```
#
# 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',
```

```
120:         'clearscreen', 'colormode', 'delay', 'exitonclick', 'getcanvas',
121:         'getshapes', 'listen', 'mainloop', 'mode', 'numinput',
122:         'onkey', 'onkeypress', 'onkeyrelease', 'onscreenclick', 'ontimer',
123:         'register_shape', 'resetscreen', 'screensize', 'setup',
124:         'setworldcoordinates', 'textinput', 'title', 'tracer', 'turtles', 'update',
125:         'window_height', 'window_width']
126: _tg_turtle_functions = ['back', 'backward', 'begin_fill', 'begin_poly', 'bk',
127:         'circle', 'clear', 'clearstamp', 'clearstamps', 'clone', 'color',
128:         'degrees', 'distance', 'dot', 'down', 'end_fill', 'end_poly', 'fd',
129:         'fillcolor', 'filling', 'forward', 'get_poly', 'getpen', 'getscreen',
     'get_shapepoly',
130:         'getturtle', 'goto', 'heading', 'hideturtle', 'home', 'ht', 'isdown',
131:         'isvisible', 'left', 'lt', 'onclick', 'ondrag', 'onrelease', 'pd',
132:         'pen', 'pencolor', 'pendown', 'pensize', 'penup', 'pos', 'position',
133:         'pu', 'radians', 'right', 'reset', 'resizemode', 'rt',
134:         'seth', 'setheading', 'setpos', 'setposition', 'settiltangle',
135:         'setundobuffer', 'setx', 'sety', 'shape', 'shapesize', 'shapetransform',
     'shearfactor', 'showturtle',
136:         'speed', 'st', 'stamp', 'tilt', 'tiltangle', 'towards',
137:         'turtlesize', 'undo', 'undobufferentries', 'up', 'width',
138:         'write', 'xcor', 'ycor']
139: _tg_utilities = ['write_docstringdict', 'done']
140:
141: __all__ = (_tg_classes + _tg_screen_functions + _tg_turtle_functions +
142:           _tg_utilities) # + _math_functions)
143:
144: _alias_list = ['addshape', 'backward', 'bk', 'fd', 'ht', 'lt', 'pd', 'pos',
145:                'pu', 'rt', 'seth', 'setpos', 'setposition', 'st',
146:                'turtlesize', 'up', 'width']
147:
148: _CFG = {"width" : 0.5,               # Screen
149:         "height" : 0.75,
150:         "canvwidth" : 400,
151:         "canvheight": 300,
152:         "leftright": None,
153:         "topbottom": None,
154:         "mode": "standard",          # TurtleScreen
155:         "colormode": 1.0,
156:         "delay": 10,
157:         "undobuffersize": 1000,      # RawTurtle
158:         "shape": "classic",
159:         "pencolor" : "black",
160:         "fillcolor" : "black",
161:         "resizemode" : "noresize",
162:         "visible" : True,
163:         "language": "english",        # docstrings
164:         "exampleturtle": "turtle",
165:         "examplescreen": "screen",
166:         "title": "Python Turtle Graphics",
167:         "using_IDLE": False
168:        }
169:
170: def config_dict(filename):
171:     """Convert content of config-file into dictionary."""
172:     with open(filename, "r") as f:
173:         cfglines = f.readlines()
174:     cfgdict = {}
175:     for line in cfglines:
176:         line = line.strip()
177:         if not line or line.startswith("#"):
```

```
178:             continue
179:         try:
180:             key, value = line.split("=")
181:         except:
182:             print("Bad line in config-file %s:\n%s" % (filename,line))
183:             continue
184:         key = key.strip()
185:         value = value.strip()
186:         if value in ["True", "False", "None", "''", '""']:
187:             value = eval(value)
188:         else:
189:             try:
190:                 if "." in value:
191:                     value = float(value)
192:                 else:
193:                     value = int(value)
194:             except:
195:                 pass # value need not be converted
196:         cfgdict[key] = value
197:     return cfgdict
198: 
199: def readconfig(cfgdict):
200:     """Read config-files, change configuration-dict accordingly.
201: 
202:     If there is a turtle.cfg file in the current working directory,
203:     read it from there. If this contains an importconfig-value,
204:     say 'myway', construct filename turtle_mayway.cfg else use
205:     turtle.cfg and read it from the import-directory, where
206:     turtle.py is located.
207:     Update configuration dictionary first according to config-file,
208:     in the import directory, then according to config-file in the
209:     current working directory.
210:     If no config-file is found, the default configuration is used.
211:     """
212:     default_cfg = "turtle.cfg"
213:     cfgdict1 = {}
214:     cfgdict2 = {}
215:     if isfile(default_cfg):
216:         cfgdict1 = config_dict(default_cfg)
217:     if "importconfig" in cfgdict1:
218:         default_cfg = "turtle_%s.cfg" % cfgdict1["importconfig"]
219:     try:
220:         head, tail = split(__file__)
221:         cfg_file2 = join(head, default_cfg)
222:     except:
223:         cfg_file2 = ""
224:     if isfile(cfg_file2):
225:         cfgdict2 = config_dict(cfg_file2)
226:     _CFG.update(cfgdict2)
227:     _CFG.update(cfgdict1)
228: 
229: try:
230:     readconfig(_CFG)
231: except:
232:     print ("No configfile read, reason unknown")
233: 
234: 
235: class Vec2D(tuple):
236:     """A 2 dimensional vector class, used as a helper class
237:     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):
```

```
298:     """helper function for Scrolled Canvas"""
299:     _dict = {}
300:     __methodDict(cls, _dict)
301:     return _dict.keys()
302: 
303: __stringBody = (
304:     'def %(method)s(self, *args, **kw): return ' +
305:     'self.%(attribute)s.%(method)s(*args, **kw)')
306: 
307: def __forwardmethods(fromClass, toClass, toPart, exclude = ()):
308:     ### MANY CHANGES ###
309:     _dict_1 = {}
310:     __methodDict(toClass, _dict_1)
311:     _dict = {}
312:     mfc = __methods(fromClass)
313:     for ex in _dict_1.keys():
314:         if ex[:1] == '_' or ex[-1:] == '_' or ex in exclude or ex in mfc:
315:             pass
316:         else:
317:             _dict[ex] = _dict_1[ex]
318: 
319:     for method, func in _dict.items():
320:         d = {'method': method, 'func': func}
321:         if isinstance(toPart, str):
322:             execString = \
323:                 __stringBody % {'method' : method, 'attribute' : toPart}
324:         exec(execString, d)
325:         setattr(fromClass, method, d[method])   ### NEWU!
326: 
327: 
328: class ScrolledCanvas(TK.Frame):
329:     """Modeled after the scrolled canvas class from Grayons's Tkinter book.
330: 
331:     Used as the default canvas, which pops up automatically when
332:     using turtle graphics functions or the Turtle class.
333:     """
334:     def __init__(self, master, width=500, height=350,
335:                                           canvwidth=600, canvheight=500):
336:         TK.Frame.__init__(self, master, width=width, height=height)
337:         self._rootwindow = self.winfo_toplevel()
338:         self.width, self.height = width, height
339:         self.canvwidth, self.canvheight = canvwidth, canvheight
340:         self.bg = "white"
341:         self._canvas = TK.Canvas(master, width=width, height=height,
342:                                  bg=self.bg, relief=TK.SUNKEN, borderwidth=2)
343:         self.hscroll = TK.Scrollbar(master, command=self._canvas.xview,
344:                                     orient=TK.HORIZONTAL)
345:         self.vscroll = TK.Scrollbar(master, command=self._canvas.yview)
346:         self._canvas.configure(xscrollcommand=self.hscroll.set,
347:                                yscrollcommand=self.vscroll.set)
348:         self.rowconfigure(0, weight=1, minsize=0)
349:         self.columnconfigure(0, weight=1, minsize=0)
350:         self._canvas.grid(padx=1, in_ = self, pady=1, row=0,
351:                 column=0, rowspan=1, columnspan=1, sticky='news')
352:         self.vscroll.grid(padx=1, in_ = self, pady=1, row=0,
353:                 column=1, rowspan=1, columnspan=1, sticky='news')
354:         self.hscroll.grid(padx=1, in_ = self, pady=1, row=1,
355:                 column=0, rowspan=1, columnspan=1, sticky='news')
356:         self.reset()
357:         self._rootwindow.bind('<Configure>', self.onResize)
```

```
358:
359:     def reset(self, canvwidth=None, canvheight=None, bg = None):
360:         """Adjust canvas and scrollbars according to given canvas size."""
361:         if canvwidth:
362:             self.canvwidth = canvwidth
363:         if canvheight:
364:             self.canvheight = canvheight
365:         if bg:
366:             self.bg = bg
367:         self._canvas.config(bg=bg,
368:                         scrollregion=(-self.canvwidth//2, -self.canvheight//2,
369:                                       self.canvwidth//2, self.canvheight//2))
370:         self._canvas.xview_moveto(0.5*(self.canvwidth - self.width + 30) /
371:                                                                self.canvwidth)
372:         self._canvas.yview_moveto(0.5*(self.canvheight- self.height + 30) /
373:                                                              self.canvheight)
374:         self.adjustScrolls()
375:
376:
377:     def adjustScrolls(self):
378:         """ Adjust scrollbars according to window- and canvas-size.
379:         """
380:         cwidth = self._canvas.winfo_width()
381:         cheight = self._canvas.winfo_height()
382:         self._canvas.xview_moveto(0.5*(self.canvwidth-cwidth)/self.canvwidth)
383:         self._canvas.yview_moveto(0.5*(self.canvheight-cheight)/self.canvheight)
384:         if cwidth < self.canvwidth or cheight < self.canvheight:
385:             self.hscroll.grid(padx=1, in_ = self, pady=1, row=1,
386:                               column=0, rowspan=1, columnspan=1, sticky='news')
387:             self.vscroll.grid(padx=1, in_ = self, pady=1, row=0,
388:                               column=1, rowspan=1, columnspan=1, sticky='news')
389:         else:
390:             self.hscroll.grid_forget()
391:             self.vscroll.grid_forget()
392:
393:     def onResize(self, event):
394:         """self-explanatory"""
395:         self.adjustScrolls()
396:
397:     def bbox(self, *args):
398:         """ 'forward' method, which canvas itself has inherited...
399:         """
400:         return self._canvas.bbox(*args)
401:
402:     def cget(self, *args, **kwargs):
403:         """ 'forward' method, which canvas itself has inherited...
404:         """
405:         return self._canvas.cget(*args, **kwargs)
406:
407:     def config(self, *args, **kwargs):
408:         """ 'forward' method, which canvas itself has inherited...
409:         """
410:         self._canvas.config(*args, **kwargs)
411:
412:     def bind(self, *args, **kwargs):
413:         """ 'forward' method, which canvas itself has inherited...
414:         """
415:         self._canvas.bind(*args, **kwargs)
416:
417:     def unbind(self, *args, **kwargs):
```

```
418:         """ 'forward' method, which canvas itself has inherited...
419:         """
420:         self._canvas.unbind(*args, **kwargs)
421:
422:     def focus_force(self):
423:         """ 'forward' method, which canvas itself has inherited...
424:         """
425:         self._canvas.focus_force()
426:
427: __forwardmethods(ScrolledCanvas, TK.Canvas, '_canvas')
428:
429:
430: class _Root(TK.Tk):
431:     """Root class for Screen based on Tkinter."""
432:     def __init__(self):
433:         TK.Tk.__init__(self)
434:
435:     def setupcanvas(self, width, height, cwidth, cheight):
436:         self._canvas = ScrolledCanvas(self, width, height, cwidth, cheight)
437:         self._canvas.pack(expand=1, fill="both")
438:
439:     def _getcanvas(self):
440:         return self._canvas
441:
442:     def set_geometry(self, width, height, startx, starty):
443:         self.geometry("%dx%d%+d%+d"%(width, height, startx, starty))
444:
445:     def ondestroy(self, destroy):
446:         self.wm_protocol("WM_DELETE_WINDOW", destroy)
447:
448:     def win_width(self):
449:         return self.winfo_screenwidth()
450:
451:     def win_height(self):
452:         return self.winfo_screenheight()
453:
454: Canvas = TK.Canvas
455:
456:
457: class TurtleScreenBase(object):
458:     """Provide the basic graphics functionality.
459:        Interface between Tkinter and turtle.py.
460:
461:        To port turtle.py to some different graphics toolkit
462:        a corresponding TurtleScreenBase class has to be implemented.
463:     """
464:
465:     @staticmethod
466:     def _blankimage():
467:         """return a blank image object
468:         """
469:         img = TK.PhotoImage(width=1, height=1)
470:         img.blank()
471:         return img
472:
473:     @staticmethod
474:     def _image(filename):
475:         """return an image object containing the
476:         imagedata from a gif-file named filename.
477:         """
```

```
478:         return TK.PhotoImage(file=filename)
479:
480:     def __init__(self, cv):
481:         self.cv = cv
482:         if isinstance(cv, ScrolledCanvas):
483:             w = self.cv.canvwidth
484:             h = self.cv.canvheight
485:         else:  # expected: ordinary TK.Canvas
486:             w = int(self.cv.cget("width"))
487:             h = int(self.cv.cget("height"))
488:             self.cv.config(scrollregion = (-w//2, -h//2, w//2, h//2 ))
489:         self.canvwidth = w
490:         self.canvheight = h
491:         self.xscale = self.yscale = 1.0
492:
493:     def _createpoly(self):
494:         """Create an invisible polygon item on canvas self.cv)
495:         """
496:         return self.cv.create_polygon((0, 0, 0, 0, 0, 0), fill="", outline="")
497:
498:     def _drawpoly(self, polyitem, coordlist, fill=None,
499:                   outline=None, width=None, top=False):
500:         """Configure polygonitem polyitem according to provided
501:         arguments:
502:         coordlist is sequence of coordinates
503:         fill is filling color
504:         outline is outline color
505:         top is a boolean value, which specifies if polyitem
506:         will be put on top of the canvas' displaylist so it
507:         will not be covered by other items.
508:         """
509:         cl = []
510:         for x, y in coordlist:
511:             cl.append(x * self.xscale)
512:             cl.append(-y * self.yscale)
513:         self.cv.coords(polyitem, *cl)
514:         if fill is not None:
515:             self.cv.itemconfigure(polyitem, fill=fill)
516:         if outline is not None:
517:             self.cv.itemconfigure(polyitem, outline=outline)
518:         if width is not None:
519:             self.cv.itemconfigure(polyitem, width=width)
520:         if top:
521:             self.cv.tag_raise(polyitem)
522:
523:     def _createline(self):
524:         """Create an invisible line item on canvas self.cv)
525:         """
526:         return self.cv.create_line(0, 0, 0, 0, fill="", width=2,
527:                                    capstyle = TK.ROUND)
528:
529:     def _drawline(self, lineitem, coordlist=None,
530:                   fill=None, width=None, top=False):
531:         """Configure lineitem according to provided arguments:
532:         coordlist is sequence of coordinates
533:         fill is drawing color
534:         width is width of drawn line.
535:         top is a boolean value, which specifies if polyitem
536:         will be put on top of the canvas' displaylist so it
537:         will not be covered by other items.
```

```
538:         """
539:         if coordlist is not None:
540:             cl = []
541:             for x, y in coordlist:
542:                 cl.append(x * self.xscale)
543:                 cl.append(-y * self.yscale)
544:             self.cv.coords(lineitem, *cl)
545:         if fill is not None:
546:             self.cv.itemconfigure(lineitem, fill=fill)
547:         if width is not None:
548:             self.cv.itemconfigure(lineitem, width=width)
549:         if top:
550:             self.cv.tag_raise(lineitem)
551: 
552:     def _delete(self, item):
553:         """Delete graphics item from canvas.
554:         If item is"all" delete all graphics items.
555:         """
556:         self.cv.delete(item)
557: 
558:     def _update(self):
559:         """Redraw graphics items on canvas
560:         """
561:         self.cv.update()
562: 
563:     def _delay(self, delay):
564:         """Delay subsequent canvas actions for delay ms."""
565:         self.cv.after(delay)
566: 
567:     def _iscolorstring(self, color):
568:         """Check if the string color is a legal Tkinter color string.
569:         """
570:         try:
571:             rgb = self.cv.winfo_rgb(color)
572:             ok = True
573:         except TK.TclError:
574:             ok = False
575:         return ok
576: 
577:     def _bgcolor(self, color=None):
578:         """Set canvas' backgroundcolor if color is not None,
579:         else return backgroundcolor."""
580:         if color is not None:
581:             self.cv.config(bg = color)
582:             self._update()
583:         else:
584:             return self.cv.cget("bg")
585: 
586:     def _write(self, pos, txt, align, font, pencolor):
587:         """Write txt at pos in canvas with specified font
588:         and color.
589:         Return text item and x-coord of right bottom corner
590:         of text's bounding box."""
591:         x, y = pos
592:         x = x * self.xscale
593:         y = y * self.yscale
594:         anchor = {"left":"sw", "center":"s", "right":"se" }
595:         item = self.cv.create_text(x-1, -y, text = txt, anchor = anchor[align],
596:                                         fill = pencolor, font = font)
597:         x0, y0, x1, y1 = self.cv.bbox(item)
```

```
598:         self.cv.update()
599:         return item, x1-1
600:
601: ##    def _dot(self, pos, size, color):
602: ##        """may be implemented for some other graphics toolkit"""
603:
604:     def _onclick(self, item, fun, num=1, add=None):
605:         """Bind fun to mouse-click event on turtle.
606:         fun must be a function with two arguments, the coordinates
607:         of the clicked point on the canvas.
608:         num, the number of the mouse-button defaults to 1
609:         """
610:         if fun is None:
611:             self.cv.tag_unbind(item, "<Button-%s>" % num)
612:         else:
613:             def eventfun(event):
614:                 x, y = (self.cv.canvasx(event.x)/self.xscale,
615:                         -self.cv.canvasy(event.y)/self.yscale)
616:                 fun(x, y)
617:             self.cv.tag_bind(item, "<Button-%s>" % num, eventfun, add)
618:
619:     def _onrelease(self, item, fun, num=1, add=None):
620:         """Bind fun to mouse-button-release event on turtle.
621:         fun must be a function with two arguments, the coordinates
622:         of the point on the canvas where mouse button is released.
623:         num, the number of the mouse-button defaults to 1
624:
625:         If a turtle is clicked, first _onclick-event will be performed,
626:         then _onscreensclick-event.
627:         """
628:         if fun is None:
629:             self.cv.tag_unbind(item, "<Button%s-ButtonRelease>" % num)
630:         else:
631:             def eventfun(event):
632:                 x, y = (self.cv.canvasx(event.x)/self.xscale,
633:                         -self.cv.canvasy(event.y)/self.yscale)
634:                 fun(x, y)
635:             self.cv.tag_bind(item, "<Button%s-ButtonRelease>" % num,
636:                              eventfun, add)
637:
638:     def _ondrag(self, item, fun, num=1, add=None):
639:         """Bind fun to mouse-move-event (with pressed mouse button) on turtle.
640:         fun must be a function with two arguments, the coordinates of the
641:         actual mouse position on the canvas.
642:         num, the number of the mouse-button defaults to 1
643:
644:         Every sequence of mouse-move-events on a turtle is preceded by a
645:         mouse-click event on that turtle.
646:         """
647:         if fun is None:
648:             self.cv.tag_unbind(item, "<Button%s-Motion>" % num)
649:         else:
650:             def eventfun(event):
651:                 try:
652:                     x, y = (self.cv.canvasx(event.x)/self.xscale,
653:                             -self.cv.canvasy(event.y)/self.yscale)
654:                     fun(x, y)
655:                 except:
656:                     pass
657:             self.cv.tag_bind(item, "<Button%s-Motion>" % num, eventfun, add)
```

```
658:
659:     def _onscreenclick(self, fun, num=1, add=None):
660:         """Bind fun to mouse-click event on canvas.
661:         fun must be a function with two arguments, the coordinates
662:         of the clicked point on the canvas.
663:         num, the number of the mouse-button defaults to 1
664:
665:         If a turtle is clicked, first _onclick-event will be performed,
666:         then _onscreensclick-event.
667:         """
668:         if fun is None:
669:             self.cv.unbind("<Button-%s>" % num)
670:         else:
671:             def eventfun(event):
672:                 x, y = (self.cv.canvasx(event.x)/self.xscale,
673:                         -self.cv.canvasy(event.y)/self.yscale)
674:                 fun(x, y)
675:             self.cv.bind("<Button-%s>" % num, eventfun, add)
676:
677:     def _onkeyrelease(self, fun, key):
678:         """Bind fun to key-release event of key.
679:         Canvas must have focus. See method listen
680:         """
681:         if fun is None:
682:             self.cv.unbind("<KeyRelease-%s>" % key, None)
683:         else:
684:             def eventfun(event):
685:                 fun()
686:             self.cv.bind("<KeyRelease-%s>" % key, eventfun)
687:
688:     def _onkeypress(self, fun, key=None):
689:         """If key is given, bind fun to key-press event of key.
690:         Otherwise bind fun to any key-press.
691:         Canvas must have focus. See method listen.
692:         """
693:         if fun is None:
694:             if key is None:
695:                 self.cv.unbind("<KeyPress>", None)
696:             else:
697:                 self.cv.unbind("<KeyPress-%s>" % key, None)
698:         else:
699:             def eventfun(event):
700:                 fun()
701:             if key is None:
702:                 self.cv.bind("<KeyPress>", eventfun)
703:             else:
704:                 self.cv.bind("<KeyPress-%s>" % key, eventfun)
705:
706:     def _listen(self):
707:         """Set focus on canvas (in order to collect key-events)
708:         """
709:         self.cv.focus_force()
710:
711:     def _ontimer(self, fun, t):
712:         """Install a timer, which calls fun after t milliseconds.
713:         """
714:         if t == 0:
715:             self.cv.after_idle(fun)
716:         else:
717:             self.cv.after(t, fun)
```

```
718:
719:     def _createimage(self, image):
720:         """Create and return image item on canvas.
721:         """
722:         return self.cv.create_image(0, 0, image=image)
723:
724:     def _drawimage(self, item, pos, image):
725:         """Configure image item as to draw image object
726:         at position (x,y) on canvas)
727:         """
728:         x, y = pos
729:         self.cv.coords(item, (x * self.xscale, -y * self.yscale))
730:         self.cv.itemconfig(item, image=image)
731:
732:     def _setbgpic(self, item, image):
733:         """Configure image item as to draw image object
734:         at center of canvas. Set item to the first item
735:         in the displaylist, so it will be drawn below
736:         any other item ."""
737:         self.cv.itemconfig(item, image=image)
738:         self.cv.tag_lower(item)
739:
740:     def _type(self, item):
741:         """Return 'line' or 'polygon' or 'image' depending on
742:         type of item.
743:         """
744:         return self.cv.type(item)
745:
746:     def _pointlist(self, item):
747:         """returns list of coordinate-pairs of points of item
748:         Example (for insiders):
749:         >>> from turtle import *
750:         >>> getscreen()._pointlist(getturtle().turtle._item)
751:         [(0.0, 9.9999999999999982), (0.0, -9.9999999999999982),
752:         (9.9999999999999982, 0.0)]
753:         >>> """
754:         cl = self.cv.coords(item)
755:         pl = [(cl[i], -cl[i+1]) for i in range(0, len(cl), 2)]
756:         return  pl
757:
758:     def _setscrollregion(self, srx1, sry1, srx2, sry2):
759:         self.cv.config(scrollregion=(srx1, sry1, srx2, sry2))
760:
761:     def _rescale(self, xscalefactor, yscalefactor):
762:         items = self.cv.find_all()
763:         for item in items:
764:             coordinates = list(self.cv.coords(item))
765:             newcoordlist = []
766:             while coordinates:
767:                 x, y = coordinates[:2]
768:                 newcoordlist.append(x * xscalefactor)
769:                 newcoordlist.append(y * yscalefactor)
770:                 coordinates = coordinates[2:]
771:             self.cv.coords(item, *newcoordlist)
772:
773:     def _resize(self, canvwidth=None, canvheight=None, bg=None):
774:         """Resize the canvas the turtles are drawing on. Does
775:         not alter the drawing window.
776:         """
777:         # needs amendment
```

```
778:         if not isinstance(self.cv, ScrolledCanvas):
779:             return self.canvwidth, self.canvheight
780:         if canvwidth is canvheight is bg is None:
781:             return self.cv.canvwidth, self.cv.canvheight
782:         if canvwidth is not None:
783:             self.canvwidth = canvwidth
784:         if canvheight is not None:
785:             self.canvheight = canvheight
786:         self.cv.reset(canvwidth, canvheight, bg)
787: 
788:     def _window_size(self):
789:         """ Return the width and height of the turtle window.
790:         """
791:         width = self.cv.winfo_width()
792:         if width <= 1:  # the window isn't managed by a geometry manager
793:             width = self.cv['width']
794:         height = self.cv.winfo_height()
795:         if height <= 1: # the window isn't managed by a geometry manager
796:             height = self.cv['height']
797:         return width, height
798: 
799:     def mainloop(self):
800:         """Starts event loop - calling Tkinter's mainloop function.
801: 
802:         No argument.
803: 
804:         Must be last statement in a turtle graphics program.
805:         Must NOT be used if a script is run from within IDLE in -n mode
806:         (No subprocess) - for interactive use of turtle graphics.
807: 
808:         Example (for a TurtleScreen instance named screen):
809:         >>> screen.mainloop()
810: 
811:         """
812:         TK.mainloop()
813: 
814:     def textinput(self, title, prompt):
815:         """Pop up a dialog window for input of a string.
816: 
817:         Arguments: title is the title of the dialog window,
818:         prompt is a text mostly describing what information to input.
819: 
820:         Return the string input
821:         If the dialog is canceled, return None.
822: 
823:         Example (for a TurtleScreen instance named screen):
824:         >>> screen.textinput("NIM", "Name of first player:")
825: 
826:         """
827:         return simpledialog.askstring(title, prompt)
828: 
829:     def numinput(self, title, prompt, default=None, minval=None, maxval=None):
830:         """Pop up a dialog window for input of a number.
831: 
832:         Arguments: title is the title of the dialog window,
833:         prompt is a text mostly describing what numerical information to input.
834:         default: default value
835:         minval: minimum value for imput
836:         maxval: maximum value for input
837: 
```

```
838:         The number input must be in the range minval .. maxval if these are
839:         given. If not, a hint is issued and the dialog remains open for
840:         correction. Return the number input.
841:         If the dialog is canceled,  return None.
842:
843:         Example (for a TurtleScreen instance named screen):
844:         >>> screen.numinput("Poker", "Your stakes:", 1000, minval=10, maxval=10000)
845:
846:         """
847:         return simpledialog.askfloat(title, prompt, initialvalue=default,
848:                                      minvalue=minval, maxvalue=maxval)
849:
850:
851: ##############################################################################
852: ###                  End of Tkinter - interface                            ###
853: ##############################################################################
854:
855:
856: class Terminator (Exception):
857:     """Will be raised in TurtleScreen.update, if _RUNNING becomes False.
858:
859:     This stops execution of a turtle graphics script.
860:     Main purpose: use in the Demo-Viewer turtle.Demo.py.
861:     """
862:     pass
863:
864:
865: class TurtleGraphicsError(Exception):
866:     """Some TurtleGraphics Error
867:     """
868:
869:
870: class Shape(object):
871:     """Data structure modeling shapes.
872:
873:     attribute _type is one of "polygon", "image", "compound"
874:     attribute _data is - depending on _type a poygon-tuple,
875:     an image or a list constructed using the addcomponent method.
876:     """
877:     def __init__(self, type_, data=None):
878:         self._type = type_
879:         if type_ == "polygon":
880:             if isinstance(data, list):
881:                 data = tuple(data)
882:         elif type_ == "image":
883:             if isinstance(data, str):
884:                 if data.lower().endswith(".gif") and isfile(data):
885:                     data = TurtleScreen._image(data)
886:                 # else data assumed to be Photoimage
887:         elif type_ == "compound":
888:             data = []
889:         else:
890:             raise TurtleGraphicsError("There is no shape type %s" % type_)
891:         self._data = data
892:
893:     def addcomponent(self, poly, fill, outline=None):
894:         """Add component to a shape of type compound.
895:
896:         Arguments: poly is a polygon, i. e. a tuple of number pairs.
897:         fill is the fillcolor of the component,
```

```
898:         outline is the outline color of the component.
899:
900:         call (for a Shapeobject namend s):
901:         --   s.addcomponent(((0,0), (10,10), (-10,10)), "red", "blue")
902:
903:         Example:
904:         >>> poly = ((0,0),(10,-5),(0,10),(-10,-5))
905:         >>> s = Shape("compound")
906:         >>> s.addcomponent(poly, "red", "blue")
907:         >>> # .. add more components and then use register_shape()
908:         """
909:         if self._type != "compound":
910:             raise TurtleGraphicsError("Cannot add component to %s Shape"
911:                                                                 % self._type)
912:         if outline is None:
913:             outline = fill
914:         self._data.append([poly, fill, outline])
915:
916:
917: class Tbuffer(object):
918:     """Ring buffer used as undobuffer for RawTurtle objects."""
919:     def __init__(self, bufsize=10):
920:         self.bufsize = bufsize
921:         self.buffer = [[None]] * bufsize
922:         self.ptr = -1
923:         self.cumulate = False
924:     def reset(self, bufsize=None):
925:         if bufsize is None:
926:             for i in range(self.bufsize):
927:                 self.buffer[i] = [None]
928:         else:
929:             self.bufsize = bufsize
930:             self.buffer = [[None]] * bufsize
931:         self.ptr = -1
932:     def push(self, item):
933:         if self.bufsize > 0:
934:             if not self.cumulate:
935:                 self.ptr = (self.ptr + 1) % self.bufsize
936:                 self.buffer[self.ptr] = item
937:             else:
938:                 self.buffer[self.ptr].append(item)
939:     def pop(self):
940:         if self.bufsize > 0:
941:             item = self.buffer[self.ptr]
942:             if item is None:
943:                 return None
944:             else:
945:                 self.buffer[self.ptr] = [None]
946:                 self.ptr = (self.ptr - 1) % self.bufsize
947:                 return (item)
948:     def nr_of_items(self):
949:         return self.bufsize - self.buffer.count([None])
950:     def __repr__(self):
951:         return str(self.buffer) + " " + str(self.ptr)
952:
953:
954:
955: class TurtleScreen(TurtleScreenBase):
956:     """Provides screen oriented methods like setbg etc.
957:
```

```
    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")
```

```
1018:         for btn in 1, 2, 3:
1019:             self.onclick(None, btn)
1020:         self.onkeypress(None)
1021:         for key in self._keys[:]:
1022:             self.onkey(None, key)
1023:             self.onkeypress(None, key)
1024:         Turtle._pen = None
1025:
1026:     def mode(self, mode=None):
1027:         """Set turtle-mode ('standard', 'logo' or 'world') and perform reset.
1028:
1029:         Optional argument:
1030:         mode -- on of the strings 'standard', 'logo' or 'world'
1031:
1032:         Mode 'standard' is compatible with turtle.py.
1033:         Mode 'logo' is compatible with most Logo-Turtle-Graphics.
1034:         Mode 'world' uses userdefined 'worldcoordinates'. *Attention*: in
1035:         this mode angles appear distorted if x/y unit-ratio doesn't equal 1.
1036:         If mode is not given, return the current mode.
1037:
1038:              Mode      Initial turtle heading     positive angles
1039:          ------------|-------------------------|-------------------
1040:            'standard'    to the right (east)       counterclockwise
1041:              'logo'        upward    (north)         clockwise
1042:
1043:         Examples:
1044:         >>> mode('logo')   # resets turtle heading to north
1045:         >>> mode()
1046:         'logo'
1047:         """
1048:         if mode is None:
1049:             return self._mode
1050:         mode = mode.lower()
1051:         if mode not in ["standard", "logo", "world"]:
1052:             raise TurtleGraphicsError("No turtle-graphics-mode %s" % mode)
1053:         self._mode = mode
1054:         if mode in ["standard", "logo"]:
1055:             self._setscrollregion(-self.canvwidth//2, -self.canvheight//2,
1056:                                   self.canvwidth//2, self.canvheight//2)
1057:             self.xscale = self.yscale = 1.0
1058:         self.reset()
1059:
1060:     def setworldcoordinates(self, llx, lly, urx, ury):
1061:         """Set up a user defined coordinate-system.
1062:
1063:         Arguments:
1064:         llx -- a number, x-coordinate of lower left corner of canvas
1065:         lly -- a number, y-coordinate of lower left corner of canvas
1066:         urx -- a number, x-coordinate of upper right corner of canvas
1067:         ury -- a number, y-coordinate of upper right corner of canvas
1068:
1069:         Set up user coodinat-system and switch to mode 'world' if necessary.
1070:         This performs a screen.reset. If mode 'world' is already active,
1071:         all drawings are redrawn according to the new coordinates.
1072:
1073:         But ATTENTION: in user-defined coordinatesystems angles may appear
1074:         distorted. (see Screen.mode())
1075:
1076:         Example (for a TurtleScreen instance named screen):
1077:         >>> screen.setworldcoordinates(-10,-0.5,50,1.5)
```

```
1078:         >>> for _ in range(36):
1079:         ...     left(10)
1080:         ...     forward(0.5)
1081:         """
1082:         if self.mode() != "world":
1083:             self.mode("world")
1084:         xspan = float(urx - llx)
1085:         yspan = float(ury - lly)
1086:         wx, wy = self._window_size()
1087:         self.screensize(wx-20, wy-20)
1088:         oldxscale, oldyscale = self.xscale, self.yscale
1089:         self.xscale = self.canvwidth / xspan
1090:         self.yscale = self.canvheight / yspan
1091:         srx1 = llx * self.xscale
1092:         sry1 = -ury * self.yscale
1093:         srx2 = self.canvwidth + srx1
1094:         sry2 = self.canvheight + sry1
1095:         self._setscrollregion(srx1, sry1, srx2, sry2)
1096:         self._rescale(self.xscale/oldxscale, self.yscale/oldyscale)
1097:         self.update()
1098: 
1099:     def register_shape(self, name, shape=None):
1100:         """Adds a turtle shape to TurtleScreen's shapelist.
1101: 
1102:         Arguments:
1103:         (1) name is the name of a gif-file and shape is None.
1104:             Installs the corresponding image shape.
1105:             !! Image-shapes DO NOT rotate when turning the turtle,
1106:             !! so they do not display the heading of the turtle!
1107:         (2) name is an arbitrary string and shape is a tuple
1108:             of pairs of coordinates. Installs the corresponding
1109:             polygon shape
1110:         (3) name is an arbitrary string and shape is a
1111:             (compound) Shape object. Installs the corresponding
1112:             compound shape.
1113:         To use a shape, you have to issue the command shape(shapename).
1114: 
1115:         call: register_shape("turtle.gif")
1116:         --or: register_shape("tri", ((0,0), (10,10), (-10,10)))
1117: 
1118:         Example (for a TurtleScreen instance named screen):
1119:         >>> screen.register_shape("triangle", ((5,-3),(0,5),(-5,-3)))
1120: 
1121:         """
1122:         if shape is None:
1123:             # image
1124:             if name.lower().endswith(".gif"):
1125:                 shape = Shape("image", self._image(name))
1126:             else:
1127:                 raise TurtleGraphicsError("Bad arguments for register_shape.\n"
1128:                                           + "Use  help(register_shape)" )
1129:         elif isinstance(shape, tuple):
1130:             shape = Shape("polygon", shape)
1131:         ## else shape assumed to be Shape-instance
1132:         self._shapes[name] = shape
1133: 
1134:     def _colorstr(self, color):
1135:         """Return color string corresponding to args.
1136: 
1137:         Argument may be a string or a tuple of three
```

```
1138:         numbers corresponding to actual colormode,
1139:         i.e. in the range 0<=n<=colormode.
1140: 
1141:         If the argument doesn't represent a color,
1142:         an error is raised.
1143:         """
1144:         if len(color) == 1:
1145:             color = color[0]
1146:         if isinstance(color, str):
1147:             if self._iscolorstring(color) or color == "":
1148:                 return color
1149:             else:
1150:                 raise TurtleGraphicsError("bad color string: %s" % str(color))
1151:         try:
1152:             r, g, b = color
1153:         except:
1154:             raise TurtleGraphicsError("bad color arguments: %s" % str(color))
1155:         if self._colormode == 1.0:
1156:             r, g, b = [round(255.0*x) for x in (r, g, b)]
1157:         if not ((0 <= r <= 255) and (0 <= g <= 255) and (0 <= b <= 255)):
1158:             raise TurtleGraphicsError("bad color sequence: %s" % str(color))
1159:         return "#%02x%02x%02x" % (r, g, b)
1160: 
1161:     def _color(self, cstr):
1162:         if not cstr.startswith("#"):
1163:             return cstr
1164:         if len(cstr) == 7:
1165:             cl = [int(cstr[i:i+2], 16) for i in (1, 3, 5)]
1166:         elif len(cstr) == 4:
1167:             cl = [16*int(cstr[h], 16) for h in cstr[1:]]
1168:         else:
1169:             raise TurtleGraphicsError("bad colorstring: %s" % cstr)
1170:         return tuple([c * self._colormode/255 for c in cl])
1171: 
1172:     def colormode(self, cmode=None):
1173:         """Return the colormode or set it to 1.0 or 255.
1174: 
1175:         Optional argument:
1176:         cmode -- one of the values 1.0 or 255
1177: 
1178:         r, g, b values of colortriples have to be in range 0..cmode.
1179: 
1180:         Example (for a TurtleScreen instance named screen):
1181:         >>> screen.colormode()
1182:         1.0
1183:         >>> screen.colormode(255)
1184:         >>> pencolor(240,160,80)
1185:         """
1186:         if cmode is None:
1187:             return self._colormode
1188:         if cmode == 1.0:
1189:             self._colormode = float(cmode)
1190:         elif cmode == 255:
1191:             self._colormode = int(cmode)
1192: 
1193:     def reset(self):
1194:         """Reset all Turtles on the Screen to their initial state.
1195: 
1196:         No argument.
1197: 
```

```
1198:         Example (for a TurtleScreen instance named screen):
1199:         >>> screen.reset()
1200:         """
1201:         for turtle in self._turtles:
1202:             turtle._setmode(self._mode)
1203:             turtle.reset()
1204: 
1205:     def turtles(self):
1206:         """Return the list of turtles on the screen.
1207: 
1208:         Example (for a TurtleScreen instance named screen):
1209:         >>> screen.turtles()
1210:         [<turtle.Turtle object at 0x00E11FB0>]
1211:         """
1212:         return self._turtles
1213: 
1214:     def bgcolor(self, *args):
1215:         """Set or return backgroundcolor of the TurtleScreen.
1216: 
1217:         Arguments (if given): a color string or three numbers
1218:         in the range 0..colormode or a 3-tuple of such numbers.
1219: 
1220:         Example (for a TurtleScreen instance named screen):
1221:         >>> screen.bgcolor("orange")
1222:         >>> screen.bgcolor()
1223:         'orange'
1224:         >>> screen.bgcolor(0.5,0,0.5)
1225:         >>> screen.bgcolor()
1226:         '#800080'
1227:         """
1228:         if args:
1229:             color = self._colorstr(args)
1230:         else:
1231:             color = None
1232:         color = self._bgcolor(color)
1233:         if color is not None:
1234:             color = self._color(color)
1235:         return color
1236: 
1237:     def tracer(self, n=None, delay=None):
1238:         """Turns turtle animation on/off and set delay for update drawings.
1239: 
1240:         Optional arguments:
1241:         n -- nonnegative  integer
1242:         delay -- nonnegative  integer
1243: 
1244:         If n is given, only each n-th regular screen update is really performed.
1245:         (Can be used to accelerate the drawing of complex graphics.)
1246:         Second arguments sets delay value (see RawTurtle.delay())
1247: 
1248:         Example (for a TurtleScreen instance named screen):
1249:         >>> screen.tracer(8, 25)
1250:         >>> dist = 2
1251:         >>> for i in range(200):
1252:         ...     fd(dist)
1253:         ...     rt(90)
1254:         ...     dist += 2
1255:         """
1256:         if n is None:
1257:             return self._tracing
```

```
1258:         self._tracing = int(n)
1259:         self._updatecounter = 0
1260:         if delay is not None:
1261:             self._delayvalue = int(delay)
1262:         if self._tracing:
1263:             self.update()
1264: 
1265:     def delay(self, delay=None):
1266:         """ Return or set the drawing delay in milliseconds.
1267: 
1268:         Optional argument:
1269:         delay -- positive integer
1270: 
1271:         Example (for a TurtleScreen instance named screen):
1272:         >>> screen.delay(15)
1273:         >>> screen.delay()
1274:         15
1275:         """
1276:         if delay is None:
1277:             return self._delayvalue
1278:         self._delayvalue = int(delay)
1279: 
1280:     def _incrementudc(self):
1281:         """Increment update counter."""
1282:         if not TurtleScreen._RUNNING:
1283:             TurtleScreen._RUNNNING = True
1284:             raise Terminator
1285:         if self._tracing > 0:
1286:             self._updatecounter += 1
1287:             self._updatecounter %= self._tracing
1288: 
1289:     def update(self):
1290:         """Perform a TurtleScreen update.
1291:         """
1292:         tracing = self._tracing
1293:         self._tracing = True
1294:         for t in self.turtles():
1295:             t._update_data()
1296:             t._drawturtle()
1297:         self._tracing = tracing
1298:         self._update()
1299: 
1300:     def window_width(self):
1301:         """ Return the width of the turtle window.
1302: 
1303:         Example (for a TurtleScreen instance named screen):
1304:         >>> screen.window_width()
1305:         640
1306:         """
1307:         return self._window_size()[0]
1308: 
1309:     def window_height(self):
1310:         """ Return the height of the turtle window.
1311: 
1312:         Example (for a TurtleScreen instance named screen):
1313:         >>> screen.window_height()
1314:         480
1315:         """
1316:         return self._window_size()[1]
1317: 
```

```
1318:     def getcanvas(self):
1319:         """Return the Canvas of this TurtleScreen.
1320: 
1321:         No argument.
1322: 
1323:         Example (for a Screen instance named screen):
1324:         >>> cv = screen.getcanvas()
1325:         >>> cv
1326:         <turtle.ScrolledCanvas instance at 0x010742D8>
1327:         """
1328:         return self.cv
1329: 
1330:     def getshapes(self):
1331:         """Return a list of names of all currently available turtle shapes.
1332: 
1333:         No argument.
1334: 
1335:         Example (for a TurtleScreen instance named screen):
1336:         >>> screen.getshapes()
1337:         ['arrow', 'blank', 'circle', ... , 'turtle']
1338:         """
1339:         return sorted(self._shapes.keys())
1340: 
1341:     def onclick(self, fun, btn=1, add=None):
1342:         """Bind fun to mouse-click event on canvas.
1343: 
1344:         Arguments:
1345:         fun -- a function with two arguments, the coordinates of the
1346:                clicked point on the canvas.
1347:         num -- the number of the mouse-button, defaults to 1
1348: 
1349:         Example (for a TurtleScreen instance named screen)
1350: 
1351:         >>> screen.onclick(goto)
1352:         >>> # Subsequently clicking into the TurtleScreen will
1353:         >>> # make the turtle move to the clicked point.
1354:         >>> screen.onclick(None)
1355:         """
1356:         self._onscreenclick(fun, btn, add)
1357: 
1358:     def onkey(self, fun, key):
1359:         """Bind fun to key-release event of key.
1360: 
1361:         Arguments:
1362:         fun -- a function with no arguments
1363:         key -- a string: key (e.g. "a") or key-symbol (e.g. "space")
1364: 
1365:         In order to be able to register key-events, TurtleScreen
1366:         must have focus. (See method listen.)
1367: 
1368:         Example (for a TurtleScreen instance named screen):
1369: 
1370:         >>> def f():
1371:         ...     fd(50)
1372:         ...     lt(60)
1373:         ...
1374:         >>> screen.onkey(f, "Up")
1375:         >>> screen.listen()
1376: 
1377:         Subsequently the turtle can be moved by repeatedly pressing
```

```
1378:         the up-arrow key, consequently drawing a hexagon
1379:
1380:         """
1381:         if fun is None:
1382:             if key in self._keys:
1383:                 self._keys.remove(key)
1384:         elif key not in self._keys:
1385:             self._keys.append(key)
1386:         self._onkeyrelease(fun, key)
1387:
1388:     def onkeypress(self, fun, key=None):
1389:         """Bind fun to key-press event of key if key is given,
1390:         or to any key-press-event if no key is given.
1391:
1392:         Arguments:
1393:         fun -- a function with no arguments
1394:         key -- a string: key (e.g. "a") or key-symbol (e.g. "space")
1395:
1396:         In order to be able to register key-events, TurtleScreen
1397:         must have focus. (See method listen.)
1398:
1399:         Example (for a TurtleScreen instance named screen
1400:         and a Turtle instance named turtle):
1401:
1402:         >>> def f():
1403:         ...     fd(50)
1404:         ...     lt(60)
1405:         ...
1406:         >>> screen.onkeypress(f, "Up")
1407:         >>> screen.listen()
1408:
1409:         Subsequently the turtle can be moved by repeatedly pressing
1410:         the up-arrow key, or by keeping pressed the up-arrow key.
1411:         consequently drawing a hexagon.
1412:         """
1413:         if fun is None:
1414:             if key in self._keys:
1415:                 self._keys.remove(key)
1416:         elif key is not None and key not in self._keys:
1417:             self._keys.append(key)
1418:         self._onkeypress(fun, key)
1419:
1420:     def listen(self, xdummy=None, ydummy=None):
1421:         """Set focus on TurtleScreen (in order to collect key-events)
1422:
1423:         No arguments.
1424:         Dummy arguments are provided in order
1425:         to be able to pass listen to the onclick method.
1426:
1427:         Example (for a TurtleScreen instance named screen):
1428:         >>> screen.listen()
1429:         """
1430:         self._listen()
1431:
1432:     def ontimer(self, fun, t=0):
1433:         """Install a timer, which calls fun after t milliseconds.
1434:
1435:         Arguments:
1436:         fun -- a function with no arguments.
1437:         t -- a number >= 0
```

```
1438:
1439:         Example (for a TurtleScreen instance named screen):
1440:
1441:         >>> running = True
1442:         >>> def f():
1443:         ...     if running:
1444:         ...             fd(50)
1445:         ...             lt(60)
1446:         ...             screen.ontimer(f, 250)
1447:         ...
1448:         >>> f()   # makes the turtle marching around
1449:         >>> running = False
1450:         """
1451:         self._ontimer(fun, t)
1452:
1453:     def bgpic(self, picname=None):
1454:         """Set background image or return name of current backgroundimage.
1455:
1456:         Optional argument:
1457:         picname -- a string, name of a gif-file or "nopic".
1458:
1459:         If picname is a filename, set the corresponding image as background.
1460:         If picname is "nopic", delete backgroundimage, if present.
1461:         If picname is None, return the filename of the current backgroundimage.
1462:
1463:         Example (for a TurtleScreen instance named screen):
1464:         >>> screen.bgpic()
1465:         'nopic'
1466:         >>> screen.bgpic("landscape.gif")
1467:         >>> screen.bgpic()
1468:         'landscape.gif'
1469:         """
1470:         if picname is None:
1471:             return self._bgpicname
1472:         if picname not in self._bgpics:
1473:             self._bgpics[picname] = self._image(picname)
1474:         self._setbgpic(self._bgpic, self._bgpics[picname])
1475:         self._bgpicname = picname
1476:
1477:     def screensize(self, canvwidth=None, canvheight=None, bg=None):
1478:         """Resize the canvas the turtles are drawing on.
1479:
1480:         Optional arguments:
1481:         canvwidth -- positive integer, new width of canvas in pixels
1482:         canvheight --  positive integer, new height of canvas in pixels
1483:         bg -- colorstring or color-tuple, new backgroundcolor
1484:         If no arguments are given, return current (canvaswidth, canvasheight)
1485:
1486:         Do not alter the drawing window. To observe hidden parts of
1487:         the canvas use the scrollbars. (Can make visible those parts
1488:         of a drawing, which were outside the canvas before!)
1489:
1490:         Example (for a Turtle instance named turtle):
1491:         >>> turtle.screensize(2000,1500)
1492:         >>> # e.g. to search for an erroneously escaped turtle ;-)
1493:         """
1494:         return self._resize(canvwidth, canvheight, bg)
1495:
1496:     onscreenclick = onclick
1497:     resetscreen = reset
```

```
1498:     clearscreen = clear
1499:     addshape = register_shape
1500:     onkeyrelease = onkey
1501: 
1502: class TNavigator(object):
1503:     """Navigation part of the RawTurtle.
1504:     Implements methods for turtle movement.
1505:     """
1506:     START_ORIENTATION = {
1507:         "standard": Vec2D(1.0, 0.0),
1508:         "world"   : Vec2D(1.0, 0.0),
1509:         "logo"    : Vec2D(0.0, 1.0)  }
1510:     DEFAULT_MODE = "standard"
1511:     DEFAULT_ANGLEOFFSET = 0
1512:     DEFAULT_ANGLEORIENT = 1
1513: 
1514:     def __init__(self, mode=DEFAULT_MODE):
1515:         self._angleOffset = self.DEFAULT_ANGLEOFFSET
1516:         self._angleOrient = self.DEFAULT_ANGLEORIENT
1517:         self._mode = mode
1518:         self.undobuffer = None
1519:         self.degrees()
1520:         self._mode = None
1521:         self._setmode(mode)
1522:         TNavigator.reset(self)
1523: 
1524:     def reset(self):
1525:         """reset turtle to its initial values
1526: 
1527:         Will be overwritten by parent class
1528:         """
1529:         self._position = Vec2D(0.0, 0.0)
1530:         self._orient =  TNavigator.START_ORIENTATION[self._mode]
1531: 
1532:     def _setmode(self, mode=None):
1533:         """Set turtle-mode to 'standard', 'world' or 'logo'.
1534:         """
1535:         if mode is None:
1536:             return self._mode
1537:         if mode not in ["standard", "logo", "world"]:
1538:             return
1539:         self._mode = mode
1540:         if mode in ["standard", "world"]:
1541:             self._angleOffset = 0
1542:             self._angleOrient = 1
1543:         else: # mode == "logo":
1544:             self._angleOffset = self._fullcircle/4.
1545:             self._angleOrient = -1
1546: 
1547:     def _setDegreesPerAU(self, fullcircle):
1548:         """Helper function for degrees() and radians()"""
1549:         self._fullcircle = fullcircle
1550:         self._degreesPerAU = 360/fullcircle
1551:         if self._mode == "standard":
1552:             self._angleOffset = 0
1553:         else:
1554:             self._angleOffset = fullcircle/4.
1555: 
1556:     def degrees(self, fullcircle=360.0):
1557:         """ Set angle measurement units to degrees.
```

```
1558:
1559:         Optional argument:
1560:         fullcircle -  a number
1561:
1562:         Set angle measurement units, i. e. set number
1563:         of 'degrees' for a full circle. Dafault value is
1564:         360 degrees.
1565:
1566:         Example (for a Turtle instance named turtle):
1567:         >>> turtle.left(90)
1568:         >>> turtle.heading()
1569:         90
1570:
1571:         Change angle measurement unit to grad (also known as gon,
1572:         grade, or gradian and equals 1/100-th of the right angle.)
1573:         >>> turtle.degrees(400.0)
1574:         >>> turtle.heading()
1575:         100
1576:
1577:         """
1578:         self._setDegreesPerAU(fullcircle)
1579:
1580:     def radians(self):
1581:         """ Set the angle measurement units to radians.
1582:
1583:         No arguments.
1584:
1585:         Example (for a Turtle instance named turtle):
1586:         >>> turtle.heading()
1587:         90
1588:         >>> turtle.radians()
1589:         >>> turtle.heading()
1590:         1.5707963267948966
1591:         """
1592:         self._setDegreesPerAU(2*math.pi)
1593:
1594:     def _go(self, distance):
1595:         """move turtle forward by specified distance"""
1596:         ende = self._position + self._orient * distance
1597:         self._goto(ende)
1598:
1599:     def _rotate(self, angle):
1600:         """Turn turtle counterclockwise by specified angle if angle > 0."""
1601:         angle *= self._degreesPerAU
1602:         self._orient = self._orient.rotate(angle)
1603:
1604:     def _goto(self, end):
1605:         """move turtle to position end."""
1606:         self._position = end
1607:
1608:     def forward(self, distance):
1609:         """Move the turtle forward by the specified distance.
1610:
1611:         Aliases: forward | fd
1612:
1613:         Argument:
1614:         distance -- a number (integer or float)
1615:
1616:         Move the turtle forward by the specified distance, in the direction
1617:         the turtle is headed.
```

```
1618:
1619:         Example (for a Turtle instance named turtle):
1620:         >>> turtle.position()
1621:         (0.00, 0.00)
1622:         >>> turtle.forward(25)
1623:         >>> turtle.position()
1624:         (25.00,0.00)
1625:         >>> turtle.forward(-75)
1626:         >>> turtle.position()
1627:         (-50.00,0.00)
1628:         """
1629:         self._go(distance)
1630:
1631:     def back(self, distance):
1632:         """Move the turtle backward by distance.
1633:
1634:         Aliases: back | backward | bk
1635:
1636:         Argument:
1637:         distance -- a number
1638:
1639:         Move the turtle backward by distance ,opposite to the direction the
1640:         turtle is headed. Do not change the turtle's heading.
1641:
1642:         Example (for a Turtle instance named turtle):
1643:         >>> turtle.position()
1644:         (0.00, 0.00)
1645:         >>> turtle.backward(30)
1646:         >>> turtle.position()
1647:         (-30.00, 0.00)
1648:         """
1649:         self._go(-distance)
1650:
1651:     def right(self, angle):
1652:         """Turn turtle right by angle units.
1653:
1654:         Aliases: right | rt
1655:
1656:         Argument:
1657:         angle -- a number (integer or float)
1658:
1659:         Turn turtle right by angle units. (Units are by default degrees,
1660:         but can be set via the degrees() and radians() functions.)
1661:         Angle orientation depends on mode. (See this.)
1662:
1663:         Example (for a Turtle instance named turtle):
1664:         >>> turtle.heading()
1665:         22.0
1666:         >>> turtle.right(45)
1667:         >>> turtle.heading()
1668:         337.0
1669:         """
1670:         self._rotate(-angle)
1671:
1672:     def left(self, angle):
1673:         """Turn turtle left by angle units.
1674:
1675:         Aliases: left | lt
1676:
1677:         Argument:
```

```
1678:         angle -- a number (integer or float)
1679:
1680:         Turn turtle left by angle units. (Units are by default degrees,
1681:         but can be set via the degrees() and radians() functions.)
1682:         Angle orientation depends on mode. (See this.)
1683:
1684:         Example (for a Turtle instance named turtle):
1685:         >>> turtle.heading()
1686:         22.0
1687:         >>> turtle.left(45)
1688:         >>> turtle.heading()
1689:         67.0
1690:         """
1691:         self._rotate(angle)
1692:
1693:     def pos(self):
1694:         """Return the turtle's current location (x,y), as a Vec2D-vector.
1695:
1696:         Aliases: pos | position
1697:
1698:         No arguments.
1699:
1700:         Example (for a Turtle instance named turtle):
1701:         >>> turtle.pos()
1702:         (0.00, 240.00)
1703:         """
1704:         return self._position
1705:
1706:     def xcor(self):
1707:         """ Return the turtle's x coordinate.
1708:
1709:         No arguments.
1710:
1711:         Example (for a Turtle instance named turtle):
1712:         >>> reset()
1713:         >>> turtle.left(60)
1714:         >>> turtle.forward(100)
1715:         >>> print turtle.xcor()
1716:         50.0
1717:         """
1718:         return self._position[0]
1719:
1720:     def ycor(self):
1721:         """ Return the turtle's y coordinate
1722:         ---
1723:         No arguments.
1724:
1725:         Example (for a Turtle instance named turtle):
1726:         >>> reset()
1727:         >>> turtle.left(60)
1728:         >>> turtle.forward(100)
1729:         >>> print turtle.ycor()
1730:         86.6025403784
1731:         """
1732:         return self._position[1]
1733:
1734:
1735:     def goto(self, x, y=None):
1736:         """Move turtle to an absolute position.
1737:
```

```
1738:         Aliases: setpos | setposition | goto:
1739:
1740:         Arguments:
1741:         x -- a number      or     a pair/vector of numbers
1742:         y -- a number             None
1743:
1744:         call: goto(x, y)         # two coordinates
1745:         --or: goto((x, y))       # a pair (tuple) of coordinates
1746:         --or: goto(vec)          # e.g. as returned by pos()
1747:
1748:         Move turtle to an absolute position. If the pen is down,
1749:         a line will be drawn. The turtle's orientation does not change.
1750:
1751:         Example (for a Turtle instance named turtle):
1752:         >>> tp = turtle.pos()
1753:         >>> tp
1754:         (0.00, 0.00)
1755:         >>> turtle.setpos(60,30)
1756:         >>> turtle.pos()
1757:         (60.00,30.00)
1758:         >>> turtle.setpos((20,80))
1759:         >>> turtle.pos()
1760:         (20.00,80.00)
1761:         >>> turtle.setpos(tp)
1762:         >>> turtle.pos()
1763:         (0.00,0.00)
1764:         """
1765:         if y is None:
1766:             self._goto(Vec2D(*x))
1767:         else:
1768:             self._goto(Vec2D(x, y))
1769:
1770:     def home(self):
1771:         """Move turtle to the origin - coordinates (0,0).
1772:
1773:         No arguments.
1774:
1775:         Move turtle to the origin - coordinates (0,0) and set its
1776:         heading to its start-orientation (which depends on mode).
1777:
1778:         Example (for a Turtle instance named turtle):
1779:         >>> turtle.home()
1780:         """
1781:         self.goto(0, 0)
1782:         self.setheading(0)
1783:
1784:     def setx(self, x):
1785:         """Set the turtle's first coordinate to x
1786:
1787:         Argument:
1788:         x -- a number (integer or float)
1789:
1790:         Set the turtle's first coordinate to x, leave second coordinate
1791:         unchanged.
1792:
1793:         Example (for a Turtle instance named turtle):
1794:         >>> turtle.position()
1795:         (0.00, 240.00)
1796:         >>> turtle.setx(10)
1797:         >>> turtle.position()
```

```
1798:         (10.00, 240.00)
1799:         """
1800:         self._goto(Vec2D(x, self._position[1]))
1801:
1802:     def sety(self, y):
1803:         """Set the turtle's second coordinate to y
1804:
1805:         Argument:
1806:         y -- a number (integer or float)
1807:
1808:         Set the turtle's first coordinate to x, second coordinate remains
1809:         unchanged.
1810:
1811:         Example (for a Turtle instance named turtle):
1812:         >>> turtle.position()
1813:         (0.00, 40.00)
1814:         >>> turtle.sety(-10)
1815:         >>> turtle.position()
1816:         (0.00, -10.00)
1817:         """
1818:         self._goto(Vec2D(self._position[0], y))
1819:
1820:     def distance(self, x, y=None):
1821:         """Return the distance from the turtle to (x,y) in turtle step units.
1822:
1823:         Arguments:
1824:         x -- a number   or  a pair/vector of numbers   or   a turtle instance
1825:         y -- a number       None                           None
1826:
1827:         call: distance(x, y)         # two coordinates
1828:         --or: distance((x, y))       # a pair (tuple) of coordinates
1829:         --or: distance(vec)          # e.g. as returned by pos()
1830:         --or: distance(mypen)        # where mypen is another turtle
1831:
1832:         Example (for a Turtle instance named turtle):
1833:         >>> turtle.pos()
1834:         (0.00, 0.00)
1835:         >>> turtle.distance(30,40)
1836:         50.0
1837:         >>> pen = Turtle()
1838:         >>> pen.forward(77)
1839:         >>> turtle.distance(pen)
1840:         77.0
1841:         """
1842:         if y is not None:
1843:             pos = Vec2D(x, y)
1844:         if isinstance(x, Vec2D):
1845:             pos = x
1846:         elif isinstance(x, tuple):
1847:             pos = Vec2D(*x)
1848:         elif isinstance(x, TNavigator):
1849:             pos = x._position
1850:         return abs(pos - self._position)
1851:
1852:     def towards(self, x, y=None):
1853:         """Return the angle of the line from the turtle's position to (x, y).
1854:
1855:         Arguments:
1856:         x -- a number   or  a pair/vector of numbers   or   a turtle instance
1857:         y -- a number       None                           None
```

```
1858:
1859:         call: distance(x, y)         # two coordinates
1860:         --or: distance((x, y))       # a pair (tuple) of coordinates
1861:         --or: distance(vec)          # e.g. as returned by pos()
1862:         --or: distance(mypen)        # where mypen is another turtle
1863:
1864:         Return the angle, between the line from turtle-position to position
1865:         specified by x, y and the turtle's start orientation. (Depends on
1866:         modes - "standard" or "logo")
1867:
1868:         Example (for a Turtle instance named turtle):
1869:         >>> turtle.pos()
1870:         (10.00, 10.00)
1871:         >>> turtle.towards(0,0)
1872:         225.0
1873:         """
1874:         if y is not None:
1875:             pos = Vec2D(x, y)
1876:         if isinstance(x, Vec2D):
1877:             pos = x
1878:         elif isinstance(x, tuple):
1879:             pos = Vec2D(*x)
1880:         elif isinstance(x, TNavigator):
1881:             pos = x._position
1882:         x, y = pos - self._position
1883:         result = round(math.atan2(y, x)*180.0/math.pi, 10) % 360.0
1884:         result /= self._degreesPerAU
1885:         return (self._angleOffset + self._angleOrient*result) % self._fullcircle
1886:
1887:     def heading(self):
1888:         """ Return the turtle's current heading.
1889:
1890:         No arguments.
1891:
1892:         Example (for a Turtle instance named turtle):
1893:         >>> turtle.left(67)
1894:         >>> turtle.heading()
1895:         67.0
1896:         """
1897:         x, y = self._orient
1898:         result = round(math.atan2(y, x)*180.0/math.pi, 10) % 360.0
1899:         result /= self._degreesPerAU
1900:         return (self._angleOffset + self._angleOrient*result) % self._fullcircle
1901:
1902:     def setheading(self, to_angle):
1903:         """Set the orientation of the turtle to to_angle.
1904:
1905:         Aliases:  setheading | seth
1906:
1907:         Argument:
1908:         to_angle -- a number (integer or float)
1909:
1910:         Set the orientation of the turtle to to_angle.
1911:         Here are some common directions in degrees:
1912:
1913:          standard - mode:          logo-mode:
1914:         -------------------|--------------------
1915:            0 - east                0 - north
1916:           90 - north              90 - east
1917:          180 - west              180 - south
```

```
1918:          270 - south             270 - west
1919:
1920:         Example (for a Turtle instance named turtle):
1921:         >>> turtle.setheading(90)
1922:         >>> turtle.heading()
1923:         90
1924:         """
1925:         angle = (to_angle - self.heading())*self._angleOrient
1926:         full = self._fullcircle
1927:         angle = (angle+full/2.)%full - full/2.
1928:         self._rotate(angle)
1929:
1930:     def circle(self, radius, extent = None, steps = None):
1931:         """ Draw a circle with given radius.
1932:
1933:         Arguments:
1934:         radius -- a number
1935:         extent (optional) -- a number
1936:         steps (optional) -- an integer
1937:
1938:         Draw a circle with given radius. The center is radius units left
1939:         of the turtle; extent - an angle - determines which part of the
1940:         circle is drawn. If extent is not given, draw the entire circle.
1941:         If extent is not a full circle, one endpoint of the arc is the
1942:         current pen position. Draw the arc in counterclockwise direction
1943:         if radius is positive, otherwise in clockwise direction. Finally
1944:         the direction of the turtle is changed by the amount of extent.
1945:
1946:         As the circle is approximated by an inscribed regular polygon,
1947:         steps determines the number of steps to use. If not given,
1948:         it will be calculated automatically. Maybe used to draw regular
1949:         polygons.
1950:
1951:         call: circle(radius)                  # full circle
1952:         --or: circle(radius, extent)          # arc
1953:         --or: circle(radius, extent, steps)
1954:         --or: circle(radius, steps=6)         # 6-sided polygon
1955:
1956:         Example (for a Turtle instance named turtle):
1957:         >>> turtle.circle(50)
1958:         >>> turtle.circle(120, 180)  # semicircle
1959:         """
1960:         if self.undobuffer:
1961:             self.undobuffer.push(["seq"])
1962:             self.undobuffer.cumulate = True
1963:         speed = self.speed()
1964:         if extent is None:
1965:             extent = self._fullcircle
1966:         if steps is None:
1967:             frac = abs(extent)/self._fullcircle
1968:             steps = 1+int(min(11+abs(radius)/6.0, 59.0)*frac)
1969:         w = 1.0 * extent / steps
1970:         w2 = 0.5 * w
1971:         l = 2.0 * radius * math.sin(w2*math.pi/180.0*self._degreesPerAU)
1972:         if radius < 0:
1973:             l, w, w2 = -l, -w, -w2
1974:         tr = self._tracer()
1975:         dl = self._delay()
1976:         if speed == 0:
1977:             self._tracer(0, 0)
```

```
1978:         else:
1979:             self.speed(0)
1980:         self._rotate(w2)
1981:         for i in range(steps):
1982:             self.speed(speed)
1983:             self._go(l)
1984:             self.speed(0)
1985:             self._rotate(w)
1986:         self._rotate(-w2)
1987:         if speed == 0:
1988:             self._tracer(tr, dl)
1989:         self.speed(speed)
1990:         if self.undobuffer:
1991:             self.undobuffer.cumulate = False
1992:
1993: ## three dummy methods to be implemented by child class:
1994:
1995:     def speed(self, s=0):
1996:         """dummy method - to be overwritten by child class"""
1997:     def _tracer(self, a=None, b=None):
1998:         """dummy method - to be overwritten by child class"""
1999:     def _delay(self, n=None):
2000:         """dummy method - to be overwritten by child class"""
2001:
2002:     fd = forward
2003:     bk = back
2004:     backward = back
2005:     rt = right
2006:     lt = left
2007:     position = pos
2008:     setpos = goto
2009:     setposition = goto
2010:     seth = setheading
2011:
2012:
2013: class TPen(object):
2014:     """Drawing part of the RawTurtle.
2015:     Implements drawing properties.
2016:     """
2017:     def __init__(self, resizemode=_CFG["resizemode"]):
2018:         self._resizemode = resizemode # or "user" or "noresize"
2019:         self.undobuffer = None
2020:         TPen._reset(self)
2021:
2022:     def _reset(self, pencolor=_CFG["pencolor"],
2023:                      fillcolor=_CFG["fillcolor"]):
2024:         self._pensize = 1
2025:         self._shown = True
2026:         self._pencolor = pencolor
2027:         self._fillcolor = fillcolor
2028:         self._drawing = True
2029:         self._speed = 3
2030:         self._stretchfactor = (1., 1.)
2031:         self._shearfactor = 0.
2032:         self._tilt = 0.
2033:         self._shapetrafo = (1., 0., 0., 1.)
2034:         self._outlinewidth = 1
2035:
2036:     def resizemode(self, rmode=None):
2037:         """Set resizemode to one of the values: "auto", "user", "noresize".
```

```
2038:
2039:         (Optional) Argument:
2040:         rmode -- one of the strings "auto", "user", "noresize"
2041:
2042:         Different resizemodes have the following effects:
2043:           - "auto" adapts the appearance of the turtle
2044:                    corresponding to the value of pensize.
2045:           - "user" adapts the appearance of the turtle according to the
2046:                    values of stretchfactor and outlinewidth (outline),
2047:                    which are set by shapesize()
2048:           - "noresize" no adaption of the turtle's appearance takes place.
2049:         If no argument is given, return current resizemode.
2050:         resizemode("user") is called by a call of shapesize with arguments.
2051:
2052:
2053:         Examples (for a Turtle instance named turtle):
2054:         >>> turtle.resizemode("noresize")
2055:         >>> turtle.resizemode()
2056:         'noresize'
2057:         """
2058:         if rmode is None:
2059:             return self._resizemode
2060:         rmode = rmode.lower()
2061:         if rmode in ["auto", "user", "noresize"]:
2062:             self.pen(resizemode=rmode)
2063:
2064:     def pensize(self, width=None):
2065:         """Set or return the line thickness.
2066:
2067:         Aliases:  pensize | width
2068:
2069:         Argument:
2070:         width -- positive number
2071:
2072:         Set the line thickness to width or return it. If resizemode is set
2073:         to "auto" and turtleshape is a polygon, that polygon is drawn with
2074:         the same line thickness. If no argument is given, current pensize
2075:         is returned.
2076:
2077:         Example (for a Turtle instance named turtle):
2078:         >>> turtle.pensize()
2079:         1
2080:         >>> turtle.pensize(10)   # from here on lines of width 10 are drawn
2081:         """
2082:         if width is None:
2083:             return self._pensize
2084:         self.pen(pensize=width)
2085:
2086:
2087:     def penup(self):
2088:         """Pull the pen up -- no drawing when moving.
2089:
2090:         Aliases: penup | pu | up
2091:
2092:         No argument
2093:
2094:         Example (for a Turtle instance named turtle):
2095:         >>> turtle.penup()
2096:         """
2097:         if not self._drawing:
```

```
2098:             return
2099:         self.pen(pendown=False)
2100:
2101:     def pendown(self):
2102:         """Pull the pen down -- drawing when moving.
2103:
2104:         Aliases: pendown | pd | down
2105:
2106:         No argument.
2107:
2108:         Example (for a Turtle instance named turtle):
2109:         >>> turtle.pendown()
2110:         """
2111:         if self._drawing:
2112:             return
2113:         self.pen(pendown=True)
2114:
2115:     def isdown(self):
2116:         """Return True if pen is down, False if it's up.
2117:
2118:         No argument.
2119:
2120:         Example (for a Turtle instance named turtle):
2121:         >>> turtle.penup()
2122:         >>> turtle.isdown()
2123:         False
2124:         >>> turtle.pendown()
2125:         >>> turtle.isdown()
2126:         True
2127:         """
2128:         return self._drawing
2129:
2130:     def speed(self, speed=None):
2131:         """ Return or set the turtle's speed.
2132:
2133:         Optional argument:
2134:         speed -- an integer in the range 0..10 or a speedstring (see below)
2135:
2136:         Set the turtle's speed to an integer value in the range 0 .. 10.
2137:         If no argument is given: return current speed.
2138:
2139:         If input is a number greater than 10 or smaller than 0.5,
2140:         speed is set to 0.
2141:         Speedstrings  are mapped to speedvalues in the following way:
2142:             'fastest' :  0
2143:             'fast'    :  10
2144:             'normal'  :  6
2145:             'slow'    :  3
2146:             'slowest' :  1
2147:         speeds from 1 to 10 enforce increasingly faster animation of
2148:         line drawing and turtle turning.
2149:
2150:         Attention:
2151:         speed = 0 : *no* animation takes place. forward/back makes turtle jump
2152:         and likewise left/right make the turtle turn instantly.
2153:
2154:         Example (for a Turtle instance named turtle):
2155:         >>> turtle.speed(3)
2156:         """
2157:         speeds = {'fastest':0, 'fast':10, 'normal':6, 'slow':3, 'slowest':1 }
```

```
2158:         if speed is None:
2159:             return self._speed
2160:         if speed in speeds:
2161:             speed = speeds[speed]
2162:         elif 0.5 < speed < 10.5:
2163:             speed = int(round(speed))
2164:         else:
2165:             speed = 0
2166:         self.pen(speed=speed)
2167: 
2168:     def color(self, *args):
2169:         """Return or set the pencolor and fillcolor.
2170: 
2171:         Arguments:
2172:         Several input formats are allowed.
2173:         They use 0, 1, 2, or 3 arguments as follows:
2174: 
2175:         color()
2176:             Return the current pencolor and the current fillcolor
2177:             as a pair of color specification strings as are returned
2178:             by pencolor and fillcolor.
2179:         color(colorstring), color((r,g,b)), color(r,g,b)
2180:             inputs as in pencolor, set both, fillcolor and pencolor,
2181:             to the given value.
2182:         color(colorstring1, colorstring2),
2183:         color((r1,g1,b1), (r2,g2,b2))
2184:             equivalent to pencolor(colorstring1) and fillcolor(colorstring2)
2185:             and analogously, if the other input format is used.
2186: 
2187:         If turtleshape is a polygon, outline and interior of that polygon
2188:         is drawn with the newly set colors.
2189:         For mor info see: pencolor, fillcolor
2190: 
2191:         Example (for a Turtle instance named turtle):
2192:         >>> turtle.color('red', 'green')
2193:         >>> turtle.color()
2194:         ('red', 'green')
2195:         >>> colormode(255)
2196:         >>> color((40, 80, 120), (160, 200, 240))
2197:         >>> color()
2198:         ('#285078', '#a0c8f0')
2199:         """
2200:         if args:
2201:             l = len(args)
2202:             if l == 1:
2203:                 pcolor = fcolor = args[0]
2204:             elif l == 2:
2205:                 pcolor, fcolor = args
2206:             elif l == 3:
2207:                 pcolor = fcolor = args
2208:             pcolor = self._colorstr(pcolor)
2209:             fcolor = self._colorstr(fcolor)
2210:             self.pen(pencolor=pcolor, fillcolor=fcolor)
2211:         else:
2212:             return self._color(self._pencolor), self._color(self._fillcolor)
2213: 
2214:     def pencolor(self, *args):
2215:         """ Return or set the pencolor.
2216: 
2217:         Arguments:
```

```
2218:         Four input formats are allowed:
2219:           - pencolor()
2220:             Return the current pencolor as color specification string,
2221:             possibly in hex-number format (see example).
2222:             May be used as input to another color/pencolor/fillcolor call.
2223:           - pencolor(colorstring)
2224:             s is a Tk color specification string, such as "red" or "yellow"
2225:           - pencolor((r, g, b))
2226:             *a tuple* of r, g, and b, which represent, an RGB color,
2227:             and each of r, g, and b are in the range 0..colormode,
2228:             where colormode is either 1.0 or 255
2229:           - pencolor(r, g, b)
2230:             r, g, and b represent an RGB color, and each of r, g, and b
2231:             are in the range 0..colormode
2232:
2233:         If turtleshape is a polygon, the outline of that polygon is drawn
2234:         with the newly set pencolor.
2235:
2236:         Example (for a Turtle instance named turtle):
2237:         >>> turtle.pencolor('brown')
2238:         >>> tup = (0.2, 0.8, 0.55)
2239:         >>> turtle.pencolor(tup)
2240:         >>> turtle.pencolor()
2241:         '#33cc8c'
2242:         """
2243:         if args:
2244:             color = self._colorstr(args)
2245:             if color == self._pencolor:
2246:                 return
2247:             self.pen(pencolor=color)
2248:         else:
2249:             return self._color(self._pencolor)
2250:
2251:     def fillcolor(self, *args):
2252:         """ Return or set the fillcolor.
2253:
2254:         Arguments:
2255:         Four input formats are allowed:
2256:           - fillcolor()
2257:             Return the current fillcolor as color specification string,
2258:             possibly in hex-number format (see example).
2259:             May be used as input to another color/pencolor/fillcolor call.
2260:           - fillcolor(colorstring)
2261:             s is a Tk color specification string, such as "red" or "yellow"
2262:           - fillcolor((r, g, b))
2263:             *a tuple* of r, g, and b, which represent, an RGB color,
2264:             and each of r, g, and b are in the range 0..colormode,
2265:             where colormode is either 1.0 or 255
2266:           - fillcolor(r, g, b)
2267:             r, g, and b represent an RGB color, and each of r, g, and b
2268:             are in the range 0..colormode
2269:
2270:         If turtleshape is a polygon, the interior of that polygon is drawn
2271:         with the newly set fillcolor.
2272:
2273:         Example (for a Turtle instance named turtle):
2274:         >>> turtle.fillcolor('violet')
2275:         >>> col = turtle.pencolor()
2276:         >>> turtle.fillcolor(col)
2277:         >>> turtle.fillcolor(0, .5, 0)
```

```
2278:         """
2279:         if args:
2280:             color = self._colorstr(args)
2281:             if color == self._fillcolor:
2282:                 return
2283:             self.pen(fillcolor=color)
2284:         else:
2285:             return self._color(self._fillcolor)
2286:
2287:     def showturtle(self):
2288:         """Makes the turtle visible.
2289:
2290:         Aliases: showturtle | st
2291:
2292:         No argument.
2293:
2294:         Example (for a Turtle instance named turtle):
2295:         >>> turtle.hideturtle()
2296:         >>> turtle.showturtle()
2297:         """
2298:         self.pen(shown=True)
2299:
2300:     def hideturtle(self):
2301:         """Makes the turtle invisible.
2302:
2303:         Aliases: hideturtle | ht
2304:
2305:         No argument.
2306:
2307:         It's a good idea to do this while you're in the
2308:         middle of a complicated drawing, because hiding
2309:         the turtle speeds up the drawing observably.
2310:
2311:         Example (for a Turtle instance named turtle):
2312:         >>> turtle.hideturtle()
2313:         """
2314:         self.pen(shown=False)
2315:
2316:     def isvisible(self):
2317:         """Return True if the Turtle is shown, False if it's hidden.
2318:
2319:         No argument.
2320:
2321:         Example (for a Turtle instance named turtle):
2322:         >>> turtle.hideturtle()
2323:         >>> print turtle.isvisible():
2324:         False
2325:         """
2326:         return self._shown
2327:
2328:     def pen(self, pen=None, **pendict):
2329:         """Return or set the pen's attributes.
2330:
2331:         Arguments:
2332:             pen -- a dictionary with some or all of the below listed keys.
2333:             **pendict -- one or more keyword-arguments with the below
2334:                          listed keys as keywords.
2335:
2336:         Return or set the pen's attributes in a 'pen-dictionary'
2337:         with the following key/value pairs:
```

```
2338:            "shown"      :   True/False
2339:            "pendown"    :   True/False
2340:            "pencolor"   :   color-string or color-tuple
2341:            "fillcolor"  :   color-string or color-tuple
2342:            "pensize"    :   positive number
2343:            "speed"      :   number in range 0..10
2344:            "resizemode" :   "auto" or "user" or "noresize"
2345:            "stretchfactor": (positive number, positive number)
2346:            "shearfactor":   number
2347:            "outline"    :   positive number
2348:            "tilt"       :   number
2349:
2350:        This dictionary can be used as argument for a subsequent
2351:        pen()-call to restore the former pen-state. Moreover one
2352:        or more of these attributes can be provided as keyword-arguments.
2353:        This can be used to set several pen attributes in one statement.
2354:
2355:
2356:        Examples (for a Turtle instance named turtle):
2357:        >>> turtle.pen(fillcolor="black", pencolor="red", pensize=10)
2358:        >>> turtle.pen()
2359:        {'pensize': 10, 'shown': True, 'resizemode': 'auto', 'outline': 1,
2360:        'pencolor': 'red', 'pendown': True, 'fillcolor': 'black',
2361:        'stretchfactor': (1,1), 'speed': 3, 'shearfactor': 0.0}
2362:        >>> penstate=turtle.pen()
2363:        >>> turtle.color("yellow","")
2364:        >>> turtle.penup()
2365:        >>> turtle.pen()
2366:        {'pensize': 10, 'shown': True, 'resizemode': 'auto', 'outline': 1,
2367:        'pencolor': 'yellow', 'pendown': False, 'fillcolor': '',
2368:        'stretchfactor': (1,1), 'speed': 3, 'shearfactor': 0.0}
2369:        >>> p.pen(penstate, fillcolor="green")
2370:        >>> p.pen()
2371:        {'pensize': 10, 'shown': True, 'resizemode': 'auto', 'outline': 1,
2372:        'pencolor': 'red', 'pendown': True, 'fillcolor': 'green',
2373:        'stretchfactor': (1,1), 'speed': 3, 'shearfactor': 0.0}
2374:        """
2375:        _pd =  {"shown"         : self._shown,
2376:                "pendown"       : self._drawing,
2377:                "pencolor"      : self._pencolor,
2378:                "fillcolor"     : self._fillcolor,
2379:                "pensize"       : self._pensize,
2380:                "speed"         : self._speed,
2381:                "resizemode"    : self._resizemode,
2382:                "stretchfactor" : self._stretchfactor,
2383:                "shearfactor"   : self._shearfactor,
2384:                "outline"       : self._outlinewidth,
2385:                "tilt"          : self._tilt
2386:               }
2387:
2388:        if not (pen or pendict):
2389:            return _pd
2390:
2391:        if isinstance(pen, dict):
2392:            p = pen
2393:        else:
2394:            p = {}
2395:        p.update(pendict)
2396:
2397:        _p_buf = {}
```

```
2398:         for key in p:
2399:             _p_buf[key] = _pd[key]
2400:
2401:         if self.undobuffer:
2402:             self.undobuffer.push(("pen", _p_buf))
2403:
2404:         newLine = False
2405:         if "pendown" in p:
2406:             if self._drawing != p["pendown"]:
2407:                 newLine = True
2408:         if "pencolor" in p:
2409:             if isinstance(p["pencolor"], tuple):
2410:                 p["pencolor"] = self._colorstr((p["pencolor"],))
2411:             if self._pencolor != p["pencolor"]:
2412:                 newLine = True
2413:         if "pensize" in p:
2414:             if self._pensize != p["pensize"]:
2415:                 newLine = True
2416:         if newLine:
2417:             self._newLine()
2418:         if "pendown" in p:
2419:             self._drawing = p["pendown"]
2420:         if "pencolor" in p:
2421:             self._pencolor = p["pencolor"]
2422:         if "pensize" in p:
2423:             self._pensize = p["pensize"]
2424:         if "fillcolor" in p:
2425:             if isinstance(p["fillcolor"], tuple):
2426:                 p["fillcolor"] = self._colorstr((p["fillcolor"],))
2427:             self._fillcolor = p["fillcolor"]
2428:         if "speed" in p:
2429:             self._speed = p["speed"]
2430:         if "resizemode" in p:
2431:             self._resizemode = p["resizemode"]
2432:         if "stretchfactor" in p:
2433:             sf = p["stretchfactor"]
2434:             if isinstance(sf, (int, float)):
2435:                 sf = (sf, sf)
2436:             self._stretchfactor = sf
2437:         if "shearfactor" in p:
2438:             self._shearfactor = p["shearfactor"]
2439:         if "outline" in p:
2440:             self._outlinewidth = p["outline"]
2441:         if "shown" in p:
2442:             self._shown = p["shown"]
2443:         if "tilt" in p:
2444:             self._tilt = p["tilt"]
2445:         if "stretchfactor" in p or "tilt" in p or "shearfactor" in p:
2446:             scx, scy = self._stretchfactor
2447:             shf = self._shearfactor
2448:             sa, ca = math.sin(self._tilt), math.cos(self._tilt)
2449:             self._shapetrafo = ( scx*ca, scy*(shf*ca + sa),
2450:                                 -scx*sa, scy*(ca - shf*sa))
2451:         self._update()
2452:
2453: ## three dummy methods to be implemented by child class:
2454:
2455:     def _newLine(self, usePos = True):
2456:         """dummy method - to be overwritten by child class"""
2457:     def _update(self, count=True, forced=False):
```

```
2458:         """dummy method - to be overwritten by child class"""
2459:     def _color(self, args):
2460:         """dummy method - to be overwritten by child class"""
2461:     def _colorstr(self, args):
2462:         """dummy method - to be overwritten by child class"""
2463: 
2464:     width = pensize
2465:     up = penup
2466:     pu = penup
2467:     pd = pendown
2468:     down = pendown
2469:     st = showturtle
2470:     ht = hideturtle
2471: 
2472: 
2473: class _TurtleImage(object):
2474:     """Helper class: Datatype to store Turtle attributes
2475:     """
2476: 
2477:     def __init__(self, screen, shapeIndex):
2478:         self.screen = screen
2479:         self._type = None
2480:         self._setshape(shapeIndex)
2481: 
2482:     def _setshape(self, shapeIndex):
2483:         screen = self.screen
2484:         self.shapeIndex = shapeIndex
2485:         if self._type == "polygon" == screen._shapes[shapeIndex]._type:
2486:             return
2487:         if self._type == "image" == screen._shapes[shapeIndex]._type:
2488:             return
2489:         if self._type in ["image", "polygon"]:
2490:             screen._delete(self._item)
2491:         elif self._type == "compound":
2492:             for item in self._item:
2493:                 screen._delete(item)
2494:         self._type = screen._shapes[shapeIndex]._type
2495:         if self._type == "polygon":
2496:             self._item = screen._createpoly()
2497:         elif self._type == "image":
2498:             self._item = screen._createimage(screen._shapes["blank"]._data)
2499:         elif self._type == "compound":
2500:             self._item = [screen._createpoly() for item in
2501:                                           screen._shapes[shapeIndex]._data]
2502: 
2503: 
2504: class RawTurtle(TPen, TNavigator):
2505:     """Animation part of the RawTurtle.
2506:     Puts RawTurtle upon a TurtleScreen and provides tools for
2507:     its animation.
2508:     """
2509:     screens = []
2510: 
2511:     def __init__(self, canvas=None,
2512:                  shape=_CFG["shape"],
2513:                  undobuffersize=_CFG["undobuffersize"],
2514:                  visible=_CFG["visible"]):
2515:         if isinstance(canvas, _Screen):
2516:             self.screen = canvas
2517:         elif isinstance(canvas, TurtleScreen):
```

```
2518:             if canvas not in RawTurtle.screens:
2519:                 RawTurtle.screens.append(canvas)
2520:             self.screen = canvas
2521:         elif isinstance(canvas, (ScrolledCanvas, Canvas)):
2522:             for screen in RawTurtle.screens:
2523:                 if screen.cv == canvas:
2524:                     self.screen = screen
2525:                     break
2526:             else:
2527:                 self.screen = TurtleScreen(canvas)
2528:                 RawTurtle.screens.append(self.screen)
2529:         else:
2530:             raise TurtleGraphicsError("bad canvas argument %s" % canvas)
2531: 
2532:         screen = self.screen
2533:         TNavigator.__init__(self, screen.mode())
2534:         TPen.__init__(self)
2535:         screen._turtles.append(self)
2536:         self.drawingLineItem = screen._createline()
2537:         self.turtle = _TurtleImage(screen, shape)
2538:         self._poly = None
2539:         self._creatingPoly = False
2540:         self._fillitem = self._fillpath = None
2541:         self._shown = visible
2542:         self._hidden_from_screen = False
2543:         self.currentLineItem = screen._createline()
2544:         self.currentLine = [self._position]
2545:         self.items = [self.currentLineItem]
2546:         self.stampItems = []
2547:         self._undobuffersize = undobuffersize
2548:         self.undobuffer = Tbuffer(undobuffersize)
2549:         self._update()
2550: 
2551:     def reset(self):
2552:         """Delete the turtle's drawings and restore its default values.
2553: 
2554:         No argument.
2555: 
2556:         Delete the turtle's drawings from the screen, re-center the turtle
2557:         and set variables to the default values.
2558: 
2559:         Example (for a Turtle instance named turtle):
2560:         >>> turtle.position()
2561:         (0.00,-22.00)
2562:         >>> turtle.heading()
2563:         100.0
2564:         >>> turtle.reset()
2565:         >>> turtle.position()
2566:         (0.00,0.00)
2567:         >>> turtle.heading()
2568:         0.0
2569:         """
2570:         TNavigator.reset(self)
2571:         TPen._reset(self)
2572:         self._clear()
2573:         self._drawturtle()
2574:         self._update()
2575: 
2576:     def setundobuffer(self, size):
2577:         """Set or disable undobuffer.
```

```
2578:
2579:         Argument:
2580:         size -- an integer or None
2581:
2582:         If size is an integer an empty undobuffer of given size is installed.
2583:         Size gives the maximum number of turtle-actions that can be undone
2584:         by the undo() function.
2585:         If size is None, no undobuffer is present.
2586:
2587:         Example (for a Turtle instance named turtle):
2588:         >>> turtle.setundobuffer(42)
2589:         """
2590:         if size is None:
2591:             self.undobuffer = None
2592:         else:
2593:             self.undobuffer = Tbuffer(size)
2594:
2595:     def undobufferentries(self):
2596:         """Return count of entries in the undobuffer.
2597:
2598:         No argument.
2599:
2600:         Example (for a Turtle instance named turtle):
2601:         >>> while undobufferentries():
2602:         ...     undo()
2603:         """
2604:         if self.undobuffer is None:
2605:             return 0
2606:         return self.undobuffer.nr_of_items()
2607:
2608:     def _clear(self):
2609:         """Delete all of pen's drawings"""
2610:         self._fillitem = self._fillpath = None
2611:         for item in self.items:
2612:             self.screen._delete(item)
2613:         self.currentLineItem = self.screen._createline()
2614:         self.currentLine = []
2615:         if self._drawing:
2616:             self.currentLine.append(self._position)
2617:         self.items = [self.currentLineItem]
2618:         self.clearstamps()
2619:         self.setundobuffer(self._undobuffersize)
2620:
2621:
2622:     def clear(self):
2623:         """Delete the turtle's drawings from the screen. Do not move turtle.
2624:
2625:         No arguments.
2626:
2627:         Delete the turtle's drawings from the screen. Do not move turtle.
2628:         State and position of the turtle as well as drawings of other
2629:         turtles are not affected.
2630:
2631:         Examples (for a Turtle instance named turtle):
2632:         >>> turtle.clear()
2633:         """
2634:         self._clear()
2635:         self._update()
2636:
2637:     def _update_data(self):
```

```
2638:         self.screen._incrementudc()
2639:         if self.screen._updatecounter != 0:
2640:             return
2641:         if len(self.currentLine)>1:
2642:             self.screen._drawline(self.currentLineItem, self.currentLine,
2643:                                   self._pencolor, self._pensize)
2644: 
2645:     def _update(self):
2646:         """Perform a Turtle-data update.
2647:         """
2648:         screen = self.screen
2649:         if screen._tracing == 0:
2650:             return
2651:         elif screen._tracing == 1:
2652:             self._update_data()
2653:             self._drawturtle()
2654:             screen._update()                  # TurtleScreenBase
2655:             screen._delay(screen._delayvalue) # TurtleScreenBase
2656:         else:
2657:             self._update_data()
2658:             if screen._updatecounter == 0:
2659:                 for t in screen.turtles():
2660:                     t._drawturtle()
2661:                 screen._update()
2662: 
2663:     def _tracer(self, flag=None, delay=None):
2664:         """Turns turtle animation on/off and set delay for update drawings.
2665: 
2666:         Optional arguments:
2667:         n -- nonnegative  integer
2668:         delay -- nonnegative  integer
2669: 
2670:         If n is given, only each n-th regular screen update is really performed.
2671:         (Can be used to accelerate the drawing of complex graphics.)
2672:         Second arguments sets delay value (see RawTurtle.delay())
2673: 
2674:         Example (for a Turtle instance named turtle):
2675:         >>> turtle.tracer(8, 25)
2676:         >>> dist = 2
2677:         >>> for i in range(200):
2678:         ...     turtle.fd(dist)
2679:         ...     turtle.rt(90)
2680:         ...     dist += 2
2681:         """
2682:         return self.screen.tracer(flag, delay)
2683: 
2684:     def _color(self, args):
2685:         return self.screen._color(args)
2686: 
2687:     def _colorstr(self, args):
2688:         return self.screen._colorstr(args)
2689: 
2690:     def _cc(self, args):
2691:         """Convert colortriples to hexstrings.
2692:         """
2693:         if isinstance(args, str):
2694:             return args
2695:         try:
2696:             r, g, b = args
2697:         except:
```

```
2698:             raise TurtleGraphicsError("bad color arguments: %s" % str(args))
2699:         if self.screen._colormode == 1.0:
2700:             r, g, b = [round(255.0*x) for x in (r, g, b)]
2701:         if not ((0 <= r <= 255) and (0 <= g <= 255) and (0 <= b <= 255)):
2702:             raise TurtleGraphicsError("bad color sequence: %s" % str(args))
2703:         return "#%02x%02x%02x" % (r, g, b)
2704: 
2705:     def clone(self):
2706:         """Create and return a clone of the turtle.
2707: 
2708:         No argument.
2709: 
2710:         Create and return a clone of the turtle with same position, heading
2711:         and turtle properties.
2712: 
2713:         Example (for a Turtle instance named mick):
2714:         mick = Turtle()
2715:         joe = mick.clone()
2716:         """
2717:         screen = self.screen
2718:         self._newLine(self._drawing)
2719: 
2720:         turtle = self.turtle
2721:         self.screen = None
2722:         self.turtle = None  # too make self deepcopy-able
2723: 
2724:         q = deepcopy(self)
2725: 
2726:         self.screen = screen
2727:         self.turtle = turtle
2728: 
2729:         q.screen = screen
2730:         q.turtle = _TurtleImage(screen, self.turtle.shapeIndex)
2731: 
2732:         screen._turtles.append(q)
2733:         ttype = screen._shapes[self.turtle.shapeIndex]._type
2734:         if ttype == "polygon":
2735:             q.turtle._item = screen._createpoly()
2736:         elif ttype == "image":
2737:             q.turtle._item = screen._createimage(screen._shapes["blank"]._data)
2738:         elif ttype == "compound":
2739:             q.turtle._item = [screen._createpoly() for item in
2740:                               screen._shapes[self.turtle.shapeIndex]._data]
2741:         q.currentLineItem = screen._createline()
2742:         q._update()
2743:         return q
2744: 
2745:     def shape(self, name=None):
2746:         """Set turtle shape to shape with given name / return current shapename.
2747: 
2748:         Optional argument:
2749:         name -- a string, which is a valid shapename
2750: 
2751:         Set turtle shape to shape with given name or, if name is not given,
2752:         return name of current shape.
2753:         Shape with name must exist in the TurtleScreen's shape dictionary.
2754:         Initially there are the following polygon shapes:
2755:         'arrow', 'turtle', 'circle', 'square', 'triangle', 'classic'.
2756:         To learn about how to deal with shapes see Screen-method register_shape.
2757: 
```

```
2758:         Example (for a Turtle instance named turtle):
2759:         >>> turtle.shape()
2760:         'arrow'
2761:         >>> turtle.shape("turtle")
2762:         >>> turtle.shape()
2763:         'turtle'
2764:         """
2765:         if name is None:
2766:             return self.turtle.shapeIndex
2767:         if not name in self.screen.getshapes():
2768:             raise TurtleGraphicsError("There is no shape named %s" % name)
2769:         self.turtle._setshape(name)
2770:         self._update()
2771: 
2772:     def shapesize(self, stretch_wid=None, stretch_len=None, outline=None):
2773:         """Set/return turtle's stretchfactors/outline. Set resizemode to "user".
2774: 
2775:         Optional arguments:
2776:            stretch_wid : positive number
2777:            stretch_len : positive number
2778:            outline  : positive number
2779: 
2780:         Return or set the pen's attributes x/y-stretchfactors and/or outline.
2781:         Set resizemode to "user".
2782:         If and only if resizemode is set to "user", the turtle will be displayed
2783:         stretched according to its stretchfactors:
2784:         stretch_wid is stretchfactor perpendicular to orientation
2785:         stretch_len is stretchfactor in direction of turtles orientation.
2786:         outline determines the width of the shapes's outline.
2787: 
2788:         Examples (for a Turtle instance named turtle):
2789:         >>> turtle.resizemode("user")
2790:         >>> turtle.shapesize(5, 5, 12)
2791:         >>> turtle.shapesize(outline=8)
2792:         """
2793:         if stretch_wid is stretch_len is outline is None:
2794:             stretch_wid, stretch_len = self._stretchfactor
2795:             return stretch_wid, stretch_len, self._outlinewidth
2796:         if stretch_wid == 0 or stretch_len == 0:
2797:             raise TurtleGraphicsError("stretch_wid/stretch_len must not be zero")
2798:         if stretch_wid is not None:
2799:             if stretch_len is None:
2800:                 stretchfactor = stretch_wid, stretch_wid
2801:             else:
2802:                 stretchfactor = stretch_wid, stretch_len
2803:         elif stretch_len is not None:
2804:             stretchfactor = self._stretchfactor[0], stretch_len
2805:         else:
2806:             stretchfactor = self._stretchfactor
2807:         if outline is None:
2808:             outline = self._outlinewidth
2809:         self.pen(resizemode="user",
2810:                  stretchfactor=stretchfactor, outline=outline)
2811: 
2812:     def shearfactor(self, shear=None):
2813:         """Set or return the current shearfactor.
2814: 
2815:         Optional argument: shear -- number, tangent of the shear angle
2816: 
2817:         Shear the turtleshape according to the given shearfactor shear,
```

```
2818:         which is the tangent of the shear angle. DO NOT change the
2819:         turtle's heading (direction of movement).
2820:         If shear is not given: return the current shearfactor, i. e. the
2821:         tangent of the shear angle, by which lines parallel to the
2822:         heading of the turtle are sheared.
2823:
2824:         Examples (for a Turtle instance named turtle):
2825:         >>> turtle.shape("circle")
2826:         >>> turtle.shapesize(5,2)
2827:         >>> turtle.shearfactor(0.5)
2828:         >>> turtle.shearfactor()
2829:         >>> 0.5
2830:         """
2831:         if shear is None:
2832:             return self._shearfactor
2833:         self.pen(resizemode="user", shearfactor=shear)
2834:
2835:     def settiltangle(self, angle):
2836:         """Rotate the turtleshape to point in the specified direction
2837:
2838:         Argument: angle -- number
2839:
2840:         Rotate the turtleshape to point in the direction specified by angle,
2841:         regardless of its current tilt-angle. DO NOT change the turtle's
2842:         heading (direction of movement).
2843:
2844:
2845:         Examples (for a Turtle instance named turtle):
2846:         >>> turtle.shape("circle")
2847:         >>> turtle.shapesize(5,2)
2848:         >>> turtle.settiltangle(45)
2849:         >>> stamp()
2850:         >>> turtle.fd(50)
2851:         >>> turtle.settiltangle(-45)
2852:         >>> stamp()
2853:         >>> turtle.fd(50)
2854:         """
2855:         tilt = -angle * self._degreesPerAU * self._angleOrient
2856:         tilt = (tilt * math.pi / 180.0) % (2*math.pi)
2857:         self.pen(resizemode="user", tilt=tilt)
2858:
2859:     def tiltangle(self, angle=None):
2860:         """Set or return the current tilt-angle.
2861:
2862:         Optional argument: angle -- number
2863:
2864:         Rotate the turtleshape to point in the direction specified by angle,
2865:         regardless of its current tilt-angle. DO NOT change the turtle's
2866:         heading (direction of movement).
2867:         If angle is not given: return the current tilt-angle, i. e. the angle
2868:         between the orientation of the turtleshape and the heading of the
2869:         turtle (its direction of movement).
2870:
2871:         Deprecated since Python 3.1
2872:
2873:         Examples (for a Turtle instance named turtle):
2874:         >>> turtle.shape("circle")
2875:         >>> turtle.shapesize(5,2)
2876:         >>> turtle.tilt(45)
2877:         >>> turtle.tiltangle()
```

```
2878:         """
2879:         if angle is None:
2880:             tilt = -self._tilt * (180.0/math.pi) * self._angleOrient
2881:             return (tilt / self._degreesPerAU) % self._fullcircle
2882:         else:
2883:             self.settiltangle(angle)
2884: 
2885:     def tilt(self, angle):
2886:         """Rotate the turtleshape by angle.
2887: 
2888:         Argument:
2889:         angle - a number
2890: 
2891:         Rotate the turtleshape by angle from its current tilt-angle,
2892:         but do NOT change the turtle's heading (direction of movement).
2893: 
2894:         Examples (for a Turtle instance named turtle):
2895:         >>> turtle.shape("circle")
2896:         >>> turtle.shapesize(5,2)
2897:         >>> turtle.tilt(30)
2898:         >>> turtle.fd(50)
2899:         >>> turtle.tilt(30)
2900:         >>> turtle.fd(50)
2901:         """
2902:         self.settiltangle(angle + self.tiltangle())
2903: 
2904:     def shapetransform(self, t11=None, t12=None, t21=None, t22=None):
2905:         """Set or return the current transformation matrix of the turtle shape.
2906: 
2907:         Optional arguments: t11, t12, t21, t22 -- numbers.
2908: 
2909:         If none of the matrix elements are given, return the transformation
2910:         matrix.
2911:         Otherwise set the given elements and transform the turtleshape
2912:         according to the matrix consisting of first row t11, t12 and
2913:         second row t21, 22.
2914:         Modify stretchfactor, shearfactor and tiltangle according to the
2915:         given matrix.
2916: 
2917:         Examples (for a Turtle instance named turtle):
2918:         >>> turtle.shape("square")
2919:         >>> turtle.shapesize(4,2)
2920:         >>> turtle.shearfactor(-0.5)
2921:         >>> turtle.shapetransform()
2922:         (4.0, -1.0, -0.0, 2.0)
2923:         """
2924:         if t11 is t12 is t21 is t22 is None:
2925:             return self._shapetrafo
2926:         m11, m12, m21, m22 = self._shapetrafo
2927:         if t11 is not None: m11 = t11
2928:         if t12 is not None: m12 = t12
2929:         if t21 is not None: m21 = t21
2930:         if t22 is not None: m22 = t22
2931:         if t11 * t22 - t12 * t21 == 0:
2932:             raise TurtleGraphicsError("Bad shape transform matrix: must not be
      singular")
2933:         self._shapetrafo = (m11, m12, m21, m22)
2934:         alfa = math.atan2(-m21, m11) % (2 * math.pi)
2935:         sa, ca = math.sin(alfa), math.cos(alfa)
2936:         a11, a12, a21, a22 = (ca*m11 - sa*m21, ca*m12 - sa*m22,
```

```
2937:                                sa*m11 + ca*m21, sa*m12 + ca*m22)
2938:         self._stretchfactor = a11, a22
2939:         self._shearfactor = a12/a22
2940:         self._tilt = alfa
2941:         self._update()
2942:
2943:
2944:     def _polytrafo(self, poly):
2945:         """Computes transformed polygon shapes from a shape
2946:         according to current position and heading.
2947:         """
2948:         screen = self.screen
2949:         p0, p1 = self._position
2950:         e0, e1 = self._orient
2951:         e = Vec2D(e0, e1 * screen.yscale / screen.xscale)
2952:         e0, e1 = (1.0 / abs(e)) * e
2953:         return [(p0+(e1*x+e0*y)/screen.xscale, p1+(-e0*x+e1*y)/screen.yscale)
2954:                                                               for (x, y) in poly]
2955:
2956:     def get_shapepoly(self):
2957:         """Return the current shape polygon as tuple of coordinate pairs.
2958:
2959:         No argument.
2960:
2961:         Examples (for a Turtle instance named turtle):
2962:         >>> turtle.shape("square")
2963:         >>> turtle.shapetransform(4, -1, 0, 2)
2964:         >>> turtle.get_shapepoly()
2965:         ((50, -20), (30, 20), (-50, 20), (-30, -20))
2966:
2967:         """
2968:         shape = self.screen._shapes[self.turtle.shapeIndex]
2969:         if shape._type == "polygon":
2970:             return self._getshapepoly(shape._data, shape._type == "compound")
2971:         # else return None
2972:
2973:     def _getshapepoly(self, polygon, compound=False):
2974:         """Calculate transformed shape polygon according to resizemode
2975:         and shapetransform.
2976:         """
2977:         if self._resizemode == "user" or compound:
2978:             t11, t12, t21, t22 = self._shapetrafo
2979:         elif self._resizemode == "auto":
2980:             l = max(1, self._pensize/5.0)
2981:             t11, t12, t21, t22 = l, 0, 0, l
2982:         elif self._resizemode == "noresize":
2983:             return polygon
2984:         return tuple([(t11*x + t12*y, t21*x + t22*y) for (x, y) in polygon])
2985:
2986:     def _drawturtle(self):
2987:         """Manages the correct rendering of the turtle with respect to
2988:         its shape, resizemode, stretch and tilt etc."""
2989:         screen = self.screen
2990:         shape = screen._shapes[self.turtle.shapeIndex]
2991:         ttype = shape._type
2992:         titem = self.turtle._item
2993:         if self._shown and screen._updatecounter == 0 and screen._tracing > 0:
2994:             self._hidden_from_screen = False
2995:             tshape = shape._data
2996:             if ttype == "polygon":
```

```
2997:                 if self._resizemode == "noresize": w = 1
2998:                 elif self._resizemode == "auto": w = self._pensize
2999:                 else: w =self._outlinewidth
3000:                 shape = self._polytrafo(self._getshapepoly(tshape))
3001:                 fc, oc = self._fillcolor, self._pencolor
3002:                 screen._drawpoly(titem, shape, fill=fc, outline=oc,
3003:                                                       width=w, top=True)
3004:             elif ttype == "image":
3005:                 screen._drawimage(titem, self._position, tshape)
3006:             elif ttype == "compound":
3007:                 for item, (poly, fc, oc) in zip(titem, tshape):
3008:                     poly = self._polytrafo(self._getshapepoly(poly, True))
3009:                     screen._drawpoly(item, poly, fill=self._cc(fc),
3010:                                      outline=self._cc(oc), width=self._outlinewidth,
      top=True)
3011:         else:
3012:             if self._hidden_from_screen:
3013:                 return
3014:             if ttype == "polygon":
3015:                 screen._drawpoly(titem, ((0, 0), (0, 0), (0, 0)), "", "")
3016:             elif ttype == "image":
3017:                 screen._drawimage(titem, self._position,
3018:                                         screen._shapes["blank"]._data)
3019:             elif ttype == "compound":
3020:                 for item in titem:
3021:                     screen._drawpoly(item, ((0, 0), (0, 0), (0, 0)), "", "")
3022:             self._hidden_from_screen = True
3023:
3024: ##############################  stamp stuff  ###############################
3025:
3026:     def stamp(self):
3027:         """Stamp a copy of the turtleshape onto the canvas and return its id.
3028:
3029:         No argument.
3030:
3031:         Stamp a copy of the turtle shape onto the canvas at the current
3032:         turtle position. Return a stamp_id for that stamp, which can be
3033:         used to delete it by calling clearstamp(stamp_id).
3034:
3035:         Example (for a Turtle instance named turtle):
3036:         >>> turtle.color("blue")
3037:         >>> turtle.stamp()
3038:         13
3039:         >>> turtle.fd(50)
3040:         """
3041:         screen = self.screen
3042:         shape = screen._shapes[self.turtle.shapeIndex]
3043:         ttype = shape._type
3044:         tshape = shape._data
3045:         if ttype == "polygon":
3046:             stitem = screen._createpoly()
3047:             if self._resizemode == "noresize": w = 1
3048:             elif self._resizemode == "auto": w = self._pensize
3049:             else: w =self._outlinewidth
3050:             shape = self._polytrafo(self._getshapepoly(tshape))
3051:             fc, oc = self._fillcolor, self._pencolor
3052:             screen._drawpoly(stitem, shape, fill=fc, outline=oc,
3053:                                                   width=w, top=True)
3054:         elif ttype == "image":
3055:             stitem = screen._createimage("")
```

```
3056:             screen._drawimage(stitem, self._position, tshape)
3057:         elif ttype == "compound":
3058:             stitem = []
3059:             for element in tshape:
3060:                 item = screen._createpoly()
3061:                 stitem.append(item)
3062:             stitem = tuple(stitem)
3063:             for item, (poly, fc, oc) in zip(stitem, tshape):
3064:                 poly = self._polytrafo(self._getshapepoly(poly, True))
3065:                 screen._drawpoly(item, poly, fill=self._cc(fc),
3066:                                  outline=self._cc(oc), width=self._outlinewidth,
      top=True)
3067:         self.stampItems.append(stitem)
3068:         self.undobuffer.push(("stamp", stitem))
3069:         return stitem
3070:
3071:     def _clearstamp(self, stampid):
3072:         """does the work for clearstamp() and clearstamps()
3073:         """
3074:         if stampid in self.stampItems:
3075:             if isinstance(stampid, tuple):
3076:                 for subitem in stampid:
3077:                     self.screen._delete(subitem)
3078:             else:
3079:                 self.screen._delete(stampid)
3080:             self.stampItems.remove(stampid)
3081:         # Delete stampitem from undobuffer if necessary
3082:         # if clearstamp is called directly.
3083:         item = ("stamp", stampid)
3084:         buf = self.undobuffer
3085:         if item not in buf.buffer:
3086:             return
3087:         index = buf.buffer.index(item)
3088:         buf.buffer.remove(item)
3089:         if index <= buf.ptr:
3090:             buf.ptr = (buf.ptr - 1) % buf.bufsize
3091:         buf.buffer.insert((buf.ptr+1)%buf.bufsize, [None])
3092:
3093:     def clearstamp(self, stampid):
3094:         """Delete stamp with given stampid
3095:
3096:         Argument:
3097:         stampid - an integer, must be return value of previous stamp() call.
3098:
3099:         Example (for a Turtle instance named turtle):
3100:         >>> turtle.color("blue")
3101:         >>> astamp = turtle.stamp()
3102:         >>> turtle.fd(50)
3103:         >>> turtle.clearstamp(astamp)
3104:         """
3105:         self._clearstamp(stampid)
3106:         self._update()
3107:
3108:     def clearstamps(self, n=None):
3109:         """Delete all or first/last n of turtle's stamps.
3110:
3111:         Optional argument:
3112:         n -- an integer
3113:
3114:         If n is None, delete all of pen's stamps,
```

```
3115:         else if n > 0 delete first n stamps
3116:         else if n < 0 delete last n stamps.
3117:
3118:         Example (for a Turtle instance named turtle):
3119:         >>> for i in range(8):
3120:         ...     turtle.stamp(); turtle.fd(30)
3121:         ...
3122:         >>> turtle.clearstamps(2)
3123:         >>> turtle.clearstamps(-2)
3124:         >>> turtle.clearstamps()
3125:         """
3126:         if n is None:
3127:             toDelete = self.stampItems[:]
3128:         elif n >= 0:
3129:             toDelete = self.stampItems[:n]
3130:         else:
3131:             toDelete = self.stampItems[n:]
3132:         for item in toDelete:
3133:             self._clearstamp(item)
3134:         self._update()
3135:
3136:     def _goto(self, end):
3137:         """Move the pen to the point end, thereby drawing a line
3138:         if pen is down. All other methods for turtle movement depend
3139:         on this one.
3140:         """
3141:         ## Version with undo-stuff
3142:         go_modes = ( self._drawing,
3143:                      self._pencolor,
3144:                      self._pensize,
3145:                      isinstance(self._fillpath, list))
3146:         screen = self.screen
3147:         undo_entry = ("go", self._position, end, go_modes,
3148:                       (self.currentLineItem,
3149:                       self.currentLine[:],
3150:                       screen._pointlist(self.currentLineItem),
3151:                       self.items[:])
3152:                       )
3153:         if self.undobuffer:
3154:             self.undobuffer.push(undo_entry)
3155:         start = self._position
3156:         if self._speed and screen._tracing == 1:
3157:             diff = (end-start)
3158:             diffsq = (diff[0]*screen.xscale)**2 + (diff[1]*screen.yscale)**2
3159:             nhops = 1+int((diffsq**0.5)/(3*(1.1**self._speed)*self._speed))
3160:             delta = diff * (1.0/nhops)
3161:             for n in range(1, nhops):
3162:                 if n == 1:
3163:                     top = True
3164:                 else:
3165:                     top = False
3166:                 self._position = start + delta * n
3167:                 if self._drawing:
3168:                     screen._drawline(self.drawingLineItem,
3169:                                      (start, self._position),
3170:                                      self._pencolor, self._pensize, top)
3171:                 self._update()
3172:             if self._drawing:
3173:                 screen._drawline(self.drawingLineItem, ((0, 0), (0, 0)),
3174:                                                 fill="", width=self._pensize)
```

```
3175:         # Turtle now at end,
3176:         if self._drawing: # now update currentLine
3177:             self.currentLine.append(end)
3178:         if isinstance(self._fillpath, list):
3179:             self._fillpath.append(end)
3180:         ######    vererbung!!!!!!!!!!!!!!!!!!!!
3181:         self._position = end
3182:         if self._creatingPoly:
3183:             self._poly.append(end)
3184:         if len(self.currentLine) > 42: # 42! answer to the ultimate question
3185:                                        # of life, the universe and everything
3186:             self._newLine()
3187:         self._update() #count=True)
3188: 
3189:     def _undogoto(self, entry):
3190:         """Reverse a _goto. Used for undo()
3191:         """
3192:         old, new, go_modes, coodata = entry
3193:         drawing, pc, ps, filling = go_modes
3194:         cLI, cL, pl, items = coodata
3195:         screen = self.screen
3196:         if abs(self._position - new) > 0.5:
3197:             print ("undogoto: HALLO-DA-STIMMT-WAS-NICHT!")
3198:         # restore former situation
3199:         self.currentLineItem = cLI
3200:         self.currentLine = cL
3201: 
3202:         if pl == [(0, 0), (0, 0)]:
3203:             usepc = ""
3204:         else:
3205:             usepc = pc
3206:         screen._drawline(cLI, pl, fill=usepc, width=ps)
3207: 
3208:         todelete = [i for i in self.items if (i not in items) and
3209:                                    (screen._type(i) == "line")]
3210:         for i in todelete:
3211:             screen._delete(i)
3212:             self.items.remove(i)
3213: 
3214:         start = old
3215:         if self._speed and screen._tracing == 1:
3216:             diff = old - new
3217:             diffsq = (diff[0]*screen.xscale)**2 + (diff[1]*screen.yscale)**2
3218:             nhops = 1+int((diffsq**0.5)/(3*(1.1**self._speed)*self._speed))
3219:             delta = diff * (1.0/nhops)
3220:             for n in range(1, nhops):
3221:                 if n == 1:
3222:                     top = True
3223:                 else:
3224:                     top = False
3225:                 self._position = new + delta * n
3226:                 if drawing:
3227:                     screen._drawline(self.drawingLineItem,
3228:                                      (start, self._position),
3229:                                      pc, ps, top)
3230:                 self._update()
3231:             if drawing:
3232:                 screen._drawline(self.drawingLineItem, ((0, 0), (0, 0)),
3233:                                            fill="", width=ps)
3234:         # Turtle now at position old,
```

```
3235:         self._position = old
3236:         ##  if undo is done during creating a polygon, the last vertex
3237:         ##  will be deleted. if the polygon is entirely deleted,
3238:         ##  creatingPoly will be set to False.
3239:         ##  Polygons created before the last one will not be affected by undo()
3240:         if self._creatingPoly:
3241:             if len(self._poly) > 0:
3242:                 self._poly.pop()
3243:             if self._poly == []:
3244:                 self._creatingPoly = False
3245:                 self._poly = None
3246:         if filling:
3247:             if self._fillpath == []:
3248:                 self._fillpath = None
3249:                 print("Unwahrscheinlich in _undogoto!")
3250:             elif self._fillpath is not None:
3251:                 self._fillpath.pop()
3252:         self._update() #count=True)
3253: 
3254:     def _rotate(self, angle):
3255:         """Turns pen clockwise by angle.
3256:         """
3257:         if self.undobuffer:
3258:             self.undobuffer.push(("rot", angle, self._degreesPerAU))
3259:         angle *= self._degreesPerAU
3260:         neworient = self._orient.rotate(angle)
3261:         tracing = self.screen._tracing
3262:         if tracing == 1 and self._speed > 0:
3263:             anglevel = 3.0 * self._speed
3264:             steps = 1 + int(abs(angle)/anglevel)
3265:             delta = 1.0*angle/steps
3266:             for _ in range(steps):
3267:                 self._orient = self._orient.rotate(delta)
3268:                 self._update()
3269:         self._orient = neworient
3270:         self._update()
3271: 
3272:     def _newLine(self, usePos=True):
3273:         """Closes current line item and starts a new one.
3274:            Remark: if current line became too long, animation
3275:            performance (via _drawline) slowed down considerably.
3276:         """
3277:         if len(self.currentLine) > 1:
3278:             self.screen._drawline(self.currentLineItem, self.currentLine,
3279:                                   self._pencolor, self._pensize)
3280:             self.currentLineItem = self.screen._createline()
3281:             self.items.append(self.currentLineItem)
3282:         else:
3283:             self.screen._drawline(self.currentLineItem, top=True)
3284:         self.currentLine = []
3285:         if usePos:
3286:             self.currentLine = [self._position]
3287: 
3288:     def filling(self):
3289:         """Return fillstate (True if filling, False else).
3290: 
3291:         No argument.
3292: 
3293:         Example (for a Turtle instance named turtle):
3294:         >>> turtle.begin_fill()
```

```
3295:         >>> if turtle.filling():
3296:         ...     turtle.pensize(5)
3297:         ... else:
3298:         ...     turtle.pensize(3)
3299:         """
3300:         return isinstance(self._fillpath, list)
3301: 
3302:     def begin_fill(self):
3303:         """Called just before drawing a shape to be filled.
3304: 
3305:         No argument.
3306: 
3307:         Example (for a Turtle instance named turtle):
3308:         >>> turtle.color("black", "red")
3309:         >>> turtle.begin_fill()
3310:         >>> turtle.circle(60)
3311:         >>> turtle.end_fill()
3312:         """
3313:         if not self.filling():
3314:             self._fillitem = self.screen._createpoly()
3315:             self.items.append(self._fillitem)
3316:         self._fillpath = [self._position]
3317:         self._newLine()
3318:         if self.undobuffer:
3319:             self.undobuffer.push(("beginfill", self._fillitem))
3320:         self._update()
3321: 
3322: 
3323:     def end_fill(self):
3324:         """Fill the shape drawn after the call begin_fill().
3325: 
3326:         No argument.
3327: 
3328:         Example (for a Turtle instance named turtle):
3329:         >>> turtle.color("black", "red")
3330:         >>> turtle.begin_fill()
3331:         >>> turtle.circle(60)
3332:         >>> turtle.end_fill()
3333:         """
3334:         if self.filling():
3335:             if len(self._fillpath) > 2:
3336:                 self.screen._drawpoly(self._fillitem, self._fillpath,
3337:                                       fill=self._fillcolor)
3338:                 if self.undobuffer:
3339:                     self.undobuffer.push(("dofill", self._fillitem))
3340:             self._fillitem = self._fillpath = None
3341:             self._update()
3342: 
3343:     def dot(self, size=None, *color):
3344:         """Draw a dot with diameter size, using color.
3345: 
3346:         Optional arguments:
3347:         size -- an integer >= 1 (if given)
3348:         color -- a colorstring or a numeric color tuple
3349: 
3350:         Draw a circular dot with diameter size, using color.
3351:         If size is not given, the maximum of pensize+4 and 2*pensize is used.
3352: 
3353:         Example (for a Turtle instance named turtle):
3354:         >>> turtle.dot()
```

```
3355:         >>> turtle.fd(50); turtle.dot(20, "blue"); turtle.fd(50)
3356:         """
3357:         if not color:
3358:             if isinstance(size, (str, tuple)):
3359:                 color = self._colorstr(size)
3360:                 size = self._pensize + max(self._pensize, 4)
3361:             else:
3362:                 color = self._pencolor
3363:                 if not size:
3364:                     size = self._pensize + max(self._pensize, 4)
3365:         else:
3366:             if size is None:
3367:                 size = self._pensize + max(self._pensize, 4)
3368:             color = self._colorstr(color)
3369:         if hasattr(self.screen, "_dot"):
3370:             item = self.screen._dot(self._position, size, color)
3371:             self.items.append(item)
3372:             if self.undobuffer:
3373:                 self.undobuffer.push(("dot", item))
3374:         else:
3375:             pen = self.pen()
3376:             if self.undobuffer:
3377:                 self.undobuffer.push(["seq"])
3378:                 self.undobuffer.cumulate = True
3379:             try:
3380:                 if self.resizemode() == 'auto':
3381:                     self.ht()
3382:                 self.pendown()
3383:                 self.pensize(size)
3384:                 self.pencolor(color)
3385:                 self.forward(0)
3386:             finally:
3387:                 self.pen(pen)
3388:             if self.undobuffer:
3389:                 self.undobuffer.cumulate = False
3390: 
3391:     def _write(self, txt, align, font):
3392:         """Performs the writing for write()
3393:         """
3394:         item, end = self.screen._write(self._position, txt, align, font,
3395:                                                   self._pencolor)
3396:         self.items.append(item)
3397:         if self.undobuffer:
3398:             self.undobuffer.push(("wri", item))
3399:         return end
3400: 
3401:     def write(self, arg, move=False, align="left", font=("Arial", 8, "normal")):
3402:         """Write text at the current turtle position.
3403: 
3404:         Arguments:
3405:         arg -- info, which is to be written to the TurtleScreen
3406:         move (optional) -- True/False
3407:         align (optional) -- one of the strings "left", "center" or right"
3408:         font (optional) -- a triple (fontname, fontsize, fonttype)
3409: 
3410:         Write text - the string representation of arg - at the current
3411:         turtle position according to align ("left", "center" or right")
3412:         and with the given font.
3413:         If move is True, the pen is moved to the bottom-right corner
3414:         of the text. By default, move is False.
```

```
3415:
3416:         Example (for a Turtle instance named turtle):
3417:         >>> turtle.write('Home = ', True, align="center")
3418:         >>> turtle.write((0,0), True)
3419:         """
3420:         if self.undobuffer:
3421:             self.undobuffer.push(["seq"])
3422:             self.undobuffer.cumulate = True
3423:         end = self._write(str(arg), align.lower(), font)
3424:         if move:
3425:             x, y = self.pos()
3426:             self.setpos(end, y)
3427:         if self.undobuffer:
3428:             self.undobuffer.cumulate = False
3429:
3430:     def begin_poly(self):
3431:         """Start recording the vertices of a polygon.
3432:
3433:         No argument.
3434:
3435:         Start recording the vertices of a polygon. Current turtle position
3436:         is first point of polygon.
3437:
3438:         Example (for a Turtle instance named turtle):
3439:         >>> turtle.begin_poly()
3440:         """
3441:         self._poly = [self._position]
3442:         self._creatingPoly = True
3443:
3444:     def end_poly(self):
3445:         """Stop recording the vertices of a polygon.
3446:
3447:         No argument.
3448:
3449:         Stop recording the vertices of a polygon. Current turtle position is
3450:         last point of polygon. This will be connected with the first point.
3451:
3452:         Example (for a Turtle instance named turtle):
3453:         >>> turtle.end_poly()
3454:         """
3455:         self._creatingPoly = False
3456:
3457:     def get_poly(self):
3458:         """Return the lastly recorded polygon.
3459:
3460:         No argument.
3461:
3462:         Example (for a Turtle instance named turtle):
3463:         >>> p = turtle.get_poly()
3464:         >>> turtle.register_shape("myFavouriteShape", p)
3465:         """
3466:         ## check if there is any poly?
3467:         if self._poly is not None:
3468:             return tuple(self._poly)
3469:
3470:     def getscreen(self):
3471:         """Return the TurtleScreen object, the turtle is drawing  on.
3472:
3473:         No argument.
3474:
```

```
3475:         Return the TurtleScreen object, the turtle is drawing  on.
3476:         So TurtleScreen-methods can be called for that object.
3477: 
3478:         Example (for a Turtle instance named turtle):
3479:         >>> ts = turtle.getscreen()
3480:         >>> ts
3481:         <turtle.TurtleScreen object at 0x0106B770>
3482:         >>> ts.bgcolor("pink")
3483:         """
3484:         return self.screen
3485: 
3486:     def getturtle(self):
3487:         """Return the Turtleobject itself.
3488: 
3489:         No argument.
3490: 
3491:         Only reasonable use: as a function to return the 'anonymous turtle':
3492: 
3493:         Example:
3494:         >>> pet = getturtle()
3495:         >>> pet.fd(50)
3496:         >>> pet
3497:         <turtle.Turtle object at 0x0187D810>
3498:         >>> turtles()
3499:         [<turtle.Turtle object at 0x0187D810>]
3500:         """
3501:         return self
3502: 
3503:     getpen = getturtle
3504: 
3505: 
3506:     ################################################################
3507:     ### screen oriented methods recurring to methods of TurtleScreen
3508:     ################################################################
3509: 
3510:     def _delay(self, delay=None):
3511:         """Set delay value which determines speed of turtle animation.
3512:         """
3513:         return self.screen.delay(delay)
3514: 
3515:     def onclick(self, fun, btn=1, add=None):
3516:         """Bind fun to mouse-click event on this turtle on canvas.
3517: 
3518:         Arguments:
3519:         fun --  a function with two arguments, to which will be assigned
3520:                 the coordinates of the clicked point on the canvas.
3521:         num --  number of the mouse-button defaults to 1 (left mouse button).
3522:         add --  True or False. If True, new binding will be added, otherwise
3523:                 it will replace a former binding.
3524: 
3525:         Example for the anonymous turtle, i. e. the procedural way:
3526: 
3527:         >>> def turn(x, y):
3528:         ...     left(360)
3529:         ...
3530:         >>> onclick(turn)  # Now clicking into the turtle will turn it.
3531:         >>> onclick(None)  # event-binding will be removed
3532:         """
3533:         self.screen._onclick(self.turtle._item, fun, btn, add)
3534:         self._update()
```

```
3535:
3536:     def onrelease(self, fun, btn=1, add=None):
3537:         """Bind fun to mouse-button-release event on this turtle on canvas.
3538:
3539:         Arguments:
3540:         fun -- a function with two arguments, to which will be assigned
3541:                 the coordinates of the clicked point on the canvas.
3542:         num --  number of the mouse-button defaults to 1 (left mouse button).
3543:
3544:         Example (for a MyTurtle instance named joe):
3545:         >>> class MyTurtle(Turtle):
3546:         ...     def glow(self,x,y):
3547:         ...             self.fillcolor("red")
3548:         ...     def unglow(self,x,y):
3549:         ...             self.fillcolor("")
3550:         ...
3551:         >>> joe = MyTurtle()
3552:         >>> joe.onclick(joe.glow)
3553:         >>> joe.onrelease(joe.unglow)
3554:
3555:         Clicking on joe turns fillcolor red, unclicking turns it to
3556:         transparent.
3557:         """
3558:         self.screen._onrelease(self.turtle._item, fun, btn, add)
3559:         self._update()
3560:
3561:     def ondrag(self, fun, btn=1, add=None):
3562:         """Bind fun to mouse-move event on this turtle on canvas.
3563:
3564:         Arguments:
3565:         fun -- a function with two arguments, to which will be assigned
3566:                the coordinates of the clicked point on the canvas.
3567:         num -- number of the mouse-button defaults to 1 (left mouse button).
3568:
3569:         Every sequence of mouse-move-events on a turtle is preceded by a
3570:         mouse-click event on that turtle.
3571:
3572:         Example (for a Turtle instance named turtle):
3573:         >>> turtle.ondrag(turtle.goto)
3574:
3575:         Subsequently clicking and dragging a Turtle will move it
3576:         across the screen thereby producing handdrawings (if pen is
3577:         down).
3578:         """
3579:         self.screen._ondrag(self.turtle._item, fun, btn, add)
3580:
3581:
3582:     def _undo(self, action, data):
3583:         """Does the main part of the work for undo()
3584:         """
3585:         if self.undobuffer is None:
3586:             return
3587:         if action == "rot":
3588:             angle, degPAU = data
3589:             self._rotate(-angle*degPAU/self._degreesPerAU)
3590:             dummy = self.undobuffer.pop()
3591:         elif action == "stamp":
3592:             stitem = data[0]
3593:             self.clearstamp(stitem)
3594:         elif action == "go":
```

```
3595:             self._undogoto(data)
3596:         elif action in ["wri", "dot"]:
3597:             item = data[0]
3598:             self.screen._delete(item)
3599:             self.items.remove(item)
3600:         elif action == "dofill":
3601:             item = data[0]
3602:             self.screen._drawpoly(item, ((0, 0),(0, 0),(0, 0)),
3603:                                   fill="", outline="")
3604:         elif action == "beginfill":
3605:             item = data[0]
3606:             self._fillitem = self._fillpath = None
3607:             if item in self.items:
3608:                 self.screen._delete(item)
3609:                 self.items.remove(item)
3610:         elif action == "pen":
3611:             TPen.pen(self, data[0])
3612:             self.undobuffer.pop()
3613:
3614:     def undo(self):
3615:         """undo (repeatedly) the last turtle action.
3616:
3617:         No argument.
3618:
3619:         undo (repeatedly) the last turtle action.
3620:         Number of available undo actions is determined by the size of
3621:         the undobuffer.
3622:
3623:         Example (for a Turtle instance named turtle):
3624:         >>> for i in range(4):
3625:         ...     turtle.fd(50); turtle.lt(80)
3626:         ...
3627:         >>> for i in range(8):
3628:         ...     turtle.undo()
3629:         ...
3630:         """
3631:         if self.undobuffer is None:
3632:             return
3633:         item = self.undobuffer.pop()
3634:         action = item[0]
3635:         data = item[1:]
3636:         if action == "seq":
3637:             while data:
3638:                 item = data.pop()
3639:                 self._undo(item[0], item[1:])
3640:         else:
3641:             self._undo(action, data)
3642:
3643:     turtlesize = shapesize
3644:
3645: RawPen = RawTurtle
3646:
3647: ###  Screen - Singleton  ########################
3648:
3649: def Screen():
3650:     """Return the singleton screen object.
3651:     If none exists at the moment, create a new one and return it,
3652:     else return the existing one."""
3653:     if Turtle._screen is None:
3654:         Turtle._screen = _Screen()
```

```
3655:     return Turtle._screen
3656:
3657: class _Screen(TurtleScreen):
3658:
3659:     _root = None
3660:     _canvas = None
3661:     _title = _CFG["title"]
3662:
3663:     def __init__(self):
3664:         # XXX there is no need for this code to be conditional,
3665:         # as there will be only a single _Screen instance, anyway
3666:         # XXX actually, the turtle demo is injecting root window,
3667:         # so perhaps the conditional creation of a root should be
3668:         # preserved (perhaps by passing it as an optional parameter)
3669:         if _Screen._root is None:
3670:             _Screen._root = self._root = _Root()
3671:             self._root.title(_Screen._title)
3672:             self._root.ondestroy(self._destroy)
3673:         if _Screen._canvas is None:
3674:             width = _CFG["width"]
3675:             height = _CFG["height"]
3676:             canvwidth = _CFG["canvwidth"]
3677:             canvheight = _CFG["canvheight"]
3678:             leftright = _CFG["leftright"]
3679:             topbottom = _CFG["topbottom"]
3680:             self._root.setupcanvas(width, height, canvwidth, canvheight)
3681:             _Screen._canvas = self._root._getcanvas()
3682:             TurtleScreen.__init__(self, _Screen._canvas)
3683:             self.setup(width, height, leftright, topbottom)
3684:
3685:     def setup(self, width=_CFG["width"], height=_CFG["height"],
3686:               startx=_CFG["leftright"], starty=_CFG["topbottom"]):
3687:         """ Set the size and position of the main window.
3688:
3689:         Arguments:
3690:         width: as integer a size in pixels, as float a fraction of the screen.
3691:           Default is 50% of screen.
3692:         height: as integer the height in pixels, as float a fraction of the
3693:           screen. Default is 75% of screen.
3694:         startx: if positive, starting position in pixels from the left
3695:           edge of the screen, if negative from the right edge
3696:           Default, startx=None is to center window horizontally.
3697:         starty: if positive, starting position in pixels from the top
3698:           edge of the screen, if negative from the bottom edge
3699:           Default, starty=None is to center window vertically.
3700:
3701:         Examples (for a Screen instance named screen):
3702:         >>> screen.setup (width=200, height=200, startx=0, starty=0)
3703:
3704:         sets window to 200x200 pixels, in upper left of screen
3705:
3706:         >>> screen.setup(width=.75, height=0.5, startx=None, starty=None)
3707:
3708:         sets window to 75% of screen by 50% of screen and centers
3709:         """
3710:         if not hasattr(self._root, "set_geometry"):
3711:             return
3712:         sw = self._root.win_width()
3713:         sh = self._root.win_height()
3714:         if isinstance(width, float) and 0 <= width <= 1:
```

```
3715:             width = sw*width
3716:         if startx is None:
3717:             startx = (sw - width) / 2
3718:         if isinstance(height, float) and 0 <= height <= 1:
3719:             height = sh*height
3720:         if starty is None:
3721:             starty = (sh - height) / 2
3722:         self._root.set_geometry(width, height, startx, starty)
3723:         self.update()
3724: 
3725:     def title(self, titlestring):
3726:         """Set title of turtle-window
3727: 
3728:         Argument:
3729:         titlestring -- a string, to appear in the titlebar of the
3730:                        turtle graphics window.
3731: 
3732:         This is a method of Screen-class. Not available for TurtleScreen-
3733:         objects.
3734: 
3735:         Example (for a Screen instance named screen):
3736:         >>> screen.title("Welcome to the turtle-zoo!")
3737:         """
3738:         if _Screen._root is not None:
3739:             _Screen._root.title(titlestring)
3740:         _Screen._title = titlestring
3741: 
3742:     def _destroy(self):
3743:         root = self._root
3744:         if root is _Screen._root:
3745:             Turtle._pen = None
3746:             Turtle._screen = None
3747:             _Screen._root = None
3748:             _Screen._canvas = None
3749:         TurtleScreen._RUNNING = True
3750:         root.destroy()
3751: 
3752:     def bye(self):
3753:         """Shut the turtlegraphics window.
3754: 
3755:         Example (for a TurtleScreen instance named screen):
3756:         >>> screen.bye()
3757:         """
3758:         self._destroy()
3759: 
3760:     def exitonclick(self):
3761:         """Go into mainloop until the mouse is clicked.
3762: 
3763:         No arguments.
3764: 
3765:         Bind bye() method to mouseclick on TurtleScreen.
3766:         If "using_IDLE" - value in configuration dictionary is False
3767:         (default value), enter mainloop.
3768:         If IDLE with -n switch (no subprocess) is used, this value should be
3769:         set to True in turtle.cfg. In this case IDLE's mainloop
3770:         is active also for the client script.
3771: 
3772:         This is a method of the Screen-class and not available for
3773:         TurtleScreen instances.
3774: 
```

```
3775:         Example (for a Screen instance named screen):
3776:         >>> screen.exitonclick()
3777:
3778:         """
3779:         def exitGracefully(x, y):
3780:             """Screen.bye() with two dummy-parameters"""
3781:             self.bye()
3782:         self.onclick(exitGracefully)
3783:         if _CFG["using_IDLE"]:
3784:             return
3785:         try:
3786:             mainloop()
3787:         except AttributeError:
3788:             exit(0)
3789:
3790:
3791: class Turtle(RawTurtle):
3792:     """RawTurtle auto-creating (scrolled) canvas.
3793:
3794:     When a Turtle object is created or a function derived from some
3795:     Turtle method is called a TurtleScreen object is automatically created.
3796:     """
3797:     _pen = None
3798:     _screen = None
3799:
3800:     def __init__(self,
3801:                  shape=_CFG["shape"],
3802:                  undobuffersize=_CFG["undobuffersize"],
3803:                  visible=_CFG["visible"]):
3804:         if Turtle._screen is None:
3805:             Turtle._screen = Screen()
3806:         RawTurtle.__init__(self, Turtle._screen,
3807:                            shape=shape,
3808:                            undobuffersize=undobuffersize,
3809:                            visible=visible)
3810:
3811: Pen = Turtle
3812:
3813: def _getpen():
3814:     """Create the 'anonymous' turtle if not already present."""
3815:     if Turtle._pen is None:
3816:         Turtle._pen = Turtle()
3817:     return Turtle._pen
3818:
3819: def _getscreen():
3820:     """Create a TurtleScreen if not already present."""
3821:     if Turtle._screen is None:
3822:         Turtle._screen = Screen()
3823:     return Turtle._screen
3824:
3825: def write_docstringdict(filename="turtle_docstringdict"):
3826:     """Create and write docstring-dictionary to file.
3827:
3828:     Optional argument:
3829:     filename -- a string, used as filename
3830:                 default value is turtle_docstringdict
3831:
3832:     Has to be called explicitly, (not used by the turtle-graphics classes)
3833:     The docstring dictionary will be written to the Python script <filname>.py
3834:     It is intended to serve as a template for translation of the docstrings
```

```
3835:     into different languages.
3836:     """
3837:     docsdict = {}
3838: 
3839:     for methodname in _tg_screen_functions:
3840:         key = "_Screen."+methodname
3841:         docsdict[key] = eval(key).__doc__
3842:     for methodname in _tg_turtle_functions:
3843:         key = "Turtle."+methodname
3844:         docsdict[key] = eval(key).__doc__
3845: 
3846:     f = open("%s.py" % filename,"w")
3847:     keys = sorted([x for x in docsdict.keys()
3848:                           if x.split('.')[1] not in _alias_list])
3849:     f.write('docsdict = {\n\n')
3850:     for key in keys[:-1]:
3851:         f.write('%s :\n' % repr(key))
3852:         f.write('        """%s\n""",\n\n' % docsdict[key])
3853:     key = keys[-1]
3854:     f.write('%s :\n' % repr(key))
3855:     f.write('        """%s\n"""\n\n' % docsdict[key])
3856:     f.write("}\n")
3857:     f.close()
3858: 
3859: def read_docstrings(lang):
3860:     """Read in docstrings from lang-specific docstring dictionary.
3861: 
3862:     Transfer docstrings, translated to lang, from a dictionary-file
3863:     to the methods of classes Screen and Turtle and - in revised form -
3864:     to the corresponding functions.
3865:     """
3866:     modname = "turtle_docstringdict_%(language)s" % {'language':lang.lower()}
3867:     module = __import__(modname)
3868:     docsdict = module.docsdict
3869:     for key in docsdict:
3870:         try:
3871: #            eval(key).im_func.__doc__ = docsdict[key]
3872:             eval(key).__doc__ = docsdict[key]
3873:         except:
3874:             print("Bad docstring-entry: %s" % key)
3875: 
3876: _LANGUAGE = _CFG["language"]
3877: 
3878: try:
3879:     if _LANGUAGE != "english":
3880:         read_docstrings(_LANGUAGE)
3881: except ImportError:
3882:     print("Cannot find docsdict for", _LANGUAGE)
3883: except:
3884:     print ("Unknown Error when trying to import %s-docstring-dictionary" %
3885:                                                                  _LANGUAGE)
3886: 
3887: 
3888: def getmethparlist(ob):
3889:     """Get strings describing the arguments for the given object
3890: 
3891:     Returns a pair of strings representing function parameter lists
3892:     including parenthesis.  The first string is suitable for use in
3893:     function definition and the second is suitable for use in function
3894:     call.  The "self" parameter is not included.
```

```
3895:     """
3896:     defText = callText = ""
3897:     # bit of a hack for methods - turn it into a function
3898:     # but we drop the "self" param.
3899:     # Try and build one for Python defined functions
3900:     args, varargs, varkw = inspect.getargs(ob.__code__)
3901:     items2 = args[1:]
3902:     realArgs = args[1:]
3903:     defaults = ob.__defaults__ or []
3904:     defaults = ["=%r" % (value,) for value in defaults]
3905:     defaults = [""] * (len(realArgs)-len(defaults)) + defaults
3906:     items1 = [arg + dflt for arg, dflt in zip(realArgs, defaults)]
3907:     if varargs is not None:
3908:         items1.append("*" + varargs)
3909:         items2.append("*" + varargs)
3910:     if varkw is not None:
3911:         items1.append("**" + varkw)
3912:         items2.append("**" + varkw)
3913:     defText = ", ".join(items1)
3914:     defText = "(%s)" % defText
3915:     callText = ", ".join(items2)
3916:     callText = "(%s)" % callText
3917:     return defText, callText
3918:
3919: def _turtle_docrevise(docstr):
3920:     """To reduce docstrings from RawTurtle class for functions
3921:     """
3922:     import re
3923:     if docstr is None:
3924:         return None
3925:     turtlename = _CFG["exampleturtle"]
3926:     newdocstr = docstr.replace("%s." % turtlename,"")
3927:     parexp = re.compile(r' \(.+ %s\):' % turtlename)
3928:     newdocstr = parexp.sub(":", newdocstr)
3929:     return newdocstr
3930:
3931: def _screen_docrevise(docstr):
3932:     """To reduce docstrings from TurtleScreen class for functions
3933:     """
3934:     import re
3935:     if docstr is None:
3936:         return None
3937:     screenname = _CFG["examplescreen"]
3938:     newdocstr = docstr.replace("%s." % screenname,"")
3939:     parexp = re.compile(r' \(.+ %s\):' % screenname)
3940:     newdocstr = parexp.sub(":", newdocstr)
3941:     return newdocstr
3942:
3943: ## The following mechanism makes all methods of RawTurtle and Turtle available
3944: ## as functions. So we can enhance, change, add, delete methods to these
3945: ## classes and do not need to change anything here.
3946:
3947:
3948: for methodname in _tg_screen_functions:
3949:     pl1, pl2 = getmethparlist(eval('_Screen.' + methodname))
3950:     if pl1 == "":
3951:         print(">>>>>>", pl1, pl2)
3952:         continue
3953:     defstr = ("def %(key)s%(pl1)s: return _getscreen().%(key)s%(pl2)s" %
3954:                                     {'key':methodname, 'pl1':pl1, 'pl2':pl2})
```

```
3955:     exec(defstr)
3956:     eval(methodname).__doc__ = _screen_docrevise(eval('_Screen.'+methodname).__doc__)
3957:
3958: for methodname in _tg_turtle_functions:
3959:     pl1, pl2 = getmethparlist(eval('Turtle.' + methodname))
3960:     if pl1 == "":
3961:         print(">>>>>>", pl1, pl2)
3962:         continue
3963:     defstr = ("def %(key)s%(pl1)s: return _getpen().%(key)s%(pl2)s" %
3964:                                       {'key':methodname, 'pl1':pl1, 'pl2':pl2})
3965:     exec(defstr)
3966:     eval(methodname).__doc__ = _turtle_docrevise(eval('Turtle.'+methodname).__doc__)
3967:
3968:
3969: done = mainloop
3970:
3971: if __name__ == "__main__":
3972:     def switchpen():
3973:         if isdown():
3974:             pu()
3975:         else:
3976:             pd()
3977:
3978:     def demo1():
3979:         """Demo of old turtle.py - module"""
3980:         reset()
3981:         tracer(True)
3982:         up()
3983:         backward(100)
3984:         down()
3985:         # draw 3 squares; the last filled
3986:         width(3)
3987:         for i in range(3):
3988:             if i == 2:
3989:                 begin_fill()
3990:             for _ in range(4):
3991:                 forward(20)
3992:                 left(90)
3993:             if i == 2:
3994:                 color("maroon")
3995:                 end_fill()
3996:             up()
3997:             forward(30)
3998:             down()
3999:         width(1)
4000:         color("black")
4001:         # move out of the way
4002:         tracer(False)
4003:         up()
4004:         right(90)
4005:         forward(100)
4006:         right(90)
4007:         forward(100)
4008:         right(180)
4009:         down()
4010:         # some text
4011:         write("startstart", 1)
4012:         write("start", 1)
4013:         color("red")
4014:         # staircase
```

```
4015:         for i in range(5):
4016:             forward(20)
4017:             left(90)
4018:             forward(20)
4019:             right(90)
4020:         # filled staircase
4021:         tracer(True)
4022:         begin_fill()
4023:         for i in range(5):
4024:             forward(20)
4025:             left(90)
4026:             forward(20)
4027:             right(90)
4028:         end_fill()
4029:         # more text
4030: 
4031:     def demo2():
4032:         """Demo of some new features."""
4033:         speed(1)
4034:         st()
4035:         pensize(3)
4036:         setheading(towards(0, 0))
4037:         radius = distance(0, 0)/2.0
4038:         rt(90)
4039:         for _ in range(18):
4040:             switchpen()
4041:             circle(radius, 10)
4042:         write("wait a moment...")
4043:         while undobufferentries():
4044:             undo()
4045:         reset()
4046:         lt(90)
4047:         colormode(255)
4048:         laenge = 10
4049:         pencolor("green")
4050:         pensize(3)
4051:         lt(180)
4052:         for i in range(-2, 16):
4053:             if i > 0:
4054:                 begin_fill()
4055:                 fillcolor(255-15*i, 0, 15*i)
4056:             for _ in range(3):
4057:                 fd(laenge)
4058:                 lt(120)
4059:             end_fill()
4060:             laenge += 10
4061:             lt(15)
4062:             speed((speed()+1)%12)
4063:         #end_fill()
4064: 
4065:         lt(120)
4066:         pu()
4067:         fd(70)
4068:         rt(30)
4069:         pd()
4070:         color("red","yellow")
4071:         speed(0)
4072:         begin_fill()
4073:         for _ in range(4):
4074:             circle(50, 90)
```

```
4075:             rt(90)
4076:             fd(30)
4077:             rt(90)
4078:         end_fill()
4079:         lt(90)
4080:         pu()
4081:         fd(30)
4082:         pd()
4083:         shape("turtle")
4084:
4085:         tri = getturtle()
4086:         tri.resizemode("auto")
4087:         turtle = Turtle()
4088:         turtle.resizemode("auto")
4089:         turtle.shape("turtle")
4090:         turtle.reset()
4091:         turtle.left(90)
4092:         turtle.speed(0)
4093:         turtle.up()
4094:         turtle.goto(280, 40)
4095:         turtle.lt(30)
4096:         turtle.down()
4097:         turtle.speed(6)
4098:         turtle.color("blue","orange")
4099:         turtle.pensize(2)
4100:         tri.speed(6)
4101:         setheading(towards(turtle))
4102:         count = 1
4103:         while tri.distance(turtle) > 4:
4104:             turtle.fd(3.5)
4105:             turtle.lt(0.6)
4106:             tri.setheading(tri.towards(turtle))
4107:             tri.fd(4)
4108:             if count % 20 == 0:
4109:                 turtle.stamp()
4110:                 tri.stamp()
4111:                 switchpen()
4112:             count += 1
4113:         tri.write("CAUGHT! ", font=("Arial", 16, "bold"), align="right")
4114:         tri.pencolor("black")
4115:         tri.pencolor("red")
4116:
4117:         def baba(xdummy, ydummy):
4118:             clearscreen()
4119:             bye()
4120:
4121:         time.sleep(2)
4122:
4123:         while undobufferentries():
4124:             tri.undo()
4125:             turtle.undo()
4126:         tri.fd(50)
4127:         tri.write("  Click me!", font = ("Courier", 12, "bold") )
4128:         tri.onclick(baba, 1)
4129:
4130:     demo1()
4131:     demo2()
4132:     exitonclick()
4133:
```