

PROGRAM FRAMEWORKS

The modules described in this chapter are frameworks that will largely dictate the structure of your program. Currently the modules described here are all oriented toward writing command-line interfaces.

The full list of modules described in this chapter is:

25.1 `turtle` — Turtle graphics

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

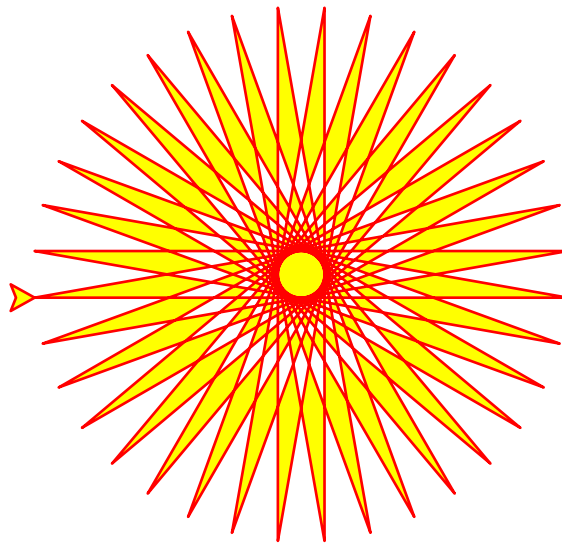
25.1.1 Introduction

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.

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()
```

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 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.

25.1.2 Overview of available Turtle and Screen methods

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()
```

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()
```

25.1.3 Methods of RawTurtle/Turtle and corresponding functions

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

Turtle motion

`turtle.forward(distance)`

`turtle.fd(distance)`

Parameters *distance* – 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)
```

Parameters

- **x** – a number or a pair/vector of numbers
- **y** – a number or None

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)
```

Parameters **x** – 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)
```

Parameters **y** – 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
```

(continues on next page)

(continued from previous page)

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

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

Parameters

- **size** – an integer ≥ 1 (if given)
- **color** – a colorstring or a numeric color tuple

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

```
>>> 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)`

Parameters `stampid` – an integer, must be return value of previous `stamp()` 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)`

Parameters `n` – 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)`

Parameters `speed` – an integer in the range 0..10 or a speedstring (see below)

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

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

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)`**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 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
```

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
```

Pen control

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()

```

(continues on next page)

(continued from previous page)

```
>>> 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
```

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')
```

(continues on next page)

(continued from previous page)

```
>>> 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'
>>> turtle.pencolor()
(50.0, 193.0, 143.0)
>>> turtle.fillcolor((50, 193, 143)) # Integers, not floats
>>> 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")
```

(continues on next page)

(continued from previous page)

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

See also: Screen method `colormode()`.

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()
```

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"))`

Parameters

- **arg** – object to be written to the TurtleScreen

- **move** – True/False
- **align** – one of the strings “left”, “center” or right”
- **font** – a triple (fontname, fontsize, fonttype)

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

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

Turtle state

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
```

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")
```

(continues on next page)

(continued from previous page)

```
>>> 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.shapesize(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 *resizemode* to “user”. If and only if *resizemode* is set to “user”, the turtle will be displayed stretched according to its stretchfactors: *stretch_wid* is stretchfactor perpendicular to 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 turtleshape 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)
- **t12** – 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 turtleshape according to the matrix consisting of first row t11, t12 and second row t21, 22. 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))
```

Using events

turtle.onclick(*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
- **btn** – 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-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)

Parameters

- **fun** – a function with two arguments which will be called with the coordinates of the clicked point on the canvas
- **btn** – 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-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 fillcolor red,
>>> turtle.onrelease(turtle.unglow)  # releasing turns it to transparent.
```

`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
- **btn** – 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 hand-drawings (if pen is down).

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()
```

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!

25.1.4 Methods of TurtleScreen/Screen and corresponding functions

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

Window control

`turtle.bgcolor(*args)`

Parameters `args` – a color string or three numbers in the range 0..colormode 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)`

Parameters

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

Set up user-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
```

Animation control

`turtle.delay(delay=None)`

Parameters **delay** – 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)`

Parameters

- **n** – nonnegative integer
- **delay** – nonnegative integer

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\(\)](#).

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)`

Parameters

- **fun** – a function with no arguments or `None`
- **key** – a string: key (e.g. “a”) or key-symbol (e.g. “space”)

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)`

Parameters

- **fun** – a function with no arguments or `None`
- **key** – a string: key (e.g. “a”) or key-symbol (e.g. “space”)

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)`

Parameters

- **fun** – a function with two arguments which will be called with the coordinates of the clicked point on the canvas
- **btn** – 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-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 TurtleScreen will
>>>                               # make the turtle move to the clicked point.
>>> 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)`

Parameters

- **fun** – a function with no arguments
- **t** – a number ≥ 0

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()
```

Input methods

`turtle.textinput(title, prompt)`

Parameters

- **title** – string
- **prompt** – string

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)`

Parameters

- **title** – string
- **prompt** – string
- **default** – number (optional)
- **minval** – number (optional)
- **maxval** – number (optional)

Pop up a dialog window for input of a number. title is the title of the dialog window, prompt is a text mostly describing what numerical information to input. default: default value, minval: minimum value for input, maxval: maximum value for input The number input must be in the range minval .. maxval if these are given. If not, a hint is issued and the dialog remains open for correction. Return the number input. If the dialog is canceled, return None.

```
>>> screen.numinput("Poker", "Your stakes:", 1000, minval=10, maxval=10000)
```

Settings and special methods

`turtle.mode(mode=None)`

Parameters **mode** – one of the strings “standard”, “logo” or “world”

Set turtle mode (“standard”, “logo” or “world”) and perform reset. If mode is not given, current mode is returned.

Mode “standard” is compatible with old *turtle*. Mode “logo” is compatible with most Logo turtle graphics. Mode “world” uses user-defined “world coordinates”. **Attention:** in this mode angles appear distorted if x/y unit-ratio doesn’t equal 1.

Mode	Initial turtle heading	positive angles
“standard”	to the right (east)	counterclockwise
“logo”	upward (north)	clockwise

```
>>> mode("logo")    # resets turtle heading to north
>>> mode()
'logo'
```

`turtle.colormode(cmode=None)`

Parameters *cmode* – one of the values 1.0 or 255

Return the colormode or set it to 1.0 or 255. Subsequently *r*, *g*, *b* values of color triples have to be in the range 0..*cmode*.

```
>>> screen.colormode(1)
>>> turtle.pencolor(240, 160, 80)
Traceback (most recent call last):
...
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
```

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
>>> screen.setup(width=.75, height=0.5, startx=None, starty=None)
>>>             # sets window to 75% of screen by 50% of screen and centers
```

`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!")
```

25.1.5 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)
```

Parameters `type_` – 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”	<code>None</code> (a compound shape has to be constructed using the <i>addcomponent()</i> method)

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

Parameters

- **poly** – a polygon, i.e. a tuple of pairs of numbers
- **fill** – a color the *poly* will be filled with
- **outline** – a color for the poly’s outline (if given)

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

25.1.6 Help and configuration

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
```

(continues on next page)

(continued from previous page)

```
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.

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")
```

Parameters `filename` – 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 glingsl@aon.at.)

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).

25.1.7 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
bytedesign	complex classical turtle graphics pattern	<code>tracer()</code> , <code>delay</code> , <code>update()</code>
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, <code>ontimer</code>
colormixer	experiment with r, g, b	<code>ondrag()</code>
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, <code>shapeseize</code>)
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	<code>onclick()</code>
peace	elementary	turtle: appearance and animation
penrose	aperiodic tiling with kites and darts	<code>stamp()</code>
planet_and_moon	simulation of gravitational system	compound shapes, <code>Vec2D</code>
round_dance	dancing turtles rotating pairwise in opposite direction	compound shapes, <code>clone</code> , <code>shapeseize</code> , <code>tilt</code> , <code>get_shapepoly</code> , <code>update</code>
sorting_animate	visual demonstration of different sorting methods	simple alignment, randomization
tree	a (graphical) breadth first tree (using generators)	<code>clone()</code>
two_canvases	simple design	turtles on two canvases
wikipedia	a pattern from the wikipedia article on turtle graphics	<code>clone()</code> , <code>undo()</code>
yinyang	another elementary example	<code>circle()</code>

Have fun!

25.1.8 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.

25.1.9 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.

25.2 cmd — Support for line-oriented command interpreters

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

The `Cmd` class provides a simple framework for writing line-oriented command interpreters. These are often useful for test harnesses, administrative tools, and prototypes that will later be wrapped in a more sophisticated interface.

class `cmd.Cmd`(*completekey='tab', stdin=None, stdout=None*)

A `Cmd` instance or subclass instance is a line-oriented interpreter framework. There is no good reason to instantiate `Cmd` itself; rather, it's useful as a superclass of an interpreter class you define yourself in order to inherit `Cmd`'s methods and encapsulate action methods.

The optional argument *completekey* is the *readline* name of a completion key; it defaults to `Tab`. If *completekey* is not `None` and *readline* is available, command completion is done automatically.

The optional arguments *stdin* and *stdout* specify the input and output file objects that the `Cmd` instance or subclass instance will use for input and output. If not specified, they will default to `sys.stdin` and `sys.stdout`.

If you want a given *stdin* to be used, make sure to set the instance's *use_rawinput* attribute to `False`, otherwise *stdin* will be ignored.

25.2.1 Cmd Objects

A `Cmd` instance has the following methods:

Cmd.cmdloop(*intro=None*)

Repeatedly issue a prompt, accept input, parse an initial prefix off the received input, and dispatch to action methods, passing them the remainder of the line as argument.

The optional argument is a banner or intro string to be issued before the first prompt (this overrides the *intro* class attribute).

If the *readline* module is loaded, input will automatically inherit **bash**-like history-list editing (e.g. **Control-P** scrolls back to the last command, **Control-N** forward to the next one, **Control-F** moves the cursor to the right non-destructively, **Control-B** moves the cursor to the left non-destructively, etc.).

An end-of-file on input is passed back as the string 'EOF'.

An interpreter instance will recognize a command name `foo` if and only if it has a method `do_foo()`. As a special case, a line beginning with the character '?' is dispatched to the method `do_help()`. As another special case, a line beginning with the character '!' is dispatched to the method `do_shell()` (if such a method is defined).

This method will return when the `postcmd()` method returns a true value. The `stop` argument to `postcmd()` is the return value from the command's corresponding `do_*`() method.

If completion is enabled, completing commands will be done automatically, and completing of commands args is done by calling `complete_foo()` with arguments *text*, *line*, *begidx*, and *endidx*. *text* is the string prefix we are attempting to match: all returned matches must begin with it. *line* is the current input line with leading whitespace removed, *begidx* and *endidx* are the beginning and ending indexes of the prefix text, which could be used to provide different completion depending upon which position the argument is in.

All subclasses of `Cmd` inherit a predefined `do_help()`. This method, called with an argument 'bar', invokes the corresponding method `help_bar()`, and if that is not present, prints the docstring of `do_bar()`, if available. With no argument, `do_help()` lists all available help topics (that is, all commands with corresponding `help_*`() methods or commands that have docstrings), and also lists any undocumented commands.

`Cmd.onecmd(str)`

Interpret the argument as though it had been typed in response to the prompt. This may be overridden, but should not normally need to be; see the `precmd()` and `postcmd()` methods for useful execution hooks. The return value is a flag indicating whether interpretation of commands by the interpreter should stop. If there is a `do_*`() method for the command *str*, the return value of that method is returned, otherwise the return value from the `default()` method is returned.

`Cmd.emptyline()`

Method called when an empty line is entered in response to the prompt. If this method is not overridden, it repeats the last nonempty command entered.

`Cmd.default(line)`

Method called on an input line when the command prefix is not recognized. If this method is not overridden, it prints an error message and returns.

`Cmd.completedefault(text, line, begidx, endidx)`

Method called to complete an input line when no command-specific `complete_*`() method is available. By default, it returns an empty list.

`Cmd.precmd(line)`

Hook method executed just before the command line *line* is interpreted, but after the input prompt is generated and issued. This method is a stub in `Cmd`; it exists to be overridden by subclasses. The return value is used as the command which will be executed by the `onecmd()` method; the `precmd()` implementation may re-write the command or simply return *line* unchanged.

`Cmd.postcmd(stop, line)`

Hook method executed just after a command dispatch is finished. This method is a stub in `Cmd`; it exists to be overridden by subclasses. *line* is the command line which was executed, and *stop* is a flag which indicates whether execution will be terminated after the call to `postcmd()`; this will be the return value of the `onecmd()` method. The return value of this method will be used as the new value for the internal flag which corresponds to *stop*; returning false will cause interpretation to continue.

`Cmd.preloop()`

Hook method executed once when `cmdloop()` is called. This method is a stub in `Cmd`; it exists to be overridden by subclasses.

`Cmd.postloop()`

Hook method executed once when `cmdloop()` is about to return. This method is a stub in `Cmd`; it

exists to be overridden by subclasses.

Instances of `Cmd` subclasses have some public instance variables:

`Cmd.prompt`

The prompt issued to solicit input.

`Cmd.identchars`

The string of characters accepted for the command prefix.

`Cmd.lastcmd`

The last nonempty command prefix seen.

`Cmd.cmdqueue`

A list of queued input lines. The `cmdqueue` list is checked in `cmdloop()` when new input is needed; if it is nonempty, its elements will be processed in order, as if entered at the prompt.

`Cmd.intro`

A string to issue as an intro or banner. May be overridden by giving the `cmdloop()` method an argument.

`Cmd.doc_header`

The header to issue if the help output has a section for documented commands.

`Cmd.misc_header`

The header to issue if the help output has a section for miscellaneous help topics (that is, there are `help_*`() methods without corresponding `do_*`() methods).

`Cmd.undoc_header`

The header to issue if the help output has a section for undocumented commands (that is, there are `do_*`() methods without corresponding `help_*`() methods).

`Cmd.ruler`

The character used to draw separator lines under the help-message headers. If empty, no ruler line is drawn. It defaults to '='.

`Cmd.use_rawinput`

A flag, defaulting to true. If true, `cmdloop()` uses `input()` to display a prompt and read the next command; if false, `sys.stdout.write()` and `sys.stdin.readline()` are used. (This means that by importing `readline`, on systems that support it, the interpreter will automatically support Emacs-like line editing and command-history keystrokes.)

25.2.2 Cmd Example

The `cmd` module is mainly useful for building custom shells that let a user work with a program interactively.

This section presents a simple example of how to build a shell around a few of the commands in the `turtle` module.

Basic turtle commands such as `forward()` are added to a `Cmd` subclass with method named `do_forward()`. The argument is converted to a number and dispatched to the turtle module. The docstring is used in the help utility provided by the shell.

The example also includes a basic record and playback facility implemented with the `precmd()` method which is responsible for converting the input to lowercase and writing the commands to a file. The `do_playback()` method reads the file and adds the recorded commands to the `cmdqueue` for immediate playback:

```
import cmd, sys
from turtle import *

class TurtleShell(cmd.Cmd):
```

(continues on next page)

(continued from previous page)

```

intro = 'Welcome to the turtle shell.  Type help or ? to list commands.\n'
prompt = '(turtle) '
file = None

# ----- basic turtle commands -----
def do_forward(self, arg):
    'Move the turtle forward by the specified distance:  FORWARD 10'
    forward(*parse(arg))
def do_right(self, arg):
    'Turn turtle right by given number of degrees:  RIGHT 20'
    right(*parse(arg))
def do_left(self, arg):
    'Turn turtle left by given number of degrees:  LEFT 90'
    left(*parse(arg))
def do_goto(self, arg):
    'Move turtle to an absolute position with changing orientation.  GOTO 100 200'
    goto(*parse(arg))
def do_home(self, arg):
    'Return turtle to the home position:  HOME'
    home()
def do_circle(self, arg):
    'Draw circle with given radius an options extent and steps:  CIRCLE 50'
    circle(*parse(arg))
def do_position(self, arg):
    'Print the current turtle position:  POSITION'
    print('Current position is %d %d\n' % position())
def do_heading(self, arg):
    'Print the current turtle heading in degrees:  HEADING'
    print('Current heading is %d\n' % (heading(),))
def do_color(self, arg):
    'Set the color:  COLOR BLUE'
    color(arg.lower())
def do_undo(self, arg):
    'Undo (repeatedly) the last turtle action(s):  UNDO'
def do_reset(self, arg):
    'Clear the screen and return turtle to center:  RESET'
    reset()
def do_bye(self, arg):
    'Stop recording, close the turtle window, and exit:  BYE'
    print('Thank you for using Turtle')
    self.close()
    bye()
    return True

# ----- record and playback -----
def do_record(self, arg):
    'Save future commands to filename:  RECORD rose.cmd'
    self.file = open(arg, 'w')
def do_playback(self, arg):
    'Playback commands from a file:  PLAYBACK rose.cmd'
    self.close()
    with open(arg) as f:

```

(continues on next page)

(continued from previous page)

```

        self.cmdqueue.extend(f.read().splitlines())
    def precmd(self, line):
        line = line.lower()
        if self.file and 'playback' not in line:
            print(line, file=self.file)
        return line
    def close(self):
        if self.file:
            self.file.close()
            self.file = None

def parse(arg):
    'Convert a series of zero or more numbers to an argument tuple'
    return tuple(map(int, arg.split()))

if __name__ == '__main__':
    TurtleShell().cmdloop()

```

Here is a sample session with the turtle shell showing the help functions, using blank lines to repeat commands, and the simple record and playback facility:

```

Welcome to the turtle shell.   Type help or ? to list commands.

(turtle) ?

Documented commands (type help <topic>):
=====
bye      color      goto      home      playback  record    right
circle  forward  heading  left      position  reset     undo

(turtle) help forward
Move the turtle forward by the specified distance:  FORWARD 10
(turtle) record spiral.cmd
(turtle) position
Current position is 0 0

(turtle) heading
Current heading is 0

(turtle) reset
(turtle) circle 20
(turtle) right 30
(turtle) circle 40
(turtle) right 30
(turtle) circle 60
(turtle) right 30
(turtle) circle 80
(turtle) right 30
(turtle) circle 100
(turtle) right 30
(turtle) circle 120
(turtle) right 30

```

(continues on next page)

(continued from previous page)

```

(turtle) circle 120
(turtle) heading
Current heading is 180

(turtle) forward 100
(turtle)
(turtle) right 90
(turtle) forward 100
(turtle)
(turtle) right 90
(turtle) forward 400
(turtle) right 90
(turtle) forward 500
(turtle) right 90
(turtle) forward 400
(turtle) right 90
(turtle) forward 300
(turtle) playback spiral.cmd
Current position is 0 0

Current heading is 0

Current heading is 180

(turtle) bye
Thank you for using Turtle

```

25.3 shlex — Simple lexical analysis

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

The *shlex* class makes it easy to write lexical analyzers for simple syntaxes resembling that of the Unix shell. This will often be useful for writing minilanguages, (for example, in run control files for Python applications) or for parsing quoted strings.

The *shlex* module defines the following functions:

shlex.split(*s*, *comments=False*, *posix=True*)

Split the string *s* using shell-like syntax. If *comments* is *False* (the default), the parsing of comments in the given string will be disabled (setting the *commenters* attribute of the *shlex* instance to the empty string). This function operates in POSIX mode by default, but uses non-POSIX mode if the *posix* argument is false.

Note: Since the *split()* function instantiates a *shlex* instance, passing *None* for *s* will read the string to split from standard input.

shlex.quote(*s*)

Return a shell-escaped version of the string *s*. The returned value is a string that can safely be used as one token in a shell command line, for cases where you cannot use a list.

This idiom would be unsafe:

```
>>> filename = 'somefile; rm -rf ~'
>>> command = 'ls -l {}'.format(filename)
>>> print(command) # executed by a shell: boom!
ls -l somefile; rm -rf ~
```

`quote()` lets you plug the security hole:

```
>>> from shlex import quote
>>> command = 'ls -l {}'.format(quote(filename))
>>> print(command)
ls -l 'somefile; rm -rf ~'
>>> remote_command = 'ssh home {}'.format(quote(command))
>>> print(remote_command)
ssh home 'ls -l \''somefile; rm -rf ~\''
```

The quoting is compatible with UNIX shells and with `split()`:

```
>>> from shlex import split
>>> remote_command = split(remote_command)
>>> remote_command
['ssh', 'home', 'ls -l 'somefile; rm -rf ~']
>>> command = split(remote_command[-1])
>>> command
['ls', '-l', 'somefile; rm -rf ~']
```

New in version 3.3.

The `shlex` module defines the following class:

class `shlex.shlex`(*instream=None*, *infile=None*, *posix=False*, *punctuation_chars=False*)

A `shlex` instance or subclass instance is a lexical analyzer object. The initialization argument, if present, specifies where to read characters from. It must be a file-/stream-like object with `read()` and `readline()` methods, or a string. If no argument is given, input will be taken from `sys.stdin`. The second optional argument is a filename string, which sets the initial value of the `infile` attribute. If the `instream` argument is omitted or equal to `sys.stdin`, this second argument defaults to “stdin”. The `posix` argument defines the operational mode: when `posix` is not true (default), the `shlex` instance will operate in compatibility mode. When operating in POSIX mode, `shlex` will try to be as close as possible to the POSIX shell parsing rules. The `punctuation_chars` argument provides a way to make the behaviour even closer to how real shells parse. This can take a number of values: the default value, `False`, preserves the behaviour seen under Python 3.5 and earlier. If set to `True`, then parsing of the characters `();<>|&` is changed: any run of these characters (considered punctuation characters) is returned as a single token. If set to a non-empty string of characters, those characters will be used as the punctuation characters. Any characters in the `wordchars` attribute that appear in `punctuation_chars` will be removed from `wordchars`. See *Improved Compatibility with Shells* for more information.

Changed in version 3.6: The `punctuation_chars` parameter was added.

See also:

Module `configparser` Parser for configuration files similar to the Windows `.ini` files.

25.3.1 shlex Objects

A `shlex` instance has the following methods:

`shlex.get_token()`

Return a token. If tokens have been stacked using `push_token()`, pop a token off the stack. Otherwise,

read one from the input stream. If reading encounters an immediate end-of-file, *eof* is returned (the empty string ('') in non-POSIX mode, and *None* in POSIX mode).

shlex.push_token(*str*)

Push the argument onto the token stack.

shlex.read_token()

Read a raw token. Ignore the pushback stack, and do not interpret source requests. (This is not ordinarily a useful entry point, and is documented here only for the sake of completeness.)

shlex.sourcehook(*filename*)

When *shlex* detects a source request (see *source* below) this method is given the following token as argument, and expected to return a tuple consisting of a filename and an open file-like object.

Normally, this method first strips any quotes off the argument. If the result is an absolute pathname, or there was no previous source request in effect, or the previous source was a stream (such as *sys.stdin*), the result is left alone. Otherwise, if the result is a relative pathname, the directory part of the name of the file immediately before it on the source inclusion stack is prepended (this behavior is like the way the C preprocessor handles *#include "file.h"*).

The result of the manipulations is treated as a filename, and returned as the first component of the tuple, with *open()* called on it to yield the second component. (Note: this is the reverse of the order of arguments in instance initialization!)

This hook is exposed so that you can use it to implement directory search paths, addition of file extensions, and other namespace hacks. There is no corresponding ‘close’ hook, but a *shlex* instance will call the *close()* method of the sourced input stream when it returns EOF.

For more explicit control of source stacking, use the *push_source()* and *pop_source()* methods.

shlex.push_source(*newstream*, *newfile=None*)

Push an input source stream onto the input stack. If the filename argument is specified it will later be available for use in error messages. This is the same method used internally by the *sourcehook()* method.

shlex.pop_source()

Pop the last-pushed input source from the input stack. This is the same method used internally when the lexer reaches EOF on a stacked input stream.

shlex.error_leader(*infile=None*, *lineno=None*)

This method generates an error message leader in the format of a Unix C compiler error label; the format is *"%s", line %d: '*, where the *%s* is replaced with the name of the current source file and the *%d* with the current input line number (the optional arguments can be used to override these).

This convenience is provided to encourage *shlex* users to generate error messages in the standard, parseable format understood by Emacs and other Unix tools.

Instances of *shlex* subclasses have some public instance variables which either control lexical analysis or can be used for debugging:

shlex.commenters

The string of characters that are recognized as comment beginners. All characters from the comment beginner to end of line are ignored. Includes just '#' by default.

shlex.wordchars

The string of characters that will accumulate into multi-character tokens. By default, includes all ASCII alphanumerics and underscore. In POSIX mode, the accented characters in the Latin-1 set are also included. If *punctuation_chars* is not empty, the characters *~./*?='*, which can appear in filename specifications and command line parameters, will also be included in this attribute, and any characters which appear in *punctuation_chars* will be removed from *wordchars* if they are present there.

shlex.whitespace

Characters that will be considered whitespace and skipped. Whitespace bounds tokens. By default, includes space, tab, linefeed and carriage-return.

shlex.escape

Characters that will be considered as escape. This will be only used in POSIX mode, and includes just `'\'` by default.

shlex.quotes

Characters that will be considered string quotes. The token accumulates until the same quote is encountered again (thus, different quote types protect each other as in the shell.) By default, includes ASCII single and double quotes.

shlex.escapedquotes

Characters in *quotes* that will interpret escape characters defined in *escape*. This is only used in POSIX mode, and includes just `''` by default.

shlex.whitespace_split

If `True`, tokens will only be split in whitespaces. This is useful, for example, for parsing command lines with *shlex*, getting tokens in a similar way to shell arguments. If this attribute is `True`, *punctuation_chars* will have no effect, and splitting will happen only on whitespaces. When using *punctuation_chars*, which is intended to provide parsing closer to that implemented by shells, it is advisable to leave *whitespace_split* as `False` (the default value).

shlex.infile

The name of the current input file, as initially set at class instantiation time or stacked by later source requests. It may be useful to examine this when constructing error messages.

shlex.instream

The input stream from which this *shlex* instance is reading characters.

shlex.source

This attribute is `None` by default. If you assign a string to it, that string will be recognized as a lexical-level inclusion request similar to the `source` keyword in various shells. That is, the immediately following token will be opened as a filename and input will be taken from that stream until EOF, at which point the *close()* method of that stream will be called and the input source will again become the original input stream. Source requests may be stacked any number of levels deep.

shlex.debug

If this attribute is numeric and 1 or more, a *shlex* instance will print verbose progress output on its behavior. If you need to use this, you can read the module source code to learn the details.

shlex.lineno

Source line number (count of newlines seen so far plus one).

shlex.token

The token buffer. It may be useful to examine this when catching exceptions.

shlex.eof

Token used to determine end of file. This will be set to the empty string (`''`), in non-POSIX mode, and to `None` in POSIX mode.

shlex.punctuation_chars

Characters that will be considered punctuation. Runs of punctuation characters will be returned as a single token. However, note that no semantic validity checking will be performed: for example, `'>>>'` could be returned as a token, even though it may not be recognised as such by shells.

New in version 3.6.

25.3.2 Parsing Rules

When operating in non-POSIX mode, *shlex* will try to obey to the following rules.

- Quote characters are not recognized within words (Do"NotSeparate is parsed as the single word Do"NotSeparate);
- Escape characters are not recognized;
- Enclosing characters in quotes preserve the literal value of all characters within the quotes;
- Closing quotes separate words ("DoSeparate is parsed as "Do" and Separate);
- If *whitespace_split* is *False*, any character not declared to be a word character, whitespace, or a quote will be returned as a single-character token. If it is *True*, *shlex* will only split words in whitespaces;
- EOF is signaled with an empty string ('');
- It's not possible to parse empty strings, even if quoted.

When operating in POSIX mode, *shlex* will try to obey to the following parsing rules.

- Quotes are stripped out, and do not separate words ("Do"NotSeparate" is parsed as the single word DoNotSeparate);
- Non-quoted escape characters (e.g. '\') preserve the literal value of the next character that follows;
- Enclosing characters in quotes which are not part of *escapedquotes* (e.g. '"') preserve the literal value of all characters within the quotes;
- Enclosing characters in quotes which are part of *escapedquotes* (e.g. '"') preserves the literal value of all characters within the quotes, with the exception of the characters mentioned in *escape*. The escape characters retain its special meaning only when followed by the quote in use, or the escape character itself. Otherwise the escape character will be considered a normal character.
- EOF is signaled with a *None* value;
- Quoted empty strings (') are allowed.

25.3.3 Improved Compatibility with Shells

New in version 3.6.

The *shlex* class provides compatibility with the parsing performed by common Unix shells like *bash*, *dash*, and *sh*. To take advantage of this compatibility, specify the *punctuation_chars* argument in the constructor. This defaults to *False*, which preserves pre-3.6 behaviour. However, if it is set to *True*, then parsing of the characters `()<>|&` is changed: any run of these characters is returned as a single token. While this is short of a full parser for shells (which would be out of scope for the standard library, given the multiplicity of shells out there), it does allow you to perform processing of command lines more easily than you could otherwise. To illustrate, you can see the difference in the following snippet:

```
>>> import shlex
>>> text = "a && b; c && d || e; f >'abc'; (def \"ghi\")"
>>> list(shlex.shlex(text))
['a', '&', '&', 'b', ';', 'c', '&', '&', 'd', '|', '|', 'e', ';', 'f', '>',
'"abc"', ';', '(', 'def', '"ghi"', ')']
>>> list(shlex.shlex(text, punctuation_chars=True))
['a', '&&', 'b', ';', 'c', '&&', 'd', '||', 'e', ';', 'f', '>', '"abc"',
';', '(', 'def', '"ghi"', ')']
```

Of course, tokens will be returned which are not valid for shells, and you'll need to implement your own error checks on the returned tokens.

Instead of passing `True` as the value for the `punctuation_chars` parameter, you can pass a string with specific characters, which will be used to determine which characters constitute punctuation. For example:

```
>>> import shlex
>>> s = shlex.shlex("a && b || c", punctuation_chars="|")
>>> list(s)
['a', '&', '&', 'b', '||', 'c']
```

Note: When `punctuation_chars` is specified, the *wordchars* attribute is augmented with the characters `~-./*?='`. That is because these characters can appear in file names (including wildcards) and command-line arguments (e.g. `--color=auto`). Hence:

```
>>> import shlex
>>> s = shlex.shlex('~ /a && b-c --color=auto || d *.py?',
...                 punctuation_chars=True)
>>> list(s)
['~/a', '&&', 'b-c', '--color=auto', '||', 'd', '*.py?']
```

For best effect, `punctuation_chars` should be set in conjunction with `posix=True`. (Note that `posix=False` is the default for *shlex*.)