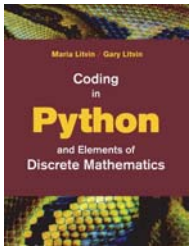


Coding in **Python** and Elements of Discrete Mathematics

Maria Litvin
Phillips Academy, Andover, Massachusetts

Gary Litvin
Skylight Software, Inc.

Skylight Publishing
Andover, Massachusetts

Skylight Publishing
9 Bartlet Street, Suite 70
Andover, MA 01810

web: http://www.skylit.com
e-mail: sales@skylit.com
support@skylit.com

Library of Congress Control Number: 2019905086

ISBN 978-0-9972528-4-2

1 2 3 4 5 6 7 23 22 21 20 19

Printed in the United States of America

Turtle Graphics

9.1 Prologue

> Alice thought to herself, "I don't see how he can *ever* finish, if he doesn't begin." But she waited patiently.
>
> "Once," said the Mock Turtle at last, with a deep sigh, "I was a real Turtle."
>
> *Alice's Adventures in Wonderland* by Lewis Carroll

The idea of using computers and robots for teaching young children arose over half a century ago. In the late 1960s, three researchers, Wally Feurzeig and Cynthia Solomon from the research firm Bolt, Beranek and Newman (BBN), and Seymour Papert, a fellow at the Artificial Intelligence (AI) lab at the Massachusetts Institute of Technology (MIT), designed a first programming language for children. They called their language Logo, from the Greek word "logos," which means "word" or "thought." In those days, computers were big and expensive and used only for "serious" applications (military, data processing, research); to many people the idea of kids using valuable computer time sounded crazy. Yet Logo thrived, and within a few years it became popular among teachers and was introduced in many schools.

At first, Logo was meant to introduce young kids to AI ideas and methods. But one of Logo's features was a *virtual* (not physically existing) robot that could follow simple commands and draw pictures on the computer screen. Papert's group called the robot a "turtle" in honor of earlier "turtle" robots created by Grey Walter in the late 1940s (Figure 9-1). (The name "turtle" was reportedly inspired by the Mock Turtle character in Lewis Carroll's *Alice in Wonderland*.)

A real turtle robot that executed Logo instructions was built at MIT in 1969. In 1972, BBN engineer Paul Wexelblat designed and built the first wireless floor turtle (Figure 9-2).

Logo's "turtle graphics" capability quickly overshadowed Logo's other features, and it became known primarily as the turtle graphics language. Logo is alive and well today: many Logo versions and apps exist as free downloads, and turtle graphics ideas are implemented in other graphics packages and programming languages such as Scratch and, of course, Python's turtle graphics *module* (library of functions).

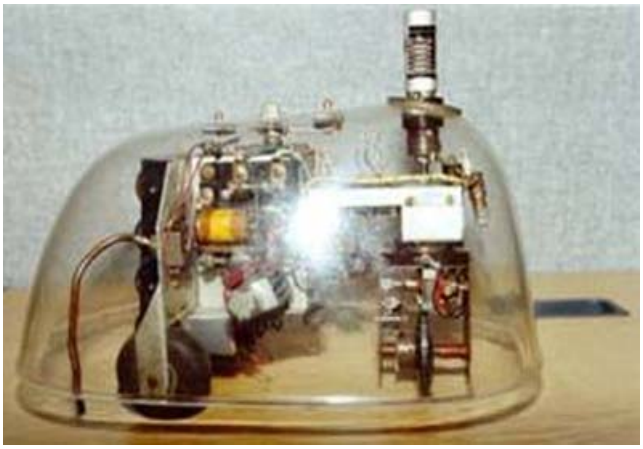


Figure 9-1. A reproduction of one of Grey Walter's "turtle" robots

Courtesy http://roamerrobot.tumblr.com/post/23079345849/the-history-of-turtle-robots

Figure 9-2. Paul Keelboat's wireless turtle, 1972

Courtesy http://cyberneticzoo.com/cyberneticanimals/
1969-the-logo-turtle-seymour-papert-marvin-minsky-et-al-american/

9.2 The `turtle` Module Basics

Python's `turtle` module comes with the standard Python installation from python.org. If you want to use `turtle`, you need to import it into your program:

```
from turtle import *  # import everything from the turtle module
```

If you wish, you can experiment with `turtle` commands (functions) directly from the Python shell. Try this:

```
>>> from turtle import *
>>> shape("turtle")
>>> forward(100)
```

A window will pop up with a line drawn by the turtle:

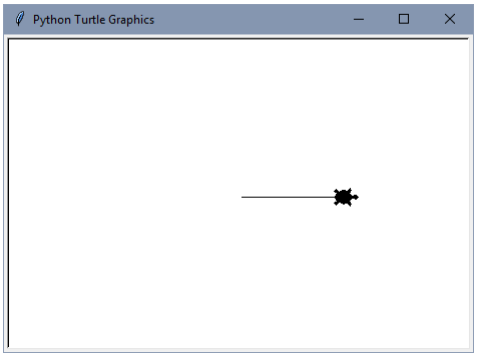


Table 9-1 shows the basic turtle commands (functions) needed to get you started. https://docs.python.org/3.7/library/turtle.html describes all the `turtle` and `screen` functions.

`turtle` functions are convenient and easy to use. Several names may be supported for the same function: a fully spelled-out name and an abbreviated name, such as `fd` for `forward`.

> **We will use full names of `turtle` functions in our programs for better readability and recommend you do the same — no need to save a few keystrokes.**

Function	Action
`shape(name)`	Choose turtle's shape: `'arrow'`, `'turtle'`, `'circle'`, `'square'`, `'triangle'`, or `'classic'` (default). You can define your own shape.
`speed(v)`	Set turtle's moving and drawing speed: `'fastest'` or 0, `'fast'` or 10, `'normal'` or 6, `'slow'` or 3 (default), `'slowest'` or 1.
`color(colorname)`	Set pen and fill colors. `colorname` can be a string, for example, `color('red')`. (Other formats are supported — see Section 9.3.)

`penup()`
`pu()`
`up()`
	Lift the "pen" from the "paper" (ready to move without drawing).

`pendown()`
`pd()`
`down()`
	Place the "pen" on the "paper" (ready to draw).

`forward(d)` `fd(d)`	Move forward by `d` units (while drawing or not).

`backward(d)`
`bk(d)`
`back(d)`
	Move backward by `d` units.

`right(deg)` `rt(deg)`	Turn ↻ (clockwise) by `deg` degrees.
`left(deg)` `lt(deg)`	Turn ↺ (counterclockwise) by `deg` degrees.
`showturtle()` or `st()` `hideturtle()` or `ht()`	Make the turtle visible. Make the turtle invisible.

Table 9-1. Basic `turtle` functions

The distances in `turtle` functions are in *pixels* ("picture elements") by default.

If the screen resolution in your device is listed, say, as 1920 by 1200, it means that the full screen is 1920 pixels horizontally and 1200 pixels vertically. The dimensions of the `turtle` graphics window are returned by the `window_width()` and `window_height()` functions. For example:

```
>>> from turtle import *
>>> window_width(), window_height()
(960, 900)
```

`setup(width, height)` defines your own custom size.

When a turtle is first created, it is placed at the center of the window, facing east (to the right), with its pen down, ready for drawing.

If you have trouble figuring out turtle graphics code, imagine that *you* are the turtle and try following the commands. Just don't draw on the rug!

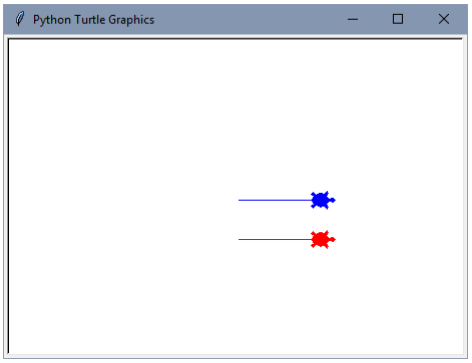
The statement `from turtle import *` not only imports all `turtle` functions into your program, but also creates an anonymous `turtle` object whose functions can be called without any name-dot prefix. For example:

```
from turtle import *
shape('turtle')
speed('fastest')
forward(100)
```

You can create any number of other turtles and give them names. To call a named turtle's functions, you need to use the name-dot prefix. For example:

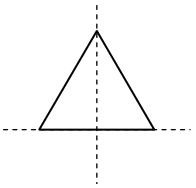
```
from turtle import *

setup(width=200, height=200)
alice = Turtle(shape='turtle')
alice.color('blue')
alice.forward(80)
bob = Turtle(shape='turtle')
bob.color('red')
bob.penup()
bob.right(90)
bob.forward(40)
bob.left(90)
bob.pendown()
bob.forward(80)
```

This displays



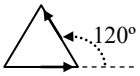
Example 1

Draw an equilateral triangle with a side length of 80 and a horizontal base centered in the middle of the graphics window:



Solution

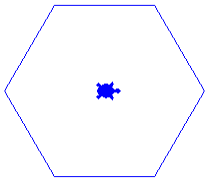
If the triangle is traced counterclockwise, the turtle needs to turn left by 120º after drawing each side:




```
from turtle import *

penup()
backward(40)
pendown()
for k in range(3):
    forward(80)
    left(120)
```

Example 2

Draw a blue regular hexagon (a hexagon whose sides are all the same length and whose angles are all the same) that is centered at the center of the graphics window and has sides 100 pixels long. Return the turtle to the center of the window when done:



Solution

```
from turtle import *

shape('turtle')
color('blue')
penup()
backward(100)
right(60)
pendown()
for k in range(6):
    forward(100)
    left(60)
penup()
left(60)
forward(100)
```


Example 3

Write and test a function that draws a rectangle with given dimensions using a specified turtle, starting from that turtle's current position and direction.

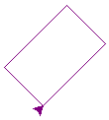
Solution

```
def draw_rectangle(t, w, h):
    """Draw a rectangle of width w and height h using the turtle t,
       going counterclockwise from its current position and direction
       and in its current color. Leave the turtle with its pen up.
    """
    t.pendown()
    for k in range(2):
        t.forward(w)
        t.left(90)
        t.forward(h)
        t.left(90)
    t.penup()
```

Then

```
escher = Turtle()
escher.color('purple')
escher.left(45)
draw_rectangle(escher, 89, 55)
```

draws



❖ ❖ ❖

`turtle` can also draw filled shapes. Table 9-2 shows the relevant functions.

Function	Action
`begin_fill()`	Register the current position as the starting position for a filled shape.
`end_fill()`	End registering and fill the registered shape.
`color(c1, c2)`	Set the pen color to `c1` and the fill color to `c2`.

Table 9-2. Functions for filling shapes

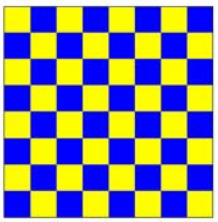
The `begin_fill()` call tells the turtle to register the current position as the starting position for a filled shape. The `end_fill()` call fills the area within all the lines drawn since the `begin_fill` call. `t.color(c)` sets both `t`'s pen color and fill color to c, but you can specify different pen and fill colors by calling `color` with two parameters. For example,

```
>>> color('red', 'yellow')
```

sets the pen color to red and the fill color to yellow. The calls `pencolor(c)` and `fillcolor(c)` set the pen color and the fill color, respectively.

Example 4

Write and test a function `draw_chessboard(t, size, colors)` that uses the turtle `t` to draw a chessboard:



`size` is the size of each square; `colors` is a tuple of two colors, for the dark and light squares. Use the `draw_rectangle` function from the previous example.

Solution

```
def draw_chessboard(t, size, colors):
    """Draw a chessboard using the turtle t with squares
       of a given size in given colors.
    """
    for row in range(8):
        for col in range(8):
            t.color(colors[(row + col)%2])
            t.begin_fill()
            draw_rectangle(t, size, size)
            t.end_fill()
            t.forward(size+1)
        t.back(8*size+8)
        t.right(90)
        t.forward(size+1)
        t.left(90)
```

```
    # Draw the border:
    t.back(1)
    t.left(90)
    t.forward(size)
    t.right(90)
    t.color('black')
    draw_rectangle(t, 8*size+8, 8*size+8)

deepblue = Turtle()  # IBM's Deep Blue chess supercomputer beat Garry
                     # Kasparov, then the chess world champion, in 1997
deepblue.speed('fastest')
deepblue.hideturtle()  # to speed up the drawing
deepblue.penup()
deepblue.back(200)
draw_chessboard(deepblue, 40, ('yellow', 'blue'))
```

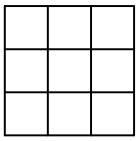
Or use the default anonymous turtle:

```
speed('fastest')
hideturtle()
penup()
back(200)
draw_chessboard(getturtle(), 40, ('yellow', 'blue'))
```

`getturtle()` returns the anonymous turtle.

Section 9.2 ~ Exercises

1. Draw a tic-tac-toe grid

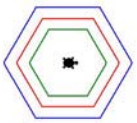


✓

2. Write a function `draw_polygon(n, a)` that uses a default turtle to draw a regular polygon with n sides of length a, in the current color. ≤ Hint: if you have n sides, you need to turn n times and cover the whole 360º angle at the end, so each turn is $\frac{360}{n}$ degrees. ≥

3. (a) Convert the code from Example 2 into a function `draw_hexagon(side)` that draws a hexagon with a given side length in the current color, centered at the current position. ✓

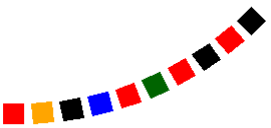
(b) Use the function from Part (a) to draw three concentric hexagons of different colors. For example:



✓

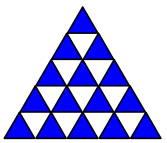
(c) Add a few lines of code to your solution for Part (b) to make the innermost hexagon filled, like this:

4. Using the `draw_rectangle` function from Example 3 or the `draw_polygon` function from Question **2**, draw a path, made of ten flagstones, that bends slightly upward. The colors of the flagstones should vary, chosen at random among red, blue, dark green, black, and orange. For example:

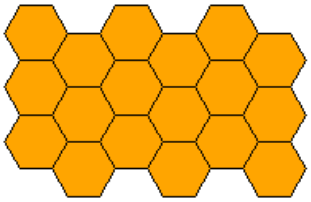


ⵑ Hint: Recall that the function `choice` from the `random` module returns a randomly chosen element of a list. ⵒ

5.■ Convert the code from Example 1 into the `draw_triangle` function and draw a pyramid of triangles like this:



6.◆ Using the `draw_polygon` function from Question 2, draw a honeycomb filled with "honey" (orange color):



7.◆ Write and test a function to draw a flower with a specified number of petals:

⦙ Hint: For a prettier flower, the number of petals should be an odd number. Add 1 if the specified number of petals is even. ⦙

9.3 Coordinates and Text

`turtle`'s coordinate system is similar to Cartesian coordinates in math: the *x*-axis is horizontal and points to the right; the *y*-axis is vertical and points up (unlike many other computer graphics packages where the *y*-axis points down). The origin is at the center of the graphics window (Figure 9-3). The default units are pixels. Angles are measured as in math, starting from the positive direction of the *x*-axis and going counterclockwise; the default units for angles are degrees.

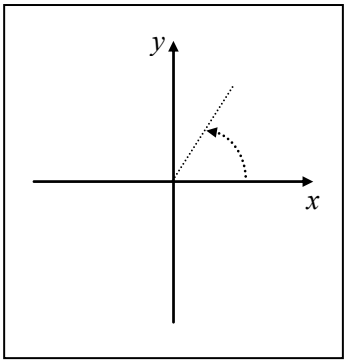


Figure 9-3. Python turtle graphics default coordinates

`turtle` has functions that change the size of the graphics window, the origin of the coordinate system, and the units, but we will stay with the defaults.

So far we have used turtle functions that move and turn the turtle relative to its current position and direction: `forward`, `backward`, `left` and `right`. These commands are easy for a robot to handle, and, in fact, for humans, too. But `turtle` also has several functions that deal with absolute coordinates and angles. These functions are summarized in Table 9-3.

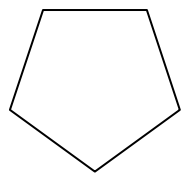
`turtle` also has a function `circle` that draws a circle of a given radius, starting at the current position and direction and going counterclockwise. `circle` also can draw an arc if the `extent` parameter is given. For example:

```
t = Turtle()
t.left(180)
t.circle(80, extent=180)
```

draws a semicircle:

`circle` will actually draw a polygon if the `steps` parameter is specified. For example:

```
>>> from turtle import *
>>> circle(80, steps=5)
>>> hideturtle()
```



Function	Action
`position()` `pos()`	Return turtle's current *x*-*y* coordinates (as a tuple).
`xcor()` `ycor()`	Return turtle's current *x*- or *y*-coordinate, respectively.
`distance(x, y)`	Return the distance from the turtle to the point (*x*, *y*).
`heading()`	Return turtle's current direction.
`towards(x, y)`	Return the angle from the *x*-axis to the vector (line) from the turtle to the point (*x*, *y*).

`setposition(x, y)`
`setpos(x, y)`
`goto(x, y)`
	Move the turtle to the point (*x*, *y*); the turtle's direction remains unchanged.

`setx(x)` `sety(y)`	Set the turtle's respective coordinate without changing the other coordinate or direction.
`setheading(to_angle)` `seth((to_angle)`	Change the turtle's direction to `to_angle`.
`home()`	Return the turtle to the origin, pointing east.

Table 9-3. `turtle` functions that use absolute coordinates and angles

The `dot(diameter, c)` function will draw a circular dot of the given diameter, filled with the color `c` and centered at the current turtle's position. `dot(diameter)` draws a dot in the current color.

Example 1

Draw a smiley face:

Solution

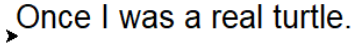
```
pensize(2)  # for thicker lines
circle(120)
penup()
setposition(-50, 140)
dot(30)
setposition(50, 140)
dot(30)
setposition(-40, 60)
setheading(-53.13) # the angle in the 3-4-5 triangle is arctan(4/3)
pendown()
pensize(4)
circle(50, extent=2*53.13)
penup()
hideturtle()
```

❖ ❖ ❖

`turtle`'s `write(msg, font=fnt)` function displays the string `msg` in a specified font. For example:

```
t = Turtle()
t.write('Once I was a real turtle.', font=('arial', 20))
```

displays



The text is displayed in the turtle's current color. The current direction of the turtle does not affect the text and remains unchanged.

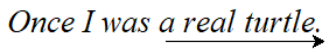
The `font` parameter is a tuple that includes the font name and size, which can be followed by `'bold'`, `'italic'`, and/or `'underline'` in any combination and order. For example:

```
t = Turtle('turtle')
t.color('blue')
t.write('Once I was a real turtle.', font=('Arial', 20, 'bold', 'italic'))
```

The available font names are those installed in your operating system, but in a portable program it is advisable to use only common fonts that are available in most systems, such as `'Arial'`, `'Times'`, and `'Courier'`, or just write `None` for the font name to use the default font.

By default, the left end of the baseline of text will be at the current turtle position. An optional parameter, `align='center'` or `align='right'`, will place the center or the right end of the baseline at the current position. The optional parameter, `move=True` will move the turtle to the end of the baseline (and draw if the pen is down). For example:

```
t.write('Once I was a real turtle.',
        font=('times', 20, 'italic'), align='center', move=True)
```



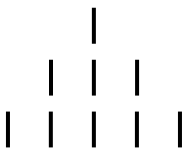
If the text string contains `'\n'` characters, `write` will correctly display multiple lines.

Section 9.3 ~ Exercises

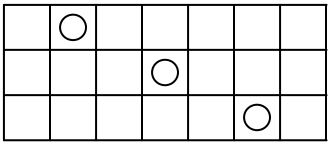
1. Draw a "hamburger" button

✓

2. The game of *Nim* is, theoretically, played with piles of stones, but it is commonly played with rows of sticks instead. Draw a configuration with three rows of sticks:

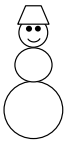


3. Another, *isomorphic* (mathematically identical) representation of Nim is tokens moving from left to right on a rectangular board. Draw the Nim configuration with three tokens:



(It is identical to the three rows of sticks in the previous question.) ✓

4.■ Draw a snowman:



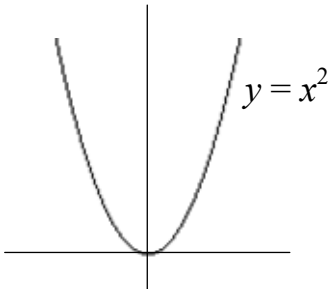
5. In the Tower of Hanoi puzzle, you need to transfer a pyramid of disks from one peg to another, using the third peg as a "spare." You can only move one disk at a time, and you may place it only on top of a larger disk or on the base. Draw a two-dimensional sketch of the puzzle with five disks:

6.■ Draw a stop sign:

Don't worry about an exact font match — Arial will do for this exercise. ✓

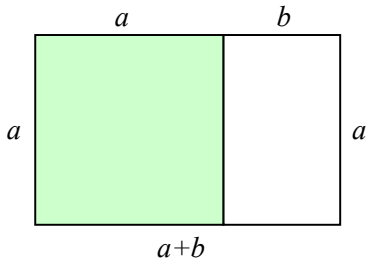
7.■ Display the code that draws a hexagon to the right of the hexagon it draws (see Example 2 in the previous section). Use the Courier font for the code.

8.■ Draw a fairly smooth graph of the parabola $y = x^2$ with a label to its right:



ⵑ Hint: generate the segment of the parabola for $-3 \le x \le 3$ but scale the graph by a factor of 30 or 40. ⵑ ✓

9.◆ Draw a diagram that illustrates the *golden ratio* (actually taken from the next chapter of this book). Add the equation to the right of the rectangle:



$$\frac{a+b}{a} = \frac{a}{b}$$

9.4 Colors

In turtle graphics a virtual turtle draws on virtual paper with a virtual pen. No pen exists, of course. What you see on your computer screen is ultimately determined by the contents of the video memory (VRAM) on the *graphics adapter* card or the graphics processor chip. VRAM represents a rectangular array of *pixels* (picture elements). Each pixel has a particular color, which can be represented as a mix of red, green, and blue components, each with its own intensity. A typical graphics adapter uses eight bits to represent each of the red, green, and blue (RGB) values (in the range from 0 to 255). The image on the screen is produced by setting the color of each pixel in VRAM. The video hardware scans the whole video memory continuously and refreshes the image on the screen.

A graphics processor is what we call a *raster* device: each individual pixel can be set separately from other pixels. (This is different from a *vector* device, such as a plotter, which actually draws lines on paper directly from point *A* to point *B*, with a pen of a particular color.) To draw a red line or a circle on a raster device, you need to set just the right group of pixels to the red color. That's where a graphics package helps: you certainly don't want to program all those functions for setting pixels yourself.

A graphics package has to provide functions for setting colors. Python's `turtle` inherits screen and color handling from the `tkinter` package (`Tk` interface), which is Python's standard toolkit for GUI (Graphical User Interface) development. `tkinter` uses names assigned to several hundred selected colors. (These names are standard in web app development environments.) You can find some of the named colors with their RGB components in hex and/or decimal form on many web sites, for example, https://trinket.io/docs/colors. A complete list of named colors is available at https://www.tcl.tk/man/tcl8.4/TkCmd/colors.htm.

Table 9-4 summarizes `turtle`'s color functions.

Function	Action

`color(c)`
`color(c1, c2)`
`color()`
	Set pen color and fill color to `c`.
	Set pen color to `c1` and fill color to `c2`.
	Return turtle's current pen color and fill color (each as an RGB tuple or name).

`pencolor(c)` `pencolor()`	Set pen color. Return turtle's current pen color.
`fillcolor(c)` `fillcolor()`	Set fill color. Return turtle's current fill color.
`pensize(w)`	Set the width of strokes to `w`; the default is 1.
`Screen().colormode(256)`	Set RGB tuples scale to 0-255.

Table 9-4. `turtle` color handling functions

The `color` function has two forms: `color(c)` sets the color `c` as both the pen color and the fill color. `color(c1, c2)` sets `c1` as the pen color and `c2` as the fill color.

The parameter `c` can be a literal string that holds the color name. It can also be a tuple of three values, the RGB components, or a literal string that holds `'#'` followed by six hex digits, two for each RGB component. (The RGB values are often expressed in hex because it is convenient to use two hex digits for each component.) For example,

```
>>> color('#25D3A0')
```

sets the red component to 0x25 (decimal 37), the green component to 0xD3 (decimal 211) and the blue component to 0xA0 (decimal 160).

`'#000000'` means black and `'#FFFFFF'` means white.

The `turtle` module uses two modes for representing RGB values in a tuple of three elements. In the first mode, these values are real numbers, scaled to the range from 0 to 1. This is the default mode.

To scale these values back to integers in the usual range, from 0 to 255, use

```
Screen().colormode(255)
```

`Screen()` returns the object `screen` associated with the drawing window. It has functions that control the window size, coordinate units, and other settings. See `turtle` documentation for the list of `screen`'s functions.

❖ ❖ ❖

There are two other functions that set color — `pencolor(c)` and `fillcolor(c)`; They set the pen color and the fill color, respectively. The supported formats for the parameter `c` in these functions are the same as for `color`.

`color()`, when called without parameters, returns the current pen and fill colors either as their symbolic names (if they were set like that) or as a tuple of RGB values (according to the current `colormode`).

Example 1

What are the RGB values for `'dark salmon'`?

Solution

According to https://www.tcl.tk/man/tcl8.4/TkCmd/colors.htm/ and other web sites, the RGB components for this color are 233, 150, 122.

Example 2

Does the color `'#ff69b4'` have a symbolic name?

Solution

Google "#ff69b4" to find out.

> **Python's `turtle` module and the `screen` object have many more functions — for setting window dimensions, defining stroke width, creating new turtle shapes, getting user input, creating animations, capturing mouse clicks and keyboard events, and so on.**

See https://docs.python.org/3.7/library/turtle.html#module-turtledemo for examples.

Section 9.4 ~ Exercises

1. If each of the three RGB color components is represented in one byte, how many different RGB colors are there? ✓

2. If the screen resolution is 1920 by 1200 and three bytes per pixel are used for color, what is the required size of the video memory?

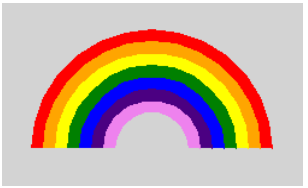
3. Is the 1920 by 1200 screen aspect ratio close to the golden ratio? ✓

4. What are the RGB values for the color `'maroon'`?

5. Draw a color swatch for the 27 colors formed by combinations of 0, 127 and 255 values for each R, G, and B components:

✓

6. Draw a rainbow of seven colors on a light gray background:



The rainbow colors are red, orange, yellow, green, blue, indigo, and violet.

ⵑ Hints:

(a) `Screen().bgcolor(c)` sets the background color of the graphics window.

(b) Draw the rings as overlapping filled semicircles, starting with the largest one and ending with the smallest semicircle in the background color.

ⵒ

7. Display a smooth gradient from pale pink to bright red:

Now reproduce the famous optical illusion by adding a rectangle of solid mid-range red in the middle:

9.5 Review

Terms introduced in this chapter:

Logo
Turtle graphics
Module
Virtual
Pixel
Raster device
Graphics adapter

Some of the Python features introduced in this chapter:

```
from turtle import * # import everything
```

`t = Turtle()` or `t = Turtle('turtle')`

`turtle` functions:
`speed`, `forward`, `backward`, `left`, `right`, `penup`, `pendown`,
`begin_fill`, `end_fill`
`showturtle`, `hideturtle`
`setheading`, `goto`, `setx`, `sety`, `home`

`turtle` color functions:
`color`, `pencolor`, `fillcolor`, `pensize`

```
Screen().colormode(255)
```