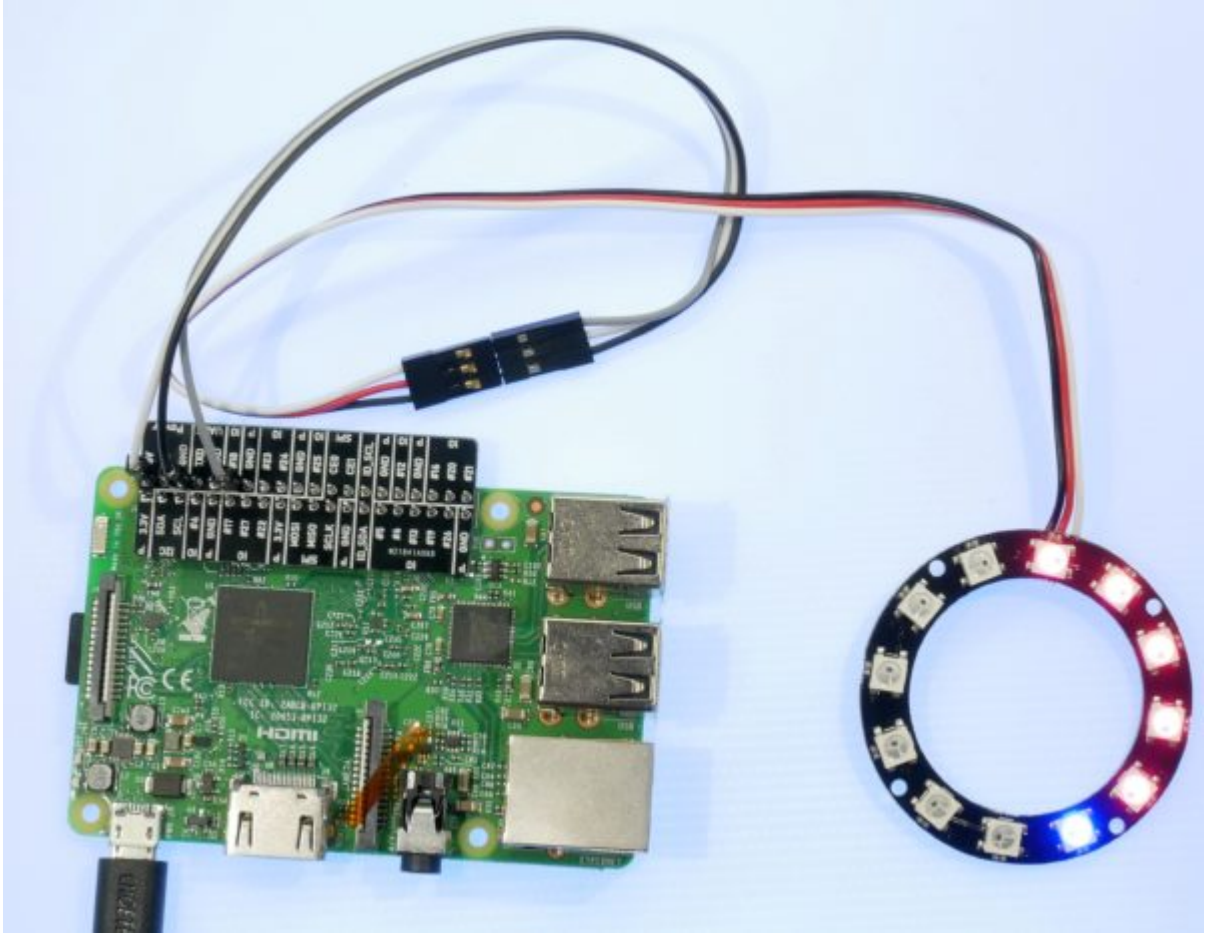
RPi Python: LED Ring Clock WS2812



Prereqs:

Have Raspbian Stretch installed, running, and connected to the internet.
You should refresh on how to easily edit files and navigate the filesystem using a terminal in these tutorials:
Terminal Intro
Editing Files Intro

Next some configuration,
To correctly output PWM to control the LED ring we first need to disable the native Raspberry Pi soundcard, to do this navigate to /boot/config.txt and comment out the line

‘dtparam=audio=on’

Enable audio (loads snd_bcm2835)
#dtparam=audio=on

^ so it looks like this.

Next step is to install some utilities we will need for setting up our led control libraries and the python environment to talk to them; open a terminal and enter

apt-get install gcc make build-essential python-dev git scons swig

this installs some compilers and utilities we will use in the next step

Next navigate to the folder where you want to setup your LED script in a terminal, use ‘mkdir’ to create a new directory if you want; here we will download the library using git
For more help on navigating and creating directories from within the terminal, see this article -> click here

git clone https://github.com/jgarff/rpi_ws281x

Enter the new rpi_ws281x directory and use the ‘scons’ utility we installed earlier to start building the library into something we can use:

sudo scons

Next enter the python directory, and run the setup and build scripts:

sudo python setup.py build install

Hardware:

First off here is a link to the finished project as a reference
To test it, download the file, rename to clock.py, and run

as sudo python clock.py

Next to writing the actual script itself…
The first step is a bunch of boilerplate, like telling our library how many LEDs we have and what pin to use ect.
We start off by importing the libraries to control the leds, and read the date and time from the system

```
from neopixel import *
import time
import datetime
```

next are the constants that tell which pin to use, and which kind of led ring we are using

```
LED_COUNT          = 12          # Number of LED pixels.
LED_PIN             = 18           # GPIO pin connected to the pixels (18 usesPWM!).
LED_FREQ_HZ         = 800000  # LED signal frequency in hertz (usually 800khz)
LED_DMA             = 10  # DMA channel to use for generating signal (try 10)
LED_BRIGHTNESS = 10         # Set to 0 for darkest and 255 for brightest
LED_INVERT         = False   # True to invert the signal (when using NPN transistor level shift)
LED_CHANNEL         = 0           # set to '1' for GPIOs 13, 19, 41, 45 or 53
LED_STRIP           = ws.WS2811_STRIP_GRB   # Strip type and colour ordering


strip = Adafruit_NeoPixel(LED_COUNT, LED_PIN, LED_FREQ_HZ, LED_DMA, LED_INVERT, LED_BRIGHTNESS, LED_CHANNEL, LED_STRIP)
```

As a brief overview, the important ones for us are:
LED_COUNT: the number of LEDs to try and address, if you have a bigger led ring than normal you might change this; 12 is for the one we gave you

LED_PIN: here we use 18; other pins might be tricky so its

probably best to stick with 18

LED_BRIGHTNESS: on full brightness the leds can be super super bright, so we lower it all the way down to 10. Wear sunglasses if you turn it up

Finally with strip we setup the strip using all the constants we just defined.

Logic:

Now we can actually start writing functions and logic for our program. Python scripts usually execute straight from top to bottom in sequential order; but we can essentially jump around our control flow using functions. Functions take arguments, do things based on these arguments and then exit back to where they were called.

We'll start with utility functions that will make controlling the ring easier later, before we write the actual clock part. (remember to use tabs for indentation levels).
We start with a function to clear all the leds, essentially turning them all off

```
def clear (strip):
 for i in range(LED_COUNT):
     strip.setPixelColor(i,Color(0,0,0))
     strip.show()
```

Let's go through this function line by line: First we define the function and give it a name 'clear', followed by a list of arguments it takesLet's go through this function line by line:

```
def clear (strip):
```

Next we start a loop to address all the leds, this pattern will appear alot so pay attention!

for i in range(LED_COUNT):

Notice we use the LED_COUNT variable we named earlier, the loop starts i at 0, and then runs its block over and over until ‘i’ reaches the LED_COUNT value

strip.setPixelColor(i,Color(0,0,0))

Here we actually start setting the led colors, we gave this function an argument called ‘strip’ and here its accessing some functionality that the ‘strip’ object holds. In this case ‘setPixelColor’. The ‘setPixelColor’ function demands a pixel number and a color as its arguments so that’s what we give it. We create a color out of R G B values (red green blue) using the ‘Color’ function.

strip.show()

Finally we actually get the leds to update on the physical ring.

And that’s the end of our first function!

Next we’ll make a ‘fill’ function which does almost the same thing; except instead of setting all the leds to off it’ll set them to a color we give as an argument.

The new function will follow the same basic pattern as our old one.

```
def fill (strip, pixel, color):
 for i in range(pixel):
    strip.setPixelColor(i,color)
    strip.show()
```

If you compare this and the old function side by side it should be easy to see how they match up and differentiate.

Now we can actually test out our functions and make sure they work.If you compare this and the old function side by side it should be easy to see how they match up and differentiate.

On the line below our functions we can start writing code that will actually execute.

```
strip.begin()
fill(strip,12,color(255,255,255))
time.sleep(.5)
clear(strip)
fill(strip,12,color(255,0,255))
time.sleep(1)
clear(strip)
```

```
strip.begin()
fill(strip,12,color(255,255,255))
time.sleep(.5)
clear(strip)
fill(strip,12,color(255,0,255))
time.sleep(1)
clear(strip)
```

We'll go over the first few lines to get an idea of how it works:

```
strip.begin()
```

First we use the 'begin' method on our strip to get it ready for any commands

```
fill(strip,12,color(255,255,255))
```

Next we actually call our fill command from earlier, sending it the strip, the number of pixels we want, and a color.

```
time.sleep(.5)
```

Next we wait for half a second so we get a chance to admire our work.

```
clear(strip)
```

We repeat the process again for another flash, this time a different color. Of course it would be more impressive if it flashed more than twice, but we don't want to just keep copying and pasting the same function calls over and over

either. And eventually we want our clock to be running indefinitely not just for a few seconds; so we will setup an infinite loop.

```
strip.begin()
while(True):
 fill(strip,12,color(255,255,255))
 time.sleep(.5)
 clear(strip)
 fill(strip,12,color(255,0,255))
 time.sleep(.5)
```

When running this script, you can use Control-C to interrupt and exit the program.A while loop repeats everything in its block until the statement we give it isn't true; 'True' is always true so in this case it just loops forever.

Otherwise it'll just keep flashing your led ring forever!.

Now we have some nice utility tools setup we can actually start writing the clock itself.

First we need to get the time! Luckily we imported the time library earlier.

So start off by erasing our While loop from earlier, because we'll be writing another one.

```
strip.begin()
while(True):
 now = datetime.datetime.now()
 time.sleep(.5)
```

Below our other two functions but above our while loop we'll put our clock function.Here we fill a variable called 'now' with the current time every .5 seconds, but we don't do anything with it yet. For that we need another function.

```
def clock (strip, now):
 hours = now.second%2
 hourPixel = (now.hour % LED_COUNT) + 1
 minutePixel = (now.minute / (60/LED_COUNT) % LED_COUNT) + 1
```

```
 hourColor = Color(255,0,0)
 minuteColor = Color(0,0,255)

 if (hours):
    fill(strip,hourPixel,hourColor)
 else:
    fill(strip,minutePixel,minuteColor)
```

hours = now.second%2This turned out to be a pretty simple function so ill go over it section by section.
The most criptic part is the first line:

This is where we work out if we'll show the hour or the minute hand; the modulo operator is explained in more depth below but basically we are testing if the time in seconds is an even number or not. If it it's even we set hours to 0 (false) and if it's odd we set hours to 1 (true).

Next we calculate the hour and minute pixels; which again is explained in the last section below.

```
hourColor = Color(255,0,0)
minuteColor = Color(0,0,255)
```

Here we set the colours we want to use for the hour and minute hands using RGB values

```
if (hours):
fill(strip,hourPixel,hourColor)
```

An if statement runs its block of code only if the expression its given is true, the hours variable is set if the seconds are even remember, so every other second we choose to display the hours or the minute hand.
Then we call our fill function from ealier with the desired results.

Finally we put our clock function in our infinite loop from earlier, and our clock is done!

```
strip.begin()
while(True):
```

```
 now = datetime.datetime.now()
 time.sleep(.5)
 clock(strip,now)
```

And here is the final project again to help fix any errors in your own:Remember to run the script as sudo, like at the start with the strandtest.py example!

https://github.com/keptan/T3RasberryClock/blob/master/alternating_clock.py

Calculating hourPixel and minutePixel:

Looking back over our long clock() function, the most complicated part is the ‘hourPixel’ and ‘minutePixel’ logic; so i’ll go over them in depth here.

```
hourPixel = (now.hour % LED_COUNT) + 1
```

hourPixel is a variable which holds a number, the number being which pixel we want to illuminate upto to represent the hour hand of the clock.
We start off with the hour in a 24 hour clock format from now.hour

So if it was 2pm the now.hour would equal 14.

To emulate a 12 hour analog clock we need to convert 14 down to 2.

The ‘%’ operator called ‘modulo’ returns the remainder after doing an integer division; this is very useful for testing if a number if perfectly divisible (the remainder would be zero) and for creating ‘circular numbers’ which never pass a certain value.

If we look test out the modulus operator we can see the pattern:

1 % 12 == 1

2 % 12 == 2

3 % 12 == 3

…

11 % 12 == 11

12 % 12 == 0 (perfect divisible)

13 % 12 == 1

14 % 12 == 2

So if we enter ‘13’ hours (1pm) into our modulo operator we get a nice conversion straight to 12 hour time (1).

Because our led ring is addressed from 1 instead of 0, (lighting up the 0th led doesn’t work with this library) we add 1 to the result.

So a straight conversion from 13 hours to 1pm, turns into the second led on the ring.

Leaving us with our final algorithm

hourPixel = (now.hour % LED_COUNT) + 1

The math works in the same way but with slightly more complicated variables for the minutePixel algorithm, it’s a good exercise to try and work out how it works yourself.

minutePixel = (now.minute / (60/LED_COUNT) % LED_COUNT) + 1

Further Exercise Ideas:

- Change the colour of the hands depending on if its AM or PM
- Add a seconds hand to the clock
- Display the hours and minutes hand simultaneously with different colours for overlapping areas (see github for a hint!)
- Make a stopwatch
- Make a counting down timer with minutes and seconds
- A rainbow effect on the hands