Turtle Braille User Guide

# Introduction

The goal of this guide is to aid the user of this software. Our assumption here is that, for a large part, we live a multidimensional world and multidimensional programming examples help. Besides, graphics is fun! As a programing teaching aid, we use Python and particularly the module turtle to emphasize graphics. In this, the visually impaired are generally left behind. We are attempting to improve the situation by providing additional help to their “seeing” graphics. We add both tactile and auditory feedback to graphical python turtle programs. These are simple programs written or modified by the student.

Our software enhances the user’s “viewing” experience by adding **extended graphics view**s to programs containing turtle statements. An **extended graphics view** provides an additional user sense to the standard turtle display.

We have added the following **extended graphics view** displays to each program’s normal display:

* **Text output**: rectangular array of letters to be sent to Braille machine which produces a coarse resolution picture
  + Currently 32 columns by 25 rows, representing the turtle window display
  + This display text is placed in the program’s standard output (console) when the user program calls turtle function **done** / **mainloop**.
  + Additionally, this picture text is placed in the system clipboard to aid the user in copying the text.
  + Actual transmission to Braille embossing machine and producing of output is not currently part of the program.
  + A few transformations are made after the cell generation to facilitate Braille reading:
    - Each of the non-trailing spaces is converted to a “,”
    - The figures are moved closer to the left edge and top edge.
* **Audio Drawing Window**: A window presenting a visual rendition of the Braille machine output. This window can be navigated by the user, producing spoken or tonal sound feedback, indicating the current cursor position.
  + Currently 32 columns by 25 rows
  + This display window is created when the user program calls turtle function **done** / **mainloop**.
  + Braille is simulated with rectangular cells containing dots arranged to simulate actual Braille. This is done to aid the sighted.

# A good time to get a live demonstration

Hopefully of “Hello World!” simplicity, with a bit of graphics flair. In truth, we went a bit further to better demonstrate the capabilities or limitations of our displays. We normally begin with a single-colored square.

# spokes.py

# Display a star with spokes

from turtle\_braille\_link import \* # Set link to library

#from turtle import \* # Bring in turtle graphic functions

speed(*"fastest"*)

for i in range(7): # Do things 7 times

if i == 0:

color(*"red"*)

elif i == 1:

color(*"orange"*)

elif i == 2:

color(*"yellow"*)

elif i == 3:

color(*"green"*)

elif i == 4:

color(*"blue"*)

elif i == 5:

color(*"indigo"*)

else:

color(*"violet"*)

forward(300)

dot(100)

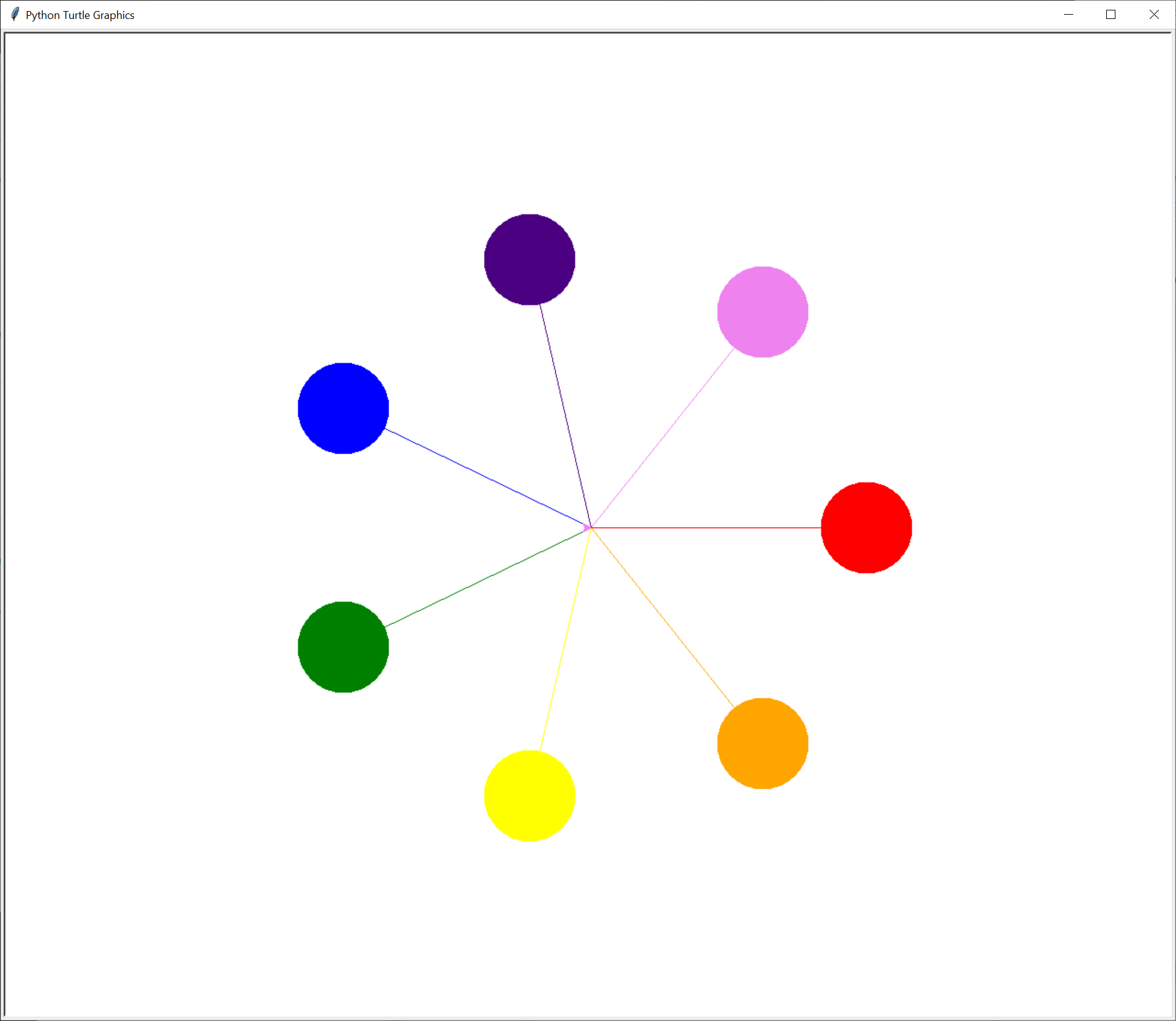
backward(300)

right(360/7)

done()

## Sample Program Output

### Turtle Graphics Screen



### Standard Output, Including Text for Brallier

resource\_lib\_proj/src is already in path

resource\_lib\_proj/src is already in path

*… Omitted for brevity …*

Braille Display - Braille Print Output

,,,,,,,,,,,,,iii

,,,,,,,,,,,,iiiiii

,,,,,,,,,,,,iiiiii,,,,,,,,vvv

,,,,,,,,,,,,iiiii,,,,,,,vvvvvv

,,,,,,,,,,,,,,,i,,,,,,,,vvvvvv

,,,bbb,,,,,,,,,ii,,,,,,,,vvvvv

,,bbbbb,,,,,,,,,i,,,,,,vvv

,,bbbbbb,,,,,,,,i,,,,,vv

,,bbbbbbbb,,,,,,ii,,,vv

,,,,,,,,,,bbbb,,,i,,v,,,,,,,,,,,rr

,,,,,,,,,,,,,bbbbivv,,,,,,,,,,rrrrrr

,,,,,,,,,,,,,,,,bvvrrrrrrrrrrrrrrrrr

,,,,,,,,,,,,,ggg,yoo,,,,,,,,,,rrrrrr

,,,,g,,,,ggggg,,,y,,o,,,,,,,,,,,rr

,,gggggggg,,,,,,yy,,,oo

,,gggggg,,,,,,,,y,,,,,oo

,,ggggg,,,,,,,,,y,,,,,,ooo

,,,,g,,,,,,,,,,yy,,,,,,,,ooooo

,,,,,,,,,,,,,,,y,,,,,,,,oooooo

,,,,,,,,,,,,yyyyyy,,,,,,oooooo

,,,,,,,,,,,,yyyyyy,,,,,,,,ooo

,,,,,,,,,,,,yyyyyy

,,,,,,,,,,,,,yyy

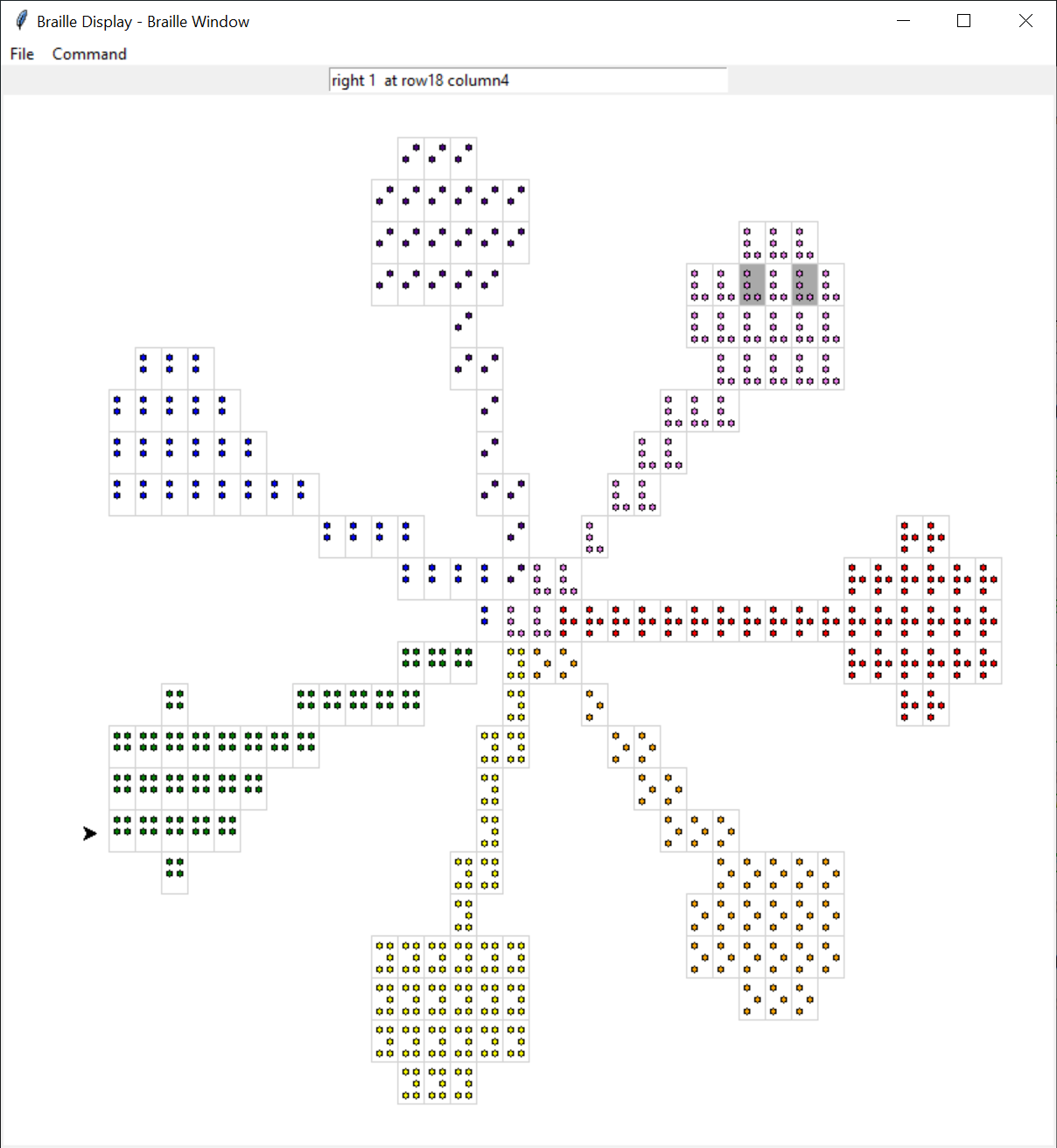
*… Omitted for brevity …*

up 2 at row17 column36

Standard Output, Including Text for Brallier

* “g” – Color is represented by first letter of the color string, e.g., “g” for green
* “,” – Each non-trailing space is replaced by a “,” whose, braille representation empty-like (only one of the possible six dots) but prevents Braille machine software from compressing multiple-space strings.
* Figure(s) moved closer to the left and top edges to make the figures easier to find.

### Braille Window – View of Braille Figure



* Surrounding rectangles are present to make it easier for me to see groupings.
* The dot color is just to remind me of the color represented by the Braille symbol.

### Braille Window Feedback

The braille window was first created as a development/documentation tool. Its primary function was to provide me, a sighted person with no access to a Braille machine, a way to better visualize the braille produced when the text output was passed to the actual Braille machine. I enhanced the Braille Window to help the user navigate around the window and report the content of the current cursor position. The user can, via keyboard key presses and mouse operations, move to around the window. Cursor position is reported as current color or the distance from the closest figure. Optionally reported is the cursor position on the Braille grid (default 25 rows, 40 columns). Position is spoken and printed by default.

Help Message (keyboard commands)

*h - say this help message*

*Up - Move up one row*

*Down - Move down one row*

*Left - Move left one column*

*Right - Move right one column*

*g - Go to closest figure*

*p - Report/Say current position*

*Escape - flush pending report output*

Keyboard commands are case insensitive

* Up, Down, Left, Right are keyboard arrow keys
* p – Report current position (default) adds current row, column to end of position reporting messages in the form “at row *N* column *M*”
* Escape – flush pending audio output
* Position Report Examples:
  + At top edge at row1 column15 – *Cursor is at top, at row 1 from top, column 15 from left edge*
  + down 9 at row4 column23 - *Figure is down 9 rows*
  + right 7 down 12 at row1 column13 – *Figure is right 7 columns, down 12 rows*
  + green at row13 column20 *– Cursor on green cell*
  + down 1 at row12 column20 *– Figure is down 1 row*
  + At right edge at row16 column40 – *Cursor is at right*
  + up 5 – Figure is directly up 5 rows, no “At row…column” reported
  + green – Cursor on green cell, no “At row…column” reported

To help with the growing number of commands, these commands were split between action-like commands such as “Up” and “Down: and state commands such as “z - stop reporting position”. The “state” commands were placed in the “Commands” menu.” These can be reached via “Alt-c”, then the corresponding short-hand letter.

Command Menu Commands

Help - list command (Alt-c) commands

h - say this help message

a - Start reporting position

z - stop reporting position

e - echo input on

o - echo off

v - visible cells

i - invisible cells

r – redraw figure

s - silent speech

t - talking speech

l - log speech

n - no log speech

p – report position

u – audio beep

d – no audio beep

Escape – flush pending report output

* a – Start reporting location (default) adds current row, column to end of position reporting messages in the form “at row *N* column *M*”
* l – log speech (default) – begin sending talk speech to standard output (console)
* n – Stop logging talk – Stop logging/printing speech. This may be helpful, if alternate speaking of the console, such as created by JAWS or NVDA programs is confusing the listener.
* s – silent - stops the talking. This may be helpful for sighted users as it greatly increases screen navigation speed.
* t – talking speech (default) starts talking reports
* v – Make figure cells visible(default) reverses w
* i – Make figure cells invisible – gives the sighted viewer a sense of the difficulty navigating when one cannot see the drawing.
* z – Stop reporting location – removes “at row *N* column *M*” from end of position reports.
* u – audio beep – Position is reported via audio beeps to reduce response delay. Cells are announced with a color determined beep frequency. Blanks have a lower frequency beep. After a cell is announced, the probable next position/cell is announced in a shorter beep to aid the user in sensing a change.
* d – no audio beep – restores talking position announcing

#### Accelerated Feedback – “audio beep”

Talking figure feedback greatly slows down display navigation. This slowdown is somewhat reduced by omitting some of the feedback such as the input echo and position announcing. In an attempt further reduce the overhead of speech we introduced an “audio beep” mode, in which the feedback was done via audio beeps produced by the Python winsound module. Blank area produces a low frequency beep. Colored cells produce a beep whose frequency indicates the color. We currently handle only the main 7 rainbow colors. Because of the limited delay of the beeps, we added a predictive beep to indicate the cell which follows, given the current search direction. We added a “p – position command” to speak the current position to make up for lack of exact position announcements in audio beep mode.

# Requirements / limitations:

* Environment
  + Extra files need to be in path (GitHub: raysmity619/resource\_lib/src/\*)
  + A directory path named “resource\_lib\_proj/src” must exist or be created somewhere within the working directory path. The extra files named above must be placed in this directory (src).
  + Installing pyttsx3 is required for program generated speech. If not installed reporting is limited to console printing. (“pip install pyttsx3” works for me)
  + The audio beeping, used for faster location info uses the python winsound module which is Microsoft Windows specific.
* User coding
  + Use “from turtle\_braille\_link import \*” instead of “from turtle import \*”.
* Turtle feature limitations
  + 70-80% of language – no animation support
  + Limited color support
  + circle not well supported – no time
  + shape plus some others – no time

# Background

Before outlining inner details, I will give a short outline as to how this program came about.

I’ve been a programmer for over half a century. I started programming in college. There I was going to be a Mathematician. But sadly, when I took my first non-required math course, late in my sophomore year, I thought I *might* be able to do this stuff but my heart isn’t in it. I scrambled to find a different major. Classical Electronics was out – to many around were much more ahead of me. Digital Electronics – a bit better – less abstract than Math but still logical. Programming – that’s it – no knowledge required here, just like a game. A career was started. And I got paid.

Currently, I am, depending on my mood, self-employed, retired, or unemployed. During the recent few years, I’ve been volunteer teaching programming to those who have “missed the programming boat”. These people are not programmers, but are interested in finding if programming might be for them. They might be curious at least at what programming can be – not just the mad typing on a console as one might see on TV or in a movie. My courses are not for those programmers who use programming language *X* and want to pick up the programming language Python, which I demonstrate in my teaching. There are many good in-person courses, online courses, and tutorials. My courses start by presenting the student a ready-made program starting very simple like ‘print(“Hello world!”)’. I run it for them. They, in turn, modify the program, and run it again. But I found that many programs, while instructive, were often too abstract, without much “action” or feedback. I hoped to increase the “action” to hold the fledging programmer’s interest. We moved to simple graphical program examples, e.g., display a square, that would provide additional visual impact. To facilitate this, we made use of Python’s **turtle** graphics module.

Along the way, aided by the presence of the famed **Perkins School for the Blind** a few miles from my home, it hit me – these fine graphical examples were much less impactful to the prospective programmer who can’t see the pictures. What’s left? Should we restrict programming examples to text-based output? This would be possible, but would, sadly, abandon a whole area of programming. Faced with the scarcity of widely available economical graphical display devices for those who can’t see, we were stuck. However, rather than trying to tackle the very large problem of general graphical presentation for the visually impaired, I decided, to concentrate on making simple graphical programs more visible.

# How It Works

## Overview

### Functional flow

Our general approach captures each turtle command, e.g., **forward**, first sending this command, unchanged, to the turtle module. Secondly the turtle command is imitated, creating a similar graphical part of an **extended graphics view**. At the end of the program display process, during the call to **done** or **mainloop**, the completed **extended graphics view** is rendered into one or more displays.

### A Detailed Flow Example – **forward** turtlecommand

Conventions used below:

* Code file names are in bold to right.

**file\_name.py**

* Timing comments are as such:

Relative execution time point: Action

* Program code snippet:

Code from file

**square.py**

0.1: Setup linkage from user written turtle statements to our program processing

from turtle\_braille\_link import \*

### Linkage

* Searches for directory resource\_lib\_proj/src, if not already in search path, from the current directory up and add this to the search path.
* Does a “from turtle\_braille import \*” to add turtle function names to programs name space.

turtle\_braille\_link.py

0.2: Add TurtleBraille/TurtleDisplay code to path

if not in\_path:

sys.path.append(dir\_check)

0.3: Add links to shadow turtle functions

from turtle\_braille import \*

**turtle\_braile.py**

0.5: Create an instance of BrailleDisplay, the kernel of the shadow turtle production, and the link to this kernel.

from braille\_display import BrailleDisplay

bd = BrailleDisplay(win\_width=None,

win\_height=None,

grid\_width=32,

grid\_height=25)

def **mainloop**():

return bd.mainloop()

def **done**():

return bd.done()

**square.py**

forward(200)

1: Turtle calls in user program

A brief review of turtle:

* forward() – move pen forward specified amount, drawing a line.
* right() – turn right specified number of degrees.
* done() – complete drawing, continue display till done.

**braille\_display.py**

2: All turtle function calls except done(), and mainloop() go, directly to turtle member function with no connection to the TurtleBraille system,

3: User program calls done() which calls BrailleDisplay.mainloop() which calls BrailleDisplay.dislpay()

class BrailleDisplay:

def \_\_init\_\_(self, …args):

…setup

*self.tu\_screen = tur.Screen()*

*…* more initialization

def done(*self): # called by user's program*

*self.mainloop()*

def mainloop(*self):*

*self.display(title=title,…args)*

class BraileDisplay continued

4: display() sets up displays.

def display(self, …args):

…setup

tu\_canvas = self.tu\_screen.getcanvas()

self.canvas\_grid = CanvasGrid(master=mw,

base=tu\_canvas,…more args)

…

self.aud\_win = self.canvas\_grid.create\_audio\_window(

title=tib)

7: Scanning turtle’s tkinter.Canvas

**canvas\_grid.py**

**class CanvasGrid** is the repository of the canvas scanning code**.** The base argument, stored in the base data member is used to support operations on a separate canvas object.

class CanvasGrid(tk.Canvas):

def \_\_init\_\_(*self, master, base=None,*

*…more args*, \*\*kwargs):

if base is None:

base = *self*

*self.base = base*

Instances of using the base canvas

In our case, that is Turtle's canvas

cx1,cy1,cx2,cy2 = self.get\_grid\_ullr(ix=ix, iy=iy,

xs=xs, ys=ys)

item\_ids\_over\_raw =

**self.base**.find\_overlapping(cx1,cy1,cx2,cy2)

…

color\_tuple = **self.base**.itemconfigure(top\_id,

"fill")

Finding canvas items that overlap grid regions

def get\_canvas\_items(*self,*

xmin=None, xmax=None,

ymin=None, ymax=None,

ncols=None, nrows=None,

… other parameters):

# Get grid limits - defaulting to self.values

xs,ys = *self.get\_grid\_lims(xmin=xmin,*

*xmax=xmax, ymin=ymin,ymax=ymax,*

ncols=ncols,nrows=nrows)

ixy\_ids\_list = [] # Building list of

# ((ix,iy), [overlapping

# ids])

… debug / tracking statements

for ix in range(len(xs)-1):

for iy in range(len(ys)-1):

cx1,cy1,cx2,cy2 =

*self.get\_grid\_ullr(ix=ix, iy=iy,*

*xs=xs, ys=ys)*

item\_ids\_over\_raw =

self.base*.find\_overlapping(*

*cx1,cy1,cx2,cy2)*

if len(item\_ids\_over\_raw) == 0:

continue # Skip - no overlap

… code for selection on item type

if chosen:

item\_ids\_over.append(item\_id)

if len(item\_ids\_over) > 0:

ixy\_ids\_list.append(((ix,iy),

item\_infos\_over))

return ixy\_ids\_list

8: Create specified displays

9: Create Braille Window with audio feedback

10. Create text picture for Braille machine

def **print\_braille**(*self*, title=None):

*""" Output braille*

*"""*

if title is not None:

print(title)

if *self*.shift\_to\_edge:

*self*.find\_edges()

left\_edge = *self*.left\_edge

top\_edge = *self*.top\_edge

else:

left\_edge = 0

top\_edge = *self*.grid\_height-1

for iy in reversed(range(top\_edge)):

line = *""*

for ix in range(left\_edge, *self*.grid\_width):

cell\_ixy = (ix,iy)

if cell\_ixy in *self*.cells:

cell = *self*.cells[cell\_ixy]

color = cell.color\_str()

line += color[0]

else:

line += *" "*

line = line.rstrip()

if *self*.blank\_char != *" "*:

line = line.replace(*" "*, *self*.blank\_char)

###print(f"{iy:2}", end=":")

print(line)

### BrailleCell – Unit of Display

Each instance of the class BrailleCell represents a small rectangle display region. These regions are members of a grid covering the turtle display. These regions will be rendered to produce the **extended graphics views**

### Rectangle / Cell / BrailleCell Population and Figure Drawing

An **extended graphics view,** representing the displayed turtle graphics view**,** is composed of BrailleCell instances which represent a rectangular region. The **extended graphics view** picture is created in three steps:

1. Scan turtle tkinter.Canvas, collecting canvas items overlapping each of rectangles making up the 32 by 25 grid covering the canvas
2. Create a BrailleCell object for each of the regions containing at least one canvas items.
3. Scan the 32 by 25 grid for display cells:
   1. To generate the text picture one must go left to right for each line, placing spaces for missing (ix,iy) cells.
   2. Generating the Braille display does not require a special ordering because the cell drawing only depends on the (ix, iy) tuple and nonexistent cells are left undrawn.

# Admissions and Apologies

It’s time to fess-up, come clean, and so on. Here are a few, not a complete list, of things we could/will do better. Note the programming here is mine. For “we”, where it is admirable substitute myself plus folks who have taught or advised me, where it is a mistake or disaster substitute “me”.

## OK – I’ts not done

Most of my recent works are mostly prototypes. The rest are strictly prototypes. My greatest goals have been to investigate how my programming could make something interesting, fun, and maybe useful. Almost Fifty years ago Fred Brooks published a great software engineering book *The Mythical Man-Month: Essays on Software Engineering*. In this book, referring to projects much, much grander than mine, he advised one to “Build two systems and throw the first away”. In which he asserted that, in the first, we could learn the “big” lessons, and in the second we could “make it right”. Well, I’m mostly trying to learn the ”big” lessons. Possibly, for those short on time, we might follow the abridged plan – “Build one system and through the first away”. Of course, Mr. Brooks would strongly agree that this does not absolve us, the designers, implementers, deliverers, from using all our efforts in doing the best we can on the first **and** subsequent versions. I’m doing the best that I can. *It’s hard to be humble*…

## Little or no Testing

Outside a few instances, testing, really just exercising, is relegated to the self-test code at the end of the major class files. As this is a heavily interactive program, it’s in major need of testing.

## Not a standard Installable Package

I should learn more about this.

|  |
| --- |
|  |