

## 24.1. `turtle` — Turtle graphics

Source code: [Lib/turtle.py](#)

### 24.1.1. Introduction

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.

The `turtle` module is an extended reimplementation of the same-named module from the Python standard distribution up to version Python 2.5.

It tries to keep the merits of the old turtle module 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.

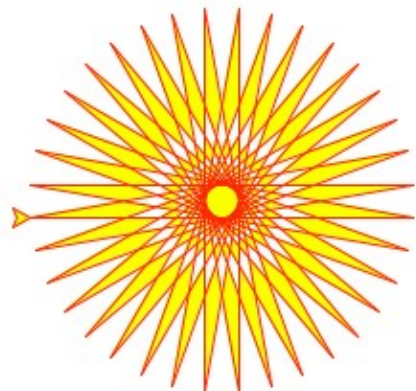
The `turtle` module provides turtle graphics primitives, in both object-oriented and procedure-oriented ways. Because it uses `tkinter` for the underlying graphics, it needs a version of Python installed with Tk support.

The object-oriented interface uses essentially two+two classes:

1. The `TurtleScreen` class defines graphics windows as a playground for

#### Turtle star

Turtle can draw intricate shapes using programs that repeat simple moves.



```
from turtle import *
color('red', 'yellow')
begin_fill()
while True:
    forward(200)
    left(170)
    if abs(pos()) < 1:
        break
end_fill()
done()
```

the drawing turtles. Its constructor needs a `tkinter.Canvas` or a `ScrolledCanvas` as argument. It should be used when `turtle` is used as part of some application.

The function `Screen()` returns a singleton object of a `TurtleScreen` subclass. This function should be used when `turtle` is used as a standalone tool for doing graphics. As a singleton object, inheriting from its class is not possible.

All methods of `TurtleScreen`/`Screen` also exist as functions, i.e. as part of the procedure-oriented interface.

2. `RawTurtle` (alias: `RawPen`) defines Turtle objects which draw on a `TurtleScreen`. Its constructor needs a `Canvas`, `ScrolledCanvas` or `TurtleScreen` as argument, so the `RawTurtle` objects know where to draw.

Derived from `RawTurtle` is the subclass `Turtle` (alias: `Pen`), which draws on “the” `Screen` instance which is automatically created, if not already present.

All methods of `RawTurtle`/`Turtle` also exist as functions, i.e. part of the procedure-oriented interface.

The procedural interface provides functions which are derived from the methods of the classes `Screen` and `Turtle`. They have the same names as the corresponding methods. A screen object is automatically created whenever a function derived from a `Screen` method is called. An (unnamed) turtle object is automatically created whenever any of the functions derived from a `Turtle` method is called.

To use multiple turtles on a screen one has to use the object-oriented interface.

**Note:** In the following documentation the argument list for functions is given. Methods, of course, have the additional first argument *self* which is omitted here.

## 24.1.2. Overview of available Turtle and Screen methods

### 24.1.2.1. Turtle methods

Turtle motion

Move and draw

```
forward() | fd()  
backward() | bk() | back()  
right() | rt()  
left() | lt()  
goto() | setpos() | setposition()
```

```
setx()  
sety()  
setheading() | seth()  
home()  
circle()  
dot()  
stamp()  
clearstamp()  
clearstamps()  
undo()  
speed()
```

#### Tell Turtle's state

```
position() | pos()  
towards()  
xcor()  
ycor()  
heading()  
distance()
```

#### Setting and measurement

```
degrees()  
radians()
```

#### Pen control

##### Drawing state

```
pendown() | pd() | down()  
penup() | pu() | up()  
pensize() | width()  
pen()  
isdown()
```

##### Color control

```
color()  
pencolor()  
fillcolor()
```

##### Filling

```
filling()  
begin_fill()  
end_fill()
```

##### More drawing control

```
reset()  
clear()  
write()
```

## Turtle state

### Visibility

```
showturtle() | st()  
hideturtle() | ht()  
isvisible()
```

### Appearance

```
shape()  
resizemode()  
shapeseize() | turtlesize()  
shearfactor()  
settiltangle()  
tiltangle()  
tilt()  
shapetransform()  
get_shapepoly()
```

## Using events

```
onclick()  
onrelease()  
ondrag()
```

## Special Turtle methods

```
begin_poly()  
end_poly()  
get_poly()  
clone()  
getturtle() | getpen()  
getscreen()  
setundobuffer()  
undobufferentries()
```

## 24.1.2.2. Methods of TurtleScreen/Screen

### Window control

```
bgcolor()  
bgpic()  
clear() | clearscreen()  
reset() | resetscreen()  
screensize()  
setworldcoordinates()
```

### Animation control

```
delay()  
tracer()  
update()
```

### Using screen events

```
listen()  
onkey() | onkeyrelease()  
onkeypress()  
onclick() | onscreenclick()  
ontimer()  
mainloop() | done()
```

### Settings and special methods

```
mode()  
colormode()  
getcanvas()  
getshapes()  
register_shape() | addshape()  
turtles()  
window_height()  
window_width()
```

### Input methods

```
textinput()  
numinput()
```

### Methods specific to Screen

```
bye()  
exitonclick()  
setup()  
title()
```

## 24.1.3. Methods of RawTurtle/Turtle and corresponding functions

Most of the examples in this section refer to a Turtle instance called `turtle`.

### 24.1.3.1. Turtle motion

```
turtle.forward(distance)  
turtle.fd(distance)
```

<b>Parameters:</b>	<b>distance</b> – a number (integer or float)
--------------------	---

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

```
>>> turtle.position()  
(0.00,0.00)
```

```
>>> turtle.forward(25)
>>> turtle.position()
(25.00,0.00)
>>> turtle.forward(-75)
>>> turtle.position()
(-50.00,0.00)
```

`turtle.back(distance)`  
`turtle.bk(distance)`  
`turtle.backward(distance)`

**Parameters:** **distance** – a number

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

```
>>> turtle.position()
(0.00,0.00)
>>> turtle.backward(30)
>>> turtle.position()
(-30.00,0.00)
```

`turtle.right(angle)`  
`turtle.rt(angle)`

**Parameters:** **angle** – a number (integer or float)

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

```
>>> turtle.heading()
22.0
>>> turtle.right(45)
>>> turtle.heading()
337.0
```

`turtle.left(angle)`  
`turtle.lt(angle)`

**Parameters:** **angle** – a number (integer or float)

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

```
>>> turtle.heading()
22.0
```

```
>>> turtle.left(45)
>>> turtle.heading()
67.0
```

`turtle.goto(x, y=None)`  
`turtle.setpos(x, y=None)`  
`turtle.setposition(x, y=None)`

<b>Parameters:</b>	<ul style="list-style-type: none"><li>• <b>x</b> – a number or a pair/vector of numbers</li><li>• <b>y</b> – a number or None</li></ul>
--------------------	---

If *y* is None, *x* must be a pair of coordinates or a [Vec2D](#) (e.g. as returned by [pos\(\)](#)).

Move turtle to an absolute position. If the pen is down, draw line. Do not change the turtle's orientation.

```
>>> tp = turtle.pos()
>>> tp
(0.00,0.00)
>>> turtle.setpos(60,30)
>>> turtle.pos()
(60.00,30.00)
>>> turtle.setpos((20,80))
>>> turtle.pos()
(20.00,80.00)
>>> turtle.setpos(tp)
>>> turtle.pos()
(0.00,0.00)
```

`turtle.setx(x)`

<b>Parameters:</b>	<b>x</b> – a number (integer or float)
--------------------	--

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

```
>>> turtle.position()
(0.00,240.00)
>>> turtle.setx(10)
>>> turtle.position()
(10.00,240.00)
```

`turtle.sety(y)`

<b>Parameters:</b>	<b>y</b> – a number (integer or float)
--------------------	--

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

```
>>> turtle.position()
(0.00,40.00)
>>> turtle.sety(-10)
>>> turtle.position()
(0.00,-10.00)
```

`turtle.setheading(to_angle)`  
`turtle.seth(to_angle)`

**Parameters:** **to\_angle** – a number (integer or float)

Set the orientation of the turtle to *to\_angle*. Here are some common directions in degrees:

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

```
>>> turtle.setheading(90)
>>> turtle.heading()
90.0
```

`turtle.home()`

Move turtle to the origin – coordinates (0,0) – and set its heading to its start-orientation (which depends on the mode, see [mode\(\)](#)).

```
>>> turtle.heading()
90.0
>>> turtle.position()
(0.00,-10.00)
>>> turtle.home()
>>> turtle.position()
(0.00,0.00)
>>> turtle.heading()
0.0
```

`turtle.circle(radius, extent=None, steps=None)`

**Parameters:**

- **radius** – a number
- **extent** – a number (or None)
- **steps** – an integer (or None)



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

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

```
>>> turtle.home()
>>> turtle.position()
(0.00,0.00)
>>> turtle.heading()
0.0
>>> turtle.circle(50)
>>> turtle.position()
(-0.00,0.00)
>>> turtle.heading()
0.0
>>> turtle.circle(120, 180) # draw a semicircle
>>> turtle.position()
(0.00,240.00)
>>> turtle.heading()
180.0
```

**turtle.dot(size=None, \*color)**

<b>Parameters:</b>	<ul style="list-style-type: none"><li>• <b>size</b> – an integer <math>\geq 1</math> (if given)</li><li>• <b>color</b> – a colorstring or a numeric color tuple</li></ul>
--------------------	---

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

```
>>> turtle.home()
>>> turtle.dot()
>>> turtle.fd(50); turtle.dot(20, "blue"); turtle.fd(50)
>>> turtle.position()
(100.00,-0.00)
>>> turtle.heading()
0.0
```

**turtle.stamp()**

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

```
>>> turtle.color("blue")
>>> turtle.stamp()
11
>>> turtle.fd(50)
```

**turtle.clearstamp(*stampid*)**

<b>Parameters:</b>	<b>stampid</b> – an integer, must be return value of previous <code>stamp()</code> call
--------------------	---

Delete stamp with given *stampid*.

```
>>> turtle.position()
(150.00,-0.00)
>>> turtle.color("blue")
>>> astamp = turtle.stamp()
>>> turtle.fd(50)
>>> turtle.position()
(200.00,-0.00)
>>> turtle.clearstamp(astamp)
>>> turtle.position()
(200.00,-0.00)
```

**turtle.clearstamps(*n=None*)**

<b>Parameters:</b>	<b>n</b> – an integer (or None)
--------------------	---------------------------------

Delete all or first/last *n* of turtle's stamps. If *n* is None, delete all stamps, if *n* > 0 delete first *n* stamps, else if *n* < 0 delete last *n* stamps.

```
>>> for i in range(8):
...     turtle.stamp(); turtle.fd(30)
13
14
15
16
17
18
19
20
>>> turtle.clearstamps(2)
>>> turtle.clearstamps(-2)
>>> turtle.clearstamps()
```

**turtle.undo()**

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

```
>>> for i in range(4):
...     turtle.fd(50); turtle.lt(80)
...
>>> for i in range(8):
...     turtle.undo()
```

`turtle.speed(speed=None)`

<b>Parameters:</b>	<b>speed</b> – an integer in the range 0..10 or a speedstring (see below)
--------------------	---

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

If input is a number greater than 10 or smaller than 0.5, speed is set to 0. Speedstrings are mapped to speedvalues as follows:

- "fastest": 0
- "fast": 10
- "normal": 6
- "slow": 3
- "slowest": 1

Speeds from 1 to 10 enforce increasingly faster animation of line drawing and turtle turning.

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

```
>>> turtle.speed()
3
>>> turtle.speed('normal')
>>> turtle.speed()
6
>>> turtle.speed(9)
>>> turtle.speed()
9
```

### 24.1.3.2. Tell Turtle's state

`turtle.position()`

`turtle.pos()`

Return the turtle's current location (x,y) (as a [Vec2D](#) vector).

```
>>> turtle.pos()
(440.00, -0.00)
```

`turtle.towards(x, y=None)`

- |                    |  |
|--------------------|--|
| <b>Parameters:</b> | <ul style="list-style-type: none"><li>• <b>x</b> – a number or a pair/vector of numbers or a turtle instance</li><li>• <b>y</b> – a number if x is a number, else None</li></ul> |
|--------------------|--|

Return the angle between the line from turtle position to position specified by (x,y), the vector or the other turtle. This depends on the turtle's start orientation which depends on the mode - "standard"/"world" or "logo").

```
>>> turtle.goto(10, 10)
>>> turtle.towards(0,0)
225.0
```

`turtle.xcor()`

Return the turtle's x coordinate.

```
>>> turtle.home()
>>> turtle.left(50)
>>> turtle.forward(100)
>>> turtle.pos()
(64.28,76.60)
>>> print(round(turtle.xcor(), 5))
64.27876
```

`turtle.ycor()`

Return the turtle's y coordinate.

```
>>> turtle.home()
>>> turtle.left(60)
>>> turtle.forward(100)
>>> print(turtle.pos())
(50.00,86.60)
>>> print(round(turtle.ycor(), 5))
86.60254
```

`turtle.heading()`

Return the turtle's current heading (value depends on the turtle mode, see [mode\(\)](#)).

```
>>> turtle.home()
>>> turtle.left(67)
>>> turtle.heading()
67.0
```

`turtle.distance(x, y=None)`

--	--

**Parameters:**

- **x** – a number or a pair/vector of numbers or a turtle instance
- **y** – a number if x is a number, else None

Return the distance from the turtle to (x,y), the given vector, or the given other turtle, in turtle step units.

```
>>> turtle.home()
>>> turtle.distance(30,40)
50.0
>>> turtle.distance((30,40))
50.0
>>> joe = Turtle()
>>> joe.forward(77)
>>> turtle.distance(joe)
77.0
```

### 24.1.3.3. Settings for measurement

**turtle.degrees**(*fullcircle=360.0*)

**Parameters:**

**fullcircle** – a number

Set angle measurement units, i.e. set number of “degrees” for a full circle. Default value is 360 degrees.

```
>>> turtle.home()
>>> turtle.left(90)
>>> turtle.heading()
90.0

Change angle measurement unit to grad (also known as gon,
grade, or gradian and equals 1/100-th of the right angle.)
>>> turtle.degrees(400.0)
>>> turtle.heading()
100.0
>>> turtle.degrees(360)
>>> turtle.heading()
90.0
```

**turtle.radians**()

Set the angle measurement units to radians. Equivalent to degrees (2\*math.pi).

```
>>> turtle.home()
>>> turtle.left(90)
>>> turtle.heading()
90.0
```

```
>>> turtle.radians()  
>>> turtle.heading()  
1.5707963267948966
```

## 24.1.3.4. Pen control

### 24.1.3.4.1. Drawing state

```
turtle.pendown()  
turtle.pd()  
turtle.down()
```

Pull the pen down – drawing when moving.

```
turtle.penup()  
turtle.pu()  
turtle.up()
```

Pull the pen up – no drawing when moving.

```
turtle.pensize(width=None)  
turtle.width(width=None)
```

**Parameters:** **width** – a positive number

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

```
>>> turtle.pensize()  
1  
>>> turtle.pensize(10)  # from here on lines of width 10 are drawn
```

```
turtle.pen(pen=None, **pendict)
```

**Parameters:**

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

Return or set the pen’s attributes in a “pen-dictionary” with the following key/value pairs:

- “shown”: True/False
- “pendown”: True/False
- “pencolor”: color-string or color-tuple
- “fillcolor”: color-string or color-tuple
- “pensize”: positive number
- “speed”: number in range 0..10

- “resizemode”: “auto” or “user” or “noresize”
- “stretchfactor”: (positive number, positive number)
- “outline”: positive number
- “tilt”: number

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

```
>>> turtle.pen(fillcolor="black", pencolor="red", pensize=10)
>>> sorted(turtle.pen().items())
[('fillcolor', 'black'), ('outline', 1), ('pencolor', 'red'),
 ('pendown', True), ('pensize', 10), ('resizemode', 'noresize'),
 ('shearfactor', 0.0), ('shown', True), ('speed', 9),
 ('stretchfactor', (1.0, 1.0)), ('tilt', 0.0)]
>>> penstate=turtle.pen()
>>> turtle.color("yellow", "")
>>> turtle.penup()
>>> sorted(turtle.pen().items())[:3]
[('fillcolor', ''), ('outline', 1), ('pencolor', 'yellow')]
>>> turtle.pen(penstate, fillcolor="green")
>>> sorted(turtle.pen().items())[:3]
[('fillcolor', 'green'), ('outline', 1), ('pencolor', 'red')]
```

### `turtle.isdown()`

Return True if pen is down, False if it's up.

```
>>> turtle.penup()
>>> turtle.isdown()
False
>>> turtle.pendown()
>>> turtle.isdown()
True
```

## 24.1.3.4.2. Color control

### `turtle.pencolor(*args)`

Return or set the pencolor.

Four input formats are allowed:

`pencolor()`

Return the current pencolor as color specification string or as a tuple (see example). May be used as input to another color/pencolor/fillcolor call.

`pencolor(colorstring)`

Set pencolor to *colorstring*, which is a Tk color specification string, such as "red", "yellow", or "#33cc8c".

pencolor((r, g, b))

Set pencolor to the RGB color represented by the tuple of *r*, *g*, and *b*. Each of *r*, *g*, and *b* must be in the range 0..colormode, where colormode is either 1.0 or 255 (see [colormode\(\)](#)).

pencolor(r, g, b)

Set pencolor to the RGB color represented by *r*, *g*, and *b*. Each of *r*, *g*, and *b* must be in the range 0..colormode.

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

```
>>> colormode()
1.0
>>> turtle.pencolor()
'red'
>>> turtle.pencolor("brown")
>>> turtle.pencolor()
'brown'
>>> tup = (0.2, 0.8, 0.55)
>>> turtle.pencolor(tup)
>>> turtle.pencolor()
(0.2, 0.8, 0.5490196078431373)
>>> colormode(255)
>>> turtle.pencolor()
(51.0, 204.0, 140.0)
>>> turtle.pencolor('#32c18f')
>>> turtle.pencolor()
(50.0, 193.0, 143.0)
```

**turtle.fillcolor(\*args)**

Return or set the fillcolor.

Four input formats are allowed:

fillcolor()

Return the current fillcolor as color specification string, possibly in tuple format (see example). May be used as input to another color/pencolor/fillcolor call.

fillcolor(colorstring)

Set fillcolor to *colorstring*, which is a Tk color specification string, such as "red", "yellow", or "#33cc8c".

fillcolor((r, g, b))



Set fillcolor to the RGB color represented by the tuple of *r*, *g*, and *b*. Each of *r*, *g*, and *b* must be in the range 0..colormode, where colormode is either 1.0 or 255 (see `colormode()`).

`fillcolor(r, g, b)`

Set fillcolor to the RGB color represented by *r*, *g*, and *b*. Each of *r*, *g*, and *b* must be in the range 0..colormode.

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

```
>>> turtle.fillcolor("violet")
>>> turtle.fillcolor()
'violet'
>>> col = turtle.pencolor()
>>> col
(50.0, 193.0, 143.0)
>>> turtle.fillcolor(col)
>>> turtle.fillcolor()
(50.0, 193.0, 143.0)
>>> turtle.fillcolor('#ffffff')
>>> turtle.fillcolor()
(255.0, 255.0, 255.0)
```

`turtle.color(*args)`

Return or set pencolor and fillcolor.

Several input formats are allowed. They use 0 to 3 arguments as follows:

`color()`

Return the current pencolor and the current fillcolor as a pair of color specification strings or tuples as returned by `pencolor()` and `fillcolor()`.

`color(colorstring)`, `color((r,g,b))`, `color(r,g,b)`

Inputs as in `pencolor()`, set both, fillcolor and pencolor, to the given value.

`color(colorstring1, colorstring2)`, `color((r1,g1,b1), (r2,g2,b2))`

Equivalent to `pencolor(colorstring1)` and `fillcolor(colorstring2)` and analogously if the other input format is used.

If `turtleshape` is a polygon, outline and interior of that polygon is drawn with the newly set colors.

```
>>> turtle.color("red", "green")
>>> turtle.color()
('red', 'green')
>>> color("#285078", "#a0c8f0")
```

```
>>> color()
((40.0, 80.0, 120.0), (160.0, 200.0, 240.0))
```

See also: Screen method [colormode\(\)](#).

#### 24.1.3.4.3. Filling

**turtle.filling()**

Return fillstate (True if filling, False else).

```
>>> turtle.begin_fill()
>>> if turtle.filling():
...     turtle.pensize(5)
... else:
...     turtle.pensize(3)
```

**turtle.begin\_fill()**

To be called just before drawing a shape to be filled.

**turtle.end\_fill()**

Fill the shape drawn after the last call to [begin\\_fill\(\)](#).

```
>>> turtle.color("black", "red")
>>> turtle.begin_fill()
>>> turtle.circle(80)
>>> turtle.end_fill()
```

#### 24.1.3.4.4. More drawing control

**turtle.reset()**

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

```
>>> turtle.goto(0, -22)
>>> turtle.left(100)
>>> turtle.position()
(0.00, -22.00)
>>> turtle.heading()
100.0
>>> turtle.reset()
>>> turtle.position()
(0.00, 0.00)
>>> turtle.heading()
0.0
```

**turtle.clear()**

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

`turtle.write(arg, move=False, align="left", font=("Arial", 8, "normal"))`

- |                    |  |
|--------------------|--|
| <b>Parameters:</b> | <ul style="list-style-type: none"><li>• <b>arg</b> – object to be written to the TurtleScreen</li><li>• <b>move</b> – True/False</li><li>• <b>align</b> – one of the strings “left”, “center” or right</li><li>• <b>font</b> – a triple (fontname, fontsize, fonttype)</li></ul> |
|--------------------|--|

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

```
>>> turtle.write("Home = ", True, align="center")
>>> turtle.write((0,0), True)
```

## 24.1.3.5. Turtle state

### 24.1.3.5.1. Visibility

`turtle.hideturtle()`

`turtle.ht()`

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

```
>>> turtle.hideturtle()
```

`turtle.showturtle()`

`turtle.st()`

Make the turtle visible.

```
>>> turtle.showturtle()
```

`turtle.isvisible()`

Return True if the Turtle is shown, False if it's hidden.

```
>>> turtle.hideturtle()
>>> turtle.isvisible()
False
>>> turtle.showturtle()
>>> turtle.isvisible()
True
```

### 24.1.3.5.2. Appearance

`turtle.shape(name=None)`

**Parameters:** **name** – a string which is a valid shapename

Set turtle shape to shape with given *name* or, if name is not given, return name of current shape. Shape with *name* must exist in the TurtleScreen's shape dictionary. Initially there are the following polygon shapes: "arrow", "turtle", "circle", "square", "triangle", "classic". To learn about how to deal with shapes see Screen method [register\\_shape\(\)](#).

```
>>> turtle.shape()
'classic'
>>> turtle.shape("turtle")
>>> turtle.shape()
'turtle'
```

`turtle.resizemode(rmode=None)`

**Parameters:** **rmode** – one of the strings "auto", "user", "noresize"

Set `resizemode` to one of the values: "auto", "user", "noresize". If *rmode* is not given, return current `resizemode`. Different `resizemodes` have the following effects:

- "auto": adapts the appearance of the turtle corresponding to the value of `pensize`.
- "user": adapts the appearance of the turtle according to the values of `stretchfactor` and `outlinewidth` (outline), which are set by [shapessize\(\)](#).
- "noresize": no adaption of the turtle's appearance takes place.

`resizemode("user")` is called by [shapessize\(\)](#) when used with arguments.

```
>>> turtle.resizemode()
'noresize'
>>> turtle.resizemode("auto")
>>> turtle.resizemode()
'auto'
```

`turtle.shapessize(stretch_wid=None, stretch_len=None, outline=None)`  
`turtle.turtlessize(stretch_wid=None, stretch_len=None, outline=None)`

**Parameters:**

- **stretch\_wid** – positive number
- **stretch\_len** – positive number
- **outline** – positive number

Return or set the pen's attributes x/y-stretchfactors and/or outline. Set resize-mode to "user". If and only if `resizemode` is set to "user", the turtle will be displayed stretched according to its stretchfactors: *stretch\_wid* is stretchfactor perpendicular to its orientation, *stretch\_len* is stretchfactor in direction of its orientation, *outline* determines the width of the shapes's outline.

```
>>> turtle.shapesize()
(1.0, 1.0, 1)
>>> turtle.resizemode("user")
>>> turtle.shapesize(5, 5, 12)
>>> turtle.shapesize()
(5, 5, 12)
>>> turtle.shapesize(outline=8)
>>> turtle.shapesize()
(5, 5, 8)
```

`turtle.shearfactor(shear=None)`

**Parameters:** **shear** – number (optional)

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

```
>>> turtle.shape("circle")
>>> turtle.shapesize(5,2)
>>> turtle.shearfactor(0.5)
>>> turtle.shearfactor()
0.5
```

`turtle.tilt(angle)`

**Parameters:** **angle** – a number

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

```
>>> turtle.reset()
>>> turtle.shape("circle")
>>> turtle.shapesize(5,2)
>>> turtle.tilt(30)
>>> turtle.fd(50)
>>> turtle.tilt(30)
>>> turtle.fd(50)
```

`turtle.settiltangle(angle)`

**Parameters:** **angle** – a number

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

```
>>> turtle.reset()
>>> turtle.shape("circle")
>>> turtle.shapesize(5,2)
>>> turtle.settiltangle(45)
>>> turtle.fd(50)
>>> turtle.settiltangle(-45)
>>> turtle.fd(50)
```

*Deprecated since version 3.1.*

`turtle.tiltangle(angle=None)`

**Parameters:** **angle** – a number (optional)

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

```
>>> turtle.reset()
>>> turtle.shape("circle")
>>> turtle.shapesize(5,2)
>>> turtle.tilt(45)
>>> turtle.tiltangle()
45.0
```

`turtle.shapetransform(t11=None, t12=None, t21=None, t22=None)`

**Parameters:**

- **t11** – a number (optional)
- **t12** – a number (optional)
- **t21** – a number (optional)
- **t22** – a number (optional)

Set or return the current transformation matrix of the turtle shape.

If none of the matrix elements are given, return the transformation matrix as a tuple of 4 elements. Otherwise set the given elements and transform the turtle-shape according to the matrix consisting of first row *t11*, *t12* and second row *t21*, *t22*. The determinant  $t11 * t22 - t12 * t21$  must not be zero, otherwise an error is raised. Modify stretchfactor, shearfactor and tiltangle according to the given matrix.

```
>>> turtle = Turtle()
>>> turtle.shape("square")
>>> turtle.shapesize(4,2)
>>> turtle.shearfactor(-0.5)
>>> turtle.shapetransform()
(4.0, -1.0, -0.0, 2.0)
```

### `turtle.get_shapepoly()`

Return the current shape polygon as tuple of coordinate pairs. This can be used to define a new shape or components of a compound shape.

```
>>> turtle.shape("square")
>>> turtle.shapetransform(4, -1, 0, 2)
>>> turtle.get_shapepoly()
((50, -20), (30, 20), (-50, 20), (-30, -20))
```

## 24.1.3.6. Using events

### `turtle.onclick(fun, btn=1, add=None)`

- |                    |  |
|--------------------|--|
| <b>Parameters:</b> | <ul style="list-style-type: none"> <li>• <b>fun</b> – a function with two arguments which will be called with the coordinates of the clicked point on the canvas</li> <li>• <b>num</b> – number of the mouse-button, defaults to 1 (left mouse button)</li> <li>• <b>add</b> – True or False – if True, a new binding will be added, otherwise it will replace a former binding</li> </ul> |
|--------------------|--|

Bind *fun* to mouse-click events on this turtle. If *fun* is None, existing bindings are removed. Example for the anonymous turtle, i.e. the procedural way:

```
>>> def turn(x, y):
...     left(180)
...
>>> onclick(turn)  # Now clicking into the turtle will turn it.
>>> onclick(None)  # event-binding will be removed
```

### `turtle.onrelease(fun, btn=1, add=None)`

- |                    |  |
|--------------------|--|
| <b>Parameters:</b> | <ul style="list-style-type: none"> <li>• <b>fun</b> – a function with two arguments which will be called with the coordinates of the clicked point on the canvas</li> <li>• <b>num</b> – number of the mouse-button, defaults to 1 (left mouse button)</li> <li>• <b>add</b> – True or False – if True, a new binding will be added, otherwise it will replace a former binding</li> </ul> |
|--------------------|--|

Bind *fun* to mouse-button-release events on this turtle. If *fun* is None, existing bindings are removed.

```

>>> class MyTurtle(Turtle):
...     def glow(self,x,y):
...         self.fillcolor("red")
...     def unglow(self,x,y):
...         self.fillcolor("")
...
>>> turtle = MyTurtle()
>>> turtle.onclick(turtle.glow)      # clicking on turtle turns fil
>>> turtle.onrelease(turtle.unglow) # releasing turns it to transp

```

`turtle.ondrag(fun, btn=1, add=None)`

**Parameters:**

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

Bind *fun* to mouse-move events on this turtle. If *fun* is None, existing bindings are removed.

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

```

>>> turtle.ondrag(turtle.goto)

```

Subsequently, clicking and dragging the Turtle will move it across the screen thereby producing handdrawings (if pen is down).

### 24.1.3.7. Special Turtle methods

`turtle.begin_poly()`

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

`turtle.end_poly()`

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

`turtle.get_poly()`

Return the last recorded polygon.

```

>>> turtle.home()
>>> turtle.begin_poly()

```



```

>>> turtle.fd(100)
>>> turtle.left(20)
>>> turtle.fd(30)
>>> turtle.left(60)
>>> turtle.fd(50)
>>> turtle.end_poly()
>>> p = turtle.get_poly()
>>> register_shape("myFavouriteShape", p)

```

### **turtle.clone()**

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

```

>>> mick = Turtle()
>>> joe = mick.clone()

```

### **turtle.getturtle()**

### **turtle.getpen()**

Return the Turtle object itself. Only reasonable use: as a function to return the “anonymous turtle”:

```

>>> pet = getturtle()
>>> pet.fd(50)
>>> pet
<turtle.Turtle object at 0x...>

```

### **turtle.getscreen()**

Return the [TurtleScreen](#) object the turtle is drawing on. TurtleScreen methods can then be called for that object.

```

>>> ts = turtle.getscreen()
>>> ts
<turtle._Screen object at 0x...>
>>> ts.bgcolor("pink")

```

### **turtle.setundobuffer(size)**

**Parameters:** **size** – an integer or None

Set or disable undobuffer. If *size* is an integer an empty undobuffer of given size is installed. *size* gives the maximum number of turtle actions that can be undone by the [undo\(\)](#) method/function. If *size* is None, the undobuffer is disabled.

```

>>> turtle.setundobuffer(42)

```

### **turtle.undobufferentries()**

Return number of entries in the undobuffer.

```
>>> while undobufferentries():  
...     undo()
```

### 24.1.3.8. Compound shapes

To use compound turtle shapes, which consist of several polygons of different color, you must use the helper class `Shape` explicitly as described below:

1. Create an empty Shape object of type “compound”.
2. Add as many components to this object as desired, using the `addcomponent()` method.

For example:

```
>>> s = Shape("compound")  
>>> poly1 = ((0,0),(10,-5),(0,10),(-10,-5))  
>>> s.addcomponent(poly1, "red", "blue")  
>>> poly2 = ((0,0),(10,-5),(-10,-5))  
>>> s.addcomponent(poly2, "blue", "red")
```

3. Now add the Shape to the Screen’s shapelist and use it:

```
>>> register_shape("myshape", s)  
>>> shape("myshape")
```

**Note:** The `Shape` class is used internally by the `register_shape()` method in different ways. The application programmer has to deal with the Shape class *only* when using compound shapes like shown above!

## 24.1.4. Methods of TurtleScreen/Screen and corresponding functions

Most of the examples in this section refer to a TurtleScreen instance called `screen`.

### 24.1.4.1. Window control

`turtle.bgcolor(*args)`

<b>Parameters:</b>	<b>args</b> – a color string or three numbers in the range 0..color-mode or a 3-tuple of such numbers
--------------------	---

Set or return background color of the TurtleScreen.

```
>>> screen.bgcolor("orange")
>>> screen.bgcolor()
'orange'
>>> screen.bgcolor("#800080")
>>> screen.bgcolor()
(128.0, 0.0, 128.0)
```

`turtle.bgpic(picname=None)`

**Parameters:** **picname** – a string, name of a gif-file or "nopic", or None

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

```
>>> screen.bgpic()
'nopic'
>>> screen.bgpic("landscape.gif")
>>> screen.bgpic()
"landscape.gif" >>>
```

`turtle.clear()`

`turtle.clearscreen()`

Delete all drawings and all turtles from the TurtleScreen. Reset the now empty TurtleScreen to its initial state: white background, no background image, no event bindings and tracing on.

**Note:** This TurtleScreen method is available as a global function only under the name `clearscreen`. The global function `clear` is a different one derived from the Turtle method `clear`.

`turtle.reset()`

`turtle.resetscreen()`

Reset all Turtles on the Screen to their initial state.

**Note:** This TurtleScreen method is available as a global function only under the name `resetscreen`. The global function `reset` is another one derived from the Turtle method `reset`.

`turtle.screensize(canvwidth=None, canvheight=None, bg=None)`

**Parameters:** • **canvwidth** – positive integer, new width of canvas in pixels

- **canvheight** – positive integer, new height of canvas in pixels
- **bg** – colorstring or color-tuple, new background color

If no arguments are given, return current (canvaswidth, canvasheight). Else resize the canvas the turtles are drawing on. Do not alter the drawing window. To observe hidden parts of the canvas, use the scrollbars. With this method, one can make visible those parts of a drawing which were outside the canvas before.

```
>>> screen.screensize()
(400, 300)
>>> screen.screensize(2000,1500)
>>> screen.screensize()
(2000, 1500) >>>
```

e.g. to search for an erroneously escaped turtle ;-)

**turtle.setworldcoordinates**(*llx*, *lly*, *urx*, *ury*)

- |                    |  |
|--------------------|--|
| <b>Parameters:</b> | <ul style="list-style-type: none"> <li>• <b>llx</b> – a number, x-coordinate of lower left corner of canvas</li> <li>• <b>lly</b> – a number, y-coordinate of lower left corner of canvas</li> <li>• <b>urx</b> – a number, x-coordinate of upper right corner of canvas</li> <li>• <b>ury</b> – a number, y-coordinate of upper right corner of canvas</li> </ul> |
|--------------------|--|

Set up user-defined coordinate system and switch to mode “world” if necessary. This performs a `screen.reset()`. If mode “world” is already active, all drawings are redrawn according to the new coordinates.

**ATTENTION:** in user-defined coordinate systems angles may appear distorted.

```
>>> screen.reset()
>>> screen.setworldcoordinates(-50,-7.5,50,7.5)
>>> for _ in range(72):
...     left(10)
...
>>> for _ in range(8):
...     left(45); fd(2)    # a regular octagon
```

## 24.1.4.2. Animation control

**turtle.delay**(*delay=None*)

- |                    |                                 |
|--------------------|---------------------------------|
| <b>Parameters:</b> | <b>delay</b> – positive integer |
|--------------------|---------------------------------|

Set or return the drawing *delay* in milliseconds. (This is approximately the time interval between two consecutive canvas updates.) The longer the drawing delay, the slower the animation.

Optional argument:

```
>>> screen.delay()
10
>>> screen.delay(5)
>>> screen.delay()
5
```

`turtle.tracer(n=None, delay=None)`

<b>Parameters:</b>	<ul style="list-style-type: none"><li>• <b>n</b> – nonnegative integer</li><li>• <b>delay</b> – nonnegative integer</li></ul>
--------------------	---

Turn turtle animation on/off and set delay for update drawings. If *n* is given, only each *n*-th regular screen update is really performed. (Can be used to accelerate the drawing of complex graphics.) When called without arguments, returns the currently stored value of *n*. Second argument sets delay value (see [delay\(\)](#)).

```
>>> screen.tracer(8, 25)
>>> dist = 2
>>> for i in range(200):
...     fd(dist)
...     rt(90)
...     dist += 2
```

`turtle.update()`

Perform a TurtleScreen update. To be used when tracer is turned off.

See also the RawTurtle/Turtle method [speed\(\)](#).

### 24.1.4.3. Using screen events

`turtle.listen(xdummy=None, ydummy=None)`

Set focus on TurtleScreen (in order to collect key-events). Dummy arguments are provided in order to be able to pass [listen\(\)](#) to the onclick method.

`turtle.onkey(fun, key)`

`turtle.onkeyrelease(fun, key)`

<b>Parameters:</b>	<ul style="list-style-type: none"><li>• <b>fun</b> – a function with no arguments or None</li><li>• <b>key</b> – a string: key (e.g. “a”) or key-symbol (e.g. “space”)</li></ul>
--------------------	--

Bind *fun* to key-release event of key. If *fun* is None, event bindings are removed. Remark: in order to be able to register key-events, TurtleScreen must have the focus. (See method [listen\(\)](#).)

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

`turtle.onkeypress(fun, key=None)`

<b>Parameters:</b>	<ul style="list-style-type: none"><li>• <b>fun</b> – a function with no arguments or None</li><li>• <b>key</b> – a string: key (e.g. “a”) or key-symbol (e.g. “space”)</li></ul>
--------------------	--

Bind *fun* to key-press event of key if key is given, or to any key-press-event if no key is given. Remark: in order to be able to register key-events, TurtleScreen must have focus. (See method [listen\(\)](#).)

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

`turtle.onclick(fun, btn=1, add=None)`

`turtle.onscreenclick(fun, btn=1, add=None)`

<b>Parameters:</b>	<ul style="list-style-type: none"><li>• <b>fun</b> – a function with two arguments which will be called with the coordinates of the clicked point on the canvas</li><li>• <b>num</b> – number of the mouse-button, defaults to 1 (left mouse button)</li><li>• <b>add</b> – True or False – if True, a new binding will be added, otherwise it will replace a former binding</li></ul>
--------------------	--

Bind *fun* to mouse-click events on this screen. If *fun* is None, existing bindings are removed.

Example for a TurtleScreen instance named `screen` and a Turtle instance named `turtle`:

```
>>> screen.onclick(turtle.goto) # Subsequently clicking into the T
>>>                               # make the turtle move to the clic
>>> screen.onclick(None)        # remove event binding again
```

<

>

**Note:** This TurtleScreen method is available as a global function only under the name `onscreenclick`. The global function `onclick` is another one derived from the Turtle method `onclick`.

`turtle.ontimer(fun, t=0)`

<b>Parameters:</b>	<ul style="list-style-type: none"><li>• <b>fun</b> – a function with no arguments</li><li>• <b>t</b> – a number <math>\geq 0</math></li></ul>
--------------------	---

Install a timer that calls *fun* after *t* milliseconds.

```
>>> running = True
>>> def f():
...     if running:
...         fd(50)
...         lt(60)
...         screen.ontimer(f, 250)
>>> f()    ### makes the turtle march around
>>> running = False
```

`turtle.mainloop()`

`turtle.done()`

Starts event loop - calling Tkinter's `mainloop` function. Must be the last statement in a turtle graphics program. Must *not* be used if a script is run from within IDLE in -n mode (No subprocess) - for interactive use of turtle graphics.

```
>>> screen.mainloop()
```

```
>>>
```

## 24.1.4.4. Input methods

`turtle.textinput(title, prompt)`

<b>Parameters:</b>	<ul style="list-style-type: none"><li>• <b>title</b> – string</li><li>• <b>prompt</b> – string</li></ul>
--------------------	--

Pop up a dialog window for input of a string. Parameter *title* is the title of the dialog window, *prompt* is a text mostly describing what information to input. Return the string input. If the dialog is canceled, return `None`.

```
>>> screen.textinput("NIM", "Name of first player:")
```

```
>>>
```

`turtle.numinput(title, prompt, default=None, minval=None, maxval=None)`

<b>Parameters:</b>	<ul style="list-style-type: none"><li>• <b>title</b> – string</li><li>• <b>prompt</b> – string</li><li>• <b>default</b> – number (optional)</li></ul>
--------------------	---





```
TurtleGraphicsError: bad color sequence: (240, 160, 80)
>>> screen.colormode()
1.0
>>> screen.colormode(255)
>>> screen.colormode()
255
>>> turtle.pencolor(240,160,80)
```

### `turtle.getcanvas()`

Return the Canvas of this TurtleScreen. Useful for insiders who know what to do with a Tkinter Canvas.

```
>>> cv = screen.getcanvas()
>>> cv
<turtle.ScrolledCanvas object ...>
```

### `turtle.getshapes()`

Return a list of names of all currently available turtle shapes.

```
>>> screen.getshapes()
['arrow', 'blank', 'circle', ..., 'turtle']
```

### `turtle.register_shape(name, shape=None)`

### `turtle.addshape(name, shape=None)`

There are three different ways to call this function:

1. *name* is the name of a gif-file and *shape* is None: Install the corresponding image shape.

```
>>> screen.register_shape("turtle.gif")
```

```
>>>
```

**Note:** Image shapes *do not* rotate when turning the turtle, so they do not display the heading of the turtle!

2. *name* is an arbitrary string and *shape* is a tuple of pairs of coordinates: Install the corresponding polygon shape.

```
>>> screen.register_shape("triangle", ((5,-3), (0,5), (-5,-3))
< >
```

3. *name* is an arbitrary string and *shape* is a (compound) [Shape](#) object: Install the corresponding compound shape.

Add a turtle shape to TurtleScreen's shapelist. Only thusly registered shapes can be used by issuing the command `shape(shapename)`.

`turtle.turtles()`

Return the list of turtles on the screen.

```
>>> for turtle in screen.turtles():  
...     turtle.color("red")
```

`turtle.window_height()`

Return the height of the turtle window.

```
>>> screen.window_height()  
480
```

>>>

`turtle.window_width()`

Return the width of the turtle window.

```
>>> screen.window_width()  
640
```

>>>

## 24.1.4.6. Methods specific to Screen, not inherited from TurtleScreen

`turtle.bye()`

Shut the turtlegraphics window.

`turtle.exitonclick()`

Bind `bye()` method to mouse clicks on the Screen.

If the value “using\_IDLE” in the configuration dictionary is `False` (default value), also enter mainloop. Remark: If IDLE with the `-n` switch (no subprocess) is used, this value should be set to `True` in `turtle.cfg`. In this case IDLE’s own mainloop is active also for the client script.

`turtle.setup(width=_CFG["width"], height=_CFG["height"], startx=_CFG["leftright"], starty=_CFG["topbottom"])`

Set the size and position of the main window. Default values of arguments are stored in the configuration dictionary and can be changed via a `turtle.cfg` file.

### Parameters:

- **width** – if an integer, a size in pixels, if a float, a fraction of the screen; default is 50% of screen
- **height** – if an integer, the height in pixels, if a float, a fraction of the screen; default is 75% of screen
- **startx** – if positive, starting position in pixels from the left edge of the screen, if negative from the right edge, if `None`, center window horizontally

- **starty** – if positive, starting position in pixels from the top edge of the screen, if negative from the bottom edge, if None, center window vertically

```
>>> screen.setup (width=200, height=200, startx=0, starty=0)
>>>                 # sets window to 200x200 pixels, in upper left of
>>> screen.setup(width=.75, height=0.5, startx=None, starty=None)
>>>                 # sets window to 75% of screen by 50% of screen a
```

`turtle.title(titlestring)`

**Parameters:** **titlestring** – a string that is shown in the titlebar of the turtle graphics window

Set title of turtle window to *titlestring*.

```
>>> screen.title("Welcome to the turtle zoo!")
```

## 24.1.5. Public classes

`class turtle.RawTurtle(canvas)`

`class turtle.RawPen(canvas)`

**Parameters:** **canvas** – a `tkinter.Canvas`, a [ScrolledCanvas](#) or a [TurtleScreen](#)

Create a turtle. The turtle has all methods described above as “methods of Turtle/RawTurtle”.

`class turtle.Turtle`

Subclass of `RawTurtle`, has the same interface but draws on a default [Screen](#) object created automatically when needed for the first time.

`class turtle.TurtleScreen(cv)`

**Parameters:** **cv** – a `tkinter.Canvas`

Provides screen oriented methods like `setbg()` etc. that are described above.

`class turtle.Screen`

Subclass of `TurtleScreen`, with [four methods added](#).

`class turtle.ScrolledCanvas(master)`

**Parameters:** **master** – some Tkinter widget to contain the `ScrolledCanvas`, i.e. a Tkinter-canvas with scrollbars added

---

Used by class `Screen`, which thus automatically provides a `ScrolledCanvas` as playground for the turtles.

`class turtle.` **Shape**(*type\_*, *data*)

<b>Parameters:</b>	<b>type_</b> – one of the strings “polygon”, “image”, “compound”
--------------------	--

Data structure modeling shapes. The pair (*type\_*, *data*) must follow this specification:

<i>type_</i>	<i>data</i>
“polygon”	a polygon-tuple, i.e. a tuple of pairs of coordinates
“image”	an image (in this form only used internally!)
“compound”	None (a compound shape has to be constructed using the <a href="#">addcomponent()</a> method)

**addcomponent**(*poly*, *fill*, *outline=None*)

<b>Parameters:</b>	<ul style="list-style-type: none"><li>• <b>poly</b> – a polygon, i.e. a tuple of pairs of numbers</li><li>• <b>fill</b> – a color the <i>poly</i> will be filled with</li><li>• <b>outline</b> – a color for the poly’s outline (if given)</li></ul>
--------------------	--

Example:

```
>>> poly = ((0,0),(10,-5),(0,10),(-10,-5))
>>> s = Shape("compound")
>>> s.addcomponent(poly, "red", "blue")
>>> # ... add more components and then use register_shape()
```

See [Compound shapes](#).

`class turtle.` **Vec2D**(*x*, *y*)

A two-dimensional vector class, used as a helper class for implementing turtle graphics. May be useful for turtle graphics programs too. 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
- `abs(a)` absolute value of *a*
- `a.rotate(angle)` rotation

## 24.1.6. Help and configuration

### 24.1.6.1. How to use help

The public methods of the Screen and Turtle classes are documented extensively via docstrings. So these can be used as online-help via the Python help facilities:

- When using IDLE, tooltips show the signatures and first lines of the docstrings of typed in function-/method calls.
- Calling `help()` on methods or functions displays the docstrings:

```
>>> help(Screen.bgcolor)
Help on method bgcolor in module turtle:

bgcolor(self, *args) unbound turtle.Screen method
    Set or return backgroundcolor of the TurtleScreen.

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

    >>> screen.bgcolor("orange")
    >>> screen.bgcolor()
    "orange"
    >>> screen.bgcolor(0.5,0,0.5)
    >>> screen.bgcolor()
    "#800080"

>>> help(Turtle.penup)
Help on method penup in module turtle:

penup(self) unbound turtle.Turtle method
    Pull the pen up -- no drawing when moving.

    Aliases: penup | pu | up

    No argument

    >>> turtle.penup()
```

- The docstrings of the functions which are derived from methods have a modified form:

```
>>> help(bgcolor)
Help on function bgcolor in module turtle:

bgcolor(*args)
    Set or return backgroundcolor of the TurtleScreen.
```

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

Example::

```
>>> bgcolor("orange")
>>> bgcolor()
"orange"
>>> bgcolor(0.5,0,0.5)
>>> bgcolor()
"#800080"
```

```
>>> help(penup)
Help on function penup in module turtle:

penup()
    Pull the pen up -- no drawing when moving.

Aliases: penup | pu | up

No argument

Example:
>>> penup()
```

These modified docstrings are created automatically together with the function definitions that are derived from the methods at import time.

## 24.1.6.2. Translation of docstrings into different languages

There is a utility to create a dictionary the keys of which are the method names and the values of which are the docstrings of the public methods of the classes `Screen` and `Turtle`.

`turtle.write_docstringdict(filename="turtle_docstringdict")`

<b>Parameters:</b>	<b>filename</b> – a string, used as filename
--------------------	--

Create and write docstring-dictionary to a Python script with the given filename. This function has to be called explicitly (it is not used by the turtle graphics classes). The docstring dictionary will be written to the Python script `filename.py`. It is intended to serve as a template for translation of the docstrings into different languages.

If you (or your students) want to use `turtle` with online help in your native language, you have to translate the docstrings and save the resulting file as e.g. `turtle_docstringdict_german.py`.

If you have an appropriate entry in your `turtle.cfg` file this dictionary will be read in at import time and will replace the original English docstrings.

At the time of this writing there are docstring dictionaries in German and in Italian. (Requests please to [glingl@aon.at](mailto:glingl@aon.at).)

### 24.1.6.3. How to configure Screen and Turtles

The built-in default configuration mimics the appearance and behaviour of the old turtle module in order to retain best possible compatibility with it.

If you want to use a different configuration which better reflects the features of this module or which better fits to your needs, e.g. for use in a classroom, you can prepare a configuration file `turtle.cfg` which will be read at import time and modify the configuration according to its settings.

The built in configuration would correspond to the following `turtle.cfg`:

```
width = 0.5
height = 0.75
leftright = None
topbottom = None
canvwidth = 400
canvheight = 300
mode = standard
colormode = 1.0
delay = 10
undobuffersize = 1000
shape = classic
pencolor = black
fillcolor = black
resizemode = noresize
visible = True
language = english
exampleturtle = turtle
examplescreen = screen
title = Python Turtle Graphics
using_IDLE = False
```

Short explanation of selected entries:

- The first four lines correspond to the arguments of the `Screen.setup()` method.
- Line 5 and 6 correspond to the arguments of the method `Screen.screensize()`.
- *shape* can be any of the built-in shapes, e.g: arrow, turtle, etc. For more info try `help(shape)`.

- If you want to use no fillcolor (i.e. make the turtle transparent), you have to write `fillcolor = ""` (but all nonempty strings must not have quotes in the `cfg`-file).
- If you want to reflect the turtle its state, you have to use `resizemode = auto`.
- If you set e.g. `language = italian` the `docstringdict_turtle_docstringdict_italian.py` will be loaded at import time (if present on the import path, e.g. in the same directory as `turtle`).
- The entries `exampleturtle` and `examplescreen` define the names of these objects as they occur in the docstrings. The transformation of method-docstrings to function-docstrings will delete these names from the docstrings.
- `using_IDLE`: Set this to `True` if you regularly work with IDLE and its `-n` switch (“no subprocess”). This will prevent `exitonclick()` to enter the mainloop.

There can be a `turtle.cfg` file in the directory where `turtle` is stored and an additional one in the current working directory. The latter will override the settings of the first one.

The `Lib/turtledemo` directory contains a `turtle.cfg` file. You can study it as an example and see its effects when running the demos (preferably not from within the demo-viewer).

## 24.1.7. `turtledemo` — Demo scripts

The `turtledemo` package includes a set of demo scripts. These scripts can be run and viewed using the supplied demo viewer as follows:

```
python -m turtledemo
```

Alternatively, you can run the demo scripts individually. For example,

```
python -m turtledemo.bytedesign
```

The `turtledemo` package directory contains:

- A demo viewer `__main__.py` which can be used to view the sourcecode of the scripts and run them at the same time.
- Multiple scripts demonstrating different features of the `turtle` module. Examples can be accessed via the Examples menu. They can also be run standalone.
- A `turtle.cfg` file which serves as an example of how to write and use such files.

The demo scripts are:

Name	Description	Features



Name	Description	Features
bytedesign	complex classical turtle graphics pattern	tracer(), delay, update()
chaos	graphs Verhulst dynamics, shows that computer's computations can generate results sometimes against the common sense expectations	world coordinates
clock	analog clock showing time of your computer	turtles as clock's hands, ontimer
colormixer	experiment with r, g, b	ondrag()
forest	3 breadth-first trees	randomization
fractalcurves	Hilbert & Koch curves	recursion
lindenmayer	ethnomathematics (indian kolams)	L-System
minimal_hanoi	Towers of Hanoi	Rectangular Turtles as Hanoi discs (shape, shapesize)
nim	play the classical nim game with three heaps of sticks against the computer.	turtles as nimsticks, event driven (mouse, keyboard)
paint	super minimalistic drawing program	onclick()
peace	elementary	turtle: appearance and animation
penrose	aperiodic tiling with kites and darts	stamp()
planet_and_moon	simulation of gravitational system	compound shapes, Vec2D
round_dance	dancing turtles rotating pairwise in opposite direction	compound shapes, clone shapesize, tilt, get_shapepoly, update
sorting_animate	visual demonstration of different sorting methods	simple alignment, randomization
tree	a (graphical) breadth first tree (using generators)	clone()
two_canvases	simple design	turtles on two canvases
wikipedia	a pattern from the wikipedia article on turtle graphics	clone(), undo()
yinyang	another elementary example	circle()

Have fun!

## 24.1.8. Changes since Python 2.6

- The methods `Turtle.tracer()`, `Turtle.window_width()` and `Turtle.window_height()` have been eliminated. Methods with these names and functionality are now available only as methods of `Screen`. The functions derived from these remain available. (In fact already in Python 2.6 these methods were merely duplications of the corresponding `TurtleScreen/Screen`-methods.)
- The method `Turtle.fill()` has been eliminated. The behaviour of `begin_fill()` and `end_fill()` have changed slightly: now every filling-process must be completed with an `end_fill()` call.
- A method `Turtle.filling()` has been added. It returns a boolean value: `True` if a filling process is under way, `False` otherwise. This behaviour corresponds to a `fill()` call without arguments in Python 2.6.

## 24.1.9. Changes since Python 3.0

- The methods `Turtle.shearfactor()`, `Turtle.shapetransform()` and `Turtle.get_shapepoly()` have been added. Thus the full range of regular linear transforms is now available for transforming turtle shapes. `Turtle.tiltangle()` has been enhanced in functionality: it now can be used to get or set the tiltangle. `Turtle.settiltangle()` has been deprecated.
- The method `Screen.onkeypress()` has been added as a complement to `Screen.onkey()` which in fact binds actions to the keyrelease event. Accordingly the latter has got an alias: `Screen.onkeyrelease()`.
- The method `Screen.mainloop()` has been added. So when working only with `Screen` and `Turtle` objects one must not additionally import `mainloop()` anymore.
- Two input methods has been added `Screen.textinput()` and `Screen.numinput()`. These popup input dialogs and return strings and numbers respectively.
- Two example scripts `tdemo_nim.py` and `tdemo_round_dance.py` have been added to the `Lib/turtledemo` directory.