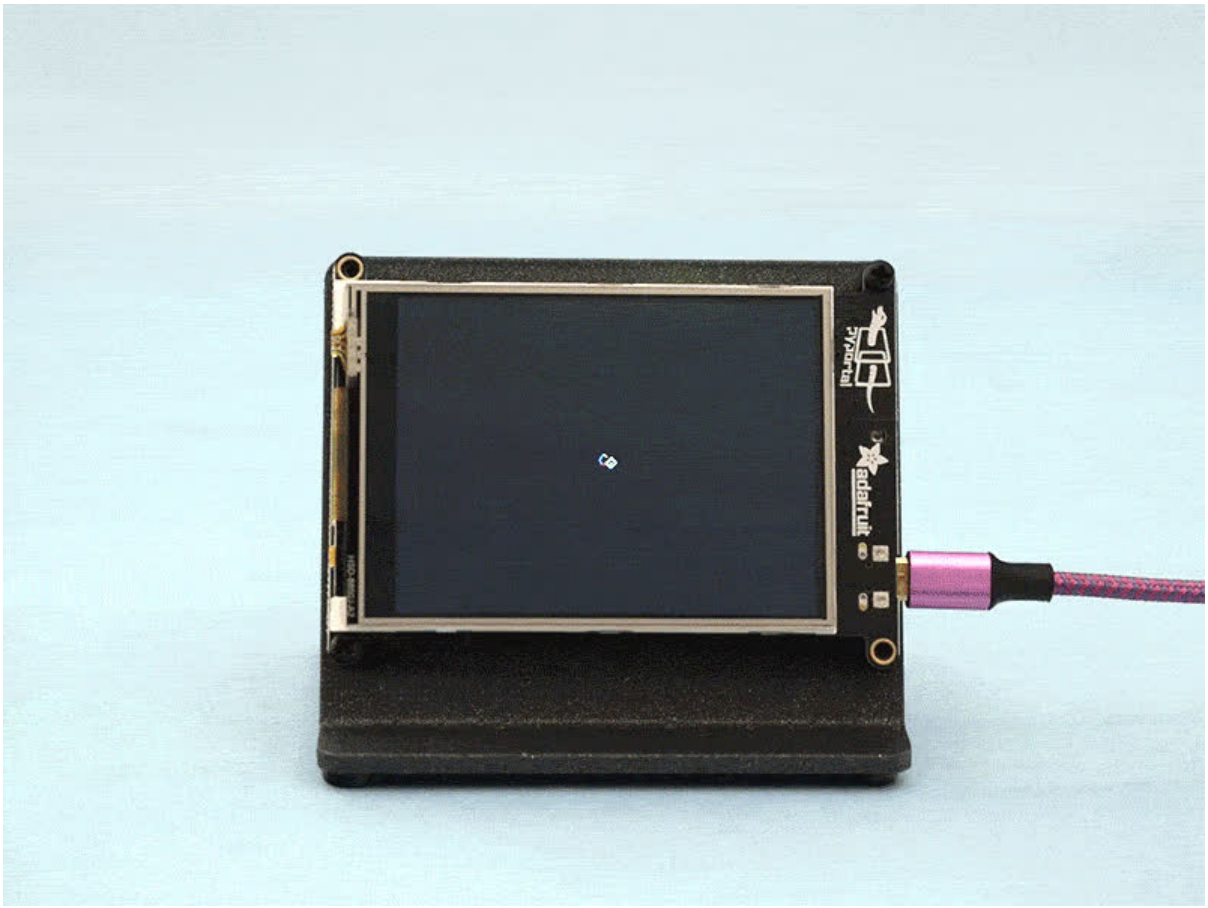
CircuitPython Turtle Graphics

Created by Dave Astels



https://learn.adafruit.com/circuitpython-turtle-graphics

Last updated on 2024-06-03 02:49:32 PM EDT

Table of Contents

Overview

You may have heard of, or even played around with, turtle graphics (https://adafru.it/FaL). Simply put, this is a metaphor for drawing vector images where you control a turtle that can drag a pen along with it. Commands to the turtle include things like move forward some distance, turn left or right by some angle, lift the pen off the paper (so that moving won't make a mark), or put it on the paper (so that moving will make a mark).

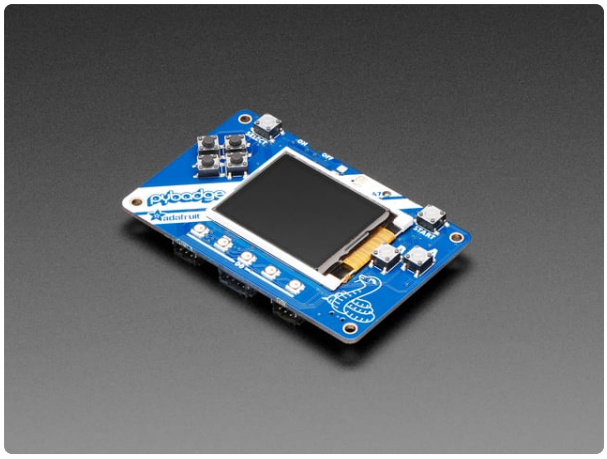
There have been many on-screen implementations of turtle graphics, most notably the LOGO (https://adafru.it/FaM) programming language. Versions have also been made that control a robot that has a pen so as to create on-paper copies of turtle drawings.

This guide introduces a turtle graphics library for CircuitPython that is built on top of `displayio` for use on display based boards such as the PyPortal, PyBadge, and PyGamer.

Adafruit PyPortal - CircuitPython Powered Internet Display

PyPortal, our easy-to-use IoT device that allows you to create all the things for the “Internet of Things” in minutes. Make custom touch screen interface...

https://www.adafruit.com/product/4116



[Adafruit PyBadge for MakeCode Arcade, CircuitPython, or Arduino]
What's the size of a credit card and can run CircuitPython, MakeCode Arcade or Arduino? That's right, its the Adafruit PyBadge! We wanted to see how much we...
https://www.adafruit.com/product/4200

[Adafruit PyGamer Starter Kit]
Please note: you may get a royal blue or purple case with your starter kit (they're both lovely colors)What fits in your pocket, is fully Open...
https://www.adafruit.com/product/4277

[Adafruit PyGamer for MakeCode Arcade, CircuitPython or Arduino]
What fits in your pocket, is fully Open Source, and can run CircuitPython, MakeCode Arcade or Arduino games you write yourself? That's right, it's the Adafruit...
https://www.adafruit.com/product/4242

[Pink and Purple Braided USB A to Micro B Cable - 2 meter long]
This cable is super-fashionable with a woven pink and purple Blinka-like pattern! First let's talk about the cover and over-molding. We got these in custom colors,...
https://www.adafruit.com/product/4148

Setting up your Device

CircuitPython is a programming language based on Python, one of the fastest growing programming languages in the world. It is specifically designed to simplify experimenting and learning to code on low-cost microcontroller boards. Here is a guide which covers the basics:

- Welcome to CircuitPython! (https://adafru.it/cpy-welcome)

Be sure you have the latest CircuitPython for the PyPortal loaded onto your board, as described here (https://adafru.it/F6N). You will want at least version 4.1 (possibly a beta version of it) for maximum graphics performance. While we'll reference the PyPortal in this guide, **adafruit_turtle** will work on all of the TFT based CircuitPython boards such as PyBadge and PyGamer.

CircuitPython is easiest to use within the Mu Editor. If you haven't previously used Mu, this guide will get you started (https://adafru.it/ANO).

Libraries

Plug your PyPortal board into your computer via a USB cable. Please be sure the cable is a good power+data cable so the computer can talk to the board.

A new disk should appear in your computer's file explorer/finder called **CIRCUITPY**. This is the place we'll copy the code and code library. If you can only get a drive named **PORTALBOOT**, load CircuitPython per the guide mentioned above (https://adafru.it/F6N).

Create a new directory on the **CIRCUITPY** drive named **lib**.

Download the latest CircuitPython driver package to your computer using the green button below. **Match the library you get to the version of CircuitPython you are using**. Save to your computer's hard drive where you can find it.

Go to GitHub to get the latest CircuitPython library bundle

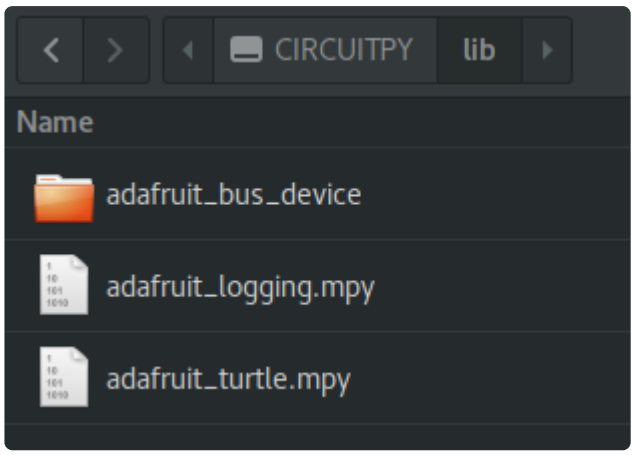
https://adafru.it/zB-

With your file explorer/finder, browse to the bundle and open it up. Copy the following folders and files from the library bundle to your **CIRCUITPY lib** directory you made earlier:

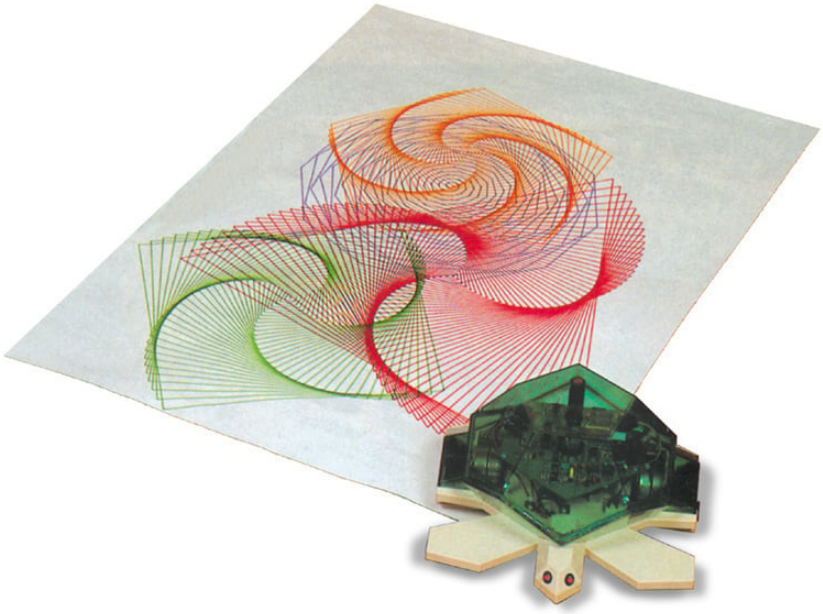
- **adafruit_bus_device**
- **adafruit_logging.mpy**
- **adafruit_turtle.mpy**

All of the other necessary libraries are baked into CircuitPython!

Your **CIRCUITPY/lib** directory should look like the snapshot below.



The Turtle API



Controlling the Pen

`pendown()`

Lower the pen, causing subsequent commands will draw.

Aliases: `pd()`, `down()`

`penup()`

Raise the pen, causing subsequent commands to not draw.

Alaises: `pu()`, `up()`

`pencolor(color=None)`

The form without an argument will return the current pen color as a 24-bit integer. The other form sets the pen color to the specified value. The Color class should be used for this value: `WHITE`, `BLACK`, `RED`, `ORANGE`, `YELLOW`, `GREEN`, `BLUE`, `PURPLE`, `PINK`.

E.g. `turtle.pencolor(adafruit_turtle.Color.RED)`

Moving

`forward(distance)`

Move the turtle forward (on its current heading) by the specified distance. The distance is a number of pixels if the turtle is moving exactly vertically or horizontally.

Alias: `fd`

`backward(distance)`

Move the turtle backward (opposite its current heading) by the specified distance. The distance is a number of pixels if the turtle is moving exactly vertically or horizontally.

Aliases: `bk`, `back`

`setx(x)`

Set the turtle's horizontal coordinate.

`sety(y)`

Set the turtle's vertical coordinate.

`goto(x, y)`

Set both coordinates of the turtle.

Heading

Drawing in a straight line isn't that interesting, so the direction that the turtle is facing (and thus moving) can be changed. This is called its heading. The first two functions below set what it means to change the heading by some value. The result of calling these methods stay in effect until the next call to one of them. By default, degrees are used, with a change of 1 corresponding to 1 degree.

`degrees(fullcircle=360)`

A full circle is 360 degrees and, by default, changing the heading by 1 means changing it by one degree. Supplying a different value will serve to scale those

incremental heading changes. For example, if you call `degrees(180)`, changing the heading by 1 will change it by 2 degrees.

`radians()`

Use radians to turn the turtle. Changing the heading by 1 now means changing it by 1 radian (about 57.3 degrees).

The remaining methods change the turtle's heading.

`left(angle)`

Turn the turtle left by angle (what that means is subject to the above methods).

Alias: **`lt`**

`right(angle)`

Turn the turtle right by angle (what that means is subject to the above methods).

Alias: **`rt`**

`setheading(angle)`

Set the heading of the turtle to angle. Up is an angle of 0, right is 90.

Alias: **`seth`**

Shapes

`dot(radius=None, color=None)`

Draw a filled-in circle centered on the turtle's current position. Turtle position and heading are unchanged.

If radius is omitted a reasonable one is used, otherwise radius is used as the radius of the dot. If color is omitted the current pen color is used, otherwise color is used. This does not change the pen's color. As above, colors are available from the `Color` class.

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

Draw a unfilled circle using the turtle's current pen color. This uses a computed sequence of calls to `forward` and `left` so the turtle's position and heading are changed. Since the circle drawing uses `left`, it draws counterclockwise by default. If radius is negative (i.e. < 0) drawing is done clockwise.

The radius of the desired circle is specified by radius, which is a number of pixels from the center to the edge.

The argument extent specifies what portion of the circle to draw and is specified in the same units as calls to `left` and `right`, as determined by the most recent call to `degrees` or `radians`. The default is to draw the complete circle. Calling `circle(10, extent=180)` will draw half a circle of radius 10, assuming that `radians` hasn't been called and neither has `degrees` with an argument other than 360. The turtle's heading is always what it was after drawing the last part of the circle, regardless extent. This will always be at the tangent to the circle. You can use this in your drawings. E.g. to draw a closed half-circle:

```
turtle.circle(20, extent=180)
turtle.left(90)
turtle.forward(40)
```

The final argument, steps, determines how many sides the circle has. By default (steps is `None`) as many are used as required to create a smooth circle. Specifying a value for steps has the effect of drawing that many line segments instead. So circle(radius r, steps=6) will draw a hexagon.

If you draw a full circle, the turtle will end up back where it started with the heading it started with. By drawing circles (of other polygons by using steps) repeatedly and turning between each, some interesting designs can be created.

```
for _ in range(36):
  turtle.circle(50, steps=6)
  turtle.left(10)
```

More control

`home()`

Move the turtle to its initial position and heading.

`clear()`

Clear the screen, the position and heading of the turtle is unaffected.

Queries

These methods let you ask the turtle about aspects of it's current state.

`pos()`

Returns the turtle's (x, y) position.

`xcor()`

Returns the turtle's x coordinate.

`ycor()`

Returns the turtle's y coordinate.

`heading()`

Returns the turtle's heading.

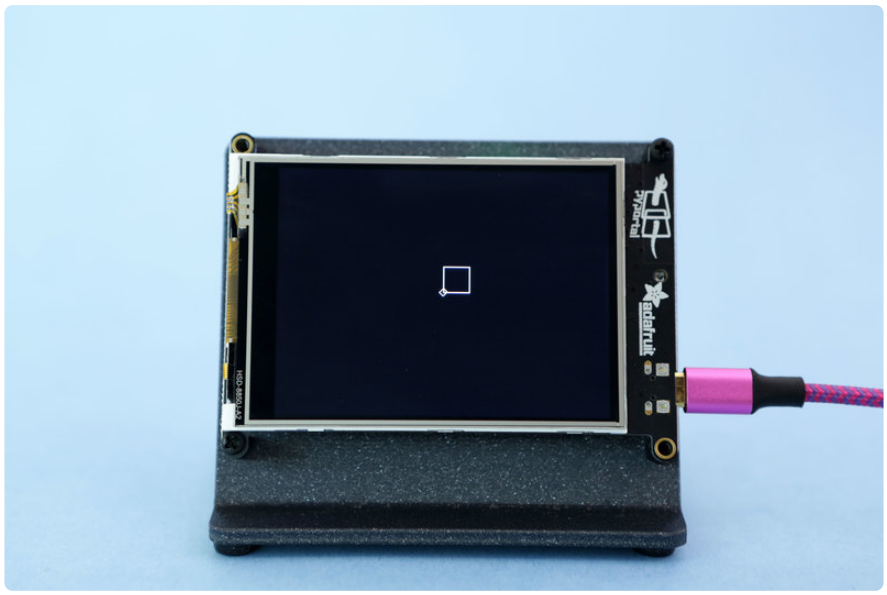
`isdown()`

Returns whether the pen is down (i.e. drawing).

Example Scripts

Lines

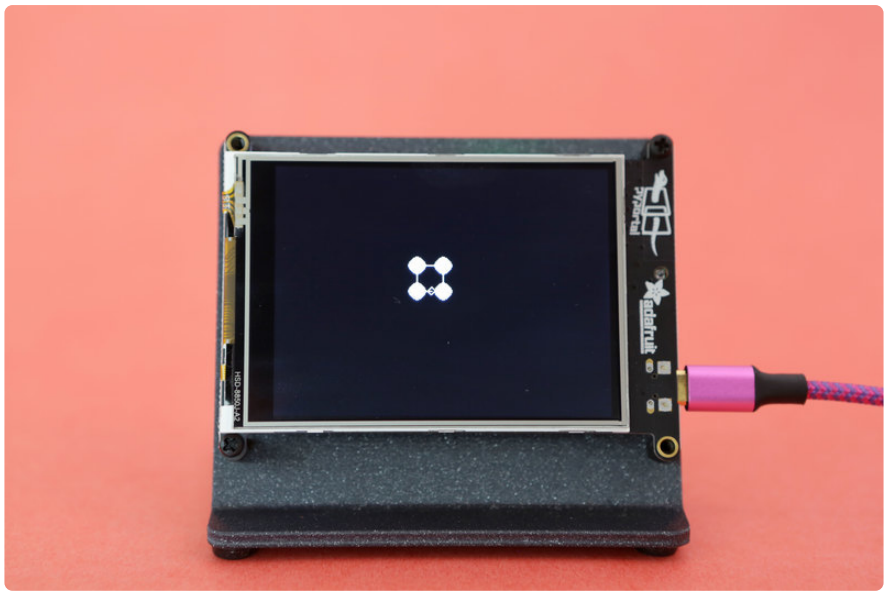
Let's start simply to get warmed up. This will draw a square that is 25 pixels on a side.



```
import board
from adafruit_turtle import Color, turtle

turtle = turtle(board.DISPLAY)
print("Turtle time! Lets draw a simple square")

turtle.pencolor(Color.WHITE)
turtle.pendown()

for _ in range(4):
    turtle.forward(25)
    turtle.left(90)

while True:
    pass
```

Dots

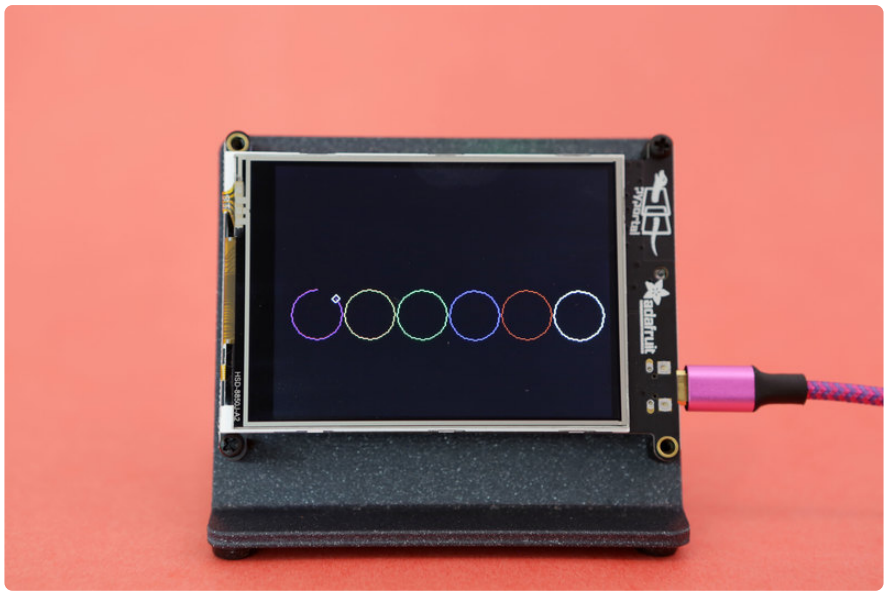
Next, put a 8 pixel radius dot at each corner. Remember that `dot()` doesn't effect the turtle's position or heading, so dots can be easily added to an existing script.



```
import board
from adafruit_turtle import turtle, Color

print("Turtle time! Lets draw a square with dots")

turtle = turtle(board.DISPLAY)
turtle.pendown()

for _ in range(4):
    turtle.dot(8)
    turtle.left(90)
    turtle.forward(25)

while True:
    pass
```

Circles

Circles are a little tricker since the outline starts with the initial turtle position and heading. The nice thing is that the turtle ends up where it started, and with the same heading. The code below moves the turtle to near the right side of the screen and then points it to the left. It then draws 6 circles in different colors, moving left between each. By moving by the diameter we use, the circles don't overlap.

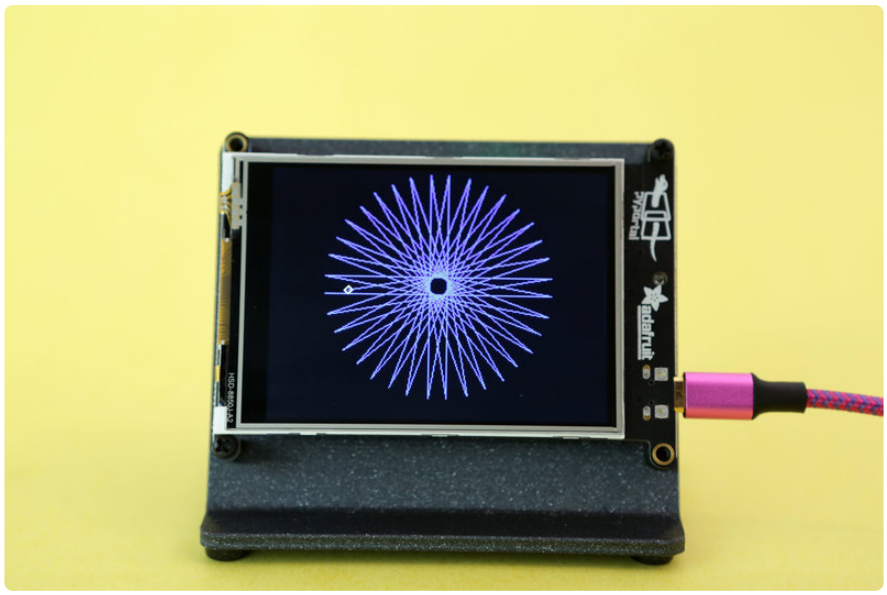


```
import board
from adafruit_turtle import Color, turtle

turtle = turtle(board.DISPLAY)

mycolors = [Color.WHITE, Color.RED, Color.BLUE, Color.GREEN, Color.ORANGE,
Color.PURPLE]
turtle.penup()
turtle.forward(130)
turtle.right(180)
turtle.pendown()

for i in range(6):
    turtle.pencolor(mycolors[i])
    turtle.circle(25)
    turtle.penup()
    turtle.forward(50)
    turtle.pendown()

while True:
    pass
```

To have them overlap, reduce how far the turtle moves between each one.

Scaling to the Screen

Since **adafruit_turtle** can be used on boards with different sized screens, it can be a good idea to scale drawings to fit on the screen. The library will clip drawings to the screen so if you draw off the edge your script won't crash, but the result might not look great. The script below shows an approach to scaling to the screen. It also shows how to detect that the turtle is back where it started, accounting for some rounding error.

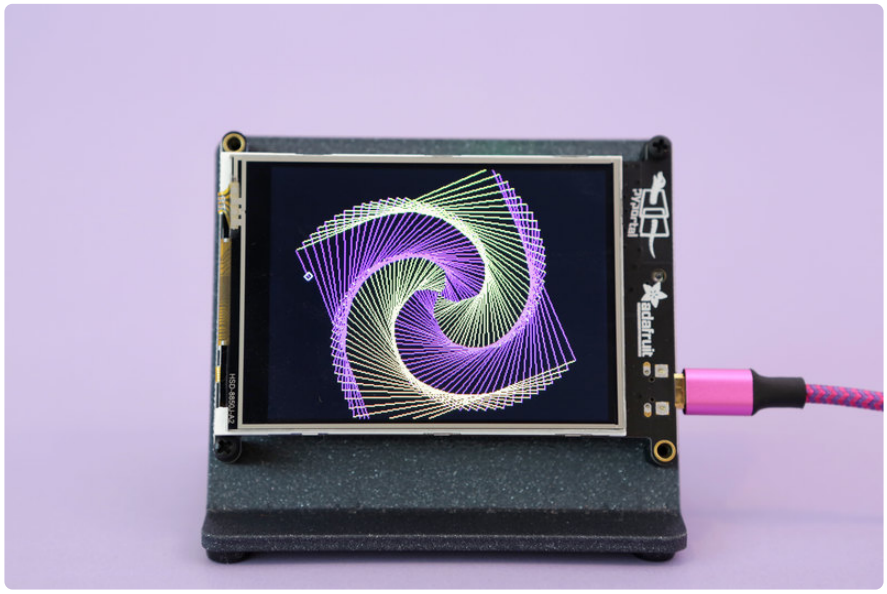


```
import board
from adafruit_turtle import Color, turtle

turtle = turtle(board.DISPLAY)
starsize = min(board.DISPLAY.width, board.DISPLAY.height) * 0.9  # 90% of screensize

print("Turtle time! Lets draw a star")

turtle.pencolor(Color.BLUE)

turtle.penup()
turtle.goto(-starsize/2, 0)
turtle.pendown()

start = turtle.pos()
while True:
    turtle.forward(starsize)
    turtle.left(170)
    if abs(turtle.pos() - start) &lt; 1:
        break

while True:
    pass
```

Using Color

Changing color as you draw can produce startling results, as these scripts show:

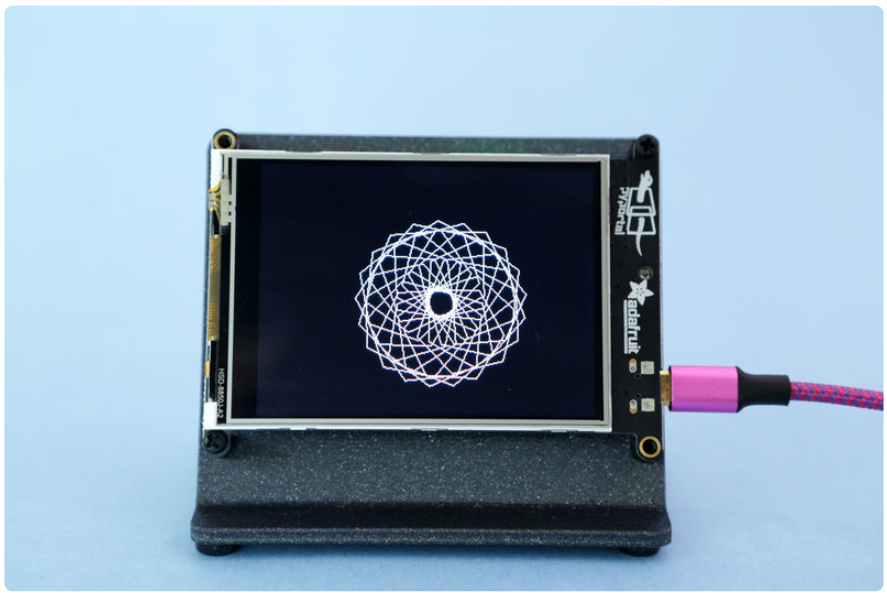
```
import board
from adafruit_turtle import Color, turtle

turtle = turtle(board.DISPLAY)
benzsize = min(board.DISPLAY.width, board.DISPLAY.height) * 0.5

print("Turtle time! Lets draw a rainbow benzene")

colors = (Color.RED, Color.ORANGE, Color.YELLOW, Color.GREEN, Color.BLUE,
Color.PURPLE)

turtle.pendown()
start = turtle.pos()

for x in range(benzsize):
    turtle.pencolor(colors[x%6])
    turtle.forward(x)
    turtle.left(59)

while True:
    pass
```



```
import board
from adafruit_turtle import turtle, Color

turtle = turtle(board.DISPLAY)


turtle.pendown()

colors = [Color.ORANGE, Color.PURPLE]

for x in range(400):
    turtle.pencolor(colors[x % 2])
    turtle.forward(x)
    turtle.left(91)

while True:
    pass
```

Using `circle` for Polygons

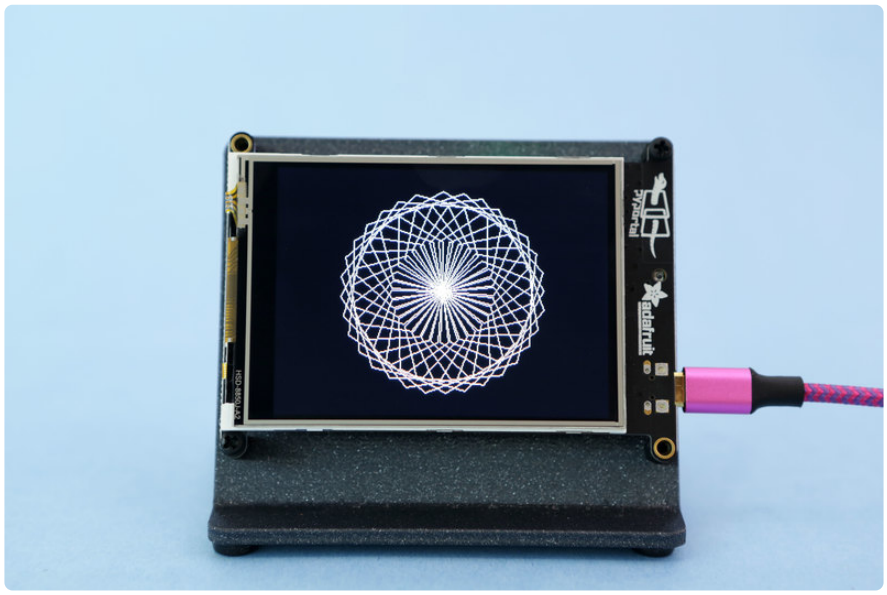
Below is a design and script to make it using just lines and turns.



```
import board
from adafruit_turtle import turtle

turtle = turtle(board.DISPLAY)


# turtle.penup()
# turtle.right(45)
# turtle.forward(90)
# turtle.right(75)

turtle.pendown()
for _ in range(21):
    for _ in range(6):
        turtle.forward(50)
        turtle.right(61)
    turtle.right(11.1111)
```

```
while True:
    pass
```

You can do something very similar using circles and the step parameter, though you may need to tweak some things since you're dealing with the radius instead of side length. This results is quite a bit less code. The following image and script show this.

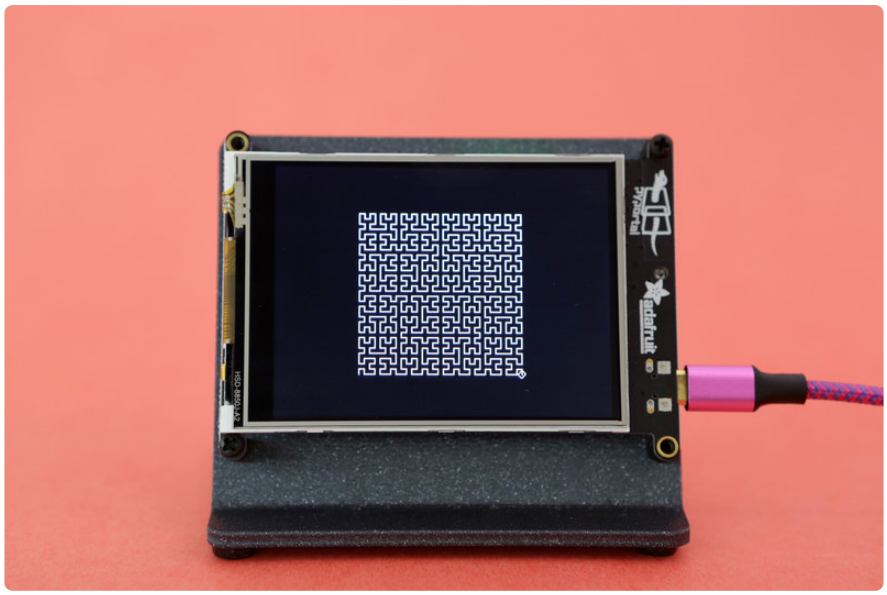


```
import board
from adafruit_turtle import turtle

turtle = turtle(board.DISPLAY)

turtle.pendown()
for _ in range(32):
    turtle.circle(50, steps=6)
    turtle.right(11.1111)

while True:
    pass
```

Fractals

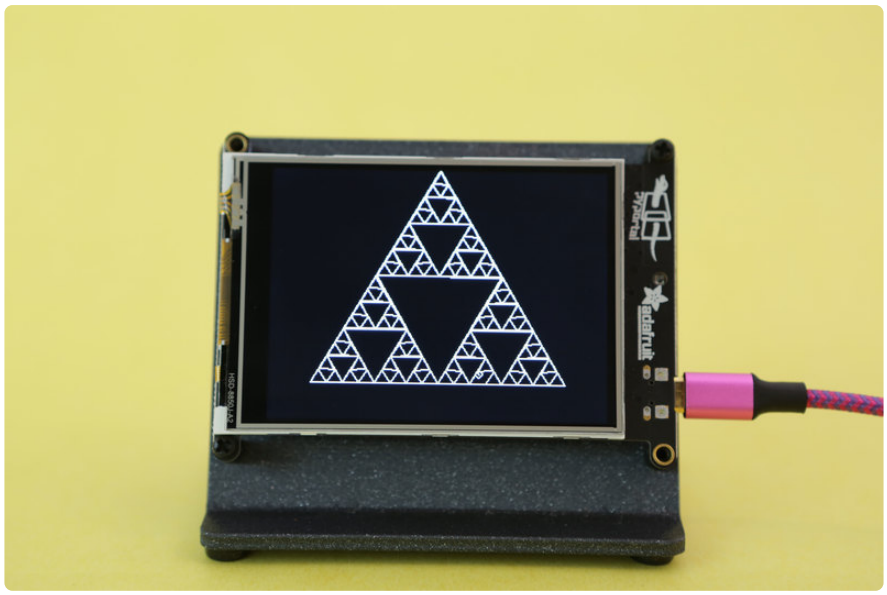
Turtle graphics are quite handy for drawing fractals (https://adafru.it/FaN).

The first example is a hilbert curve (https://adafru.it/FaO).



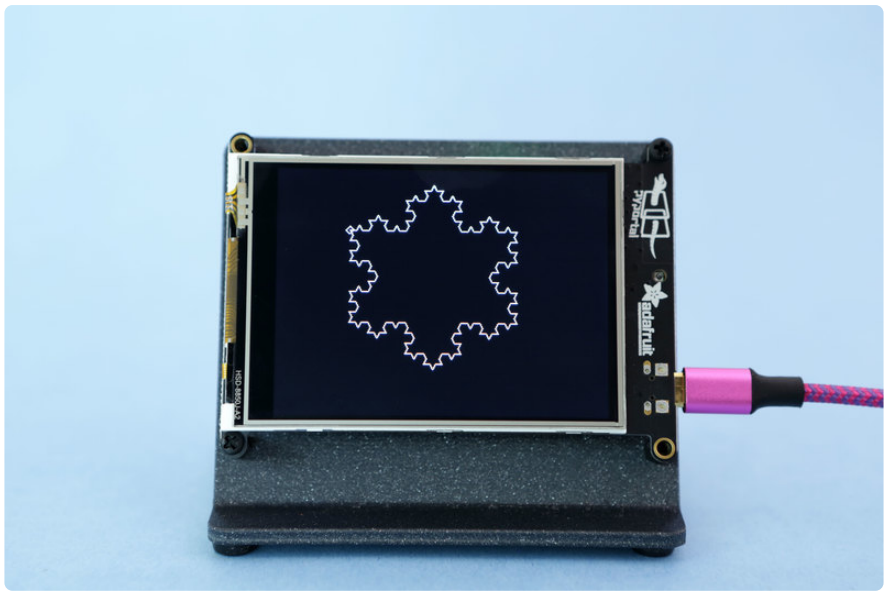
```
import board
from adafruit_turtle import turtle

def hilbert2(step, rule, angle, depth, t):
    if depth &gt; 0:
        a = lambda: hilbert2(step, "a", angle, depth - 1, t)
        b = lambda: hilbert2(step, "b", angle, depth - 1, t)
        left = lambda: t.left(angle)
        right = lambda: t.right(angle)
        forward = lambda: t.forward(step)
        if rule == "a":
            left(); b(); forward(); right(); a(); forward(); a(); right();
forward(); b(); left()
        if rule == "b":
            right(); a(); forward(); left(); b(); forward(); b(); left();
forward(); a(); right()

turtle = turtle(board.DISPLAY)
turtle.penup()

turtle.goto(-80, -80)
turtle.pendown()
hilbert2(5, "a", 90, 5, turtle)

while True:
    pass
```

Another interesting fractal is the Sierpinski triangle (https://adafru.it/FaP), shown below.



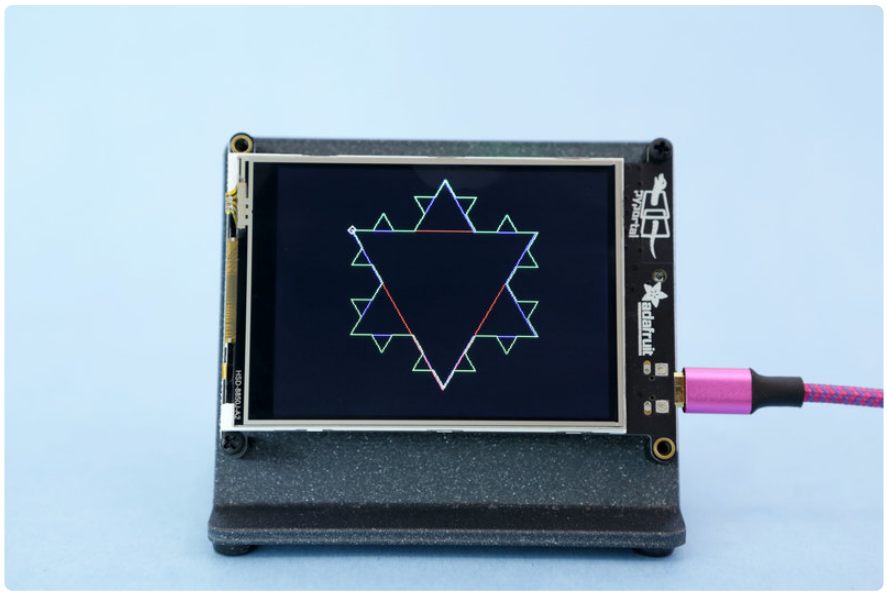
```
import board
from adafruit_turtle import turtle

def getMid(p1,p2):
    return ( (p1[0]+p2[0]) / 2, (p1[1] + p2[1]) / 2) #find midpoint

def triangle(points, depth):

    turtle.penup()
    turtle.goto(points[0][0], points[0][1])
    turtle.pendown()
    turtle.goto(points[1][0], points[1][1])
    turtle.goto(points[2][0], points[2][1])
    turtle.goto(points[0][0], points[0][1])

    if depth &gt; 0:
        triangle([points[0],
                  getMid(points[0], points[1]),
                  getMid(points[0], points[2])],
                 depth-1)
        triangle([points[1],
                  getMid(points[0], points[1]),
                  getMid(points[1], points[2])],
                 depth-1)
        triangle([points[2],
                  getMid(points[2], points[1]),
                  getMid(points[0], points[2])],
                 depth-1)

turtle = turtle(board.DISPLAY)
big = min(board.DISPLAY.width / 2, board.DISPLAY.height / 2)
little = big / 1.4
seed_points = [[-big, -little],[0,big],[big,-little]] #size of triangle
triangle(seed_points,4)

while True:
    pass
```

The final example is a Koch snowflake (https://adafru.it/FaQ). On the small displays we can get 4 generations before rounding errors start messing things up too much. Even so, it's a nice design. See below.



```
import board
from adafruit_turtle import turtle

def f(side_length, depth, generation):
    if depth == 0:
        side = turtle.forward(side_length)
    else:
        side = lambda: f(side_length / 3, depth - 1, generation + 1)
        side()
        turtle.left(60)
        side()
        turtle.right(120)
        side()
        turtle.left(60)
        side()

turtle = turtle(board.DISPLAY)

unit= min(board.DISPLAY.width / 3, board.DISPLAY.height / 4)
top_len = unit * 3
print(top_len)
turtle.penup()
turtle.goto(-1.5 * unit, unit)
turtle.pendown()

num_generations = 3
top_side = lambda: f(top_len, num_generations, 0)

top_side()
turtle.right(120)
top_side()
turtle.right(120)
top_side()

while True:
    pass
```

The script below is much the same, but overlays 3 generations in different colors. Beyond 3, rounding causes subsequent generations to not line up.



```
import board
from adafruit_turtle import turtle, Color

generation_colors = [Color.RED, Color.BLUE, Color.GREEN]

def f(side_length, depth, generation):
    if depth == 0:
        side = turtle.forward(side_length)
    else:
        side = lambda: f(side_length / 3, depth - 1, generation + 1)
        side()
        turtle.left(60)
        side()
        turtle.right(120)
        side()
        turtle.left(60)
        side()

def snowflake(num_generations, generation_color):
    top_side = lambda: f(top_len, num_generations, 0)
    turtle.pencolor(generation_color)
    top_side()
    turtle.right(120)
    top_side()
    turtle.right(120)
    top_side()


turtle = turtle(board.DISPLAY)

unit= min(board.DISPLAY.width / 3, board.DISPLAY.height / 4)
top_len = unit * 3
print(top_len)
turtle.penup()
turtle.goto(-1.5 * unit, unit)
turtle.pendown()

for generations in range(3):
    snowflake(generations, generation_colors[generations])
    turtle.right(120)

while True:
    pass
```

Where to Find More

You can find much more information about turtle graphics as well as example scripts through an Internet search of "turtle python graphics examples" or some variation. Since turtle graphics are mainly in terms of moving and turning they can usually be converted to CircuitPython and the **adafruit_turtle** library.

Check out these example sites:

- Michael0x2a Examples (https://adafru.it/Fd6)
- Geeks for Geeks Examples (https://adafru.it/Fd7)
- Vivax Solutions Examples (https://adafru.it/Fd8)