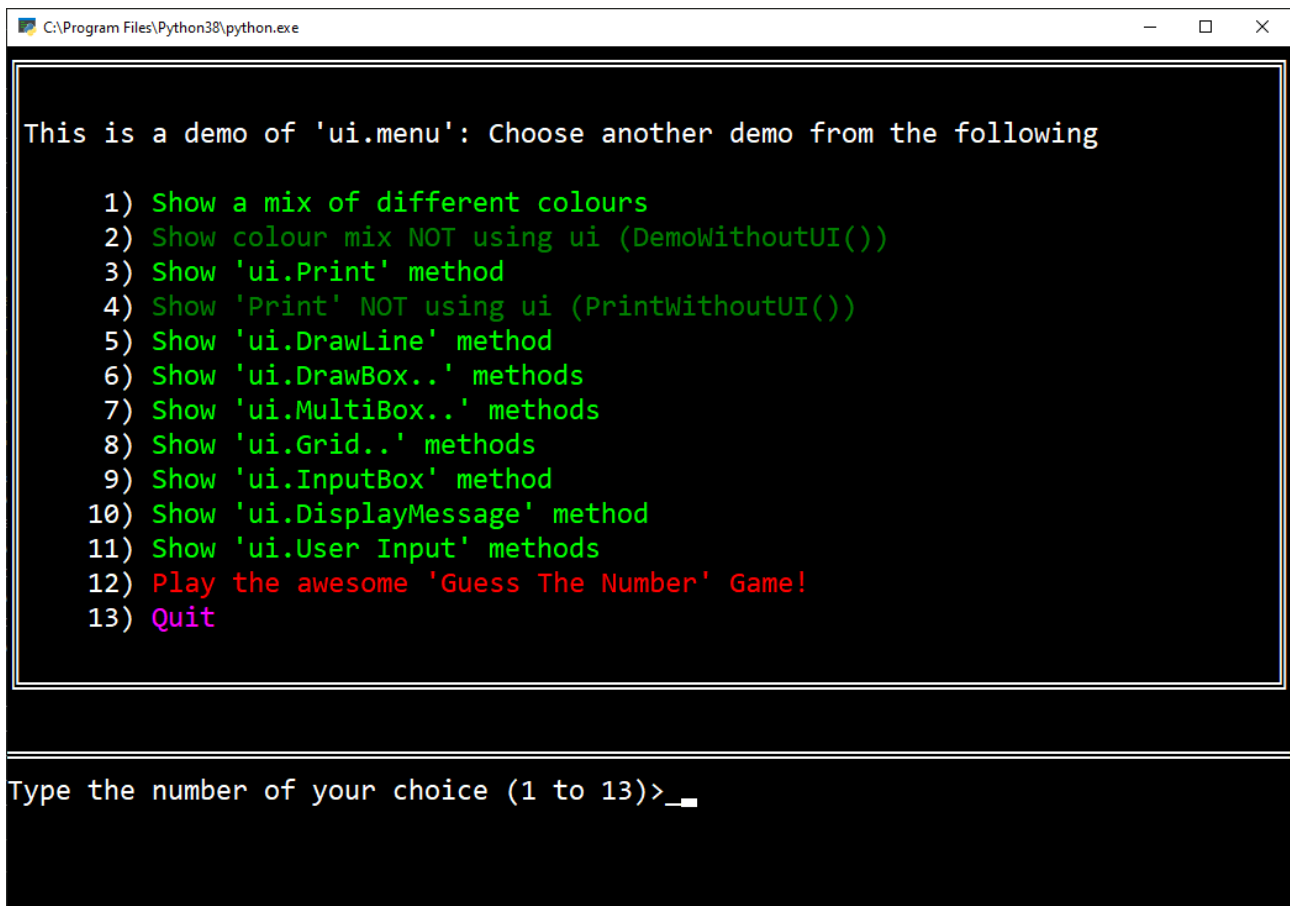


Colo(u)r Console Python



```
C:\Program Files\Python38\python.exe

This is a demo of 'ui.menu': Choose another demo from the following

1) Show a mix of different colours
2) Show colour mix NOT using ui (DemoWithoutUI())
3) Show 'ui.Print' method
4) Show 'Print' NOT using ui (PrintWithoutUI())
5) Show 'ui.DrawLine' method
6) Show 'ui.DrawBox..' methods
7) Show 'ui.MultiBox..' methods
8) Show 'ui.Grid..' methods
9) Show 'ui.InputBox' method
10) Show 'ui.DisplayMessage' method
11) Show 'ui.User Input' methods
12) Play the awesome 'Guess The Number' Game!
13) Quit

Type the number of your choice (1 to 13)>_
```

This tutorial shows you how to make the best of the standard console in Windows and Terminal in Linux by using coloured text and UTF8 characters to allow the drawing of lines and boxes to simulate some sort of UI. The spelling of colour and grey is correct UK English, but concessions have been made in code names to accommodate US spelling

The code used has been kept as simple as possible to allow translation to C#, Lua and Java to achieve the same effect, so there are probably better ways of doing it in Python that are not so easy to translate.

There is a Python module called Colorama which can be used (even on Windows) to output coloured text. This will NOT work in any IDE, the code has to be run in an external console or terminal. <https://pypi.org/project/colorama/>

If you are using:

1. Wing Personal IDE, set output to external console.
2. Visual Studio Code works ok, but the internal terminal cannot be re-sized programatically.
3. If you are using Idle or a text editor, be prepared to execute your program through Cmd / Powershell on Windows, or Terminal in Linux
4. If using a school computer, ask your IT manager to run .py scripts with Python.exe rather than opening with an editor when double-clicked. This is similar to running from a cmd prompt.

Installation:

The following code found in ui.py will auto-install Colorama. If you have Admin privileges in Windows, run colorconsole.py as administrator to ensure the library is installed for all users. On a school system it will install to %AppData% on the local workstation you are using, under your account name. This means anybody else using that workstation will have to run it themselves, and if you move to another pc, start all over again! It will NOT install to your networked user account unless you have a very clued up network administrator.

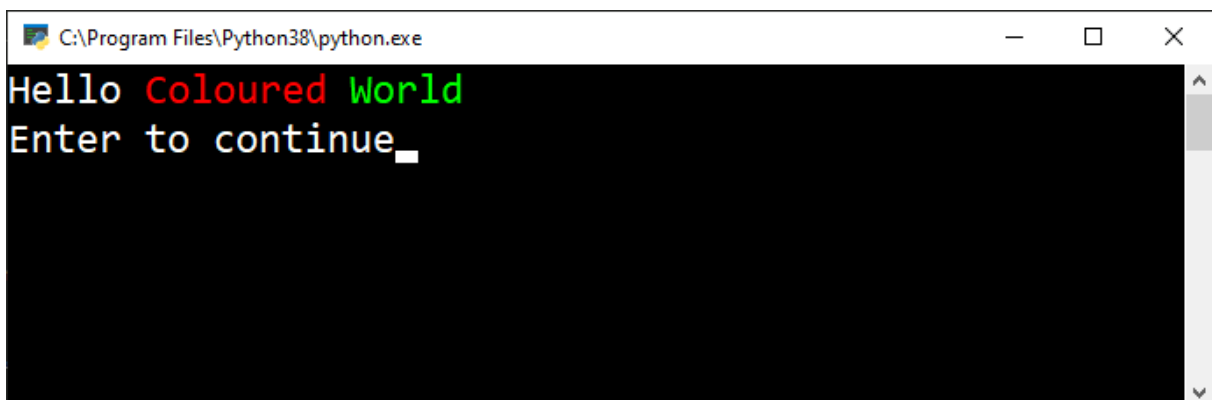
```
import os, time, sys, subprocess, math

try:
    import colorama
except ImportError:
    print("Attempting to import colorama")
    print("\nIf you get an error concerning colorama.init()")
    print("you may have to re-start this script before it works correctly")
    input("Press Enter to continue")
    subprocess.check_call([sys.executable, "-m", "pip", "install", 'colorama'])
finally:
    import colorama

colorama.init()
```

Colorama is designed to be used in an interpolated string:

```
print(f'Hello{colorama.Fore.RED + colorama.Style.BRIGHT} Coloured {colorama.Fore.GREEN + colorama.Style.BRIGHT}World{colorama.Style.RESET_ALL}')
```



This works ok, but is going to be extremely tedious to write an epic game following this technique.
How about:

```
print(f"Hello{RED} Coloured {GREEN}World{RESET}") # same output using constants
```

This example uses interpolated strings, starting with `f` so the values of the variables/constants inside `{}` are converted by the `ColorPrint` function to output colour.

To get this working a series of constants are required:

```
# colorama foreColour constants
BLACK   = colorama.Fore.BLACK + colorama.Style.BRIGHT      # '\x1b[30m'
WHITE   = colorama.Fore.LIGHTWHITE_EX                      # '\x1b[97m'
GREEN   = colorama.Fore.LIGHTGREEN_EX                      # '\x1b[92m'
RED     = colorama.Fore.LIGHTRED_EX                        # '\x1b[91m'
#backColour constants:
BLACKBG = colorama.Back.BLACK + colorama.Style.BRIGHT      # '\x1b[40m'
WHITEBG = colorama.Back.LIGHTWHITE_EX                      # '\x1b[107m'
RESET   = colorama.Style.RESET_ALL                         # '\x1b[0m'
```

These constants are used in the print statement above

Another usefull construct is to use an embedded colour code in the text, which is easier when importing text files:

```
colorprint("Hello~red~ Coloured ~green~World") # same output using embedded colour tags
```

This requires some additional code:

```
sep = '~' # default separator
black = sep + 'black' + sep # ~black~
white = sep + 'white' + sep # ~white~
red = sep + 'red' + sep # ~red~
green = sep + 'green' + sep # ~green~

blackbg = sep + 'blackbg' + sep # ~blackbg~
whitebg = sep + 'whitebg' + sep # ~whitebg~
```

This gives the required constants, but there is a need for interpreting them in code. To help with that a dictionary of colours would be helpful:

```
colors = {}
colors.update({"black":BLACK})
colors.update({"white": WHITE})
colors.update({"green": GREEN})
colors.update({"red": RED})
colors.update({"blackbg": BLACKBG})
colors.update({"whitebg": WHITEBG})
colors.update({f"{sep}black{sep}":BLACK})
colors.update({f"{sep}white{sep}":WHITE})
colors.update({f"{sep}red{sep}":RED})
colors.update({f"{sep}green{sep}":GREEN})
```

```

colors.update({f"{sep}blackbg{sep}":BLACKBG})
colors.update({f"{sep}whitebg{sep}":WHITEBG})
colors.update({f"{sep}reset{sep}":RESET})

```

The supplied text needs to be broken up depending on the presence of the separator tag ~ and then each part can be assessed as to whether it is a colour instruction, or simply part of the text:

```

def colorprint(value, newline = True, reset = True):
    num_lines = 0
    num_lines = num_lines + value.count('\n') #embedded \n newline chars
    lineparts = split_clean(value, sep)
    for line in lineparts:
        if line in colors: #red, blackbg
            if is_coloured:
                if line == "reset":
                    print(RESET, end='')
                else:
                    print(colors[line], end = '')
            else:
                print(line, end='')
    if is_coloured:
        if newline and reset:
            print(f'{RESET}')
            num_lines = num_lines + 1
        elif reset:
            print(f'{RESET}', end='')
    else:
        print()
        num_lines = num_lines + 1

    return num_lines

```

How does this work?

A series of variables has been started, consisting of a colour name assigned a value comprising a tag character (default ~) surrounding the colour name eg.

```
white = "~white~"
```

This will need to be done for all 6 colours: red, blue, green, cyan, magenta, yellow plus black and white and their dark equivalents. This combination creates 2 grey shades from the black/white/dark/bright possibilities to give Gray and DarkGray, a total of 16.

With the addition of the letters BG added to the name, 16 background colours are created, eg:

```
blackbg = "~blackbg~"
```

The dictionary colors has been partially populated with the same set of colours as keys, with equivalent ANSI codes as values. The remaining foreground and background colours need to be added.

These are used as in-line text tags eg

```
colorprint("~red~This is red text")
```

The colour tags can be upper/lower or mixed case eg

```
~RED~ = ~red~ = ~rEd~
```

The reason for using the variable 'sep' is to allow users to substitute the ~ character for something else if required. As sep is a string, using sep[0] will provide the char equivalent for functions requiring a (char), such as .Split()

colorprint()

This takes 1 required parameter 'text' and 2 optional 'newline' and 'reset'.

The text is split into a list using the current separator character (default ~)

The items of this array are iterated with a foreach loop, with appropriate action taken in the conditional block:

IF if an item is found in the color dictionary keys, it is then further checked to see if it contains "BG" indicating it is a background colour, or if it contains "RESET". The appropriate ANSI code is sent to the console/terminal.

ELSE the item is output using print(text, end='') which does not force a newline.

Once the list iteration is completed the colours are RESET by default, and a newline character output also by default.

The reason for making these optional is to give greater flexibility. For example:

```
colorprint("~green~Type your name>_~red~", false, false);
```

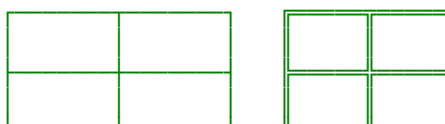
This will write "Type your name>_" in green and leave the cursor on the same line. The user's input will appear in red as they type because the colours were NOT reset.

Lines and Boxes

The use of lines and boxes can make a console based UI look very professional. There are a set of UTF8 characters for this purpose eg:

```
┌ = ┐ └ ┘ ┌ ┘ ┌ ┘ ┌ ┘
```

Single and double line boxes:



These characters are difficult to enter via the keyboard, so are best copy/pasted from specialised websites such as https://www.w3schools.com/charsets/ref_utf_box.asp

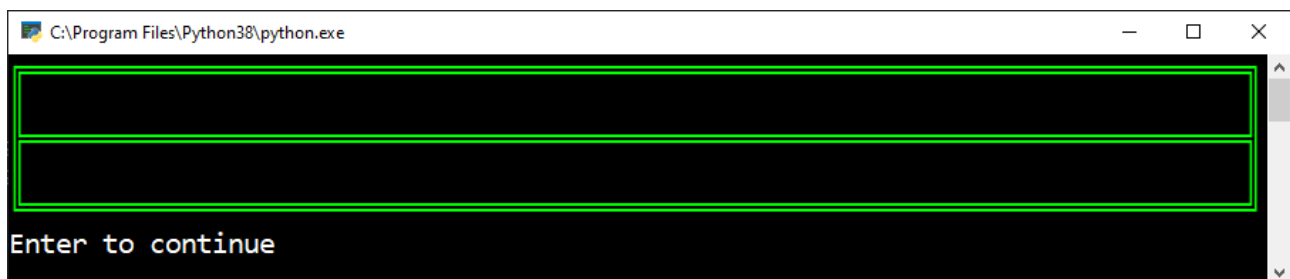
If they are kept in lists and/or dictionaries they are easier to manipulate:

```
# much of the code here copy/pasted from C# so variable names are camelCase not snake_case
sSymbolsTop    = [ "┌", "─", "┐", "┴" ]
sSymbolsBottom = [ "└", "─", "┘", "┴" ]
sSymbolsBody   = [ "│", " ", "│", "│" ]
sSymbolsMid    = [ "├", "─", "┤", "┴" ]

dSymbolsTop    = [ "┌", "=", "┐", "┴" ]
dSymbolsBottom = [ "└", "=", "┘", "┴" ]
dSymbolsBody   = [ "│", " ", "│", "│" ]
dSymbolsMid    = [ "├", "=", "┤", "┴" ]
```

The following code using the ColorPrint() functions draws a double box:

```
colorprint(f"{GREEN}{dSymbolsTop[0]}{''.ljust(78, dSymbolsTop[1])}{dSymbolsTop[2]}")
colorprint(f"{GREEN}{dSymbolsBody[0]}{''.ljust(78, dSymbolsBody[1])}{dSymbolsBody[2]}")
colorprint(f"{GREEN}{dSymbolsMid[0]}{''.ljust(78, dSymbolsMid[1])}{dSymbolsMid[2]}")
colorprint(f"{GREEN}{dSymbolsBody[0]}{''.ljust(78, dSymbolsBody[1])}{dSymbolsBody[2]}")
colorprint(f"{GREEN}{dSymbolsBottom[0]}{''.ljust(78, dSymbolsBottom[1])}{dSymbolsBottom[2]}")
```



UI Library code

With these basic concepts in place, the UI library has been written to give developers a selection of methods as seen in the screenshot at the beginning of this document.

The UI library is a Python code module, which equates to a static class consisting of 1122 lines of code with all functions and procedures in alphabetical order. As Python cannot distinguish between Public and Private functions all can be called from other classes, notably colorconsole.py which hosts main()

C# static classes can have the equivalent of a constructor, which runs once only when the class is first used. Python modules do not have this, so a function called initialise() is called directly when the module loads. This is used to initialise variables, such as populating the dictionaries:

```
def initialise():
    ''' runs on load to create lists/dictionaries of colour variables '''
    global window_width, window_height, colors, is_console
    try:
        window_width = os.get_terminal_size().columns # this fails if NOT in a console
        window_height = os.get_terminal_size().lines # this fails if NOT in a console
    except:
        window_width = 80
        window_height = 25
        is_console = False

    colors.clear()
    colors.update({"black":BLACK})
    colors.update({"grey": GREY})
    colors.update({"gray": GRAY})
    colors.update({"dgrey": DGREY})
```

From now on all functions and procedures will be referred to as 'functions', Whether or not they return a value is irrelevant in a general description.

The ideal way of using this library is to use it for any user input from the keyboard, and to display any output. Most program logic should be done in other classes.

Starting in colorconsole.main():

```
ui.set_console(80, 25, "white", "black", False) #set to default size 80 x 25, white text on black bg
```

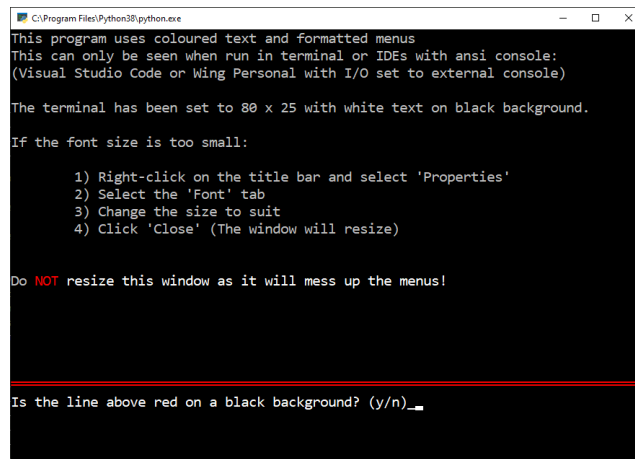
This sets the console width to 80 characters and height to 25 rows, with white foreground on a black background. The screen is cleared within the function, and the public global variables `window_width` and `window_height` are set:

```
def set_console(cols, rows, fore_color, back_color, initialise = False):
    global window_width, window_height, is_coloured
    windows = {
        'black':'0', 'red':'4', 'green':'2', 'yellow':'6',
        'blue':'1', 'magenta':'5', 'cyan':'3', 'white':'7',
        'grey' : '8', 'lightblue' : '9', 'lightgreen' : 'A', 'lightaqua' : 'B',
        'lightred' : 'C', 'lightpurple' : 'D', 'lightyellow' : 'E', 'brightwhite' : 'F'
    }
    linux_fg = {
        'black':'30', 'red':'31', 'green':'32', 'yellow':'33',
        'blue':'34', 'magenta':'35', 'cyan':'36', 'white':'37'
    }
    linux_bg = {
        'black':'40', 'red':'41', 'green':'42', 'yellow':'43',
        'blue':'44', 'magenta':'45', 'cyan':'46', 'white':'47'
    }
    # set the shared values for size and colours
    window_width = cols          # set global variable window_width
    window_height = rows         # set global variable window_height
    resize(window_width, window_height) # resize terminal using global window_height, window_width
    # set colours to global fore, back
    if is_console: #defined on module load
        if os.name == 'nt': # Windows
            os.system(f'color {windows[back_color]}{windows[fore_color]}') # white on black preferred
        else: # Linux / Mac
            #send ansi codes directly to terminal
            fc = f'''\e[{linux_fg[fore_color]}m'''
            bc = f'''\e[1{linux_bg[back_color]}m''' # 1 = set bold. 22 = set normal
            os.system("echo -e " + bc)
            os.system("echo -e " + fc)
            #reset()
        clear()

    if initialise:
        display_setup(fore_color, back_color)
    else:
        is_coloured = True
```

The dictionaries hold values for foreground and background colours for both Windows and Linux systems, and these are used to set the console colours. In Linux the same ANSI codes are used, but within an `os.execute` call.

There is an optional `ui.DisplaySetup()` call, which allows the user to confirm if they are looking at a coloured display, and sets the `isColoured` flag accordingly. This will run if the `initialise` flag is set to true when is called:



A while loop runs the menu system shown in the screenshot on page 1.

A title for the menu is stored in string title.

A List of the required menu items is created using the embedded colour tags method:

```
title = "This is a demo of 'ui.menu': Choose another demo from the following"
options = []
options.append( "~green~Show a mix of different colours")
options.append( "~dgreen~Show colour mix NOT using ui (DemoWithoutUI())")
options.append( "~green~Show 'ui.Print' method")
options.append( "~dgreen~Show 'Print' NOT using ui (PrintWithoutUI())")
options.append( "~green~Show 'ui.DrawLine' method")
...

choice = ui.menu("d", title, ">_", options)
```

UI.Menu()

```
choice = ui.menu("d", title, ">_", options)
```

"d" indicates use a double line box construction.

">_" is the prompt characters where the user is expected to type their input.

The returned int value is the equivalent index of the option chosen from the list. The list display and the return value is 0 based, and this is adjusted for display purposes to start the list at 1.

```
def menu(style, title, prompt_end, text_lines, fore_color = 'white', back_color = blackbg,
align = "left", width = 0):
```

The remaining parameters are optional.

fore_color and back_color only affect the boxes that make up the menu.

align is not currently used, but was included for further development

width can be set if less than Console.Width is required. (0 = Console.width)

The boxes and their contents are drawn using:

```
draw_box_outline()
draw_box_body()
```

Empty lines are added to fill the first 20 rows:

```
add_lines(5, num_lines)
```

A white full width double line is drawn:

```
draw_line("d", white, blackbg)
```

If the user does not respond correctly, the `get_integer()` method returns the validity as false, and a message is displayed for 2 seconds, then the whole screen is cleared and re-drawn. It is this ability to continuously re-draw the whole console that gives the best approximation of a GUI in appearance.

Most methods are self-documenting.

Use the examples in `colorconsole.py` to give you ideas for your own user interface.