Dashport

JP Etcheber

Jun 28, 2022

CONTENTS:

Dashport is a wrapper for Python's curses library to enable developers to easily make curses-based applications.

CHAPTER

ONE

USING DASHPORT

This the boilerplate code that can be used to start any application. This code won't display anything, it will just set up a curses screen, and continually refresh until the user uses Ctrl+C to break out of it. See documentation below for details on building on this boilerplate.

```python
from dashport.dash import Dashport, Info
from dashport.run import wrap

# This info function displays a better error message if
# the program crashes.
def info(stdscr):
    return Info(stdscr)


def dashport(stdscr):
    app = Dashport(stdscr)

    # This loop refreshes the screen. Put commands in this loop if you
    # want them to be refreshed every millisecond.
    while True:
        app.refresh()


if __name__ == '__main__':
    wrap(dashport, info)
```

1.1 Dashport App Options

When making a dashport app, you can hide the cursor, by setting the cursor to `False`. If you need to set the cursor back to visible during the program, use the `curs_set` method to bring it back.

```python
def dashport(stdscr):
    app = Dashport(stdscr, cursor=False)
    # bring the cursor back
    app.curs_set(True)
```

1.2 Printing Statements

Dashport contains its own print function, which is separate from Python's `print` command. To use the method, just add the method to the dashport function, as shown below.

```
def dashport(stdscr):
    app = Dashport(stdscr)
    app.print("Some text is on the screen!", x=0, y=10)
    while True:
        app.refresh()
```

The coordinate system in Dashport is `x` refers to the horizontal position (the column on the screen), and y refers to the (the row on the screen).

See *Add String Methods* for more information and other methods to add characters to your app's screen.

1.3 Coordinate layout

If you are familiar with the built-in Python curses library, you'll note that `x` and `y` for Dashport is switched, so that `x` refers to columns and `y` refers to rows. The origin `(0, 0)` is still the top-left of the screen, and values increase for `x` as you move right and increase for `y` as you move down the screen. Run the explore_screen.py example to see this in action.

This was done to make the coding experience a bit more approachable, so you can use the values of the size of the screen itself (`app.rows`, `app.cols`) to figure out the limits of whether or not a string or character can be placed.

1.4 Binding keys to the application

We probably want to make a quit command so that we can quit the program without having to press Ctrl+C. To do that, use the add_control method in the dashport function:

```
def quit(*args):
    exit(0)

def dashport(stdscr):
    app = Dashport(stdscr)
    app.add_control("q", quit, case_sensitive=False)
    ...
```

The first argument is a string, the key that is being bound. The second argument is the function itself, which will be added to the `app.controls` dict of the app we've defined. And finally, you can set an optional `case_sensitive` argument to `False`, if you would like both upper and lowercase versions of the key pressed to be bound to the same function.

In this example, the `q` key is bound to the quit function, which you can define yourself outside of the dashport function. You can also import functions from other python files or libraries, and bind those functions to other keys. Even if a function takes no arguments, the function still needs `*args` set. If you want your function to interact with the screen and have access to other Dashport functions, pass in "app" into the function:

In this example, the add_message function is bound to the `k` key. When pressed, the app will display the new message on the screen.

```
def add_message(app):
    app.print("A new message appears!")

def dashport(stdscr):
    app = Dashport(stdscr)
    app.add_control("k", add_message)
    ...
```

1.5 Native curses methods

Because this is a wrapper, and a work in progress, not all functions in curses will be implemented, but you may still use curses methods in your applications set up by Dashport. The `screen` attribute in the Dashport object is just defined as the curses object, so you can use any documented curses method with it.

In the example below, `addstr` is being used, but it's not the `addstr` method in Dashport, it's the `addstr` method native to curses, and so it takes different arguments.

```
def dashport(stdscr):
    app = Dashport(stdscr)
    app.screen.addstr(10, 10, "k")
    ...
```

CHAPTER

TWO

COLORS

Color text in curses is usually done via a `color_pairs` method. With Dashport, color pairs (foreground and background) are defined with a single string which sets both background and foreground using plain English words connected by the string `_on_`. So, if you want red text on a white background, the color definition would be `red_on_white`.

Set the `color_default` value when starting a Dashport object to set the default text color value. Setting the color_default to `None` will set the default color to the default colors of the terminal session.

```
from dashport.dash import Dashport, Info
from dashport.run import wrap

def info(stdscr):
    return Info(stdscr)

def dashport(stdscr):
    app = Dashport(stdscr, color_default="green_on_white")
    app.print("hello world", color="red_on_white")
    app.print("plain text, colored with color_default")
    app.print("black on blue text", color="black_on_blue")
    while True:
        app.refresh()


if __name__ == '__main__':
    wrap(dashport, info)
```

The `background` method will fill the background with the background for the `color` selected. It does this by setting down a space character in every text position. It's best to use background before any other printing commands.

Use the `rectangle` method to draw a rectangle. It's best to keep rectangles away from the edges or corners of the screen, or filling up the entire screen. Use `background` to fill the screen with a color, or insstr to insert a string into the corner.

```
from dashport.dash import Dashport, Info
from dashport.run import wrap

def info(stdscr):
    return Info(stdscr)

def dashport(stdscr):
    app = Dashport(stdscr, color_default="yellow_on_blue")
    app.print("this text won't show")
```

(continues on next page)

(continued from previous page)

```
    app.background(color="white_on_maroon")
    app.print("some color text", x=20, color="blue_on_yellow")
    app.rectangle(10, 10, 12, 12, color="blue_on_yellow")
    while True:
        app.refresh()


if __name__ == '__main__':
    wrap(dashport, info)
```

2.1 Default Color Definitions

In Python's implementation of curses there are 255 color pair positions available to define colors, and by default, Dashport uses those color pairs for color definitions. These are the colors Dashport defines. Run the `examples/util/color_palette.py` program in the examples folder for an interactive guide in showing how all the colors will look on your screen:

- default
- black
- maroon
- green
- olive
- navy
- purple
- teal
- silver
- grey
- red
- lime
- yellow
- blue
- fuchsia
- aqua
- white

Whenever a Dashport command, like `print`, uses a `color` argument, you can use the standard `<color>_on_<color>` formatted string to define the foreground and background.

```
# This creates blue color text with a yellow background.
app.print("some color text", color="blue_on_yellow")
```

Due to the limit of 255 color pairs, the `olive` and `teal` colors are not available as background colors, but you can still use them as background colors, simply by formatting them with an A_REVERSE argument in the print command:

```python
# This creates black text on an olive background.
app.print("some color text", color="olive_on_black", A_REVERSE=True)
```

DEFINING YOUR OWN COLOR DEFINITIONS WITH CURSES

You can still use regular color pairs you've defined in curses, if you want to get an exact color. When initiating Dashport, set the color_map option to False, and then create color pairs just as you would with the normal curses library. You can still use Dashport's `print` and other string commands, passing in the integer of the color pair you've defined, rather than a string.

```
from dashport.dash import Dashport
from dashport.run import wrap
import curses

def dashport(stdscr):
    app = Dashport(stdscr, color_map=False)
    curses.init_pair(1, 2, 4)
    app.print("plain text")
    app.print("plain text, colored with a curses color pair", color=1)
    while True:
        app.refresh()


if __name__ == '__main__':
    wrap(dashport)
```

CHAPTER

FOUR

FORMATTING TEXT

The print function can take various attributes to change the text. These text attributes are available in the Python curses documentation. Use the constants listed in the table to change the text. In the example below, A_BOLD is set, which boldens the text. Please note, that support for these attributes is very system-dependent, and some attributes may not be available on all systems. Any unavailable text formatting will be ignored by Dashport, rather than crash the program.

```python
from dashport.dash import Dashport, Info
from dashport.run import wrap

def info(stdscr):
    return Info(stdscr)

def dashport(stdscr):
    app = Dashport(stdscr, color_default="green_on_white")
    app.print("This text is bold.", A_BOLD=True)
    app.print("This text is italic.", A_ITALIC=True)
    while True:
        app.refresh()


if __name__ == '__main__':
    wrap(dashport, info)
```

CHAPTER

FIVE

STRING METHODS

5.1 insstr

The insstr (insert string) method is used when a character should be placed directly at the current cursor position without the cursor moving. If your operation requires a character to be placed in a corner of the screen, it's best to use insstr so that placing the character doesn't cause the cursor to move to the right by one and crash your program.

In the example below, this will place your character at the bottom left corner of the screen, regardless of the size of the terminal window.

```
def dashport(stdscr):
    app = Dashport(stdscr)
    app.instr("X", x=0, y=app.rows - 1)
    ...
```

5.2 addstr

The addstr method adds all the characters of a string to the x, y position.

```
def dashport(stdscr):
    app = Dashport(stdscr)
    app.addstr("This string overwrites all other content", x=0, y=app.rows - 1)
    ...
```

5.3 print

This method is meant to be close in function to Python's print method and lets you mimic a TTY on a curses screen. Like `addstr`, it will add characters to the screen, but unlike `addstr`, it will also move the cursor down by one y-coordinate to simulate a carriage return. You may opt to turn this behavior for a print statement off by passing in an `end` option, to override the default, but if that's your use case you may want to use `addstr` instead.

Like addstr, it can take `x` and `y` coordinates, but when that is implemented, the next print statement would follow below the previous print statement that had no `x` and `y` coordinates set.

```
def dashport(stdscr):
    app = Dashport(stdscr)
    app.print("This string overwrites all other content")
    app.print("This is in a different location.", x= 10, y=15, end=False)
```

(continues on next page)

(continued from previous page)

```
    app.print("This is on the next line.")
    ...
```

CHAPTER SIX

LAYOUTS

Layouts in Dashport can be used to divide the screen easily and make separate panels that you can use `print` to display text on.

In the split_screen_quad example the `split_screen_quad` method is done on the app, with `borders` enabled. You can then print to these panels, by passing in a "panel" string value to the print command. Each screen can only have one layout at a time, which the app stores in the app.panels dict under the key "layout".

Using the `print` command, select the panel, by passing in a string which tells both that print should access the "layout" key to find the panel, and the index value of the panel that should be manipulated. Example:

```
app.print("some text", panel="layout.0")
```

This would print `some text` to the first panel of the layout.

By default, the starting position of each panel is at `[0, 0]`, but you can manually set x and y to whatever position you need. Important note: the coordinates in each panel are relative to the panel, not the entire screen. Use the `panel_dimensions` attribute of each panel to evaluate the current maximum columns and rows that are available in each panel. The `panel_dimensions` attribute is a list of tuples, which give the dimensions of each panel.

```
#!/usr/bin/env python3
from dashport.dash import Dashport, Info
from dashport.run import wrap

def dashport(stdscr):
    app = Dashport(stdscr, color_default=176)
    app.add_control("q", quit, case_sensitive=False)
    app.split_screen_quad(borders=True)
    app.print("panel 0", panel="layout.0")
    app.print(f"Rows: {app.panel_dimensions[0][0]}", x=5, y=2, panel="layout.0")
    app.print(f"Columns: {app.panel_dimensions[0][1]}", x=5, y=3, panel="layout.0")
    while True:
        app.refresh()


if __name__ == '__main__':
    wrap(dashport, info)
```

Other available splits are `split_screen_columns`, which divides the screen in two equal pieces, divided vertically, and `split_screen_rows` which does the same split with the division being horizontal.

6.1 borders

You can also pass in a list of booleans into any layout to enable borders. Do this if you want some panels to have their border enabled, and some not. You can still just pass in a single boolean (not in a list), to enable borders on all panels.

```
    app.split_screen_quad(borders=[False, False, False, True])
```

6.2 border_styles

The border_styles attribute can also be used to define the border styles of the individual panels. For this, provide a list of numbered border styles (see table below for the complete list of built-in border styles) that matches the panels that will be generated. Even if some panels don't have borders (in the example below, the second panel does not have a border), still place a value for the border style as a placeholder.

```
app.split_screen_quad(border=[True, False, True, True],
                      border_styles=[0, 1, 1, None])
app.panels["layout"][3].border('M', 'M', '=', '=', ' ', ' ', ' ', ' ')
```

By using `None` as a border style, the built-in borders are not used, and the program can then manually provide a border style, using the native curses method. The panels are all in the `app.panels` attribute as a list, so borders can be set manually for any panel that exists. To disable a border you've created manually, simply define that section of the border with a space character.

Finally, there is also a `custom_border` function available in Dashport Borders which makes custom borders using unicode characters a bit more flexible than what the native curses method allows. Borders in the native curses method cannot represent a lot of unicode characters, due to the way the characters are handled in ncurses. Using dashport, you can set a list of 8 characters to represent the upper left, upper right, lower left, lower right, left vertical, right vertical, top horizontal, and bottom horizontal characters and apply that to any panel on the screen.

In the example below (from the custom_borders.py example), the border_characters list is using the import of BoxDrawing, aliased as boxes, to get the exact characters needed, and then passing that, with the panel, into the `custom_border` function.

```
#!/usr/bin/env python3
from dashport.dash import Dashport
from dashport.run import wrap
from dashport.borders import custom_border
from dashport.characters import BoxDrawing as boxes


def quit(app):
    exit(0)


def dashport(stdscr):
    app = Dashport(stdscr, color_default=176)
    app.add_control("q", quit, case_sensitive=False)
    app.layout("single_panel")
    border_characters = [
            chr(boxes.html("double_down_and_right")),  # upper left
            chr(boxes.html("double_down_and_left")),   # upper right
            chr(boxes.html("double_up_and_right")),    # lower left
```

(continues on next page)

(continued from previous page)

```
            chr(boxes.html("double_up_and_left")),     # lower right
            chr(boxes.html("double_vertical")),        # left vertical
            chr(boxes.html("double_vertical")),        # right vertical
            chr(boxes.html("double_horizontal")),      # top horizontal
            chr(boxes.html("double_horizontal"))       # bottom horizontal
    ]
    custom_border(app.panels["layout"][0], border_characters)
    app.print("panel 0", x=5, y=2, panel="layout.0")
    app.print(f"Rows: {app.panel_dimensions[0][0]}",
              x=5, y=3, panel="layout.0")
    app.print(f"Columns: {app.panel_dimensions[0][1]}",
              x=5, y=4, panel="layout.0")

    while True:
        app.refresh()


if __name__ == '__main__':
    wrap(dashport)
```

The built in border styles that are available through the dashport library are shown here:

border_styles #	Definition
0	Default borders provided by curses `.border()` method
1	dashes and pluses `('|', '|', '-', '-', '+', '+', '+', '+')`
2	slashes no corners `('\\', '/', '=', '=', ' ', ' ', ' ', ' ')`
3	double box drawing `('' ,'', '', '', '', '', '', '' )`

Please note, that you can't use all of these characters in the native curses method, and if you do want to define your own borders, it's best to use chr() to define exactly which character it should be.

CHAPTER

SEVEN

THREE WAY SPLITS

Splitting the screen in three ways means that you can determine which side of the screen the longer panel will be on (top or bottom for horizontal splits and left or right for vertical splits), and what the width of that panel will be.

When using `split_screen_three_vert`, set `long_side` to `left` or `right` to set the longer panel to the left or right, and then `long_side_width` to set the width of that panel. The other two panels will fill the screen to compensate. See split_screen_three_vert.py for an example.

When using `split_screen_three_horizontal`, set `long_side` to `top` or `bottom` to set the longer panel to the top or bottom, and then `long_side_height` to set the height of that panel. The other two panels will fill the screen to compensate. See split_screen_three_horizontal.py for an example.

CHAPTER EIGHT

WIDGETS

8.1 title_bar

The title bar widget occupies one row on either the top or bottom of the screen to display persistent information. The title bar does take away one row from the screen, so even if a `print` command designates a spot where the title bar is, those coordinates will not overlap with the title_bar.

Example:

```python
#!/usr/bin/env python3
from dashport.dash import Dashport, Info
from dashport.run import wrap


def info(stdscr):
    return Info(stdscr)


def dashport(stdscr):
    app = Dashport(stdscr)
    app.widget("title_bar", text="This is the title bar", color=256)
    app.print(content="this is text", x=0, y=0)
    while True:
        app.refresh()


if __name__ == '__main__':
    wrap(dashport, info)
```

CHAPTER

NINE

DASHPORT EXAMPLES

The examples/ folder of this repo show how Dashport can be used in a variety of ways. Each example can be run independently, and with only the Dashport library installed.

The categories are:

- *boilerplate*
- *layouts*
- *util*
- *views*

CHAPTER

TEN

BOILERPLATE

Use these examples as starters for your own projects. These examples cover the bare minimum of creating a window, writing text to the screen, and getting used to the concept of the Dashport wrapper and how the program loop works.

10.1 basic_window

10.1.1 Summary

Creates a basic window with a keyboard command to quit. Also demonstrates creating text on the screen.

10.1.2 Details

The `Info` class, which is passed into Dashport's `wrap` function, is for error handling. If curses crashes, it will show a friendly error message along with the dimensions of the terminal, which is usually a contributing factor to the crash.

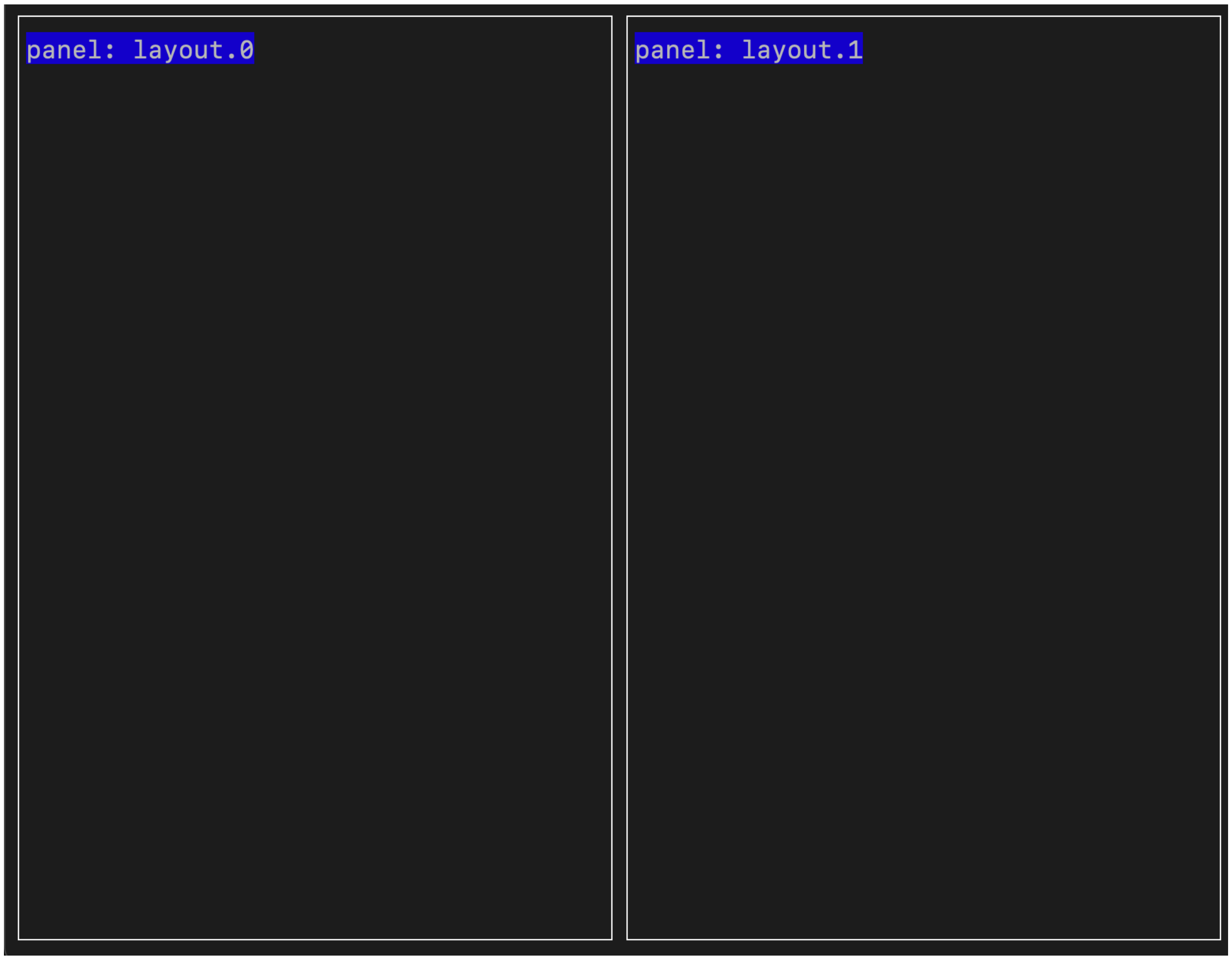
CHAPTER

ELEVEN

LAYOUTS

These examples show the different layouts that can be implemented through Dashport.

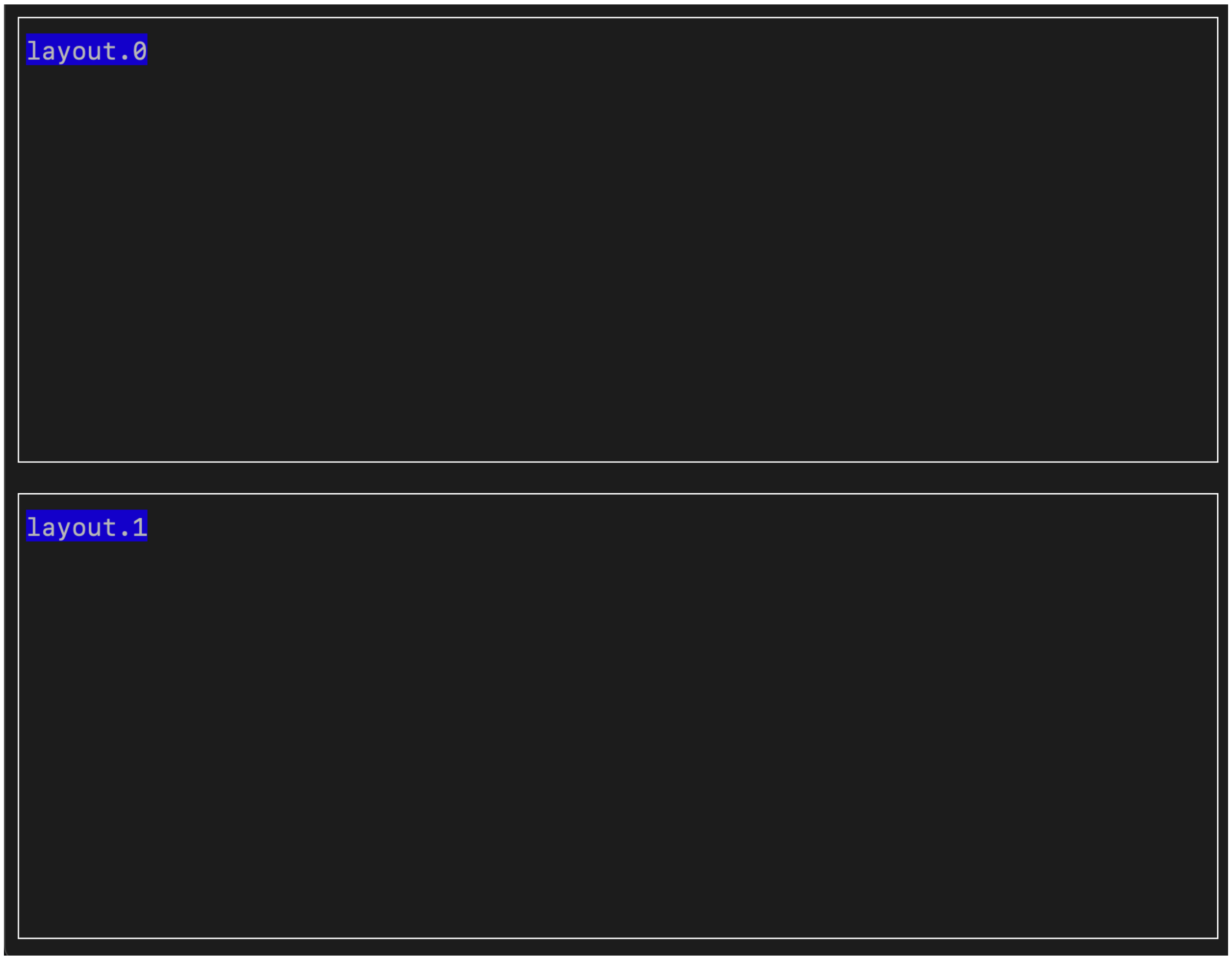
11.1 split_screen_columns



```
panel: layout.0
```

```
panel: layout.1
```

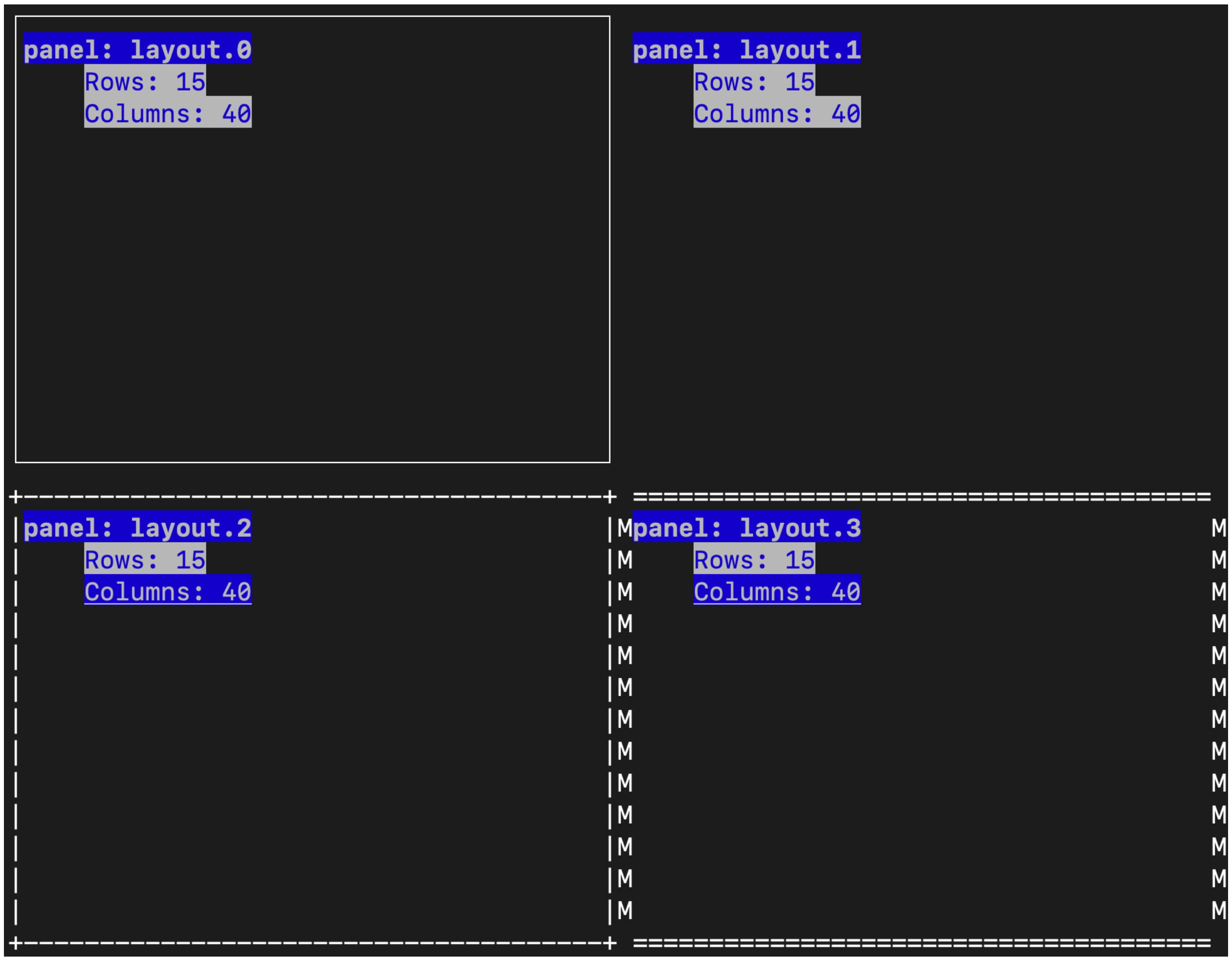
11.2 split_screen_rows



```
layout.0
```

```
layout.1
```

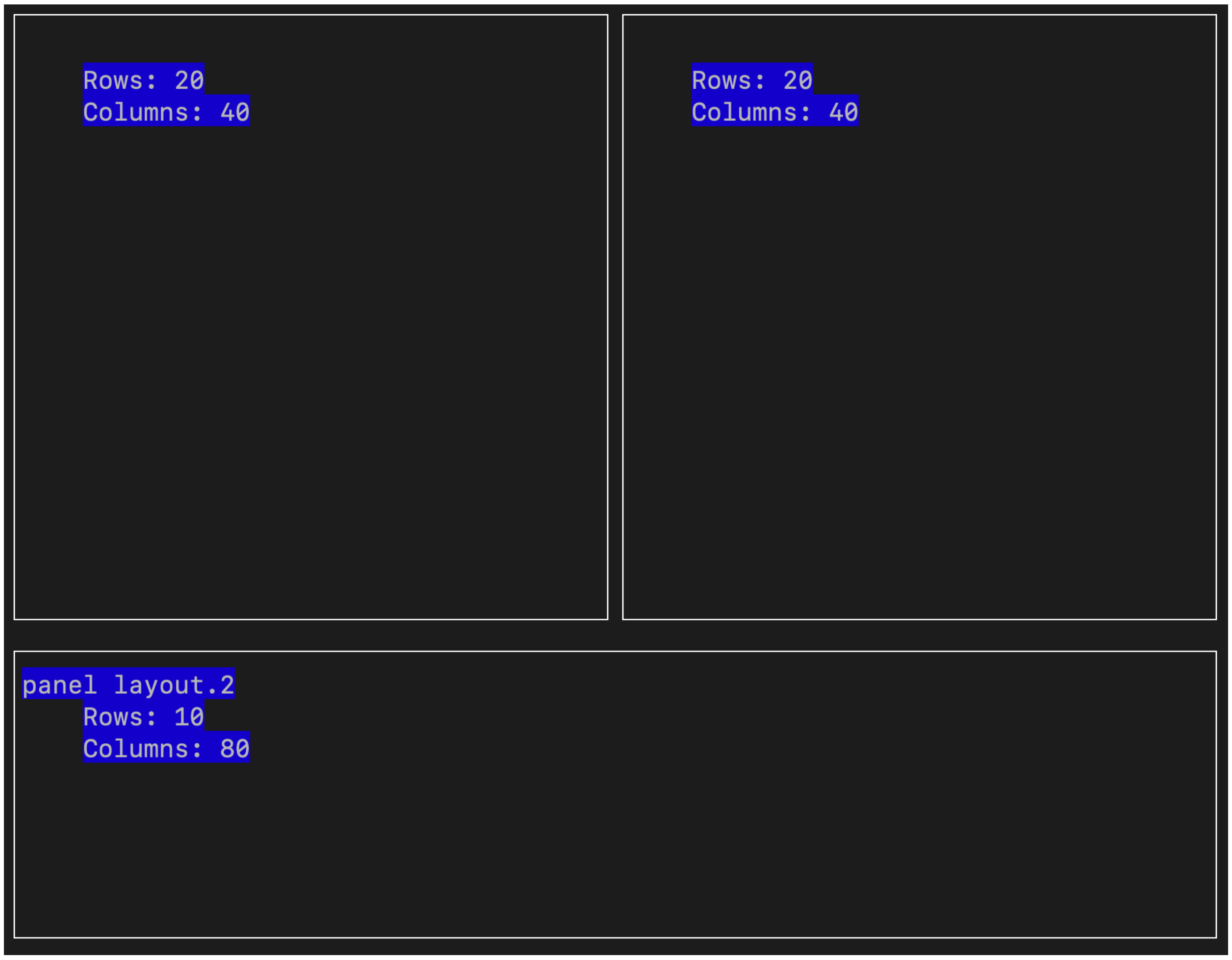
11.3 quadrants



```
 panel: layout.0                         panel: layout.1
     Rows: 15                                Rows: 15
     Columns: 40                             Columns: 40

+--------------------------------------+ ======================================
|panel: layout.2                       |Mpanel: layout.3                       M
|    Rows: 15                          |M    Rows: 15                          M
|    Columns: 40                       |M    Columns: 40                       M
|                                      |M                                      M
|                                      |M                                      M
|                                      |M                                      M
|                                      |M                                      M
|                                      |M                                      M
|                                      |M                                      M
|                                      |M                                      M
|                                      |M                                      M
|                                      |M                                      M
|                                      |M                                      M
+--------------------------------------+ ======================================
```

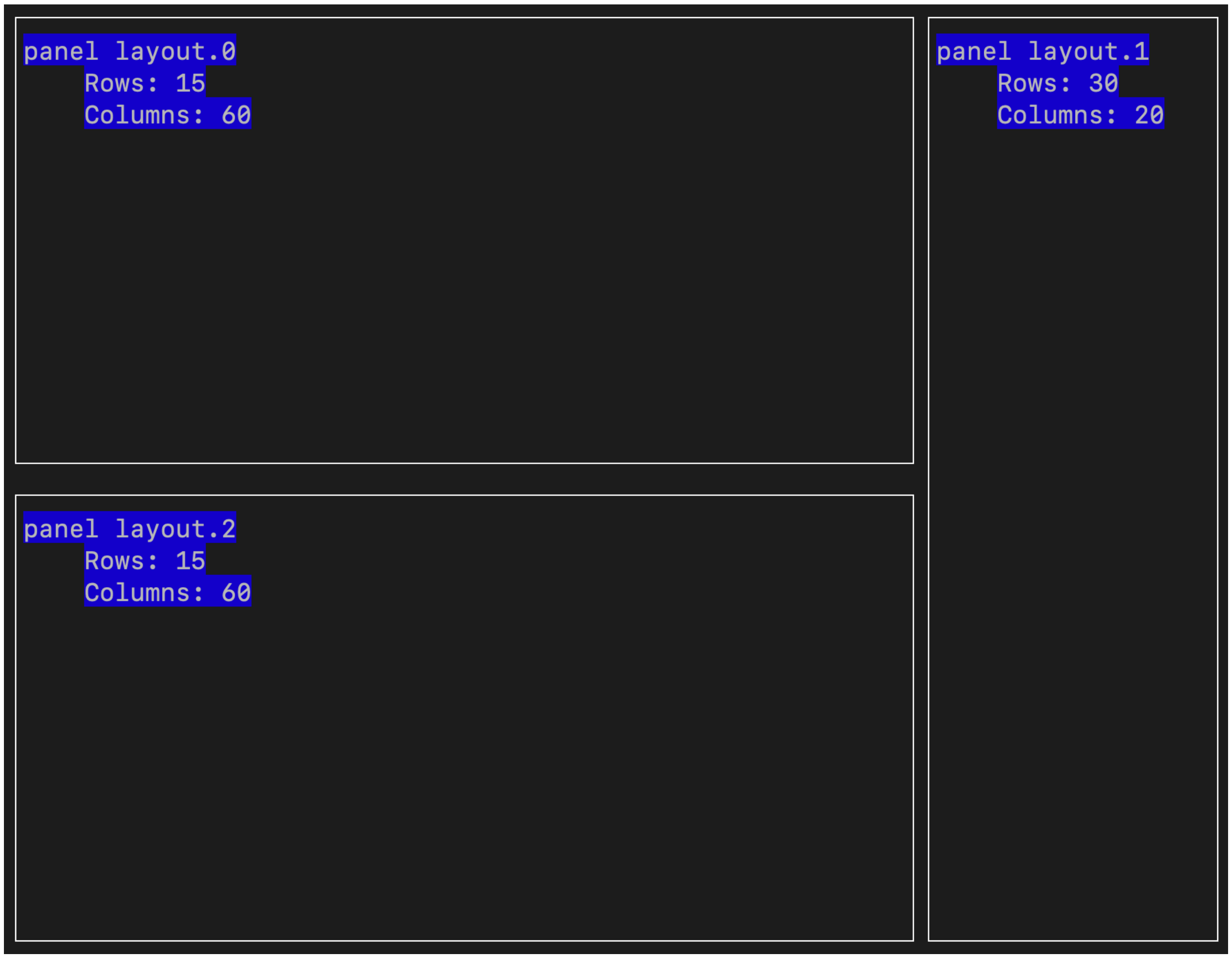
11.4 three_panels_horizontal



```
    Rows: 20                                Rows: 20
    Columns: 40                             Columns: 40

panel layout.2
    Rows: 10
    Columns: 80
```

11.5 three_panels_vertical



```
panel layout.0
    Rows: 15
    Columns: 60
```

```
panel layout.1
    Rows: 30
    Columns: 20
```

```
panel layout.2
    Rows: 15
    Columns: 60
```

CHAPTER

TWELVE

UTIL

The util examples are intended to provide interactive demos of key features of dashport. These concepts can be easily implemented into your own applications and scripts.

CHAPTER

THIRTEEN

VIEWS

These examples show some of the common views that Dashport programs can implement like scrolling panels, scrolling text on the main screen, support for multiple views, and implementing an interactive prompt.

13.1 Multiple Views

Multiple views can be accomplished by using keyboard controls and functions to switch between two screen states. The multiple_views.py example in the examples folder gives a good example of an app that puts all the output into functions that are called by the user typing the bound keys (in this case, “3” and “4”).