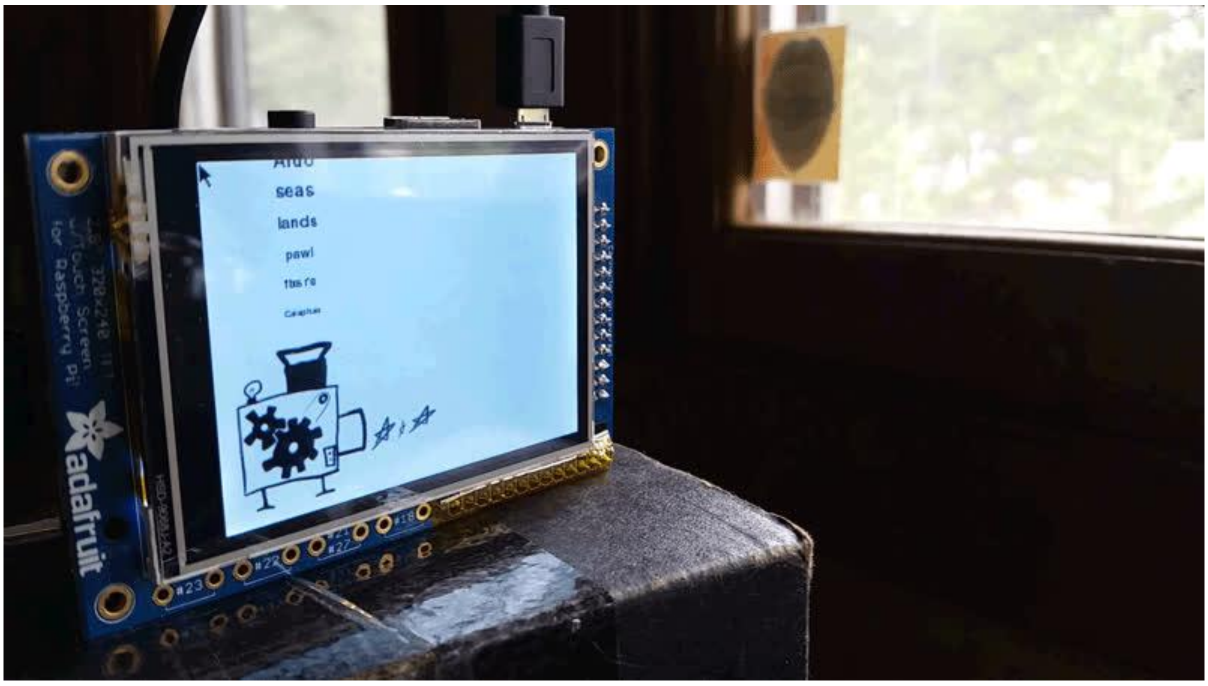
RasPipe: A Raspberry Pi Pipeline Viewer, Part 1

Created by Brennen Bearnes

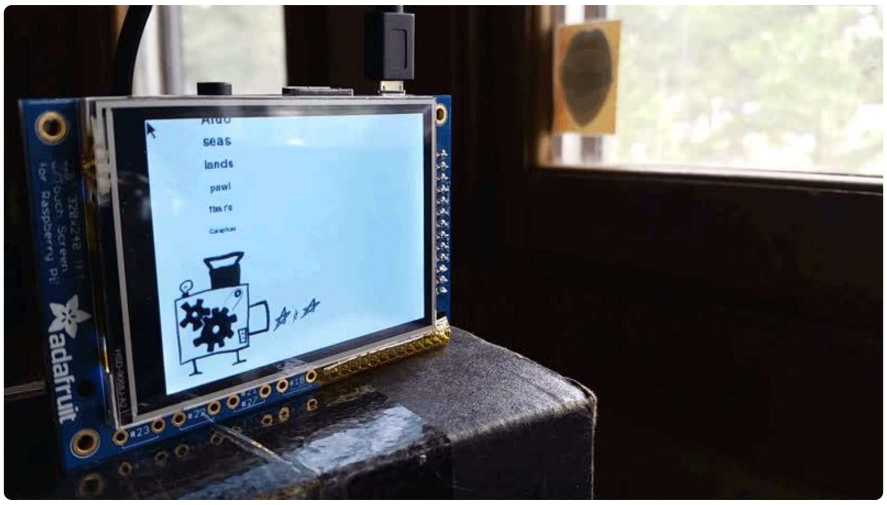


https://learn.adafruit.com/raspipe-a-raspberry-pi-pipeline-viewer

Last updated on 2023-08-29 02:46:04 PM EDT

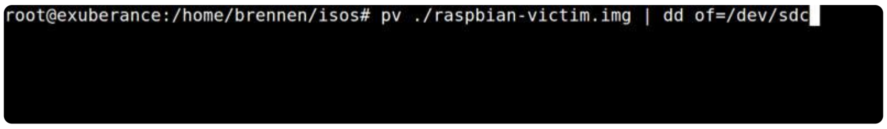
Table of Contents

Overview



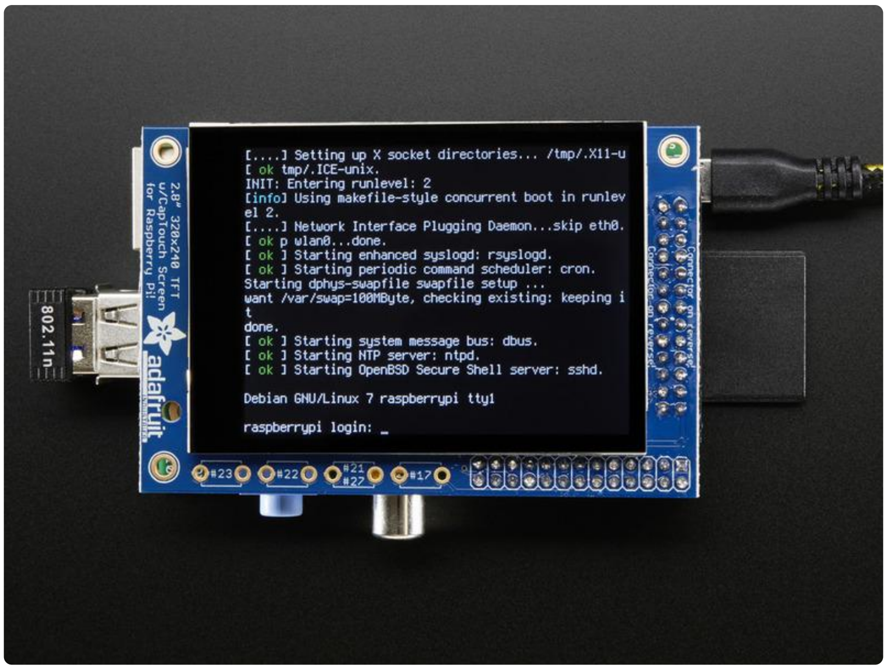
The other day I was thinking about how I could use a Raspberry Pi for a practical demonstration of some Unix shell basics. At right around the same time, I was using a neat little shell utility called pv (), for pipe viewer.

What pv does is sit in the middle of a pipeline and display a progress bar, so you can have some idea of whether a long-running process is actually processing, and how much longer you'll have to wait for it to finish. It's great for things like copying big disk images to SD cards.



```
root@exuberance:/home/brennen/isos# pv ./raspbian-victim.img | dd of=/dev/sdc
```

So ok, I thought, I have this PiTFT display (). What if I made a pipeline viewer out of the Pi and stuck it on top of my regular monitor?



```
[....] Setting up X socket directories... /tmp/.X11-u
[ ok tmp/.ICE-unix.
INIT: Entering runlevel: 2
[info] Using makefile-style concurrent boot in runlev
el 2.
[....] Network Interface Plugging Daemon...skip eth0.
[ ok p wlan0...done.
[ ok ] Starting enhanced syslogd: rsyslogd.
[ ok ] Starting periodic command scheduler: cron.
Starting dphys-swapfile swapfile setup ...
want /var/swap=100MByte, checking existing: keeping i
t
done.
[ ok ] Starting system message bus: dbus.
[ ok ] Starting NTP server: ntpd.
[ ok ] Starting OpenBSD Secure Shell server: sshd.

Debian GNU/Linux 7 raspberrypi tty1

raspberrypi login: _
```

It turns out the basics of that are pretty easy to do, but the specifics are a good chance to experiment with some interlocking little technologies to make something useful (or at least cool to look at). In part one, we'll look at:

- Pipes, filters, and basic data processing with shell tools in Bash.
- Simple Python filter scripts.
- Pygame for data visualization, optionally displayed on the PiTFT.

In part two, we'll look at

- netcat for easily slinging data over network sockets.
- SSH as a secure and authenticated channel for data pipelines between machines.
- Node.js streams.
- Creating a simple Flask () application.

This guide assumes a little familiarity with basic programming concepts and the command line (), but is generally aimed at the beginner.

Shell Filters

In An Illustrated Guide to Shell Magic: Typing Less & Doing More (), I talked about writing shell filters () in a handful of languages. Let's revisit a version of one of those, `stars.py`:

```
#!/usr/bin/env python
# encoding: utf-8

import sys
import re

line = sys.stdin.readline()
while line:
    stars = re.sub('\S', '★', line)
    sys.stdout.write(stars)
    line = sys.stdin.readline()
```

This is some fairly simple Python. It takes some text from standard input one line at a time, uses a regular expression to convert any non-space character to a ★, and writes each line back to standard output.

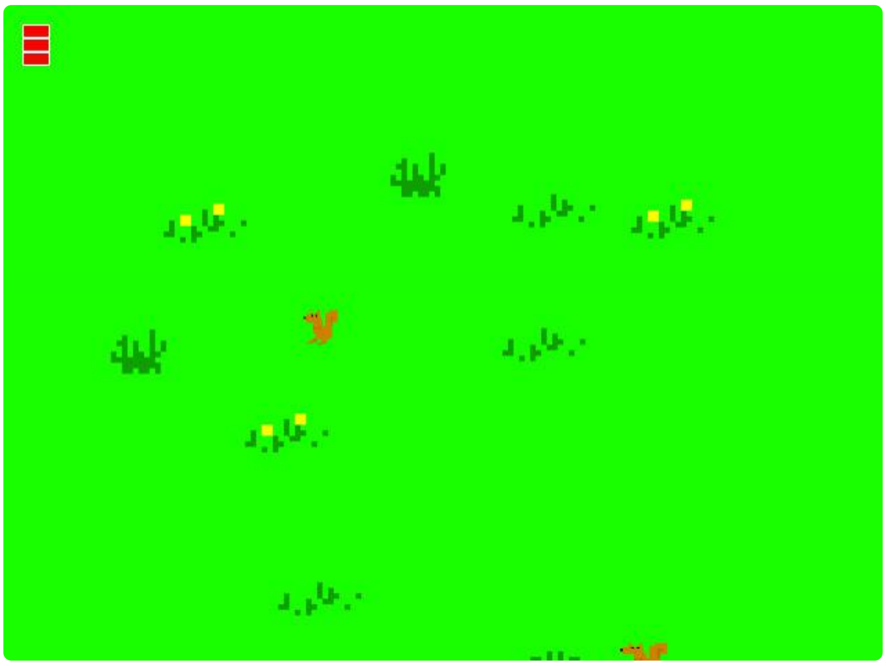
In practice, that looks like this:

```
pi@raspberrypi ~ $
```

Usually, we write code like this to search, transform, store, or otherwise munge data. But what if, to start, we just want to do something more visually interesting with the output than view some characters?

Pygame () is a set of Python libraries for making it easy to write games. In turn, it wraps up SDL (), a popular cross-platform graphics and sound library.

You probably won't be writing the next blockbuster first-person shooter in Pygame, but it's a great way to experiment with simple games that will run on all sorts of computers, including the Pi. For some good examples, you can have a look in the `python_games` folder in your home directory on the Pi. Here's `squirrel.py`, which you can run with `python squirrel.py`:

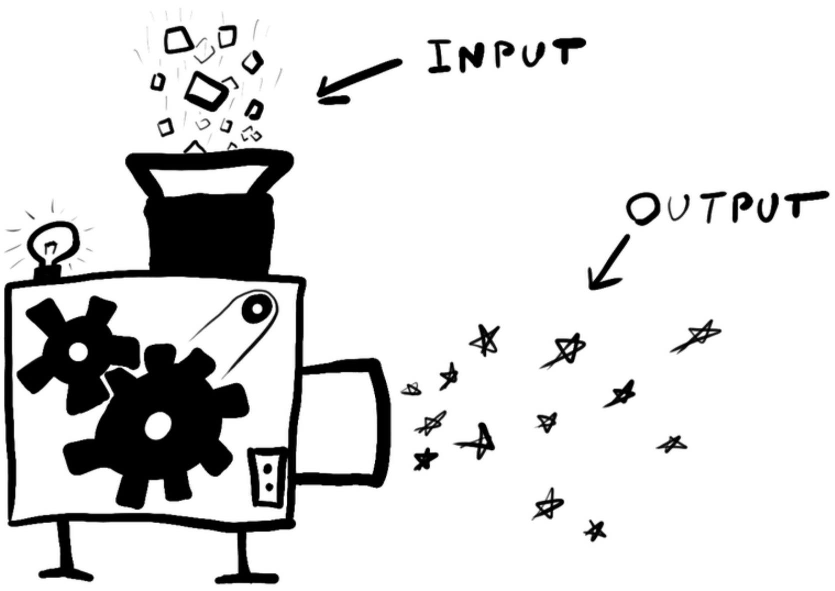


If you `cd ~/python_games` and run `nano squirrel.py`, you can see it's about [400 lines of Python]() (). This could be a little intimidating if you don't know Python - let's start with something a little simpler.

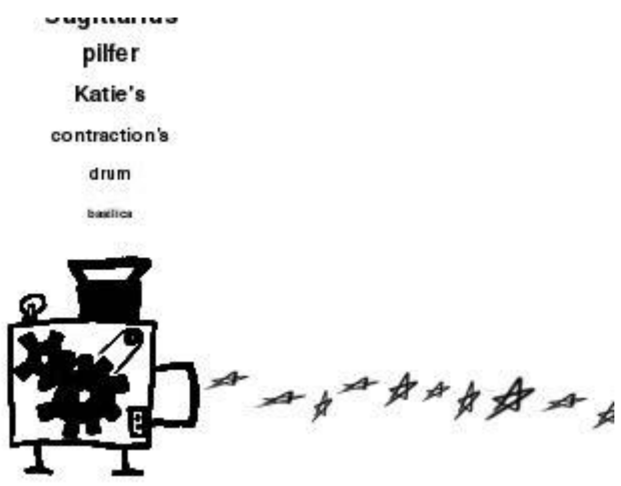
---

machine_stars.py

If you read our [guide on standard I/O]() (), you might remember this little metaphor for a filter program like `stars.py` that takes standard input and produces transformed output:



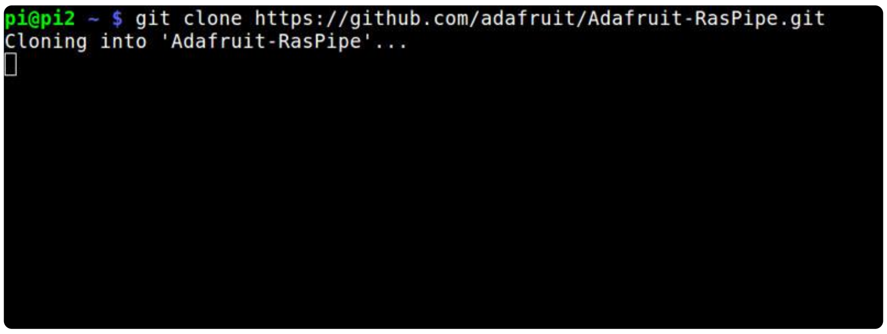
I just wrote a program called `machine_stars.py` that uses Pygame to animate this idea like so:



Clone the Adafruit-RasPipe Repo with Git

In order to experiment with this program and the other code covered in this guide, you should get a copy on your Raspberry Pi. Open up a terminal and enter the following:

```
git clone https://github.com/adafruit/Adafruit-RasPipe.git
```



```
pi@pi2 ~ $ git clone https://github.com/adafruit/Adafruit-RasPipe.git
Cloning into 'Adafruit-RasPipe'...
```

What just happened? Well, Git is a version control system, a tool used to record the changes made to a repository of files over time. GitHub is a popular hosting service for Git repositories. (In addition to cloning a local copy onto your own computer, you can browse all of the code used in this guide on GitHub ().)

You should now have a directory with some Python files and other miscellaneous things in it. Have a look around:

```
cd Adafruit-RasPipe
ls
```

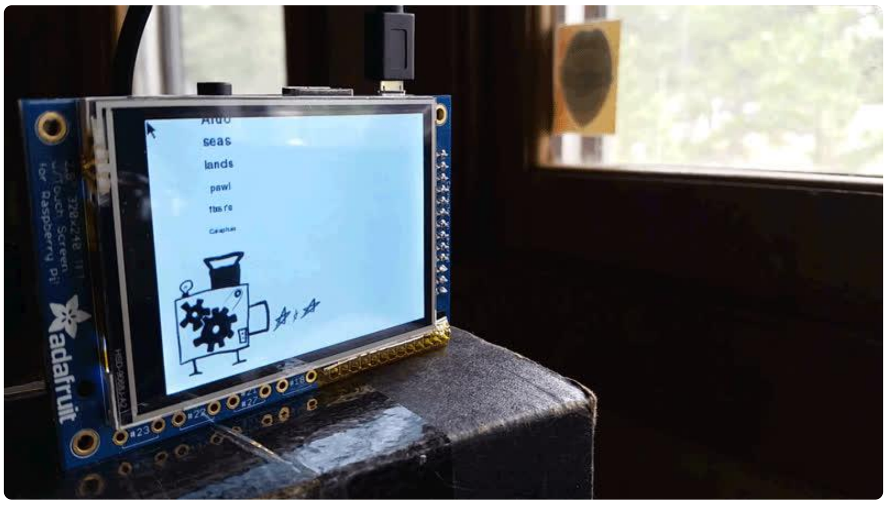
In order to test machine_stars.py, you can do something like the following:

```
shuf /usr/share/dict/words | head -50 | ./machine_stars.py
```

This will shuffle a long list of dictionary words, chop off the first 50, and pass them to the input of `machine_stars.py`, which should animate them falling into a little machine and being transformed into stars.

If you're using an Adafruit PiTFT display, do the following instead:

```
export SDL_FBDEV=/dev/fb1
shuf /usr/share/dict/words | head -50 | ./machine_stars.py
```



Line by Line

Open up `machine_stars.py` in Nano and have a look:

```
nano machine_stars.py
```

Let's step through this program a chunk at a time.

```
#!/usr/bin/env python

import sys
import random

import pygame
```

The first line is a shebang. It tells the Linux kernel to run the rest of the program using `/usr/bin/python`.

The next three are imports, telling Python that we'll be using modules or libraries which contain code already written for us.

- `sys` contains various ways to interact with the larger system - in this case stdin and stdout.

- `random` lets us grab pseudo-random numbers so we can vary certain parts of our program.
- `pygame` lets us do lots of things with graphics and sound.

```
text_color = pygame.Color(0, 0, 0)
bg_color = pygame.Color(255, 255, 255)

pygame.init()
screen = pygame.display.set_mode([320, 240])
screen.fill(bg_color)
```

These lines build the initial state of our program, starting the Pygame engine with `pygame.init()` (), and creating a 320x240 pixel screen, then painting it white.

`pygame.Color(255, 255, 255)` returns a Color object () with its red, green, and blu e values set to the maximum of 255. Each of these can be changed to anything between 0 and 255.

`pygame.display.set_mode()` () actually returns a Surface object (). You don't need to worry about that very much now, but later you'll want to remember that in Pygame, Surfaces are how you work with images in general.

```
# Set up the picture of our little machine:
machine_img = pygame.image.load('machine.png').convert_alpha()
machine_img = pygame.transform.smoothscale(machine_img, (100, 112))
machine_rect = machine_img.get_rect()
machine_rect.left = 10
machine_rect.top = 120
screen.blit(machine_img, machine_rect)

# Set up the picture of a star:
orig_star_img = pygame.image.load('star.png').convert_alpha()
```

Next we load some image files (). These are Surfaces too!

With `pygame.image.load('machine.png')`, we get a Surface for the file `machine.png`. The `.convert_alpha()` method returns a version of this image converted to display faster on our screen, complete with transparency - it's not strictly necessary, but it might make things run smoother.

If you have a Surface, you can get an object that represents the rectangle it occupies with `get_rect()` (). In turn, you can position the Surface by changing values like `machine_rect.left`. Here, we set the left edge of our machine image at 10 pixels from the left of the screen and the top at 120 pixels from the top.

`screen.blit(machine_img, machine_rect)` draws the machine image on top of the screen Surface (), at the coordinates provided by `machine_rect`. We only need to do this once, because everything else we draw will be on different parts of the screen.

We also get an image of a star. We'll display copies of this a bunch of times later on.

```
# This will hold some input lines:
stars_length = 0
offset = 0

# Start building a list of things to display from stdin:
display_lines = [sys.stdin.readline()]
```

`stars_length` will be the number of stars to display - more about that in a while.

`offset` will be used later to offset the falling words from the top of the screen.

`display_lines = [sys.stdin.readline()]` creates a list with one element: The first line in our standard input. `sys.stdin.readline()` gets the stdin one line at a time.

```
while len(display_lines) &gt; 0:
```

Next, we start a loop. While the length of the `display_lines` list is greater than zero, everything in the following indented section will be executed repeatedly.

Each trip through this loop will serve two purposes:

1. Get the next line of standard input, if there is one.
2. Perform one frame of animation.

```
    # Get the next available line from stdin:
    line = sys.stdin.readline()

    if line:
        display_lines.insert(0, line)
```

First, we get the next line of stdin.

Next, we check the truth value of `line`. The `sys.stdin.readline()` call will only return an empty string if there's no more input, and Python considers empty strings false, so we put the value of `line` at the front of the `display_lines` list only if it contains new input.

```
    # If there're more than 6 lines to display, or we're not getting
    # any more input, pop the last line off the list and turn it into
    # a number of stars to show:
    if (len(display_lines) &gt; 6) or (not line):
        stars_length = len(display_lines.pop())
    # If there's no more input, start offsetting display from the top
    # of the screen so it seems to fall downwards:
    if not line:
        offset = offset + 20
```

Next, we're going to decide how many stars to display. If we have more than six lines to display, or we're not going to get any more lines, we'll `pop()` the last line off of `display_lines`, then set `stars_length` to the number of characters it contains.

(The different ways we use a `len()` call here may be a bit confusing: In Python, it's a built-in function used to get the length of many different types of things, including both lists like `display_lines` and strings like the individual elements contained in `display_lines`.)

If we got an empty string for `line`, we'll also increment a variable called `offset`, which we'll use to push the lines of text further down the screen once no more are coming in from stdin. This makes it look like they're falling towards the machine.

```
    # Blank the areas above and right of the machine image:
    screen.fill(bg_color, [0, 0, 320, 120])
    screen.fill(bg_color, [machine_rect.right, machine_rect.top, 320, 240])

    # Display the most recent lines of stdin falling into the machine,
    # in a font that gets smaller as it falls:
    font_size = 22
    y = 0 + offset
    for render_me in display_lines:
        font_size = font_size - 2
        font = pygame.font.Font(None, font_size)
        input_text_surface = font.render(render_me.rstrip(), True, text_color)
        input_text_rect = input_text_surface.get_rect(center=(64, y))
        screen.blit(input_text_surface, input_text_rect)
        y += 20

    pygame.display.update()
```

Now we render the contents of `display_lines` falling towards the machine. The heart of this code is a `for` loop. `render_me` is set equal to each element of `display_loop` in turn, and the indented code is executed with that value.

Inside the loop, we use `pygame.font.Font()` to get a Font object which can render a Surface containing text, and then (just like with the machine image above), draw this on top of the screen, centered at 64 pixels from the left and `y` pixels from the top of the screen.

We call `render_me.rstrip()` to get a version of the string with any whitespace chopped off (including newlines, which will usually be present).

Each time through the `for` loop, we increase `y` by 20.

After all of `display_lines` has been rendered, we update the display with this frame of the animation.

```
    # Display stars leaving machine's output.  Stars are scaled to a random
    # height &amp; width between 8 and 30 pixels, then displayed at a random
    # vertical location on the screen +/- 8 pixels from 185:
    if stars_length &gt; 0:
        star_x = machine_rect.right
        for i in range(0, stars_length):
            star_w = random.randrange(8, 30)
            star_h = random.randrange(8, 30)
            star_img = pygame.transform.smoothscale(orig_star_img, (star_w, star_h))

            star_rect = star_img.get_rect()
            star_rect.left = star_x
            star_rect.top = 185 + random.randrange(-8, 8)
            screen.blit(star_img, star_rect)

            pygame.display.update()

            # Chill out for 15 milliseconds:
            pygame.time.wait(15)

            # Move start of next star to end of the current one, and quit
            # drawing stars if we've run off the edge of the screen:
            star_x += star_w
            if star_x &gt; 320:
                break

    pygame.time.wait(100)
```

Finally, we'll show `stars_length` stars emerging from the side of the machine.

`star_x = machine_rect.right` sets the first star's horizontal position to the righthand side of the machine.

`for i in range(0, stars_length)` starts a loop that will run `stars_length` times.

Inside the loop, we use `random.randrange()` to get stars of different widths and heights each time through, where `pygame.transform.smoothscale(orig_star_img, (star_w, star_h))` makes a smoothly scaled copy of the original star image. We also randomize each star's vertical placement a bit.

`pygame.time.wait(15)` pauses for 15ms between each star, and `pygame.time.wait(100)` waits for 100ms before the main `while` loop starts over.

You can play with all of these timing and randomization values to get different effects. You should also be able to pipe just about any text in. Try, for example:

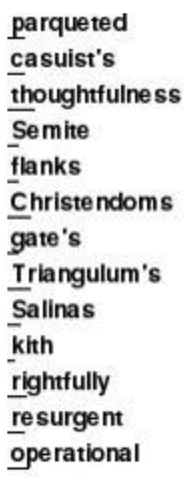
```
dmesg | tail -10 | ./machine_stars.py
```

raspipe.py

The star-making machine is an interesting visualization of a pipeline, but maybe it would be better to use more of the available screen real-estate.

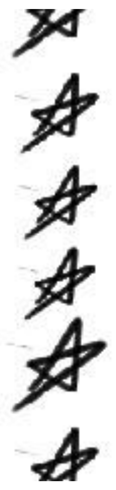
For something a bit different, try running some text through `raspipe.py`:

```
shuf /usr/share/dict/words | head -500 | ./raspipe.py
```



```
parqueted
casuist's
thoughtfulness
Semite
flanks
Christendoms
gate's
Triangulum's
Salinas
kith
rightfully
resurgent
operational
```

With raspipe.py running, try clicking anywhere in the window and the display should switch to something a little more abstract:



You can click again to switch back to the text display.

The script itself reads a lot like `machine_stars.py`, but there are a few key differences. Open it up with Nano and let's have a look at some of the important ones (if you'd rather view it in a browser, you can find the code on GitHub ()):

```
nano raspipe.py
```

Imports

The first thing you'll probably notice is that there're a couple of extra imports:

```
#!/usr/bin/env python
import getopt
import re
import sys

import pygame
```

The `re` module, which you might remember from `stars.py`, will let us match on regular expressions. We use this to let the user optionally specify a pattern to highlight. More about that when we get to the penguin.

The `getopt` module will let us easily parse some command-line options.

RasPipe is a Class

Next, instead of just writing a series of procedural statements in order, I wrote RasPipe as a class. An outline of its structure looks something like this:

```
class RasPipe:
    # Some properties are here.

    def __init__(self, infile):
        """Create a RasPipe object for a given input file."""

    def setfont(self, size):
        """Set a display font of a given size."""

    def wait_for_click(self):
        """Wait for the user to click on the screen (or quit)."""

    def toggle_stars(self):
        """Toggle whether we should display stars."""

    def scale_display(self):
       """Set the current font size and delay based on amount of input."""

    def penguin_think(self, thought):
        """Have a penguin appear and think a thought."""

    def run(self):
        """Process standard input."""
```

What the Heck is a Class?

Python, like most modern languages, provides support for a technique known as Obje ct Oriented Programming, often abbreviated OOP or OO.

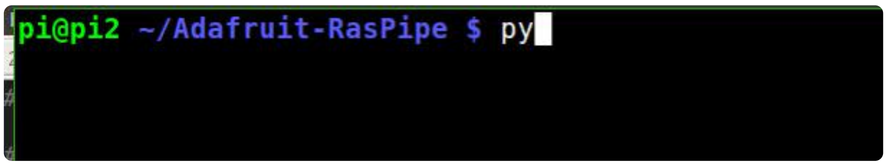
OO can be a very deep topic, but at heart it isn't as complicated as it's often made out to be. The short version is that it offers a way to bundle together state (variables, often called properties when they're part of an object) with behavior (functions, often called methods when they're part of an object).

The RasPipe class describes an object. You can think of an object as a sort of box around a collection of variables (properties) and functions (methods). Here's a very simple Python class and some code which gets an object (an instance of this class) and calls one of its methods:

```
#!/usr/bin/env python

class HelloSayer:
    name = ""

    def __init__(self, name):
        self.name = name

    def speak(self):
        print("Hello " + self.name)

s = HelloSayer("Ralph")
s.speak()
```

`def foo(param1, param2:` is how you define a function in Python. The first parameter to object methods in Python is always the object instance itself. This is how objects refer to their own properties and call their own methods.

`__init__()` is a special method that gets called when we create an object instance with something like `s = HelloSayer("Ralph")`.



```
pi@pi2 ~/Adafruit-RasPipe $ py
```

RasPipe works just the same, with more properties and methods.

raspipe.py is a Utility and Takes Options

Most of what's in `raspipe.py` is a class definition, but if you look at the bottom you'll see a few lines that turn it into a standalone utility program and let it take some useful options:

```
# Handle running this file as a standalone script.
if __name__ == '__main__':
    rp = RasPipe(sys.stdin)

    opts, args = getopt.getopt(sys.argv[1:], 'sx:y:')
    for opt, arg in opts:
        if opt == '-x':
            rp.size[0] = (int(arg))
        if opt == '-y':
            rp.size[1] = (int(arg))
        if opt == '-r':
            rp.grep_for = arg
        if opt == '-s':
            rp.toggle_stars()

    rp.run()
```

The `if __name__ == '__main__':` bit is a little magic that lets us check whether the current file is being run as the main Python program. (This way, other programs can include and reuse the RasPipe class for their own purposes - we'll get to that in Part 2.)

With `rp = RasPipe(sys.stdin)` we create an instance of RasPipe with `self.infile` set to `sys.stdin`. (To see everything that happens here, have a look at the `__init__()` defined at the top of the class.)

Next, we use the getopt () module to see if the user passed in any command line arguments. This lets us do the following:

-s	Start with stars enabled instead of scrolling text.
-x 640	Start with a screen 640 pixels wide.
-y 480	Star with a screen 480 pixels tall.
-r '.*foo.*'	Have a penguin appear onscreen thinking any line that matches the regular expression `.*foo.*`.

We could easily add other options here. As an exercise, you might try adding a feature to change a property like `bgcolor` from the command line.

Finally, we call `rp.run()`, which is the method that does most of the heavy lifting.

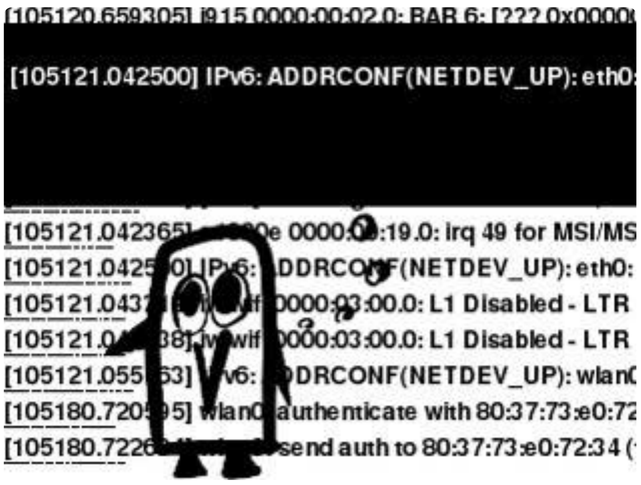
RasPipe Has a Grepping Penguin

Suppose you have a lot of output streaming through `raspipe.py`, and want to highlight lines that match a particular pattern? Normally, you'd use a tool like `grep` () to accomplish this.

Fortunately, Python supports its own variety of regular expressions, so it's pretty easy to give our program its own simple version of grep-like features. For example, if I wanted to check the output of `dmesg` for things about ethernet interfaces, I could do:

```
dmesg | ./raspipe.py -r '.*eth0.*'
```

And see someting like the following:



```
[105120.659305] i915 0000:00:02.0: BAR 6: [??? 0x0000
[105121.042500] IPv6: ADDRCONF(NETDEV_UP): eth0:
[105121.042365] [illegible]e 0000:[illegible]:19.0: irq 49 for MSI/MS
[105121.042[illegible]0] IPv6: [illegible]DDRCO[illegible]F(NETDEV_UP): eth0:
[105121.043[illegible] [illegible] 0000:[illegible]3:00.0: L1 Disabled - LTR
[105121.0[illegible]38] iwlwifi 0000:03:00.0: L1 Disabled - LTR
[105121.055[illegible]53] [illegible]v6: [illegible]DRCONF(NETDEV_UP): wlan
[105180.720[illegible]5] wlan0: authenticate with 80:37:73:e0:72
[105180.7226[illegible] send auth to 80:37:73:e0:72:34 (
```

This is done inside `run()` by checking each line to render against a regular expression:

```
            for render_line in to_render:
                # Show a penguin thinking if we have a regular expression to
                # search for and found a match:
                if self.grep_for is not None:
                    if re.match(self.grep_for, render_line):
                        matched_line = render_line
```

And later calling a special method to draw the penguin and its thought:

```
            if matched_line is not None:
                self.penguin_think(matched_line)
```

That function looks like this:

```
    def penguin_think(self, thought):
        """Have a penguin appear and think a thought."""
        r = self.penguin.get_rect(left=50, bottom=self.height)
        self.screen.fill(self.fgcolor, [0, 10, self.width, r.top])
        self.screen.blit(self.penguin, r)
        thought_surface = self.font.render(thought, True, self.bgcolor)
        thought_r = thought_surface.get_rect(left=5, top=30)
        self.screen.blit(thought_surface, thought_r)
        pygame.display.update()
```

RasPipe Handles Mouse Clicks

Pygame offers a way to process all sorts of user input.

In this case, we only really deal with one interaction: When the user clicks on the screen, we toggle the display between a scrolling-text mode and a mode that displays little star images. To see how this is done, have a look at the first few lines of `run()`:

```
    def run(self):
        """Process standard input."""
        line = self.infile.readline()
        while line:
            for event in pygame.event.get():
                if event.type == pygame.QUIT:
                    sys.exit()
                if event.type == pygame.MOUSEBUTTONDOWN:
                    self.toggle_stars()
```

Just like in `machine_stars.py`, we're looping over the contents of standard input. Here, however, we also check the Pygame event loop () and make a decision if there's been a mouse click (or if the window has been closed, which will give us a `pygame.QUIT`).

RasPipe Plays Sound

You may have noticed this bit in `__init()__`:

```
        # A little bit of sound.
        pygame.mixer.init()
        self.click = pygame.mixer.Sound('./tick.wav')
```

...and later on in `run()`:

```
            if self.tick % self.display_lines == 0:
                self.click.play()
```

This just plays a clicking noise every so often. Pygame's mixer () can be used to do all sorts of things with individual sound files, and pygame.mixer.music () allows for a custom soundtrack.

Pygame on the PiTFT and Next Steps

If you have a PiTFT display, you can use it to display the output from Pygame, and avoid the overhead of running the desktop system at the same time. Since my goal is to use a PiTFT as an auxiliary display to my main computer, I've been targetting a 320x240 display.

For this guide, I'm using the 320x240 2.8" TFT+Touchscreen for Raspberry Pi (). We have detailed installation instructions here (). Make sure you configure the console to appear on the PiTFT.

Under the hood, Pygame uses SDL for graphics and audio. SDL needs to be told what device it should use as a framebuffer display. Remember these commands from earlier?

```
export SDL_FBDEV=/dev/fb1
shuf /usr/share/dict/words | head -50 | ./machine_stars.py
```

`export` just tells the shell that a variable should be available to processes you start within the shell. `SDL_FBDEV=/dev/fb1` tells the SDL library what device file maps to the display we want it to use.

raspipe_pitft.sh & machine_stars_pitft.sh

You could remember to export this variable every time you open up a new shell, but there's a good chance you'll forget. A better approach might be to write a wrapper script like this one:

```
#!/usr/bin/env bash

# A wrapper for displaying raspipe.py output on /dev/fb1

export SDL_FBDEV=/dev/fb1
cat /dev/stdin | ./raspipe.py $@
```

Since `/dev/stdin` is just a file from the perspective of the script, we can pass it off to a command really easily. We also tack on `$@`, which is all the arguments to the script. This way we can say things like `./raspipe_pitft.sh -r '.*foo.*'` and `raspipe.py` will see the right options.

You can see this in the Adafruit-RasPipe repository, as `raspipe_pitft.sh`, and it can be invoked like so:

```
shuf /usr/share/dict/words | head -50 | ./raspipe_pitft.sh
```

If you write your own wrapper script, you'll want to tell Linux that it's executable. Let's say you decide to write the equivalent for `machine_stars.py`. First, open a new file in Nano:

```
nano machine_stars_pitft.sh
```

And then add the following:

```
#!/usr/bin/env bash

# A wrapper for displaying machine_stars.py output on /dev/fb1

export SDL_FBDEV=/dev/fb1
cat /dev/stdin | ./machine_stars.py $@
```

Finally, use `chmod` to set the executable bit on your new script:

```
chmod +x ./machine_stars_pitft.sh
```

Now you should be able to test like so:

What Now?

By now, hopefully you have a better idea of how to use Pygame for simple visualizations.

Next, we'll explore building a simple network wrapper for RasPipe so you can send it data to visualize from anywhere on the network. Stay tuned for Part 2!