.0) % (2*math.pi)
        self.pen(resizemode="user", tilt=tilt)

    def tiltangle(self, angle=None):
        """Set or return the current tilt-angle.

        Optional argument: angle -- number

        Rotate the turtleshape to point in the direction specified by angle,
        regardless of its current tilt-angle. DO NOT change the turtle's
        heading (direction of movement).
        If angle is not given: return the current tilt-angle, i. e. the angle
        between the orientation of the turtleshape and the heading of the
        turtle (its direction of movement).

        Deprecated since Python 3.1

        Examples (for a Turtle instance named turtle):
        >>> turtle.shape("circle")
        >>> turtle.shapesize(5,2)
        >>> turtle.tilt(45)
        >>> turtle.tiltangle()
        """
        if angle is None:
            tilt = -self._tilt * (180.0/math.pi) * self._angleOrient
            return (tilt / self._degreesPerAU) % self._fullcircle
        else:
            self.settiltangle(angle)

    def tilt(self, angle):
        """Rotate the turtleshape by angle.

        Argument:
        angle - a number

        Rotate the turtleshape by angle from its current tilt-angle,
        but do NOT change the turtle's heading (direction of movement).

        Examples (for a Turtle instance named turtle):
        >>> turtle.shape("circle")
        >>> turtle.shapesize(5,2)
        >>> turtle.tilt(30)
        >>> turtle.fd(50)
        >>> turtle.tilt(30)
        >>> turtle.fd(50)
        """
        self.settiltangle(angle + self.tiltangle())

    def shapetransform(self, t11=None, t12=None, t21=None, t22=None):
        """Set or return the current transformation matrix of the turtle shape.

        Optional arguments: t11, t12, t21, t22 -- numbers.

        If none of the matrix elements are given, return the transformation
        matrix.
        Otherwise set the given elements and transform the turtleshape
        according to the matrix consisting of first row t11, t12 and
        second row t21, 22.
        Modify stretchfactor, shearfactor and tiltangle according to the
        given matrix.

        Examples (for a Turtle instance named turtle):
        >>> turtle.shape("square")
        >>> turtle.shapesize(4,2)
        >>> turtle.shearfactor(-0.5)
        >>> turtle.shapetransform()
        (4.0, -1.0, -0.0, 2.0)
        """
        if t11 is t12 is t21 is t22 is None:
            return self._shapetrafo
        m11, m12, m21, m22 = self._shapetrafo
        if t11 is not None: m11 = t11
        if t12 is not None: m12 = t12
        if t21 is not None: m21 = t21
        if t22 is not None: m22 = t22
        if t11 * t22 - t12 * t21 == 0:
            raise TurtleGraphicsError("Bad shape transform matrix: must not be singular")
        self._shapetrafo = (m11, m12, m21, m22)
        alfa = math.atan2(-m21, m11) % (2 * math.pi)
        sa, ca = math.sin(alfa), math.cos(alfa)
        a11, a12, a21, a22 = (ca*m11 - sa*m21, ca*m12 - sa*m22,
                              sa*m11 + ca*m21, sa*m12 + ca*m22)
        self._stretchfactor = a11, a22
        self._shearfactor = a12/a22
        self._tilt = alfa
        self._update()


    def _polytrafo(self, poly):
        """Computes transformed polygon shapes from a shape
        according to current position and heading.
        """
        screen = self.screen
        p0, p1 = self._position
        e0, e1 = self._orient
        e = Vec2D(e0, e1 * screen.yscale / screen.xscale)
        e0, e1 = (1.0 / abs(e)) * e
        return [(p0+(e1*x+e0*y)/screen.xscale, p1+(-e0*x+e1*y)/screen.yscale)
                                                           for (x, y) in poly]

    def get_shapepoly(self):
        """Return the current shape polygon as tuple of coordinate pairs.

        No argument.

        Examples (for a Turtle instance named turtle):
        >>> turtle.shape("square")
        >>> turtle.shapetransform(4, -1, 0, 2)
        >>> turtle.get_shapepoly()
        ((50, -20), (30, 20), (-50, 20), (-30, -20))

        """
        shape = self.screen._shapes[self.turtle.shapeIndex]
        if shape._type == "polygon":
            return self._getshapepoly(shape._data, shape._type == "compound")
        # else return None

    def _getshapepoly(self, polygon, compound=False):
        """Calculate transformed shape polygon according to resizemode
        and shapetransform.
        """
        if self._resizemode == "user" or compound:
            t11, t12, t21, t22 = self._shapetrafo
        elif self._resizemode == "auto":
            l = max(1, self._pensize/5.0)
            t11, t12, t21, t22 = l, 0, 0, l
        elif self._resizemode == "noresize":
            return polygon
        return tuple([(t11*x + t12*y, t21*x + t22*y) for (x, y) in polygon])

    def _drawturtle(self):
        """Manages the correct rendering of the turtle with respect to
        its shape, resizemode, stretch and tilt etc."""
        screen = self.screen
        shape = screen._shapes[self.turtle.shapeIndex]
        ttype = shape._type
        titem = self.turtle._item
        if self._shown and screen._updatecounter == 0 and screen._tracing > 0:
            self._hidden_from_screen = False
            tshape = shape._data
            if ttype == "polygon":
                if self._resizemode == "noresize": w = 1
                elif self._resizemode == "auto": w = self._pensize
                else: w =self._outlinewidth
                shape = self._polytrafo(self._getshapepoly(tshape))
                fc, oc = self._fillcolor, self._pencolor
                screen._drawpoly(titem, shape, fill=fc, outline=oc,
                                                      width=w, top=True)
            elif ttype == "image":
                screen._drawimage(titem, self._position, tshape)
            elif ttype == "compound":
                for item, (poly, fc, oc) in zip(titem, tshape):
                    poly = self._polytrafo(self._getshapepoly(poly, True))
                    screen._drawpoly(item, poly, fill=self._cc(fc),
                                     outline=self._cc(oc), width=self._outlinewidth, top=True)
        else:
            if self._hidden_from_screen:
                return
            if ttype == "polygon":
                screen._drawpoly(titem, ((0, 0), (0, 0), (0, 0)), "", "")
            elif ttype == "image":
                screen._drawimage(titem, self._position,
                                          screen._shapes["blank"]._data)
            elif ttype == "compound":
                for item in titem:
                    screen._drawpoly(item, ((0, 0), (0, 0), (0, 0)), "", "")
            self._hidden_from_screen = True

##############################  stamp stuff  ###############################

    def stamp(self):
        """Stamp a copy of the turtleshape onto the canvas and return its id.

        No argument.

        Stamp a copy of the turtle shape onto the canvas at the current
        turtle position. Return a stamp_id for that stamp, which can be
        used to delete it by calling clearstamp(stamp_id).

        Example (for a Turtle instance named turtle):
        >>> turtle.color("blue")
        >>> turtle.stamp()
        13
        >>> turtle.fd(50)
        """
        screen = self.screen
        shape = screen._shapes[self.turtle.shapeIndex]
        ttype = shape._type
        tshape = shape._data
        if ttype == "polygon":
            stitem = screen._createpoly()
            if self._resizemode == "noresize": w = 1
            elif self._resizemode == "auto": w = self._pensize
            else: w =self._outlinewidth
            shape = self._polytrafo(self._getshapepoly(tshape))
            fc, oc = self._fillcolor, self._pencolor
            screen._drawpoly(stitem, shape, fill=fc, outline=oc,
                                                  width=w, top=True)
        elif ttype == "image":
            stitem = screen._createimage("")
            screen._drawimage(stitem, self._position, tshape)
        elif ttype == "compound":
            stitem = []
            for element in tshape:
                item = screen._createpoly()
                stitem.append(item)
            stitem = tuple(stitem)
            for item, (poly, fc, oc) in zip(stitem, tshape):
                poly = self._polytrafo(self._getshapepoly(poly, True))
                screen._drawpoly(item, poly, fill=self._cc(fc),
                                 outline=self._cc(oc), width=self._outlinewidth,
top=True)
        self.stampItems.append(stitem)
        self.undobuffer.push(("stamp", stitem))
        return stitem

    def _clearstamp(self, stampid):
        """does the work for clearstamp() and clearstamps()
        """
        if stampid in self.stampItems:
            if isinstance(stampid, tuple):
                for subitem in stampid:
                    self.screen._delete(subitem)
            else:
                self.screen._delete(stampid)
            self.stampItems.remove(stampid)
        # Delete stampitem from undobuffer if necessary
        # if clearstamp is called directly.
        item = ("stamp", stampid)
        buf = self.undobuffer
        if item not in buf.buffer:
            return
        index = buf.buffer.index(item)
        buf.buffer.remove(item)
        if index <= buf.ptr:
            buf.ptr = (buf.ptr - 1) % buf.bufsize
        buf.buffer.insert((buf.ptr+1)%buf.bufsize, [None])

    def clearstamp(self, stampid):
        """Delete stamp with given stampid

        Argument:
        stampid - an integer, must be return value of previous stamp() call.

        Example (for a Turtle instance named turtle):
        >>> turtle.color("blue")
        >>> astamp = turtle.stamp()
        >>> turtle.fd(50)
        >>> turtle.clearstamp(astamp)
        """
        self._clearstamp(stampid)
        self._update()

    def clearstamps(self, n=None):
        """Delete all or first/last n of turtle's stamps.

        Optional argument:
        n -- an integer

        If n is None, delete all of pen's stamps,
        else if n > 0 delete first n stamps
        else if n < 0 delete last n stamps.

        Example (for a Turtle instance named turtle):
        >>> for i in range(8):
        ...     turtle.stamp(); turtle.fd(30)
        ...
        >>> turtle.clearstamps(2)
        >>> turtle.clearstamps(-2)
        >>> turtle.clearstamps()
        """
        if n is None:
            toDelete = self.stampItems[:]
        elif n >= 0:
            toDelete = self.stampItems[:n]
        else:
            toDelete = self.stampItems[n:]
        for item in toDelete:
            self._clearstamp(item)
        self._update()

    def _goto(self, end):
        """Move the pen to the point end, thereby drawing a line
        if pen is down. All other methods for turtle movement depend
        on this one.
        """
        ## Version with undo-stuff
        go_modes = ( self._drawing,
                     self._pencolor,
                     self._pensize,
                     isinstance(self._fillpath, list))
        screen = self.screen
        undo_entry = ("go", self._position, end, go_modes,
                      (self.currentLineItem,
                      self.currentLine[:],
                      screen._pointlist(self.currentLineItem),
                      self.items[:])
                      )
        if self.undobuffer:
            self.undobuffer.push(undo_entry)
        start = self._position
        if self._speed and screen._tracing == 1:
            diff = (end-start)
            diffsq = (diff[0]*screen.xscale)**2 + (diff[1]*screen.yscale)**2
            nhops = 1+int((diffsq**0.5)/(3*(1.1**self._speed)*self._speed))
            delta = diff * (1.0/nhops)
            for n in range(1, nhops):
                if n == 1:
                    top = True
                else:
                    top = False
                self._position = start + delta * n
                if self._drawing:
                    screen._drawline(self.drawingLineItem,
                                     (start, self._position),
                                     self._pencolor, self._pensize, top)
                self._update()
            if self._drawing:
                screen._drawline(self.drawingLineItem, ((0, 0), (0, 0)),
                                               fill="", width=self._pensize)
        # Turtle now at end,
        if self._drawing: # now update currentLine
            self.currentLine.append(end)
        if isinstance(self._fillpath, list):
            self._fillpath.append(end)
        ######    vererbung!!!!!!!!!!!!!!!!!!!!!
        self._position = end
        if self._creatingPoly:
            self._poly.append(end)
        if len(self.currentLine) > 42: # 42! answer to the ultimate question
                                       # of life, the universe and everything
            self._newLine()
        self._update() #count=True)

    def _undogoto(self, entry):
        """Reverse a _goto. Used for undo()
        """
        old, new, go_modes, coodata = entry
        drawing, pc, ps, filling = go_modes
        cLI, cL, pl, items = coodata
        screen = self.screen
        if abs(self._position - new) > 0.5:
            print ("undogoto: HALLO-DA-STIMMT-WAS-NICHT!")
        # restore former situation
        self.currentLineItem = cLI
        self.currentLine = cL

        if pl == [(0, 0), (0, 0)]:
            usepc = ""
        else:
            usepc = pc
        screen._drawline(cLI, pl, fill=usepc, width=ps)

        todelete = [i for i in self.items if (i not in items) and
                                       (screen._type(i) == "line")]
        for i in todelete:
            screen._delete(i)
            self.items.remove(i)

        start = old
        if self._speed and screen._tracing == 1:
            diff = old - new
            diffsq = (diff[0]*screen.xscale)**2 + (diff[1]*screen.yscale)**2
            nhops = 1+int((diffsq**0.5)/(3*(1.1**self._speed)*self._speed))
            delta = diff * (1.0/nhops)
            for n in range(1, nhops):
                if n == 1:
                    top = True
                else:
                    top = False
                self._position = new + delta * n
                if drawing:
                    screen._drawline(self.drawingLineItem,
                                     (start, self._position),
                                     pc, ps, top)
                self._update()
            if drawing:
                screen._drawline(self.drawingLineItem, ((0, 0), (0, 0)),
                                               fill="", width=ps)
        # Turtle now at position old,
        self._position = old
        ##  if undo is done during creating a polygon, the last vertex
        ##  will be deleted. if the polygon is entirely deleted,
        ##  creatingPoly will be set to False.
        ##  Polygons created before the last one will not be affected by undo()
        if self._creatingPoly:
            if len(self._poly) > 0:
                self._poly.pop()
            if self._poly == []:
                self._creatingPoly = False
                self._poly = None
        if filling:
            if self._fillpath == []:
                self._fillpath = None
                print("Unwahrscheinlich in _undogoto!")
            elif self._fillpath is not None:
                self._fillpath.pop()
        self._update() #count=True)

    def _rotate(self, angle):
        """Turns pen clockwise by angle.
        """
        if self.undobuffer:
            self.undobuffer.push(("rot", angle, self._degreesPerAU))
        angle *= self._degreesPerAU
        neworient = self._orient.rotate(angle)
        tracing = self.screen._tracing
        if tracing == 1 and self._speed > 0:
            anglevel = 3.0 * self._speed
            steps = 1 + int(abs(angle)/anglevel)
            delta = 1.0*angle/steps
            for _ in range(steps):
                self._orient = self._orient.rotate(delta)
                self._update()
        self._orient = neworient
        self._update()

    def _newLine(self, usePos=True):
        """Closes current line item and starts a new one.
           Remark: if current line became too long, animation
           performance (via _drawline) slowed down considerably.
        """
        if len(self.currentLine) > 1:
            self.screen._drawline(self.currentLineItem, self.currentLine,
                                      self._pencolor, self._pensize)
            self.currentLineItem = self.screen._createline()
            self.items.append(self.currentLineItem)
        else:
            self.screen._drawline(self.currentLineItem, top=True)
        self.currentLine = []
        if usePos:
            self.currentLine = [self._position]

    def filling(self):
        """Return fillstate (True if filling, False else).

        No argument.

        Example (for a Turtle instance named turtle):
        >>> turtle.begin_fill()
        >>> if turtle.filling():
        ...     turtle.pensize(5)
        ... else:
        ...     turtle.pensize(3)
        """
        return isinstance(self._fillpath, list)

    def begin_fill(self):
        """Called just before drawing a shape to be filled.

        No argument.

        Example (for a Turtle instance named turtle):
        >>> turtle.color("black", "red")
        >>> turtle.begin_fill()
        >>> turtle.circle(60)
        >>> turtle.end_fill()
        """
        if not self.filling():
            self._fillitem = self.screen._createpoly()
            self.items.append(self._fillitem)
        self._fillpath = [self._position]
        self._newLine()
        if self.undobuffer:
            self.undobuffer.push(("beginfill", self._fillitem))
        self._update()


    def end_fill(self):
        """Fill the shape drawn after the call begin_fill().

        No argument.

        Example (for a Turtle instance named turtle):
        >>> turtle.color("black", "red")
        >>> turtle.begin_fill()
        >>> turtle.circle(60)
        >>> turtle.end_fill()
        """
        if self.filling():
            if len(self._fillpath) > 2:
                self.screen._drawpoly(self._fillitem, self._fillpath,
                                      fill=self._fillcolor)
                if self.undobuffer:
                    self.undobuffer.push(("dofill", self._fillitem))
            self._fillitem = self._fillpath = None
            self._update()

    def dot(self, size=None, *color):
        """Draw a dot with diameter size, using color.

        Optional arguments:
        size -- an integer >= 1 (if given)
        color -- a colorstring or a numeric color tuple

        Draw a circular dot with diameter size, using color.
        If size is not given, the maximum of pensize+4 and 2*pensize is used.

        Example (for a Turtle instance named turtle):
        >>> turtle.dot()
        >>> turtle.fd(50); turtle.dot(20, "blue"); turtle.fd(50)
        """
        if not color:
            if isinstance(size, (str, tuple)):
                color = self._colorstr(size)
                size = self._pensize + max(self._pensize, 4)
            else:
                color = self._pencolor
                if not size:
                    size = self._pensize + max(self._pensize, 4)
        else:
            if size is None:
                size = self._pensize + max(self._pensize, 4)
            color = self._colorstr(color)
        if hasattr(self.screen, "_dot"):
            item = self.screen._dot(self._position, size, color)
            self.items.append(item)
            if self.undobuffer:
                self.undobuffer.push(("dot", item))
        else:
            pen = self.pen()
            if self.undobuffer:
                self.undobuffer.push(["seq"])
                self.undobuffer.cumulate = True
            try:
                if self.resizemode() == 'auto':
                    self.ht()
                self.pendown()
                self.pensize(size)
                self.pencolor(color)
                self.forward(0)
            finally:
                self.pen(pen)
            if self.undobuffer:
                self.undobuffer.cumulate = False

    def _write(self, txt, align, font):
        """Performs the writing for write()
        """
        item, end = self.screen._write(self._position, txt, align, font,
                                                          self._pencolor)
        self.items.append(item)
        if self.undobuffer:
            self.undobuffer.push(("wri", item))
        return end

    def write(self, arg, move=False, align="left", font=("Arial", 8, "normal")):
        """Write text at the current turtle position.

        Arguments:
        arg -- info, which is to be written to the TurtleScreen
        move (optional) -- True/False
        align (optional) -- one of the strings "left", "center" or right"
        font (optional) -- a triple (fontname, fontsize, fonttype)

        Write text - the string representation of arg - at the current
        turtle position according to align ("left", "center" or right")
        and with the given font.
        If move is True, the pen is moved to the bottom-right corner
        of the text. By default, move is False.

        Example (for a Turtle instance named turtle):
        >>> turtle.write('Home = ', True, align="center")
        >>> turtle.write((0,0), True)
        """
        if self.undobuffer:
            self.undobuffer.push(["seq"])
            self.undobuffer.cumulate = True
        end = self._write(str(arg), align.lower(), font)
        if move:
            x, y = self.pos()
            self.setpos(end, y)
        if self.undobuffer:
            self.undobuffer.cumulate = False

    def begin_poly(self):
        """Start recording the vertices of a polygon.

        No argument.

        Start recording the vertices of a polygon. Current turtle position
        is first point of polygon.

        Example (for a Turtle instance named turtle):
        >>> turtle.begin_poly()
        """
        self._poly = [self._position]
        self._creatingPoly = True

    def end_poly(self):
        """Stop recording the vertices of a polygon.

        No argument.

        Stop recording the vertices of a polygon. Current turtle position is
        last point of polygon. This will be connected with the first point.

        Example (for a Turtle instance named turtle):
        >>> turtle.end_poly()
        """
        self._creatingPoly = False

    def get_poly(self):
        """Return the lastly recorded polygon.

        No argument.

        Example (for a Turtle instance named turtle):
        >>> p = turtle.get_poly()
        >>> turtle.register_shape("myFavouriteShape", p)
        """
        ## check if there is any poly?
        if self._poly is not None:
            return tuple(self._poly)

    def getscreen(self):
        """Return the TurtleScreen object, the turtle is drawing  on.

        No argument.

        Return the TurtleScreen object, the turtle is drawing  on.
        So TurtleScreen-methods can be called for that object.

        Example (for a Turtle instance named turtle):
        >>> ts = turtle.getscreen()
        >>> ts
        <turtle.TurtleScreen object at 0x0106B770>
        >>> ts.bgcolor("pink")
        """
        return self.screen

    def getturtle(self):
        """Return the Turtleobject itself.

        No argument.

        Only reasonable use: as a function to return the 'anonymous turtle':

        Example:
        >>> pet = getturtle()
        >>> pet.fd(50)
        >>> pet
        <turtle.Turtle object at 0x0187D810>
        >>> turtles()
        [<turtle.Turtle object at 0x0187D810>]
        """
        return self

    getpen = getturtle


    ################################################################
    ### screen oriented methods recurring to methods of TurtleScreen
    ################################################################

    def _delay(self, delay=None):
        """Set delay value which determines speed of turtle animation.
        """
        return self.screen.delay(delay)

    def onclick(self, fun, btn=1, add=None):
        """Bind fun to mouse-click event on this turtle on canvas.

        Arguments:
        fun --  a function with two arguments, to which will be assigned
                the coordinates of the clicked point on the canvas.
        num --  number of the mouse-button defaults to 1 (left mouse button).
        add --  True or False. If True, new binding will be added, otherwise
                it will replace a former binding.

        Example for the anonymous turtle, i. e. the procedural way:

        >>> def turn(x, y):
        ...     left(360)
        ...
        >>> onclick(turn)  # Now clicking into the turtle will turn it.
        >>> onclick(None)  # event-binding will be removed
        """
        self.screen._onclick(self.turtle._item, fun, btn, add)
        self._update()

    def onrelease(self, fun, btn=1, add=None):
        """Bind fun to mouse-button-release event on this turtle on canvas.

        Arguments:
        fun -- a function with two arguments, to which will be assigned
                the coordinates of the clicked point on the canvas.
        num --  number of the mouse-button defaults to 1 (left mouse button).

        Example (for a MyTurtle instance named joe):
        >>> class MyTurtle(Turtle):
        ...     def glow(self,x,y):
        ...             self.fillcolor("red")
        ...     def unglow(self,x,y):
        ...             self.fillcolor("")
        ...
        >>> joe = MyTurtle()
        >>> joe.onclick(joe.glow)
        >>> joe.onrelease(joe.unglow)

        Clicking on joe turns fillcolor red, unclicking turns it to
        transparent.
        """
        self.screen._onrelease(self.turtle._item, fun, btn, add)
        self._update()

    def ondrag(self, fun, btn=1, add=None):
        """Bind fun to mouse-move event on this turtle on canvas.

        Arguments:
        fun -- a function with two arguments, to which will be assigned
               the coordinates of the clicked point on the canvas.
        num -- number of the mouse-button defaults to 1 (left mouse button).

        Every sequence of mouse-move-events on a turtle is preceded by a
        mouse-click event on that turtle.

        Example (for a Turtle instance named turtle):
        >>> turtle.ondrag(turtle.goto)

        Subsequently clicking and dragging a Turtle will move it
        across the screen thereby producing handdrawings (if pen is
        down).
        """
        self.screen._ondrag(self.turtle._item, fun, btn, add)


    def _undo(self, action, data):
        """Does the main part of the work for undo()
        """
        if self.undobuffer is None:
            return
        if action == "rot":
            angle, degPAU = data
            self._rotate(-angle*degPAU/self._degreesPerAU)
            dummy = self.undobuffer.pop()
        elif action == "stamp":
            stitem = data[0]
            self.clearstamp(stitem)
        elif action == "go":
            self._undogoto(data)
        elif action in ["wri", "dot"]:
            item = data[0]
            self.screen._delete(item)
            self.items.remove(item)
        elif action == "dofill":
            item = data[0]
            self.screen._drawpoly(item, ((0, 0),(0, 0),(0, 0)),
                                  fill="", outline="")
        elif action == "beginfill":
            item = data[0]
            self._fillitem = self._fillpath = None
            if item in self.items:
                self.screen._delete(item)
                self.items.remove(item)
        elif action == "pen":
            TPen.pen(self, data[0])
            self.undobuffer.pop()

    def undo(self):
        """undo (repeatedly) the last turtle action.

        No argument.

        undo (repeatedly) the last turtle action.
        Number of available undo actions is determined by the size of
        the undobuffer.

        Example (for a Turtle instance named turtle):
        >>> for i in range(4):
        ...     turtle.fd(50); turtle.lt(80)
        ...
        >>> for i in range(8):
        ...     turtle.undo()
        ...
        """
        if self.undobuffer is None:
            return
        item = self.undobuffer.pop()
        action = item[0]
        data = item[1:]
        if action == "seq":
            while data:
                item = data.pop()
                self._undo(item[0], item[1:])
        else:
            self._undo(action, data)

    turtlesize = shapesize

RawPen = RawTurtle

###  Screen - Singleton  ########################

def Screen():
    """Return the singleton screen object.
    If none exists at the moment, create a new one and return it,
    else return the existing one."""
    if Turtle._screen is None:
        Turtle._screen = _Screen()
    return Turtle._screen

class _Screen(TurtleScreen):

    _root = None
    _canvas = None
    _title = _CFG["title"]

    def __init__(self):
        # XXX there is no need for this code to be conditional,
        # as there will be only a single _Screen instance, anyway
        # XXX actually, the turtle demo is injecting root window,
        # so perhaps the conditional creation of a root should be
        # preserved (perhaps by passing it as an optional parameter)
        if _Screen._root is None:
            _Screen._root = self._root = _Root()
            self._root.title(_Screen._title)
            self._root.ondestroy(self._destroy)
        if _Screen._canvas is None:
            width = _CFG["width"]
            height = _CFG["height"]
            canvwidth = _CFG["canvwidth"]
            canvheight = _CFG["canvheight"]
            leftright = _CFG["leftright"]
            topbottom = _CFG["topbottom"]
            self._root.setupcanvas(width, height, canvwidth, canvheight)
            _Screen._canvas = self._root._getcanvas()
            TurtleScreen.__init__(self, _Screen._canvas)
            self.setup(width, height, leftright, topbottom)

    def setup(self, width=_CFG["width"], height=_CFG["height"],
              startx=_CFG["leftright"], starty=_CFG["topbottom"]):
        """ Set the size and position of the main window.

        Arguments:
        width: as integer a size in pixels, as float a fraction of the screen.
          Default is 50% of screen.
        height: as integer the height in pixels, as float a fraction of the
          screen. Default is 75% of screen.
        startx: if positive, starting position in pixels from the left
          edge of the screen, if negative from the right edge
          Default, startx=None is to center window horizontally.
        starty: if positive, starting position in pixels from the top
          edge of the screen, if negative from the bottom edge
          Default, starty=None is to center window vertically.

        Examples (for a Screen instance named screen):
        >>> screen.setup (width=200, height=200, startx=0, starty=0)

        sets window to 200x200 pixels, in upper left of screen

        >>> screen.setup(width=.75, height=0.5, startx=None, starty=None)

        sets window to 75% of screen by 50% of screen and centers
        """
        if not hasattr(self._root, "set_geometry"):
            return
        sw = self._root.win_width()
        sh = self._root.win_height()
        if isinstance(width, float) and 0 <= width <= 1:
            width = sw*width
        if startx is None:
            startx = (sw - width) / 2
        if isinstance(height, float) and 0 <= height <= 1:
            height = sh*height
        if starty is None:
            starty = (sh - height) / 2
        self._root.set_geometry(width, height, startx, starty)
        self.update()

    def title(self, titlestring):
        """Set title of turtle-window

        Argument:
        titlestring -- a string, to appear in the titlebar of the
                       turtle graphics window.

        This is a method of Screen-class. Not available for TurtleScreen-
        objects.

        Example (for a Screen instance named screen):
        >>> screen.title("Welcome to the turtle-zoo!")
        """
        if _Screen._root is not None:
            _Screen._root.title(titlestring)
        _Screen._title = titlestring

    def _destroy(self):
        root = self._root
        if root is _Screen._root:
            Turtle._pen = None
            Turtle._screen = None
            _Screen._root = None
            _Screen._canvas = None
        TurtleScreen._RUNNING = True
        root.destroy()

    def bye(self):
        """Shut the turtlegraphics window.

        Example (for a TurtleScreen instance named screen):
        >>> screen.bye()
        """
        self._destroy()

    def exitonclick(self):
        """Go into mainloop until the mouse is clicked.

        No arguments.

        Bind bye() method to mouseclick on TurtleScreen.
        If "using_IDLE" - value in configuration dictionary is False
        (default value), enter mainloop.
        If IDLE with -n switch (no subprocess) is used, this value should be
        set to True in turtle.cfg. In this case IDLE's mainloop
        is active also for the client script.

        This is a method of the Screen-class and not available for
        TurtleScreen instances.

        Example (for a Screen instance named screen):
        >>> screen.exitonclick()

        """
        def exitGracefully(x, y):
            """Screen.bye() with two dummy-parameters"""
            self.bye()
        self.onclick(exitGracefully)
        if _CFG["using_IDLE"]:
            return
        try:
            mainloop()
        except AttributeError:
            exit(0)


class Turtle(RawTurtle):
    """RawTurtle auto-creating (scrolled) canvas.

    When a Turtle object is created or a function derived from some
    Turtle method is called a TurtleScreen object is automatically created.
    """
    _pen = None
    _screen = None

    def __init__(self,
                 shape=_CFG["shape"],
                 undobuffersize=_CFG["undobuffersize"],
                 visible=_CFG["visible"]):
        if Turtle._screen is None:
            Turtle._screen = Screen()
        RawTurtle.__init__(self, Turtle._screen,
                           shape=shape,
                           undobuffersize=undobuffersize,
                           visible=visible)

Pen = Turtle

def _getpen():
    """Create the 'anonymous' turtle if not already present."""
    if Turtle._pen is None:
        Turtle._pen = Turtle()
    return Turtle._pen

def _getscreen():
    """Create a TurtleScreen if not already present."""
    if Turtle._screen is None:
        Turtle._screen = Screen()
    return Turtle._screen

def write_docstringdict(filename="turtle_docstringdict"):
    """Create and write docstring-dictionary to file.

    Optional argument:
    filename -- a string, used as filename
                default value is turtle_docstringdict

    Has to be called explicitly, (not used by the turtle-graphics classes)
    The docstring dictionary will be written to the Python script <filname>.py
    It is intended to serve as a template for translation of the docstrings
```

```python
    into different languages.
    """
    docsdict = {}

    for methodname in _tg_screen_functions:
        key = "_Screen."+methodname
        docsdict[key] = eval(key).__doc__
    for methodname in _tg_turtle_functions:
        key = "Turtle."+methodname
        docsdict[key] = eval(key).__doc__

    f = open("%s.py" % filename,"w")
    keys = sorted([x for x in docsdict.keys()
                        if x.split('.')[1] not in _alias_list])
    f.write('docsdict = {\n\n')
    for key in keys[:-1]:
        f.write('%s :\n' % repr(key))
        f.write('        """%s\n""",\n\n' % docsdict[key])
    key = keys[-1]
    f.write('%s :\n' % repr(key))
    f.write('        """%s\n"""\n\n' % docsdict[key])
    f.write("}\n")
    f.close()

def read_docstrings(lang):
    """Read in docstrings from lang-specific docstring dictionary.

    Transfer docstrings, translated to lang, from a dictionary-file
    to the methods of classes Screen and Turtle and - in revised form -
    to the corresponding functions.
    """
    modname = "turtle_docstringdict_%(language)s" % {'language':lang.lower()}
    module = __import__(modname)
    docsdict = module.docsdict
    for key in docsdict:
        try:
#             eval(key).im_func.__doc__ = docsdict[key]
            eval(key).__doc__ = docsdict[key]
        except:
            print("Bad docstring-entry: %s" % key)

_LANGUAGE = _CFG["language"]

try:
    if _LANGUAGE != "english":
        read_docstrings(_LANGUAGE)
except ImportError:
    print("Cannot find docsdict for", _LANGUAGE)
except:
    print ("Unknown Error when trying to import %s-docstring-dictionary" %
                                                                  _LANGUAGE)


def getmethparlist(ob):
    """Get strings describing the arguments for the given object

    Returns a pair of strings representing function parameter lists
    including parenthesis.  The first string is suitable for use in
    function definition and the second is suitable for use in function
    call.  The "self" parameter is not included.
    """
    defText = callText = ""
    # bit of a hack for methods - turn it into a function
    # but we drop the "self" param.
    # Try and build one for Python defined functions
    args, varargs, varkw = inspect.getargs(ob.__code__)
    items2 = args[1:]
    realArgs = args[1:]
    defaults = ob.__defaults__ or []
    defaults = ["=%r" % (value,) for value in defaults]
    defaults = [""] * (len(realArgs)-len(defaults)) + defaults
    items1 = [arg + dflt for arg, dflt in zip(realArgs, defaults)]
    if varargs is not None:
        items1.append("*" + varargs)
        items2.append("*" + varargs)
    if varkw is not None:
        items1.append("**" + varkw)
        items2.append("**" + varkw)
    defText = ", ".join(items1)
    defText = "(%s)" % defText
    callText = ", ".join(items2)
    callText = "(%s)" % callText
    return defText, callText

def _turtle_docrevise(docstr):
    """To reduce docstrings from RawTurtle class for functions
    """
    import re
    if docstr is None:
        return None
    turtlename = _CFG["exampleturtle"]
    newdocstr = docstr.replace("%s." % turtlename,"")
    parexp = re.compile(r' \(.+ %s\):' % turtlename)
    newdocstr = parexp.sub(":", newdocstr)
    return newdocstr

def _screen_docrevise(docstr):
    """To reduce docstrings from TurtleScreen class for functions
    """
    import re
    if docstr is None:
        return None
    screenname = _CFG["examplescreen"]
    newdocstr = docstr.replace("%s." % screenname,"")
    parexp = re.compile(r' \(.+ %s\):' % screenname)
    newdocstr = parexp.sub(":", newdocstr)
    return newdocstr

## The following mechanism makes all methods of RawTurtle and Turtle available
## as functions. So we can enhance, change, add, delete methods to these
## classes and do not need to change anything here.


for methodname in _tg_screen_functions:
    pl1, pl2 = getmethparlist(eval('_Screen.' + methodname))
    if pl1 == "":
        print(">>>>>>", pl1, pl2)
        continue
    defstr = ("def %(key)s%(pl1)s: return _getscreen().%(key)s%(pl2)s" %
                                   {'key':methodname, 'pl1':pl1, 'pl2':pl2})
    exec(defstr)
    eval(methodname).__doc__ = _screen_docrevise(eval('_Screen.'+methodname).__doc__)

for methodname in _tg_turtle_functions:
    pl1, pl2 = getmethparlist(eval('Turtle.' + methodname))
    if pl1 == "":
        print(">>>>>>", pl1, pl2)
        continue
    defstr = ("def %(key)s%(pl1)s: return _getpen().%(key)s%(pl2)s" %
                                   {'key':methodname, 'pl1':pl1, 'pl2':pl2})
    exec(defstr)
    eval(methodname).__doc__ = _turtle_docrevise(eval('Turtle.'+methodname).__doc__)


done = mainloop

if __name__ == "__main__":
    def switchpen():
        if isdown():
            pu()
        else:
            pd()

    def demo1():
        """Demo of old turtle.py - module"""
        reset()
        tracer(True)
        up()
        backward(100)
        down()
        # draw 3 squares; the last filled
        width(3)
        for i in range(3):
            if i == 2:
                begin_fill()
            for _ in range(4):
                forward(20)
                left(90)
            if i == 2:
                color("maroon")
                end_fill()
            up()
            forward(30)
            down()
        width(1)
        color("black")
        # move out of the way
        tracer(False)
        up()
        right(90)
        forward(100)
        right(90)
        forward(100)
        right(180)
        down()
        # some text
        write("startstart", 1)
        write("start", 1)
        color("red")
        # staircase
```

```python
        for i in range(5):
            forward(20)
            left(90)
            forward(20)
            right(90)
        # filled staircase
        tracer(True)
        begin_fill()
        for i in range(5):
            forward(20)
            left(90)
            forward(20)
            right(90)
        end_fill()
        # more text

    def demo2():
        """Demo of some new features."""
        speed(1)
        st()
        pensize(3)
        setheading(towards(0, 0))
        radius = distance(0, 0)/2.0
        rt(90)
        for _ in range(18):
            switchpen()
            circle(radius, 10)
        write("wait a moment...")
        while undobufferentries():
            undo()
        reset()
        lt(90)
        colormode(255)
        laenge = 10
        pencolor("green")
        pensize(3)
        lt(180)
        for i in range(-2, 16):
            if i > 0:
                begin_fill()
                fillcolor(255-15*i, 0, 15*i)
            for _ in range(3):
                fd(laenge)
                lt(120)
            end_fill()
            laenge += 10
            lt(15)
            speed((speed()+1)%12)
        #end_fill()

        lt(120)
        pu()
        fd(70)
        rt(30)
        pd()
        color("red","yellow")
        speed(0)
        begin_fill()
        for _ in range(4):
            circle(50, 90)
            rt(90)
            fd(30)
            rt(90)
        end_fill()
        lt(90)
        pu()
        fd(30)
        pd()
        shape("turtle")

        tri = getturtle()
        tri.resizemode("auto")
        turtle = Turtle()
        turtle.resizemode("auto")
        turtle.shape("turtle")
        turtle.reset()
        turtle.left(90)
        turtle.speed(0)
        turtle.up()
        turtle.goto(280, 40)
        turtle.lt(30)
        turtle.down()
        turtle.speed(6)
        turtle.color("blue","orange")
        turtle.pensize(2)
        tri.speed(6)
        setheading(towards(turtle))
        count = 1
        while tri.distance(turtle) > 4:
            turtle.fd(3.5)
            turtle.lt(0.6)
            tri.setheading(tri.towards(turtle))
            tri.fd(4)
            if count % 20 == 0:
                turtle.stamp()
                tri.stamp()
                switchpen()
            count += 1
        tri.write("CAUGHT! ", font=("Arial", 16, "bold"), align="right")
        tri.pencolor("black")
        tri.pencolor("red")

        def baba(xdummy, ydummy):
            clearscreen()
            bye()

        time.sleep(2)

        while undobufferentries():
            tri.undo()
            turtle.undo()
        tri.fd(50)
        tri.write("  Click me!", font = ("Courier", 12, "bold") )
        tri.onclick(baba, 1)

    demo1()
    demo2()
    exitonclick()

```