Quo Documentation

Release 2023.x

Gerrishon Sirere

Apr 28, 2023

TUTORIALS EXPLANATIONS

Version	2023.x
Web	Documentation
Download	Downloads
Source	Github

`Forever Scalable`

Quo is a toolkit for writing Command-Line Interface(CLI) app lications and a TUI (Text User Interface) framework for Python. Quo is making headway towards composing speedy and orderly CLI and TUI applications while forestalling any disappointments brought about by the failure to execute a python application.

Simple to code, easy to learn, and does not come with needless baggage.

Quo requires Python `3.8` or later.

Features

- [x] Support for ANSI, RGB and Hex color models
- [x] Support for tabular presentation of data
- [x] Intuitive progressbars
- [x] Code completions
- [x] Parsing and nesting of commands
- [x] Customizable Text User Interface *(TUI)* dialogs
- [x] Automatic help page generation
- [x] Syntax highlighting
- [x] Autosuggestions
- [x] Key Binders

Quo is **simple** If you know Python you can easily use Quo and it can integrate with just about anything.

CHAPTER

ONE

INTRODUCTION

Quo is a Python based toolkit for writing Command-Line Interface(CLI) applications. Quo is making headway towards composing speedy and orderly CLI applications while forestalling any disappointments brought about by the failure to execute a CLI API. Simple to code, easy to learn, and does not come with needless baggage.

1.1 Requirements

Quo works flawlessly with Linux, OSX and Windows.

Quo requires Python `3.8` or later

1.2 Installation

You can install Quo from PyPi with *pip*

```
pip install -U quo
```

1.3 Quick Start

```
from quo import echo

echo(f"Hello World!", fg="red", italic=True, bold=True)
```

This will print `Hello World!` plus a new line to the terminal. Unlike the builtin print function, echo

function has improved support for handling formatted text.

CHAPTER

TWO

PRINTING (AND USING) FORMATTED TEXT

2.1 Formatted text

There are several ways to display colors:

- By creating a `quo.echo()` function.
- By creating a `quo.print()` function.

An instance of any of these three kinds of objects is called "formated text".

2.2 echo

`quo.echo()` prints a message plus a newline to the given file or stdout. On first sight, this looks like the print function, but it has improved support for handling Unicode, binary data and formatted text. It will emit newline by default, which cab be suppressed by passing :param:`nl=False`

» List of supported ANSI colors

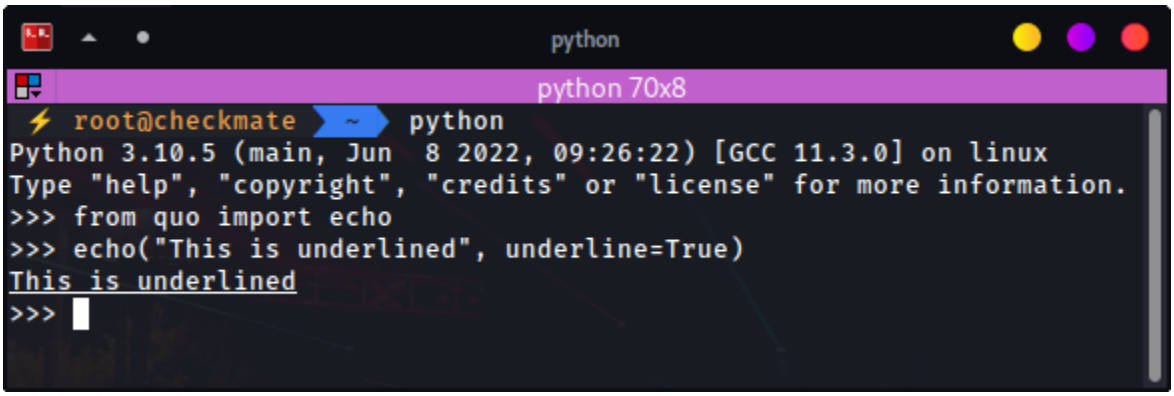
Parameters

- `text` – the string to style with ansi or rgb color codes.
- `fg` – if provided this will become the foreground color.
- `bg` – if provided this will become the background color.
- `bold` – if provided this will enable or disable bold mode.
- `dim` – if provided this will enable or disable dim mode.
- `nl` - if provided this will print a new line.
- `ul or underline` – if provided this will enable or disable underline.
- `italic` - if provided this will print data in italic.
- `blink` – if provided this will enable or disable blinking.
- `strike` -if provided this will print a strikethrough text.
- `hidden` - if privided this will prevent the input from getting printed.
- `reverse` – if provided this will enable or disable inverse rendering (foreground becomes background and the other way round).
- `reset` – by default a reset-all code is added at the end of the string which means that styles do not carry over. This can be disabled to compose styles.

```
from quo import echo

echo("Hello, world!", nl=False)
```

```
from quo import echo

echo("This is bold", bold=True)
echo("This is italic", italic=True)


# Colors from the ANSI palette

echo("This is red", fg="red")
echo("This is green", fg="green")
```

```
from quo import echo
echo("This is underlined", underline=True)
```




```
from quo import echo

echo(b'\xe2\x98\x83')
```

2.2.1 Printing to Standard error using echo

You can easily print to standard error by passing :param:`err=True`

```
from quo import echo

echo('Hello World!', err=True)
```

2.3 print

Quo ships with a `print()` function that's meant to be (as much as possible) compatible with the built-in print function, and `quo.echo()`. It also supports color and formatting just like `quo.echo()`. `print()` can be used to indicate that a string contains HTML-like formatting. It recognizes the basic tags for bold, italic and underline: <b>, <i> and <u>. *Changed since v2022.3.5*

On Linux systems, this will output VT100 escape sequences, while on Windows it will use Win32 API calls or VT100 sequences, depending on what is available.

Parameters

- `values` - Any kind of printable object, or formatted string.
- `end` - String appended after the last value, default a newline.(the default is a new line).
- `fmt` *bool* - Default is *False*, if *True*, you will be able to utilize an instance of `quo.text.FormattedText`. *Added on v2022.4*
- `color_depth` - Instance of `quo.color.ColorDepth` . This specifies the number of bits used for each color component i.e: *one_bit(2 colors black ad white), four_bit(ANSI 16 colors), eight_bit(256 colors) or twenty_four_bit(24 bit True color).* The default color depth is `eight_bit`.
- `sep` - String inserted between values, default a space.
- `style` - `quo.style.Style` instance for the color scheme.

```
from quo import print

print('<b>This is bold</b>')
print('<i>This is italic</i>')
print('<u>This is underlined</u>')
```

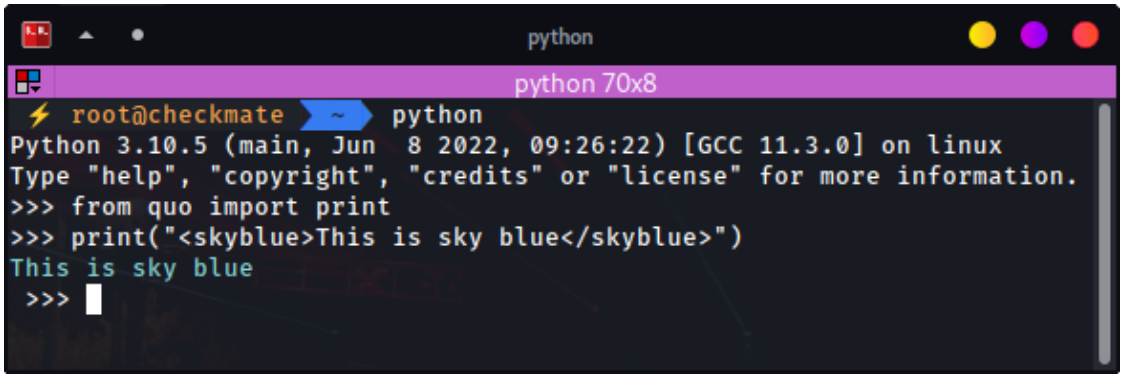
- Colors from the ANSI palette.

```
from quo import print

print('<red>This is red</red>')
print('<green>This is green</green>')
```

» List of supported ANSI colors Read more about styling.

- Named colors (256 color palette, or true color).

```
from quo import print

print('<skyblue>This is sky blue</skyblue>')
```



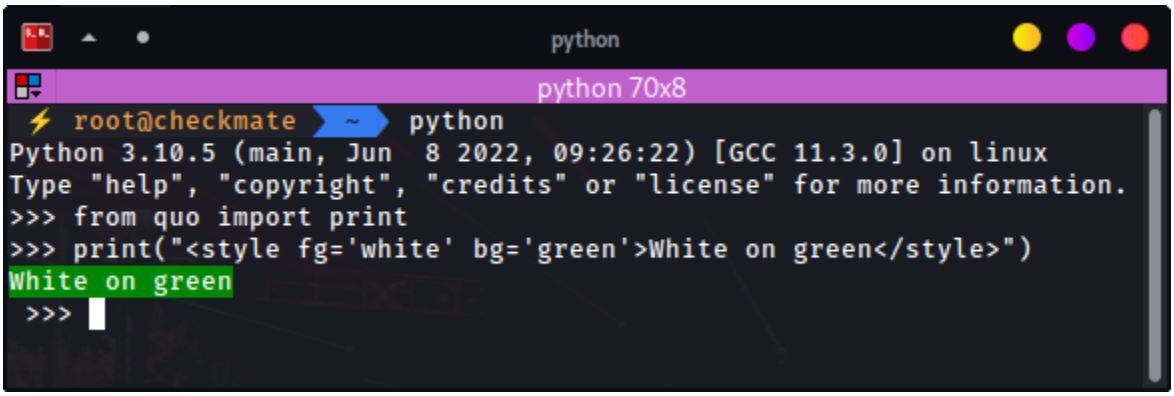
More examples

```
from quo import print
print('<seagreen>This is sea green</seagreen>')
print('<violet>This is violet</violet>')
```

» List of supported Named colors

Both foreground and background colors can also be specified setting the *fg* and *bg* attributes of any Text tag:

```
from quo import print

print('<style fg="white" bg="green">White on green</style>')
```



Note: » style tag in the example above can be anything i.e: <abc…

Underneath, all Text tags are mapped to classes from a stylesheet, so you can assign a style for a custom tag.

```
from quo import print
from quo.style import Style

style = Style.add({
    'aaa': 'fg:red',
    'bbb': 'fg:blue italic'
    })

print('<aaa>Hello</aaa> <bbb>world</bbb>!', style=style)
```

Note: This page is also useful if you'd like to learn how to use formatting in other places, like in a prompt or a toolbar.

» Check out more examples here

CHAPTER

THREE

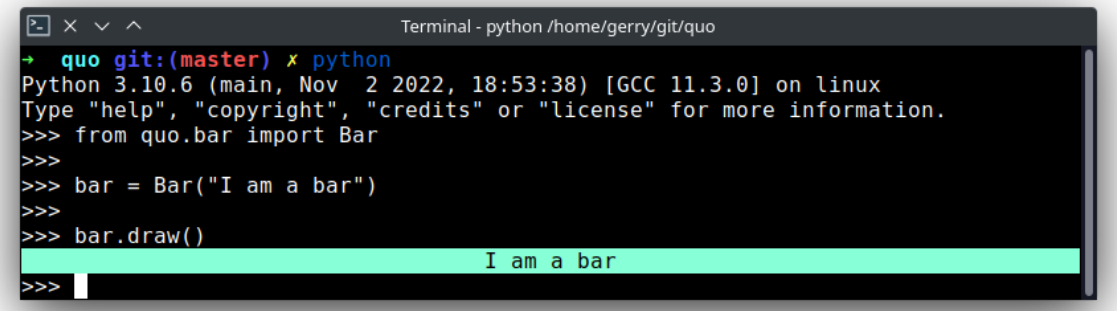
BARS

The `Bar` can be used to draw a horizontal bar with an optional title, which is a good way of dividing your terminal output in to sections. *Added on v2023.3*

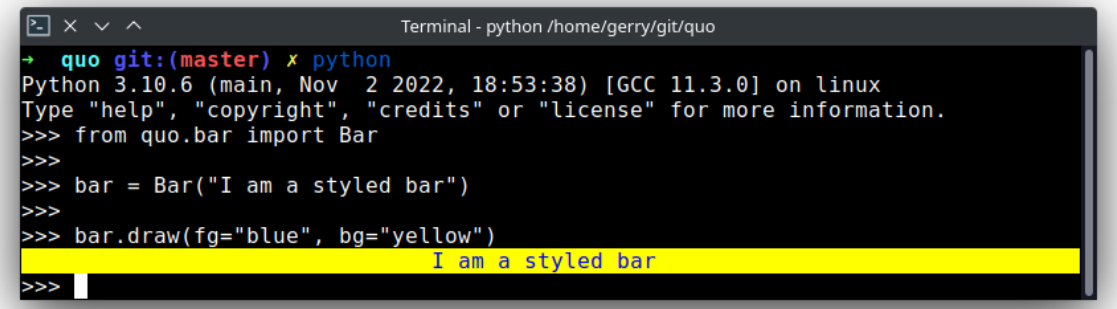
Parameters

- `message` Optional[*(str)*] – Message print on the terminal.
- `align` Optional[*(str)*] - Postion of the message to be printed. Default is `center` other options are `left` and `right`.
- `fg` Optional[*(str)*] - Foreground color to be applied.
- `bg` Optional[*(str)*] - Background color to be applied.

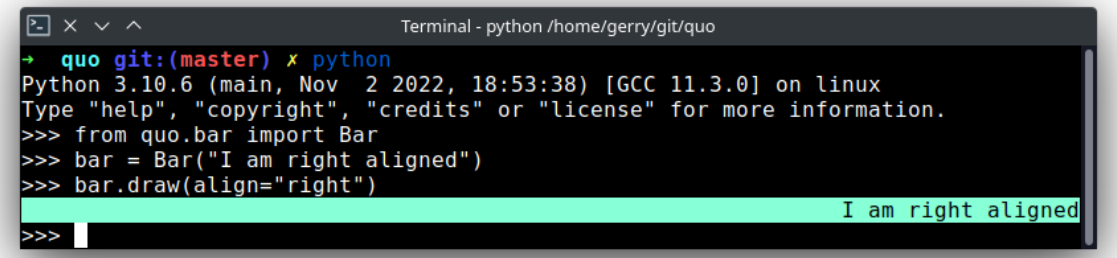
```
from quo.bar import Bar

bar = Bar("I am a bar")

bar.draw()
```




```
from quo.bar import Bar

bar = Bar("I am a styled bar")

bar.draw(fg="blue", bg="yellow")
```




```python
from quo.bar import Bar

bar = Bar("I am right aligned")

bar.draw(align="right")
```



CHAPTER

FOUR

CONSOLE API

For complete control over terminal formatting, Quo offers a `Console` class. Most applications will require a single Console instance, so you may want to create one at the module level or as an attribute of your top-level object. For example, you could add a file called "console.py" to your project:

```
from quo.console import Console
console = Console()
```

Then you can import the console from anywhere in your project like this:

```
from my_file.console import console
```

4.1 Attributes

The console will auto-detect a number of properties required when rendering.

4.2 Bell

For making a beep sound multiple times

Parameters `instance` *(int)* - The number of times to ring the bell

```
from quo.console import Console

console = Console()

console.bell(3)
```

4.3 Encoding

`quo.console.Console.encoding` will detect the default encoding of the Terminal (typically "utf-8")

```
from quo.console import Console

console = Console()

console.encoding()
```

4.4 File Opening

The logic for opening files from the `quo.types.File` type is exposed through the `quo.Console.openfile()` function. It can intelligently open stdin/stdout as well as any other file. **Parameters**

- `filename` *(str)* - The name of the file to open (or `'-'` for stdin/stdout).
- `mode` *(str)* - The mode in which to open the file.
- `encoding` Optional - The encoding to use.
- `errors` *(str)* - The error handling for this file.
- `lazy` *(bool)* - Can be flipped to true to open the file lazily.
- `atomic` *(bool)* -in atomic mode writes go into a temporary file and it's moved on close.

```
from quo.console import Console

console = Console()

stdout = console.openfile('-', 'w')
test_file = console.openfile('test.txt', 'w')
```

If stdin or stdout are returned, the return value is wrapped in a special file where the context manager will prevent the closing of the file. This makes the handling of standard streams transparent and you can always use it like this:

```
from quo.console import Console

console = Console()

with console.openfile(filename, 'w') as f:
f.write('Hello World!\n')
```

4.5 Launching Applications

Quo supports launching applications through `quo.Console.launch()`. This can be used to open the default application associated with a URL or filetype.

This can be used to launch web browsers or picture viewers, for instan ce. In addition to this, it can also launch the file manager and automatically select the provided file.

Parameters

- `url` *(str)* – URL or filename of the thing to launch.
- `wait` *(bool)* – Wait for the program to exit before returning. This only works if the launched program blocks. In particular, xdg- open on Linux does not block.
- `locate` Optional *(bool)* – if this is set to True then instead of launching the application associated with the URL it will attempt to launch a file manager with the file located. This might have weird effects if the URL does not point to the filesystem.

```
from quo.console import Console

console = Console()
```

(continues on next page)

(continued from previous page)

```
console.launch("https://quo.rtfd.io/")
```

```
from quo.console import Console

console = Console()

console.launch("/home/downloads/file.txt", locate=True)
```

4.6 Launching Text Editors

Quo supports launching editors automatically through `quo.Console.edit()`. This is very useful for asking users for multi-line input. It will automatically open the user's defined editor or fall back to sensible default. If the user closes the editor without saving, the return value will be `None`, otherwise the entered text.

Parameters

- `text` *(str)* - The text to edit.
- `editor` Optional - The editor to use. Defaults to automatic detection.
- `env` *(str)* - The environment variables to forward to the editor.
- `require_save` *(bool)* - If this is true, then not saving in the editor will make the return value become *None*.
- `extension` *(str)* - The extension to tell the editor about. This defaults to *.txt* but changing this might change syntax highlighting.
- `filename` *(str)* - If provided it will edit this file instead of the provided text contents. It will not use a temporary file as an indirection in that case.

Note: For Windows: to simplify cross-platform usage, the newlines are automatically converted from POSIX to Windows and vice versa. As such, the message here will have `\n` as newline markers

---

```
from quo.console import Console

console = Console()

def get_commit_message():
    MARKER = '# Everything below is ignored\n'
    message = console.edit('\n\n' + MARKER)
    if message is not None:
        return message.split(MARKER, 1)[0].rstrip('\n')
```

Alternatively, the function can also be used to launch editors for files by a specific filename. In this case, the return value is always *None*.

```
from quo.console import Console

console = Console()
console.edit(filename='/etc/passwd')
```

4.7 Pager

`quo.console.Console.pager()` takes a text and shows it via an environment specific pager on stdout. *Added on v2022.4*

Parameters

- `text` - The text to page, or alternatively, a generator emitting the text to page.
- `color` - controls if the pager supports ANSI colors or not.

4.8 Spin

This creates a context manager that is used to display a spinner on stdout as long as the context has not exited. *Added on v2022.5*

```
import time

from quo.console import Console

console = Console()

with console.spin():
          time.sleep(3)
          print("Hello, World")
```

4.9 Terminal size

Function `quo.console.Console.size` returns the current size of the terminal as tuple in the form `(width, height)` in columns and rows.

```
from quo.console import Console

console = Console()
console.size()
```

» Check out more examples here

CHAPTER

FIVE

DIALOGS

Quo ships with a high level API for displaying dialog boxes to the user for informational purposes, or get input from the user.

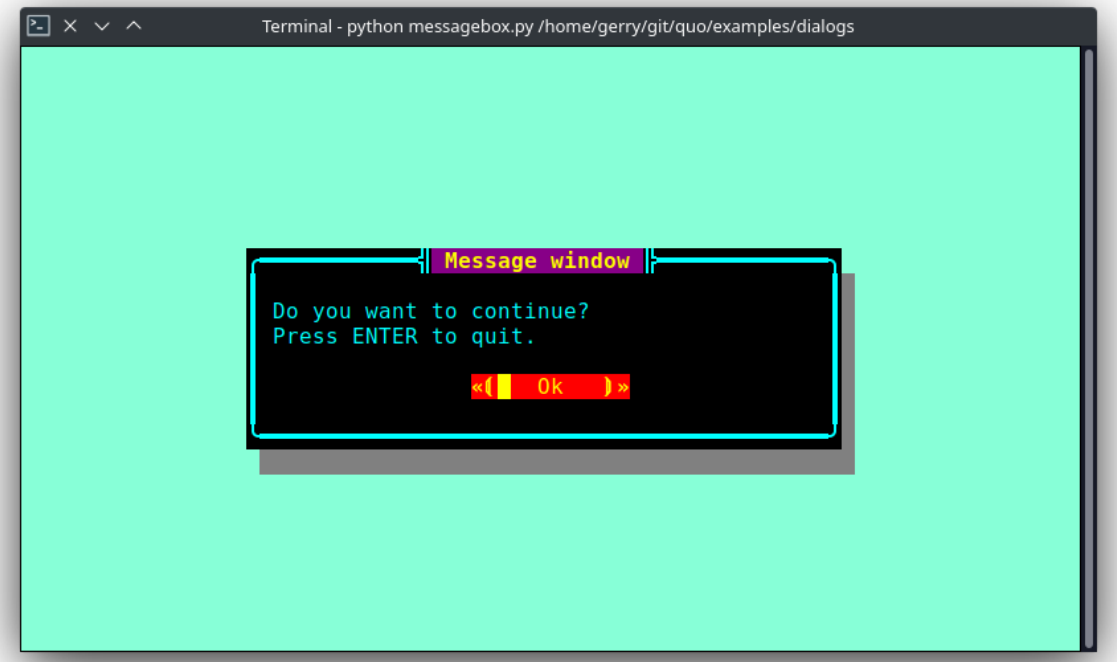
All dialogs can be passed `bg=False` option to turn off the background. *Added on v2022.4*

Deprecated :meth:`.run` on v2022.3.2

5.1 Message Box

Use the `MessageBox()` function to display a simple message box. For instance:

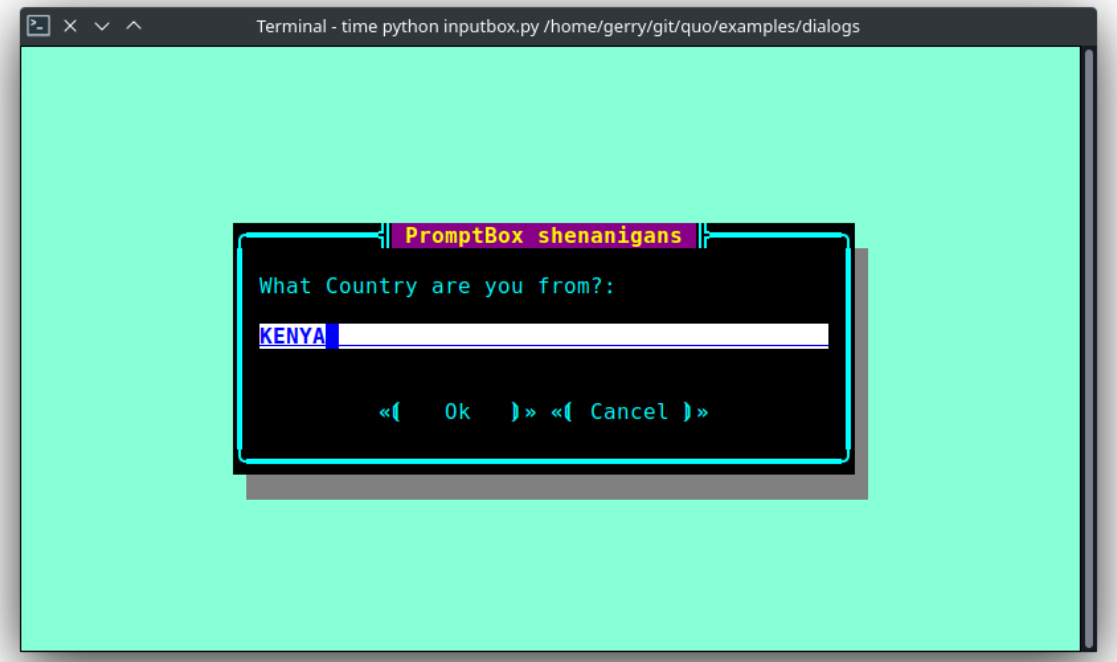
```
from quo.dialog import MessageBox

MessageBox(
    title='Message window',
    text='Do you want to continue?\nPress ENTER to quit.')
```



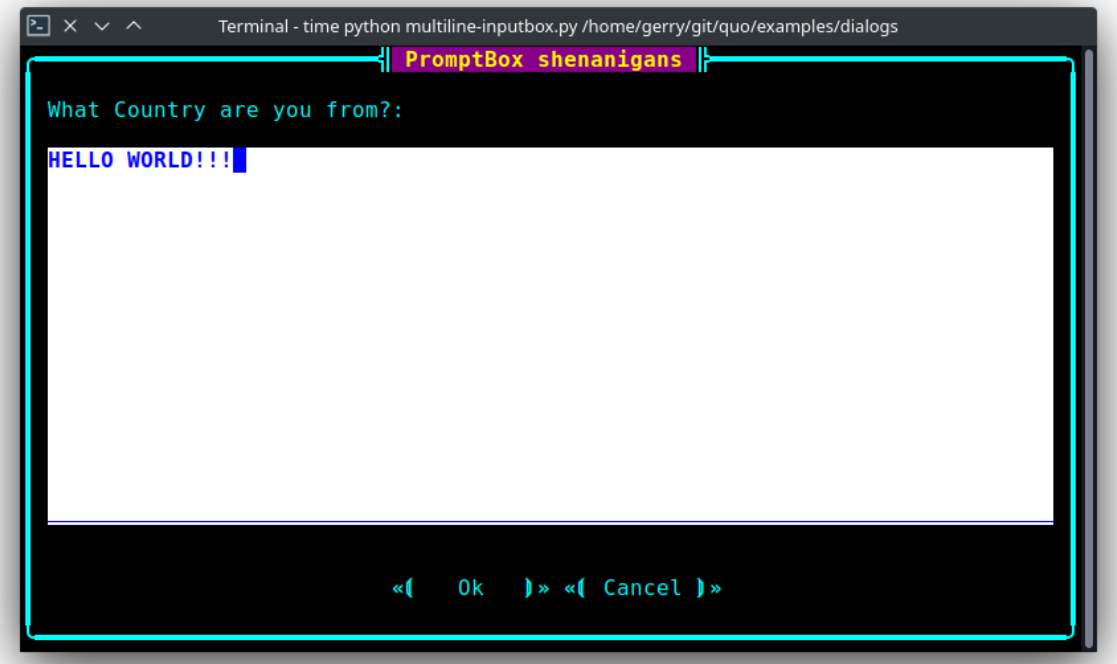
5.2 Input Box

The `InputBox()` function can display an input box. It will return the user input as a string.

```
from quo.dialog import InputBox
InputBox(
    title='PromptBox Shenanigans',
    text='What Country are you from?:')
```



The `multiline=True` parameter can be passed to turn this into a multiline Input box



The `hide=True` option can be passed to the `InputBox()` function to turn this into a password input box.

5.3 Confirm Box

The `ConfirmBox()` function displays a yes/no confirmation dialog. It will return a boolean according to the selection.

```
from quo.dialog import ConfirmBox

ConfirmBox(
    title='Yes/No example',
    text='Do you want to confirm?')
```

5.4 Choice Box

The `ChoiceBox()` function displays a dialog with choices offered as buttons. Buttons are indicated as a list of tuples, each providing the label (first) and return value if clicked (second).

```
from quo.dialog import ChoiceBox

ChoiceBox(
    title='Button dialog example',
    text='Do you want to confirm?',
    buttons=[
        ('Yes', True),
        ('No', False),
        ('Maybe...', None)
    ])
```

5.5 Radiolist Box

The `RadiolistBox()` function displays a dialog with choices offered as a radio list. The values are provided as a list of tuples, each providing the return value (first element) and the displayed value (second element).

```
from quo.dialog import RadiolistBox

RadiolistBox(
    title="RadioList dialog example",
    text="Which breakfast would you like ?",
    values=[
        ("breakfast1", "Eggs and beacon"),
        ("breakfast2", "French breakfast"),
        ("breakfast3", "Equestrian breakfast")
    ])
```

5.6 Check Box

The `CheckBox()` has the same usage and purpose than the Radiolist dialog, but allows several values to be selected and therefore returned.

```
from quo.dialog import CheckBox

CheckBox(
    title="CheckboxList dialog",
```

(continues on next page)

(continued from previous page)

```
    text="What would you like in your breakfast ?",
    values=[
        ("eggs", "Eggs"),
        ("bacon", "Bacon"),
        ("croissants", "20 Croissants"),
        ("daily", "The breakfast of the day")
    ]
)
```

5.7 Styling of dialogs

A custom `Style` instance can be passed to alldialogs to override the default style. Also, text can be styled by passing an `Text` object.

```
from quo.dialog import MessageBox
from quo.style import Style
from quo.text import Text

style = Style.add({
    'dialog': 'bg:aquamarine',
    'dialog.body': 'bg:black fg:green',
    'dialog shadow': 'bg:yellow' })

MessageBox(
    title=Text('<style bg="blue" fg="white">Styled</style> '
    '<style fg="red">dialog</style> window'),
    text='Do you want to continue?\nPress ENTER to quit.',
    style=style)
```

5.8 Styling reference sheet

In reality, the dialog commands presented above build a full-screen frame by using a list of components. The two tables below allow you to get the classnames available for each dialog therefore you will be able to provide a custom style for every element that is displayed, using the method provided above.

Note: All the dialogs use the `Dialog` component, therefore it isn't specified explicitly below.

Shortcut	Components used
quo.dialog.ConfirmationBox	• Label • Button (x2)
quo.dialog.ChoiceBox	• Label • Button
quo.dialog.PromptBox	• TextArea • Button (x2)
quo.dialog.MessageBox	• Label • Button
quo.dialog.RadiolistBox	• Label • RadioList • Button (x2)
quo.dialog.CheckBox	• Label • CheckboxList • Button (x2)
quo.dialog.ProgressBox	• Label • TextArea (locked) • ProgressBar

Components	Available classnames
Dialog	• `dialog` • `dialog.body`
TextArea	• `text-area` • `text-area.prompt`
Label	• `label`
Button	• `button` • `button.focused` • `button.arrow` • `button.text`
Frame	• `frame` • `frame.border` • `frame.label`
Shadow	• `shadow`
RadioList	• `radio-list` • `radio` • `radio-checked` • `radio-selected`
CheckboxList	• `checkbox-list` • `checkbox` • `checkbox-checked` • `checkbox-selected`
VerticalLine	• `line` • `vertical-line`
HorizontalLine	• `line` • `horizontal-line`
ProgressBar	• `progress-bar` • `progress-bar.used`

5.8.1 Example

Let's customize the example of the `Check Box`.

It uses 2 `Button`, a `CheckboxList` and a `Label`, packed inside a `Dialog`. Therefore we can customize each of these elements separately, using for instance:

```
from quo.dialog import CheckBox
from quo.style import Style

style = Style.add({
            'dialog': 'bg:green',
            'button': 'bg:red',
            'checkbox': 'fg:blue',
            'dialog.body': 'bg:yellow',
            'dialog shadow': 'bg:khaki',
            'frame.label': 'fg:black',
            'dialog.body label': 'fg:aquamarine'})

CheckBox(
       title="CheckboxList dialog",
       text="What would you like in your breakfast ?",
       values=[
           ("eggs", "Eggs"),
           ("bacon", "Bacon"),
           ("croissants", "20 Croissants"),
           ("daily", "The breakfast of the day")
       ],
       style = style)
```

» Check out more examples here

CHAPTER

SIX

PARSE

This is intended to be a gentle introduction to `Parser`, a command-line parsing class based on argparse.

Optional arguments can be added to commands using the `quo.parse.Parser`.

Optional arguments in Quo are profoundly configurable and ought not to be mistaken for positional arguments.

Parameters

- `filename` *(str)* - The name of the file to open (or `'-'` for stdin/stdout).
- `prog` - The name of the program (default:`os.path.basename(sys.argv[0])`)
- `color` *(bool)* - Print a colorful help output
- `usage` - A usage message (default: auto-generated from arguments)
- `description` - A description of what the program does
- `epilog` - Text following the argument descriptions
- `argument_default` - The default value for all arguments
- `add_help` *(bool)* - Add a -h/-help option
- `allow_abbrev` *(bool)*- Allow long options to be abbreviated unambiguously
- `exit_on_error` *(bool)* - Determines whether or not ArgumentParser exits with error info when an error occurs

6.1 How to name Optional Arguments

For the purpose of uniformity, a name is chosen in the following order

1. In the event that the name is not prefixed with – or -, it will be considered a positional argument.
2. If there is more than one name prefixed with – or -, the first one given is used as the name.

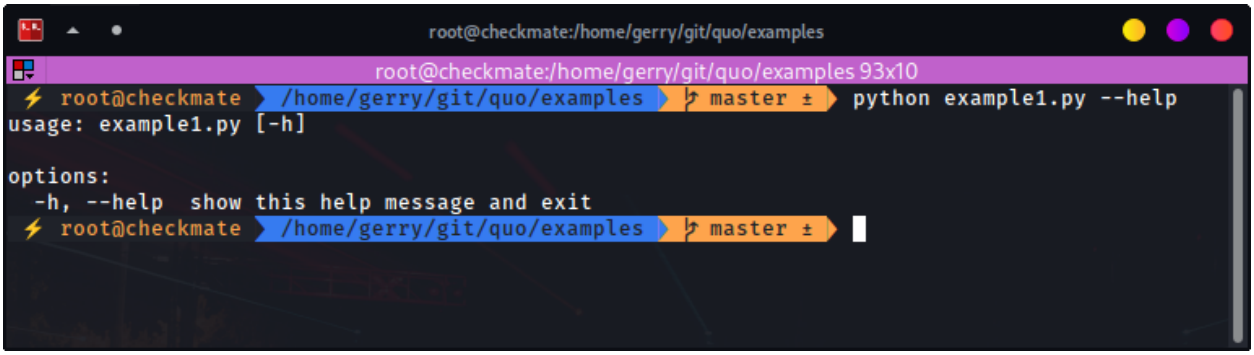
6.1.1 The basics

Let us start with a very simple example which does (almost) nothing:

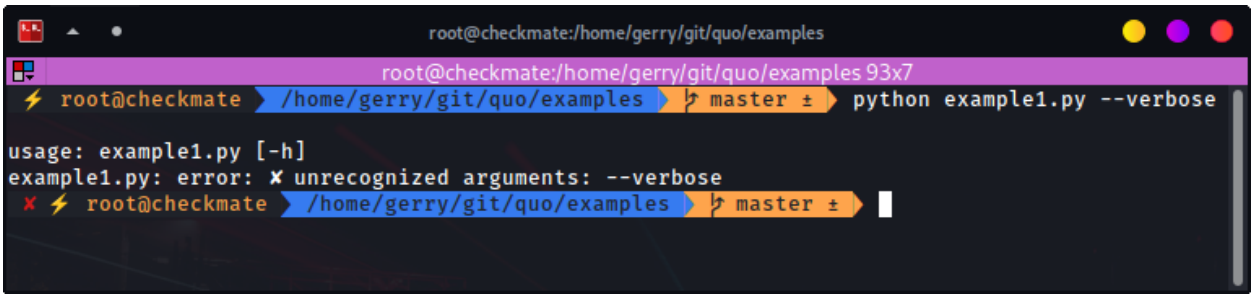
```
from quo.parse import Parser
arg = Parser()
arg.parse()
```

Following is a result of running the code:

```
python example1.py --help
```




```
python example1.py --verbose
```

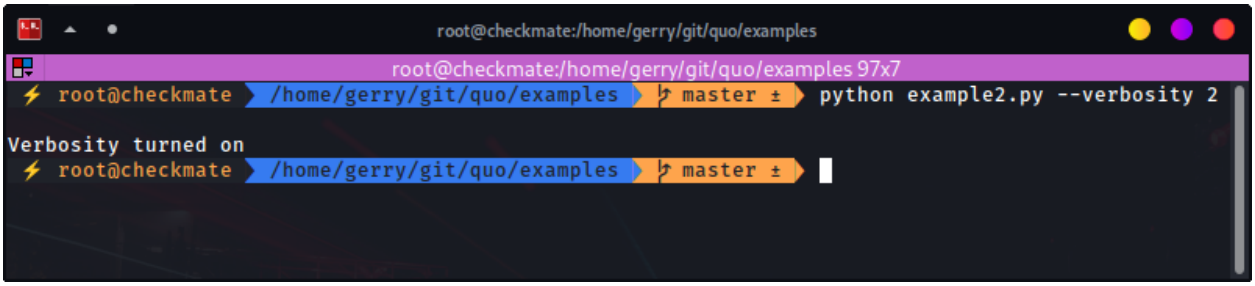


Running the script without any options results in nothing displayed to stdout. Not so useful. The second one starts to display the usefulness of `Parser`. We have done almost nothing, but already we get a nice help message.

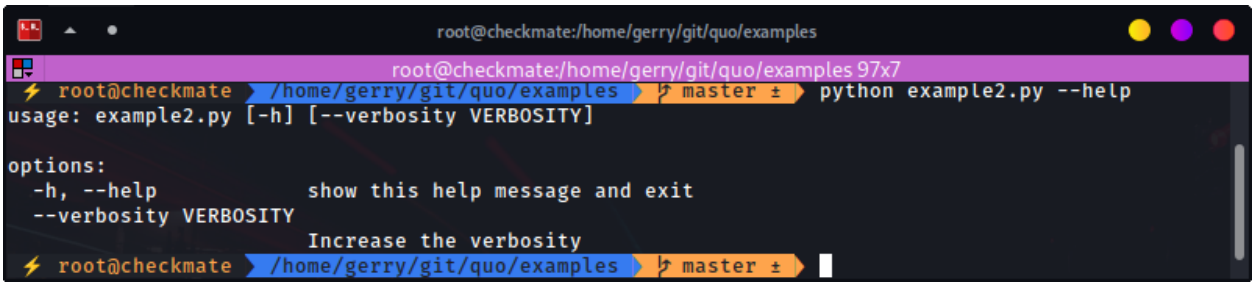
The *–help* option, which can also be shortened to *-h*, is the only option we get for free (i.e. no need to specify it). Specifying anything else results in an error. But even then, we do get a useful usage message.

```
from quo.parse import Parser

optional = Parser()
optional.argument("--verbosity", help="Increase the verbosity")
arg = optional.parse()
if arg.verbosity:
       print("Verbosity turned on")
```

```
python example2.py --verbosity 1
```




```
python example2.py --help
```




```
python example2.py --verbosity
```

The program above is written so as to display something when *–verbosity* is specified and display nothing when not specified. To show that the option is actually optional, there is no error when running the program without it.

Note: By default, if an optional argument isn't used, the relevant variable, in this case *argsverbosity*, is given None as its value.

When using the optional argument in this case *–verbosity* option, one must also specify some value, any value.

The above example accepts arbitrary integer values for –verbosity, but for our simple program, only two values are actually useful, True or False. Let's modify the code accordingly:

```
from quo.parse import Parser

optional = Parser()
optional.argument("--verbose", help="Increase the verbosity", action="store_true")
arg = optional.parse()
if arg.verbose:
       print("Verbosity turned on")
```

And the output:

```
python example2.py --verbose
```

```
python example2.py --verbose 1
```

```
python example2.py --help
```

Here is what is happening:

The option is now more of a flag than something that requires a value. We even changed the name of the option to match that idea. Note that we now specify a new keyword, action, and give it the value "store_true". This means that,

if the option is specified, assign the value True to arg.verbose. Not specifying it implies False.

It complains when you specify a value, in true spirit of what flags actually are.

Notice the different help text.

6.1.2 Short options

If you are familiar with command line usage, you will notice that I haven't yet touched on the topic of short versions of the options. It's quite simple:

```
from quo.parse import Parser

optional = Parser()
optional.argument("-v", "--verbose", help="Increase the verbosity", action="store_true")
arg = optional.parse()
if arg.verbose:
       print("Verbosity turned on")
```

And here goes:

```
python example3.py -v
```

/image/

```
python example3.py --help
```

/image/

Note that the new ability is also reflected in the help text.

import argparse The help message is a bit different.

CHAPTER

SEVEN

POSITIONAL ARGUMENTS

Introducing Positional arguments An example:

```
from quo.parse import Parser

positional = Parser()

positional.argument("echo")
arg = positional.parse()
print(arg.echo)
```

Note however that, although the help display looks nice and all, it currently is not as helpful as it can be. For example we see that we got echo as a positional argument, but we don't know what it does, other than by guessing or by reading the source code. So, let's make it a bit more useful:

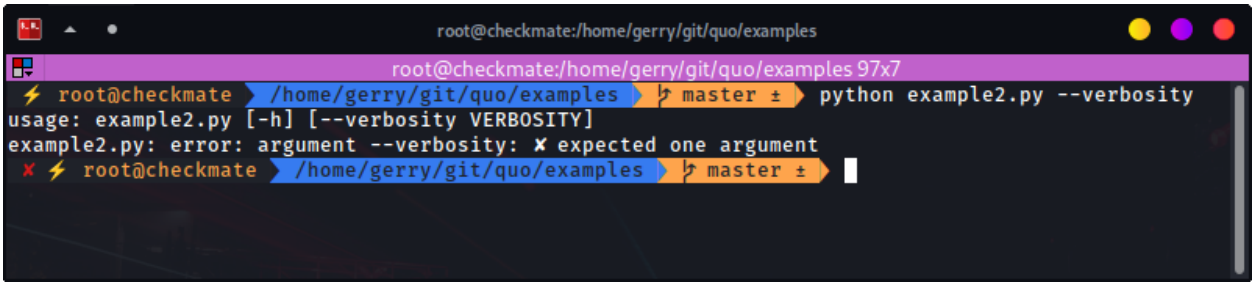
```
 from quo.parse import Parser

 positional = Parser()

 positional.argument("echo", help="echo the string you use here")
 arg = positional.parse()
 print(arg.echo)


-h, --help  show this help message and exit
```

Now, how about doing something even more useful:

```
from quo.parse import Parser

positional = Parser()

positional.argument("square", help="display a square of a given number")
arg = positional.parse()
print(arg.square**2)
```

Following is a result of running the code:

```
python prog.py 4
```

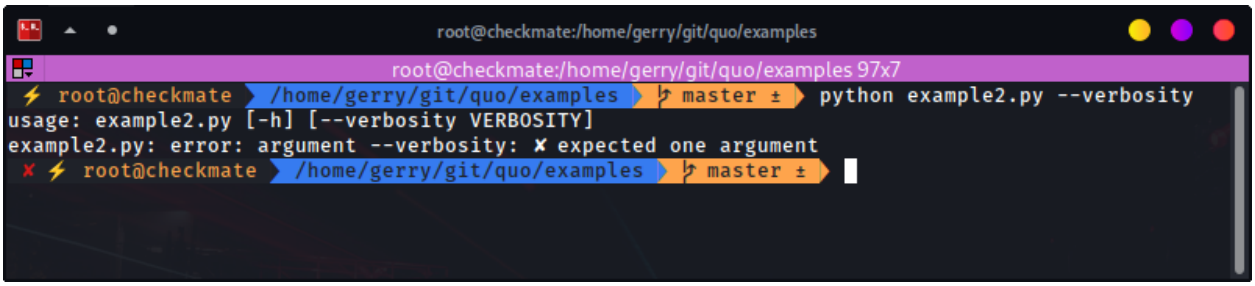


That didn't go so well. That's because `Parser` treats the options we give it as strings, unless we tell it otherwise. So, let's tell it to treat that input as an integer:

```
from quo.parse import Parser

positional = Parser()

positional.argument("square", help="display a square of a given number", type=int)
arg = positional.parse()
print(arg.square**2)
```

Following is a result of running the code:

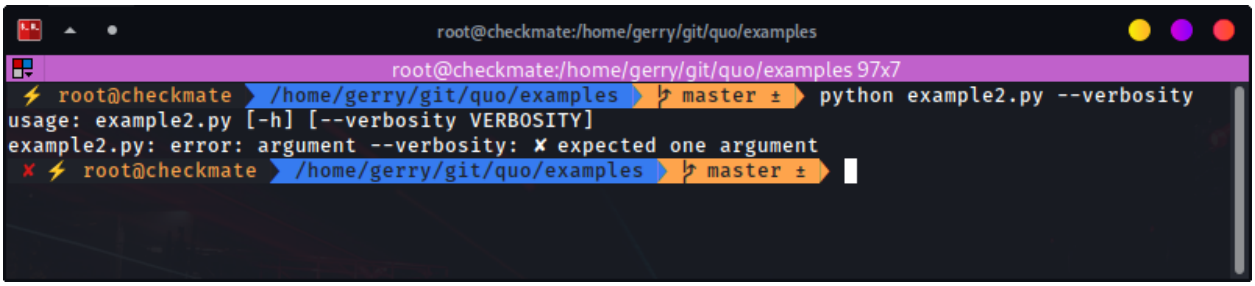
```
python prog.py 4
```



16

how about this…

```
python prog.py four
```



That went well. The program now even helpfully quit on illegal input before proceeding.

CHAPTER

EIGHT

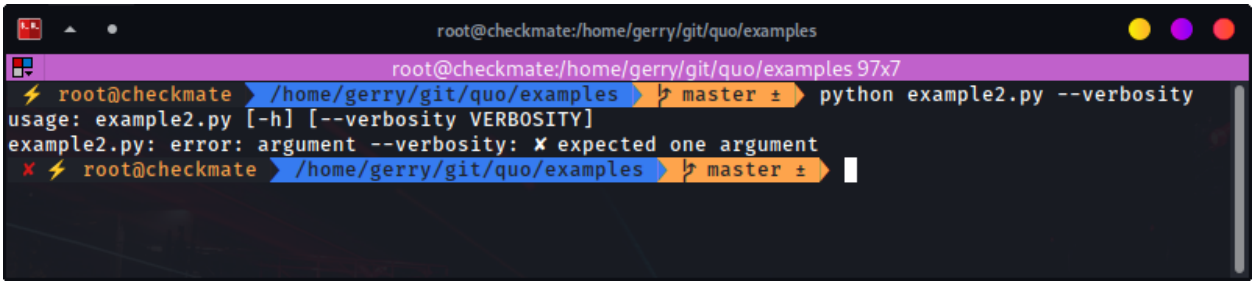
COMBINING POSITIONAL AND OPTIONAL ARGUMENTS

Our program keeps growing in complexity

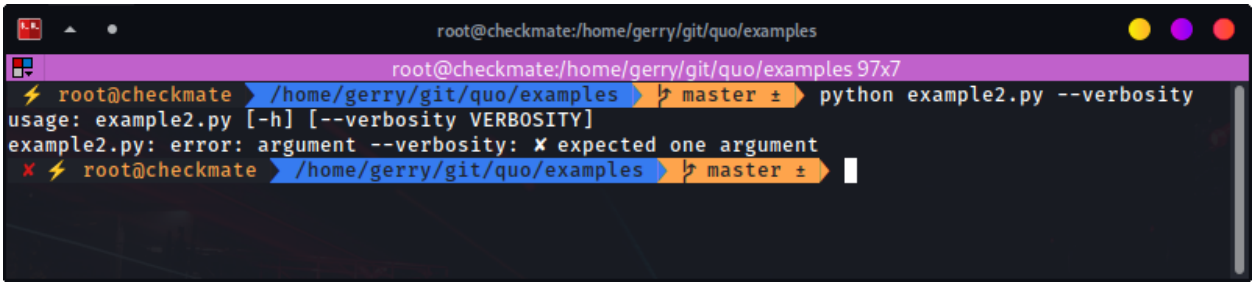
```
from quo.parse import Parser

parser = Parser()

parser.argument("square", type=int, help="display a square of a given number")
parser.argument("-v", "--verbose", action="store_true", help="increase output verbosity")
arg = parser.parse()

answer = arg.square**2

if args.verbose:
     print(f"the square of {arg.square} equals {answer}")

else:
  print(answer)
```

And now the output:

```
python prog.py
```




```
python3 prog.py 4 --verbose
```



Note that the order does not matter. The above program can be written like so:

```
python3 prog.py --verbose 4
```

How about we give this program of ours back the ability to have multiple verbosity values, and actually get to use them:

```
from quo.parse import Parser

parser = Parser()

parser.argument("square", type=int, help="display a square of a given number")
parser.argument("-v", "--verbosity", type=int, help="increase output verbosity")
arg = parser.parse()

answer = arg.square**2

if arg.verbosity == 2:
      print(f"the square of {arg.square} equals {answer}")
elif arg.verbosity == 1:
    print(f"{arg.square}^2 == {answer}")
else:
    print(answer)
```

And the output:

```
python prog.py  4
```

$ python3 prog.py 4 16

```
python prog.py 4 -v
```

$ python3 prog.py 4 -v usage: prog.py [-h] [-v VERBOSITY] square prog.py: error: argument -v/–verbosity: expected one argument

```
python prog.py 4  -v 1
```

```
python prog.py 4 -v 2
```

```
python prog.py 4 -v 3
```

These all look good except the last one, which exposes a bug in our program. Let's fix it by restricting the values the `--verbosity` option can accept:

```
from quo.parse import Parser

```

(continues on next page)

(continued from previous page)

```
parser = Parser()

parser.argument("square", type=int, help="display a square of a given number")
parser.argument("-v", "--verbosity", type=int, choices=[0, 1, 2], help="increase output␣
↪verbosity")
arg = parser.parse()

answer = arg.square**2

if arg.verbosity == 2:
    print(f"the square of {arg.square} equals {answer}")

elif arg.verbosity == 1:
    print(f"{arg.square}^2 == {answer}")

else:
 print(answer)
```

And the output:

```
python prog.py 4 -v 3
```

Note that the change also reflects both in the error message as well as the help string. .. code:: console

    python prog.py 4 -h

Now, let's use a different approach of playing with verbosity, which is pretty common. It also matches the way the CPython executable handles its own verbosity argument (check the output of python –help):

```
from quo.parse import Parser

parser = Parser()

parser.argument("square", type=int, help="display a square of a given number")
parser.argument("-v", "--verbosity", action="count", help="increase output verbosity")
arg = parser.parse()

answer = arg.square**2
if arg.verbosity == 2:
    print(f"the square of {arg.square} equals {answer}")

elif arg.verbosity == 1:
    print(f"{arg.square}^2 == {answer}")

else:
    print(answer)
```

We have introduced another action, `count`, to count the number of occurrences of specific options.

```
python prog.py 4
```

```
python prog.py 4 -v
```

```
python prog.py 4 -vv
```

```
python prog.py 4 -v 1
```

CHAPTER

NINE

GROUPING CONFLICTING OPTIONAL ARGUMENTS

`group()` allows us to specify options that conflict with each other. Let's also change the rest of the program so that the new functionality makes more sense: we'll introduce the `--quiet` option, which will be the opposite of the `--verbose` one.

```
from quo.parse import Parser

parser = Parser()

group = parser.group()
group.argument("-v", "--verbose", action="store_true")
group.argument("-q", "--quiet", action="store_true")

parser.argument("x", type=int, help="the base")
parser.argument("y", type=int, help="the exponent")

arg = parser.parse()

answer = arg.x**arg.y

if arg.quiet:
    print(answer)
elif arg.verbose:
    print(f"{arg.x} to the power {arg.y} equals {answer}")
else:
    print(f"{arg.x}^{arg.y} == {answer}")
```

Our program is now simpler, and we've lost some functionality for the sake of demonstration. Anyways, here's the output

```
python prog.py 4 2
```

Ouput:

```
4^2 == 16
```

```
$ python prog.py 4 2 -q
```

Output:

```
16
```

```
$ python3 prog.py 4 2 -v
```

Ouput:

```
4 to the power 2 equals 16
```

```
$ python prog.py 4 2 -vq
```

That should be easy to follow. I've added that last output so you can see the sort of flexibility you get, i.e. mixing long form options with short form ones.

Before we conclude, you probably want to tell your users the main purpose of your program, just in case they don't know

```
from quo.parse import Parser

parser = Parser(description="calculate X to the power of Y")

group = parser.group()

group.argument("-v", "--verbose", action="store_true")
group.argument("-q", "--quiet", action="store_true")
parser.argument("x", type=int, help="the base")
parser.argument("y", type=int, help="the exponent")

arg = parser.parse()
answer = arg.x**arg.y

if arg.quiet:
    print(answer)
elif arg.verbose:
    print("{} to the power {} equals {}".format(arg.x, arg.y, answer))
else:
    print("{}^{} == {}".format(arg.x, arg.y, answer))
```

Note that slight difference in the usage text. Note the [-v | -q], which tells us that we can either use -v or -q, but not both at the same time

```
python prog.py --help
```

CHAPTER

TEN

PROGRESS BARS

A progress bar is a user interface element that indicates the progress of an operation. Progress bar supports two modes to represent progress: determinate, and indeterminate. Showing Progress Bars Sometimes, you have command line scripts that need to process a lot of data, but you want to quickly show the user some progress about how long that will take. Quo supports simple progress bar rendering for that.

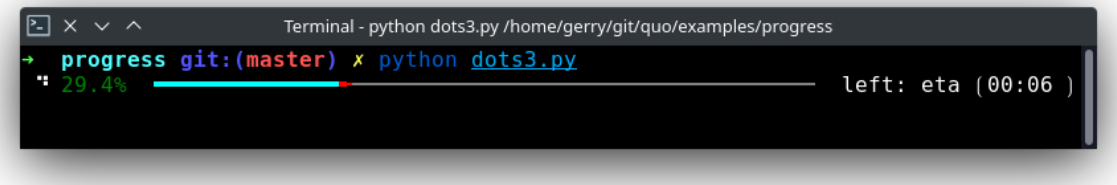
The basic usage is very simple: the idea is that you have an iterable that you want to operate on. For each item in the iterable it might take some time to do processing.

10.1 Simple progress bar

Creating a new progress bar can be done by calling the `ProgressBar`

The progress can be displayed for any iterable. This works by wrapping the iterable (like `range`) with the `ProgressBar`. This way, the progress bar knows when the next item is consumed by the forloop and when progress happens.

```
import time

from quo.progress import ProgressBar


with ProgressBar() as pb:
    for i in pb(range(800)):
        time.sleep(.01)
```



Keep in mind that not all iterables can report their total length. This happens with a typical generator. In that case, you can still pass the total as follows in order to make displaying the progress possible:

```
def some_iterable():
    yield ...
```

(continues on next page)

(continued from previous page)

```
with ProgressBar() as pb:
    for i in pb(some_iterable, total=1000):
        time.sleep(.01)
```

10.2 Autohide progressbar

Autohide the progressbar after consuming an iterator.

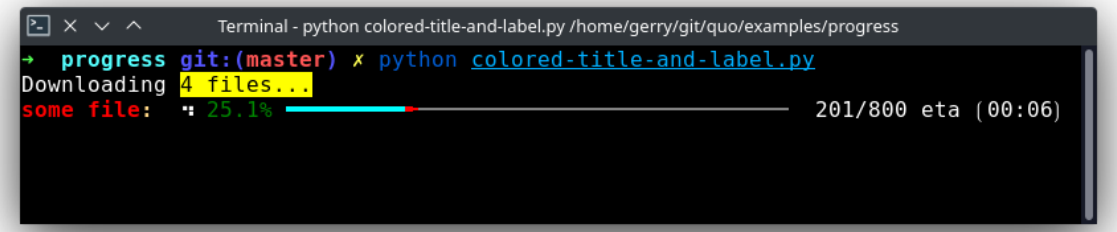
(Added on v2023.3)

```
import time

from quo.progress import ProgressBar

with ProgressBar() as pb:
     for i in pb(range(800), auto_hide=True):
        time.sleep(.01)
```

10.3 Adding a title and label

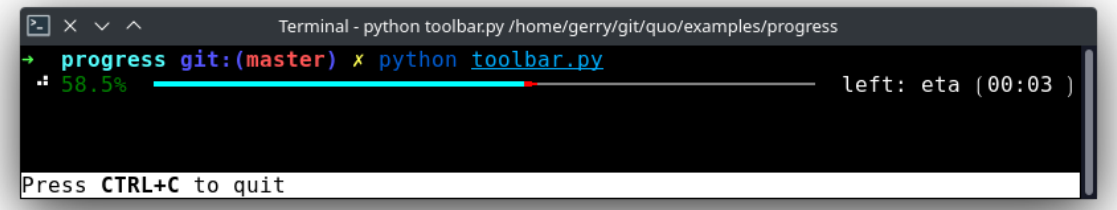
Each progress bar can have one title, and for each task an individual label.

```
import time

from quo.progress import ProgressBar

title = "<style fg='yellow' bg='black'>Downloading 4 files...</style>"

label = "<red>some file:</red>"

with ProgressBar(title) as pb:
    for i in pb(range(800), label):
      time.sleep(.01)
```



10.4 Adding a toolbar

```
import time

from quo.progress import ProgressBar

toolbar = "Press <b>CTRL+C</b> to quit"

with ProgressBar(toolbar=toolbar) as pb:
    for i in pb(range(800)):
      time.sleep(.01)
```

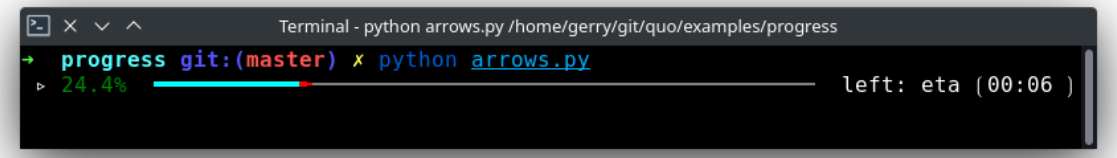


10.5 Spinner themes

(Added on v2023.3)

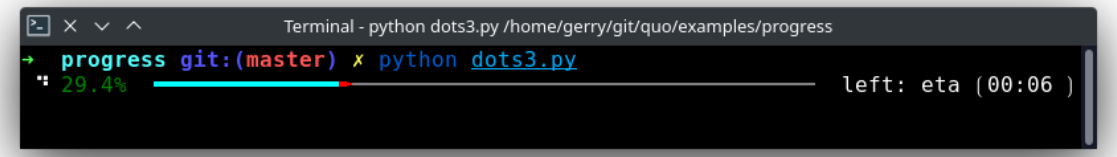
- arrows

```
import time

from quo.progress import ProgressBar

with ProgressBar(spinner="arrows") as pb:
    for i in pb(range(800)):
        time.sleep(0.01
```



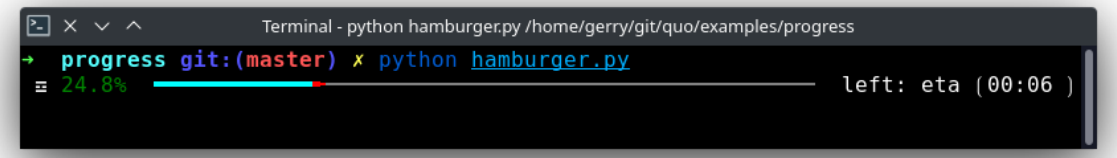
- dots3

```
import time

from quo.progress import ProgressBar

with ProgressBar(spinner="dots3") as pb:
    for i in pb(range(800)):
        time.sleep(0.01
```



- hamburger

```
import time

from quo.progress import ProgressBar

with ProgressBar(spinner="hamburger") as pb:
    for i in pb(range(800)):
        time.sleep(0.01
```



10.6 Multiple parallel tasks

A quo `ProgressBar` can display the progress of multiple tasks running in parallel. Each task can run in a separate thread and the `ProgressBar` user interface runs in its own thread.

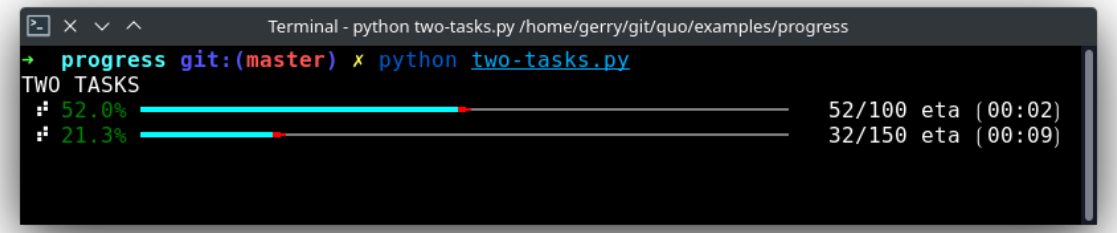
Notice that we set the "daemon" flag for both threads that run the tasks. This is because control-c will stop the progress and quit our application. We don't want the application to wait for the background threads to finish. Whether you want this depends on the application.

```
import threading
import time
```

(continues on next page)

(continued from previous page)

```
from quo.progress import ProgressBar

with ProgressBar("TWO TASKS") as pb:
    # Two parallel tasks.
    def task1():
        for i in pb(range(100)):
            time.sleep(.05)

    def task2():
        for i in pb(range(150)):
            time.sleep(.08)

    # Start threads.
    t1 = threading.Thread(target=task1)
    t2 = threading.Thread(target=task2)
    t1.daemon = True
    t2.daemon = True
    t1.start()
    t2.start()

    # Wait for the threads to finish. We use a timeout for the join() call,
    # because on Windows, join cannot be interrupted by Control-C or any other
    # signal.
    for t in [t1, t2]:
        while t.is_alive():
            t.join(timeout=.5)
```



10.7 Nested progressbars

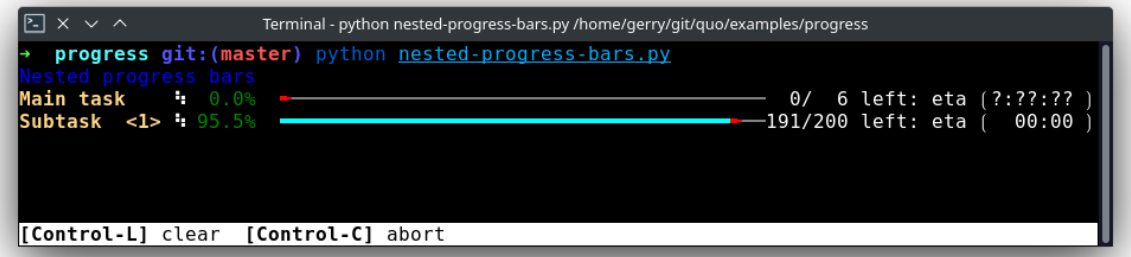
Example of nested progress bars.

```
import time

from quo.progress import ProgressBar

title='<blue>Nested progress bars</blue>'
toolbar="<b>[Control-L]</b> clear  <b>[Control-C]</b> abort"
```

(continues on next page)

(continued from previous page)

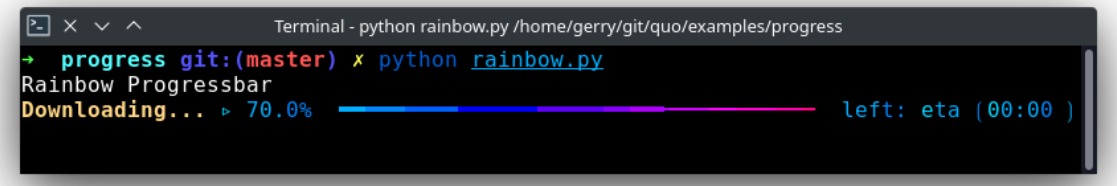
```
with ProgressBar(title, bottom_toolbar=toolbar)as pb:
    for i in pb(range(6), label="Main task"):
        for j in pb(range(200), label=f"Subtask  <%s>" % (i + 1,), auto_hide=True):
            time.sleep(0.01)
```



10.8 Rainbow progress bar

A simple progress bar, visualised with rainbow colors for fun.

```
import time

from quo.progress import ProgressBar

with ProgressBar("Rainbow Progressbar", rainbow=True, spinner="arrows") as pb:
       for i in pb(range(20), label="Downloading...", auto_hide=True):
           time.sleep(0.1)
```



10.9 Adding a key binder

Like other quo applications, we can add custom key bindings, by passing `quo.keys.bind()` which is an instance of `Bind` object

```
import os
import signal
import time

from quo.keys import bind
from quo.progress import ProgressBar

bottom_toolbar = 'Press <b>[q]</b> to Abort or <b>[x]</b>  to Send Control-C.'

# Create custom key bindings first
cancel = [False]

@bind.add("q")
def _(event):
    "Quit by setting cancel flag."
    cancel[0] = True

@bind.add("x")
def _(event):
    "Quit by sending SIGINT to the main thread."
    os.kill(os.getpid(), signal.SIGINT)


with ProgressBar(bottom_toolbar=bottom_toolbar) as pb:
     for i in pb(range(800)):
          time.sleep(0.01)

         if cancel[0]:
             break
```

when "x" is pressed, we set a cancel flag, which stops the progress. It would also be possible to send *SIGINT* to the mean thread, but that's not always considered a clean way of cancelling something.

In the example above, we also display a toolbar at the bottom which shows the key bindings.

Read more about key bindings

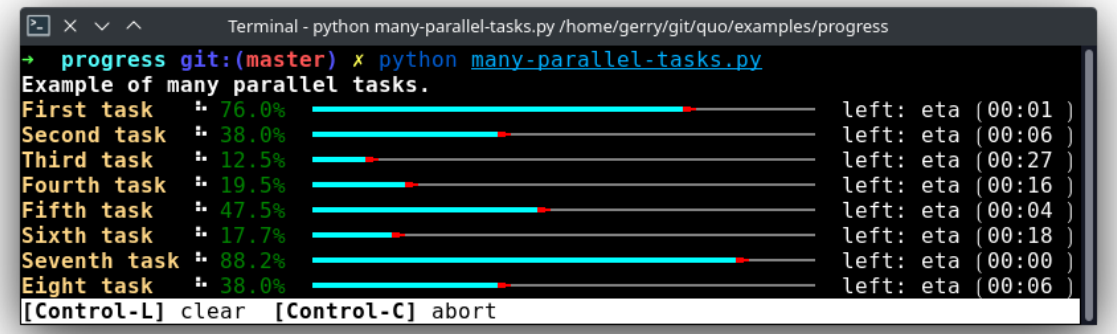
Here's a more complex demonstration of what's possible with the progress bar.

```
import threading
import time

from quo.progress import ProgressBar

title = "<b>Example of many parallel tasks.</b>"
toolbar = "<b>[Control-L]</b> clear  <b>[Control-C]</b> abort"

with ProgressBar(title, bottom_toolbar=toolbar) as pb:
    def run_task(label, total, sleep_time):
```

(continues on next page)

(continued from previous page)

```
        for i in pb(range(total), label=label):
                time.sleep(sleep_time)

    threads = [
        threading.Thread(target=run_task, args=("First task", 50, 0.1)),
        threading.Thread(target=run_task, args=("Second task", 100, 0.1)),
        threading.Thread(target=run_task, args=("Third task", 8, 3)),
        threading.Thread(target=run_task, args=("Fourth task", 200, 0.1)),
        threading.Thread(target=run_task, args=("Fifth task", 40, 0.2)),
        threading.Thread(target=run_task, args=("Sixth task", 220, 0.1)),
        threading.Thread(target=run_task, args=("Seventh task", 85, 0.05)),
        threading.Thread(target=run_task, args=("Eight task", 200, 0.05)),
    ]

    for t in threads:
        t.daemon = True
        t.start()

    # Wait for the threads to finish. We use a timeout for the join() call,
    # because on Windows, join cannot be interrupted by Control-C or any other
    # signal.
    for t in threads:
          while t.is_alive():
             t.join(timeout=0.5)
```



» Check out more examples here

CHAPTER

ELEVEN

PROMPTS

Quo supports prompts in two different places. The first is automated prompts when the parameter handling happens, and the second is to ask for prompts at a later point independently.

This can be accomplished with the `prompt()` function, which asks for valid input according to a type, or the `quo.prompt.Prompt` object, this makes it possible to create a Prompt instance followed by calling prompt() method for every input. This creates a kind of an input session and its packed with lots of features. You can also use the `quo.confirm()` function, which asks for confirmation (yes/no).

The prompt function is a quo function that displays a prompt to the user and waits for input. The purpose of the prompt function is to obtain user input from the console. It can be used to ask the user for a variety of input, including text, numbers, and boolean values. It has several optional arguments which can be used to customize the prompt and how the input is handled.

Parameters

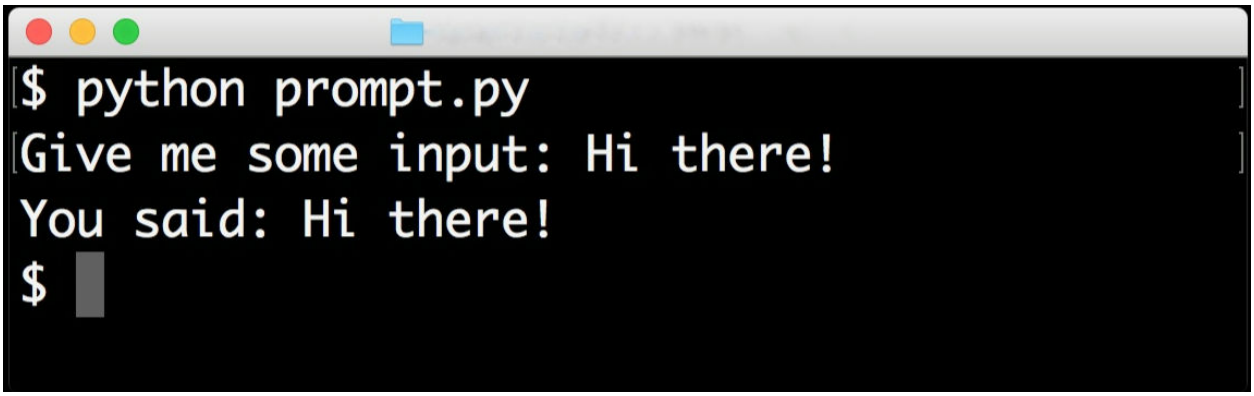
The prompt function takes several parameters, which are explained below:

> text: The text to show for the prompt. This parameter is required and must be a string. default: The default value to use if no input happens. If this is not given, it will prompt until it's aborted. This parameter is optional and can be any data type. hide: If this is set to true, then the input value will be hidden, and asterisks printed instead. This parameter is optional and defaults to False. affirm: Asks for confirmation for the value. This parameter is optional and defaults to False. type: The type to use to check the value against. This parameter is optional and defaults to str. suffix: A suffix that should be added to the prompt. This parameter is optional and defaults to an empty string. show_default: Shows or hides the default value in the prompt. This parameter is optional and defaults to True. fg: The color for the prompt text. This parameter is optional and defaults to None. bg: The color for the prompt background. This parameter is optional and defaults to None.

Examples

Here's a simple example

```
import quo

quo.prompt('Give me some input: ')
```



The prompt function returns the input value provided by the user, or the default value if no input was provided. The data type of the return value will depend on the type parameter.

```
import quo

name = quo.prompt('What is your name?', type=str)
age =  quo.prompt('What is your age?', default=18, type=int)
```

Additionally, the type will be determined automatically if a default value is provided. For instance, the following will only accept floats:

```
import quo

quo.prompt('Please enter a number', default=42.0)
```

11.1 App Prompts

App prompts are integrated into the app interface. See app-prompting for more information. Internally, it automatically calls either `quo.prompt()` or `quo.confirm()` as necessary.

11.2 Input Validation

A prompt can have a validator attached. To manually ask for user input, you can use the `quo.prompt()` function or the `quo.prompt.Prompt` object. For instance, you can ask for a valid integer:

```
from quo import prompt

prompt('Please enter a valid integer', type=int)
```

You can also pass the `affirm` flag to `quo.prompt()`

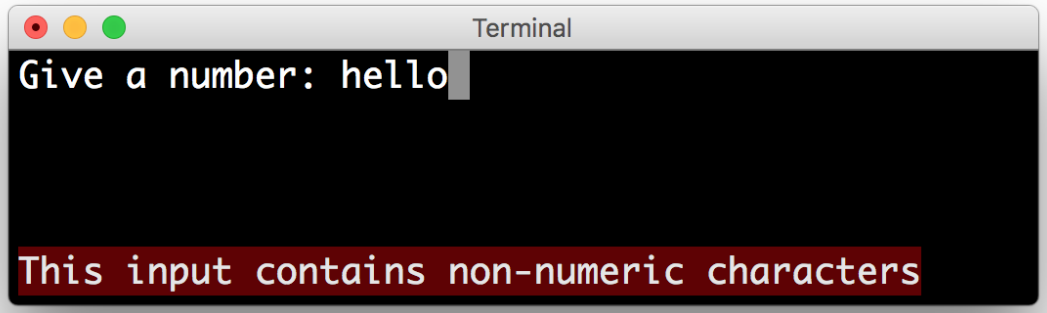
```
from quo import prompt
prompt("What is your name?: ", affirm=True)
```

Alternatively, you can use class:*quo.types.Validator* This should implement the `Validator` abstract base class. This requires only one method, named `type` that takes a `Document` as input and raises `ValidationError` when the validation fails.

Added on *v2022.4.4* :meth:`int` [bool] can be used when validating numerical characters.

11.2.1 Integer Validator

```
from quo.prompt import Prompt

session = Prompt(int=True)

number = int(session.prompt('Give a number: '))
print(f"You said: {number}")
```



By default, the input is validated in real-time while the user is typing, but Quo can also validate after the user presses the enter key:

```
session = Prompt(
              int=True,
              validate_while_typing=False
              )

session.prompt('Give a number: ')
```

If the input validation contains some heavy CPU intensive code, but you don't want to block the event loop, then it's recommended to wrap the validation class in a `ThreadedValidator`.

11.3 Input Prompts using Prompt() class

Input history can be kept between consecutive `quo.prompt()` and `quo.prompt.Prompt` calls incase you want to ask for multiple inputs, but each input call needs about the same arguments.

```
from quo import prompt

tex1 = prompt("What is your name?")
text2 = prompt("Where are you from?")
```

```
from quo.prompt import Prompt

```

(continues on next page)

(continued from previous page)

```
# Create prompt object.
session = Prompt()

# Do multiple input calls.
text1 = session.prompt("What's your name?")
text2 = session.prompt("Where are you from?")
```

11.4 Multiline Input

Reading multiline input is as easy as passing the `multiline=True` parameter.

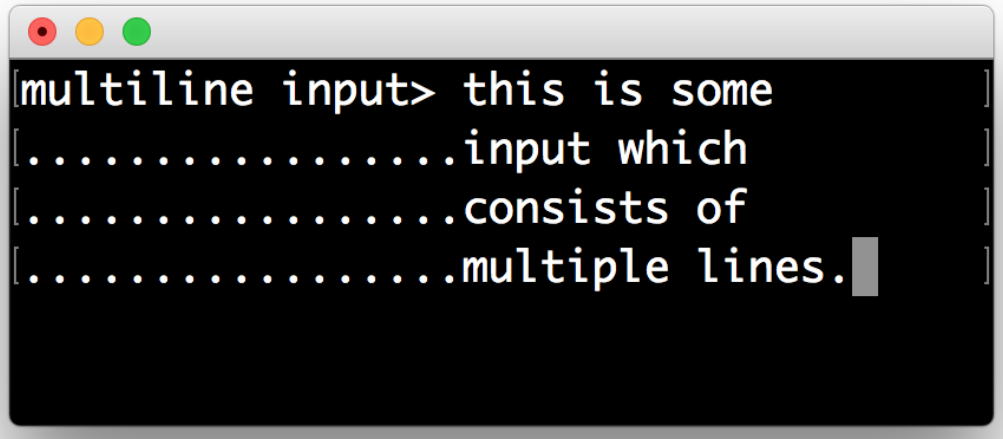
```
from quo.prompt import Prompt

session = Prompt(multiline=True)
session.prompt('> ')
```

A side effect of this is that the enter key will now insert a newline instead of accepting and returning the input. The user will now have to press `Meta+Enter` in order to accept the input. (Or `Escape` followed by `Enter`.)

It is possible to specify a continuation prompt. This works by passing :meth:`continuation` [bool] to `Prompt`. This function is supposed to return formatted text, or a list of `(style, text)` tuples. The width of the returned text should not exceed the given width. (The width of the prompt margin is defined by the prompt.)

`continuation()` was added on *v2022.4.4*

```
from quo.prompt import Prompt

session = Prompt(multiline=True, continuation=True)

session.prompt('multiline input> ')
```



11.5 Hide Input

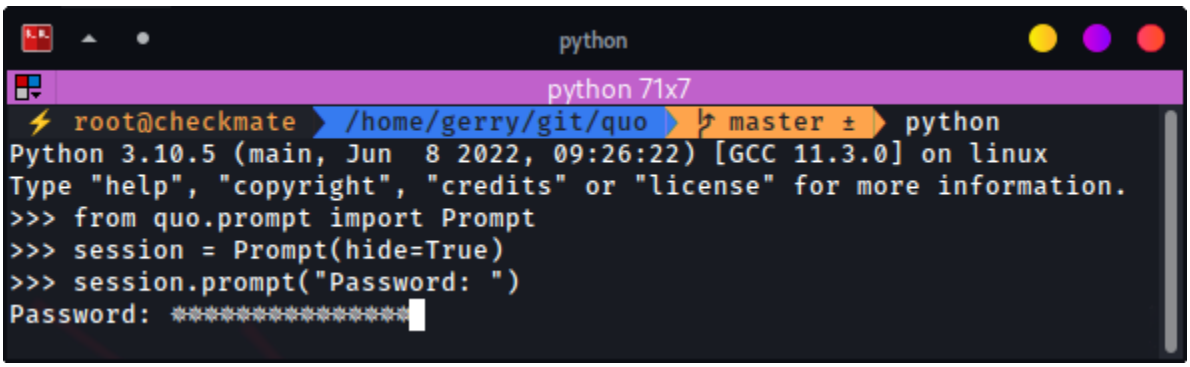
When the `hide=True` flag in `quo.prompt()` or `quo.prompt.Prompt` has been given, the input is hidden in `quo.prompt()` or replaced by asterisks (* characters) in `quo.prompt.Prompt`

11.5.1 Using function quo.prompt()

```
from quo import prompt

prompt("Enter password: ", hide=True)
```

11.5.2 Using class `quo.prompt.Prompt()

```
from quo.prompt import Prompt

session = Prompt(hide=True)

session.prompt("Password: ")
```



11.6 Confirmation Prompts

To ask if a user wants to continue with an action, the `confirm()` function comes in handy. By default, it returns the result of the prompt as a boolean value: **Parameters**

- `text` *(str)* – the question to ask.
- `default` *(Optional[str, int])* – The default value to use when no input is given. If None, repeat until input is given.
- `abort` *(Optional[bool])* – if this is set to True a negative answer aborts the exception by raising Abort.
- `suffix` *(str)* – a suffix that should be added to the prompt.
- `show_default` *(Optional[bool])* – shows or hides the default value in the prompt.
- `err` *(bool)* – if set to true the file defaults to stderr instead of stdout, the same as with echo.

```
from quo import confirm

confirm('Do you want to continue?')
```

11.7 System prompt

If you press `meta-!` or `esc-!`, you can enter system commands like *ls* or *cd*.

```
from quo.prompt import Prompt

session = Prompt(system_prompt=True)

session.prompt("Give me some input: ")
```

11.8 Suspend prompt

Pressing `ctrl-z` will suspend the process from running and then run the command *fg* to continue the process.

```
from quo.prompt import Prompt

session = Prompt(suspend=True)

sessiom.prompr("Give me some input: ")
```

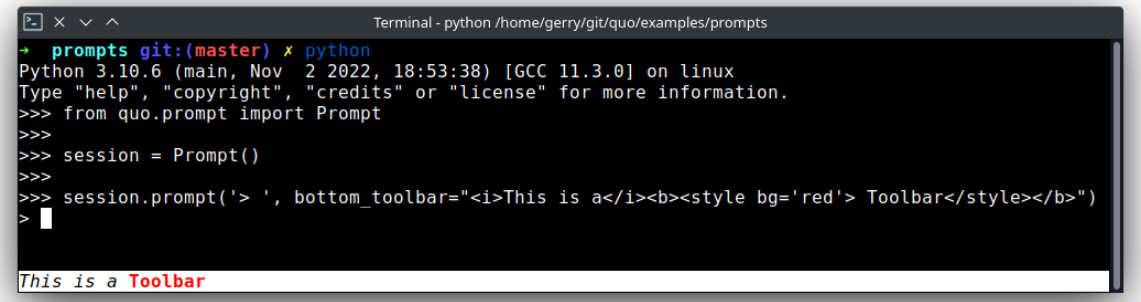
11.9 Prompt bottom toolbar

Adding a bottom toolbar is as easy as passing a bottom_toolbar argument to prompt(). This argument be either plain text, formatted text or a callable that returns plain or formatted text.

When a function is given, it will be called every time the prompt is rendered, so the bottom toolbar can be used to display dynamic information.

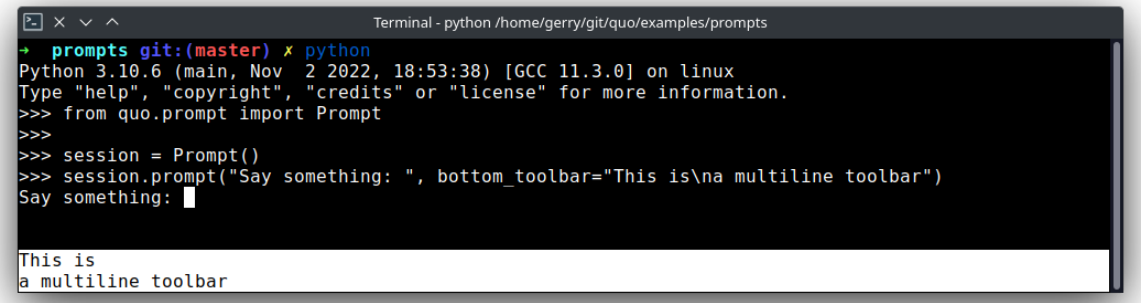
By default, the toolbar has the reversed style, which is why we are setting the background instead of the foreground.

```
from quo.prompt import Prompt

session = Prompt()

session.prompt('> ', bottom_toolbar="<i>This is a</i><b><style bg='red'> Toolbar</style>
↪</b>")
```



Here's an example of a multiline bottom toolbar.

```
from quo.prompt import Prompt

session = Prompt()

session.prompt("Say something: ", bottom_toolbar="This is\na multiline toolbar")
```



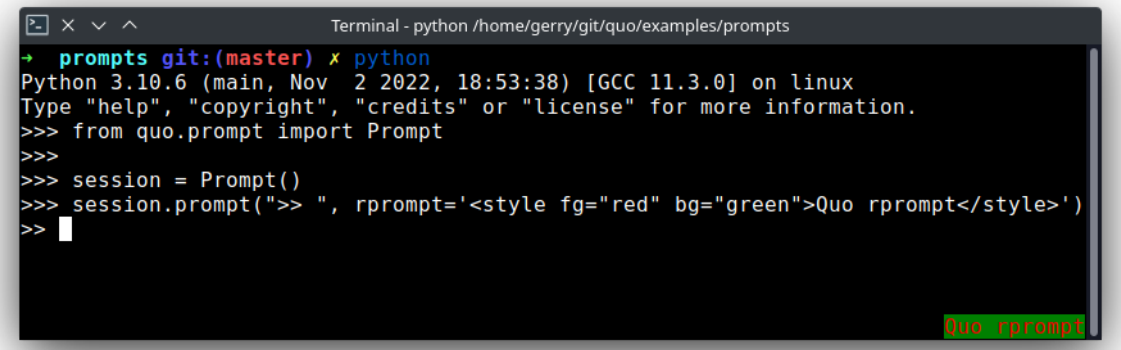
11.10 Right prompt(rprompt)

The `quo.prompt.Prompt` class has out of the box support for right prompts as well. People familiar to ZSH could recognise this as the RPROMPT option.

This can be either plain text, formatted text or a callable which returns either.

The following example returns a formatted text:

```
from quo.prompt import Prompt

session = Prompt()
session.prompt(">> ", rprompt='<style fg="red" bg="green">Quo rprompt</style>')
```

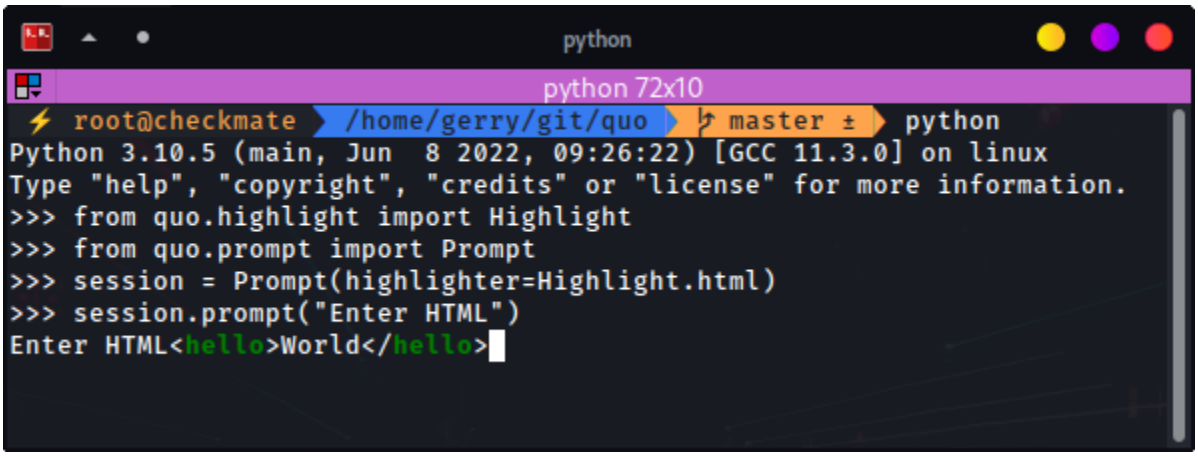


11.11 Syntax highlighting

Quo ships with an intuitive syntax highlighter. It is also possible to create a custom highlighter by implementing the `Highlight` class.

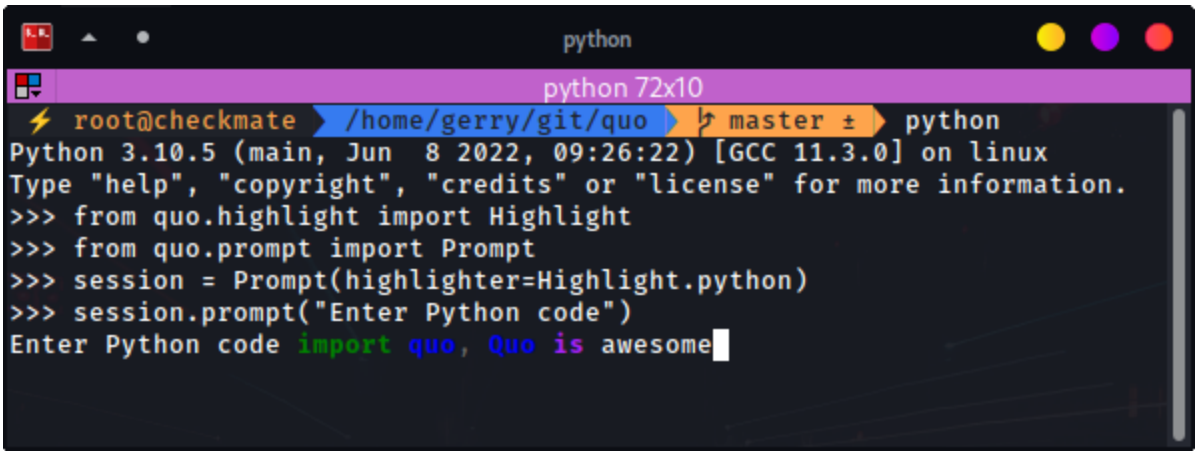
(changed since v2022.9)

```
from quo.prompt import Prompt
from quo.highlight import Highlight

session = Prompt(highlighter=Highlight.html)

session.prompt('Enter HTML: ')
```



If you want to use another style you can do the following:»

```
from quo.prompt import Prompt
from quo.highlight import Highlight

session = Prompt(highlighter=Highlight.python)
session.prompt('Enter Python code: ')
```



or:»

```
from quo.prompt import Prompt
from quo.highlight import Highlight

session = Prompt(highlighter=Highlight.css)
session.prompt('Enter css: ')
```

Syntax highlighting is as simple as adding a highlighter. All of the available syntax styles can be found here or Read more about styling.

11.12 Placeholder text

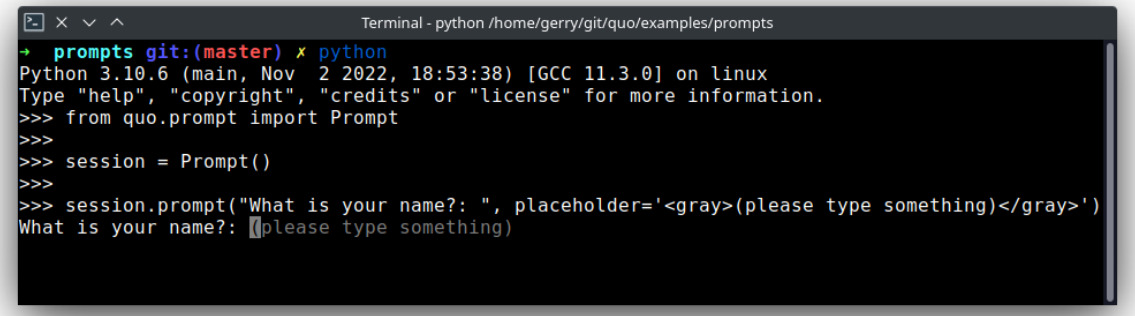
A placeholer is a text that's displayed as long as no input is given. This won't be returned as part of the output. This can be a string, formatted text or a callable that returns formatted text.

11.12.1 Plain text placeholder

```
from quo.prompt import Prompt

session = Prompt()

session.prompt("What is your name?: ", placeholder="..(please type something)")
```

11.12.2 Formatted text placeholder

```
from quo.prompt import Prompt

session = Prompt()
session.prompt("What is your name?: ", placeholder='<gray>(please type something)</gray>
↪')
```



11.13 Colors

By default, a neutral built-in color syntax is used, but any style instance can be passed to the `Prompt` class.

Note: `quo.prompt()` has different semantics and cannot output colored text but `quo.prompt.Prompt` has several ways on how this can be achieved.

11.13.1 Plain text prompt

```
from quo.prompt import Prompt

session = Prompt()

session.prompt("What is your name?: ")
```

11.13.2 Formatted text prompt

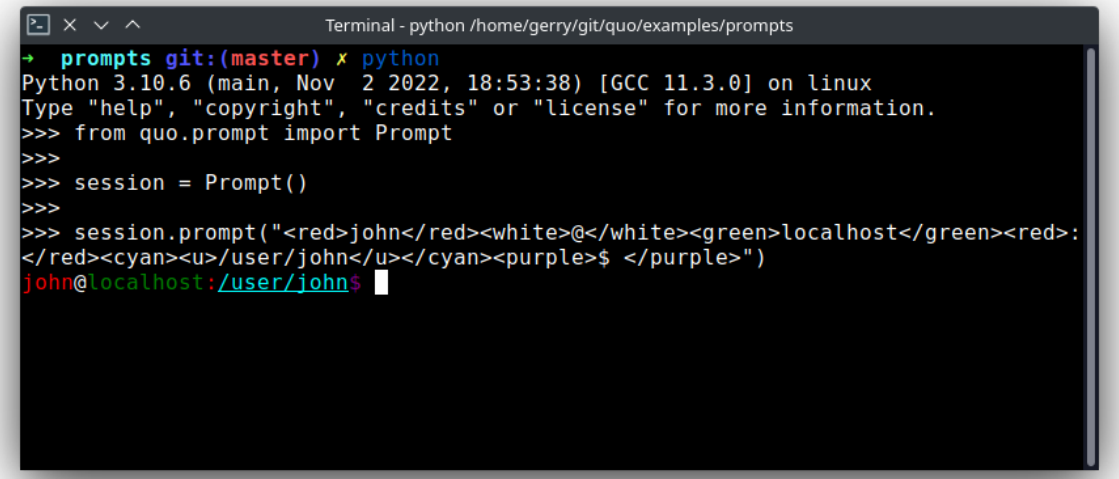
added on v2023.2

It is possible to add some colors to the prompt itself. In the following example, the prompt will be in green. *added on v2023.2*

```
from quo.prompt import Prompt

session = Prompt()
session.prompt("<green>What is your name?: </green>")
```

11.13.3 Styled prompt

```
from quo.prompt import Prompt

session = Prompt()
session.prompt("<red>john</red><white>@</white><green>localhost</green><red>:</red><cyan>
↪<u>/user/john</u></cyan><purple>$ </purple>")
```

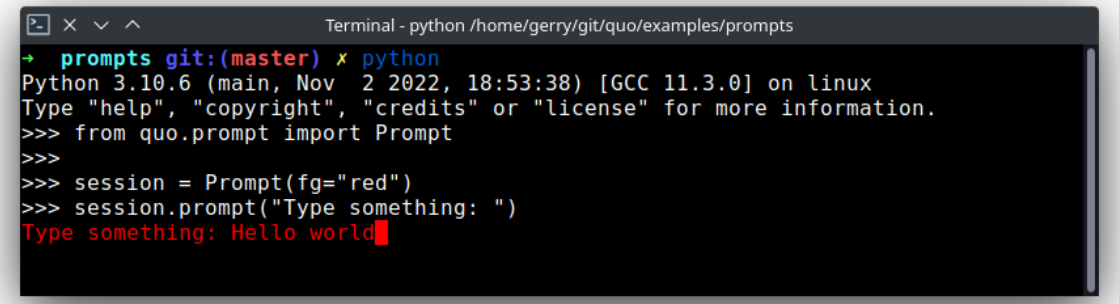


11.13.4 Coloring the prompt and the input

It is possible to add some colors to the prompt itself and the input. In the following example, the prompt and the input will be in red

version changed 2023.2

```
from quo.prompt import Prompt

session = Prompt(fg="red")
session.prompt("Type something: ")
```



fg and bg parameters added on version 2023.2 .. code:: python

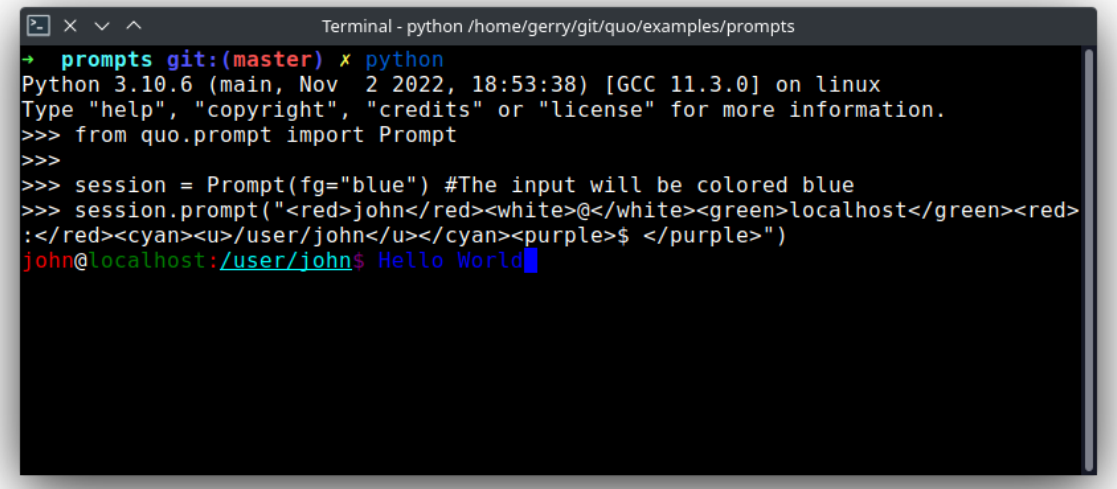
from quo.prompt import Prompt

session = Prompt(fg="red", bg="green")

session.prompt("Type something: ")

Here's an example upgrade:

```
from quo.prompt import Prompt

session = Prompt(fg="blue") #The input will be colored blue

session.prompt("<red>john</red><white>@</white><green>localhost</green><red>:</red><cyan>
↪<u>/user/john</u></cyan><purple>$ </purple>")
```



The *message* can be any kind of formatted text, as discussed here. It can also be a callable that returns some formatted text.

By default, colors are taken from the 256 color palette. If you want to have 24-bit true color, this is possible by adding

the `color_depth=ColorDepth.TRUE_COLOR` option to the `Prompt` .

```
from quo.prompt import Prompt
from quo.color import ColorDepth

session = Prompt(color_depth=ColorDepth.TRUE_COLOR)

session.prompt("<style fg='red' bg='blue'>What is your name:? </style>")
```

11.14 Completion

11.14.1 Auto suggestion

Auto suggestion is a way to propose some input completions to the user like the fish shell.

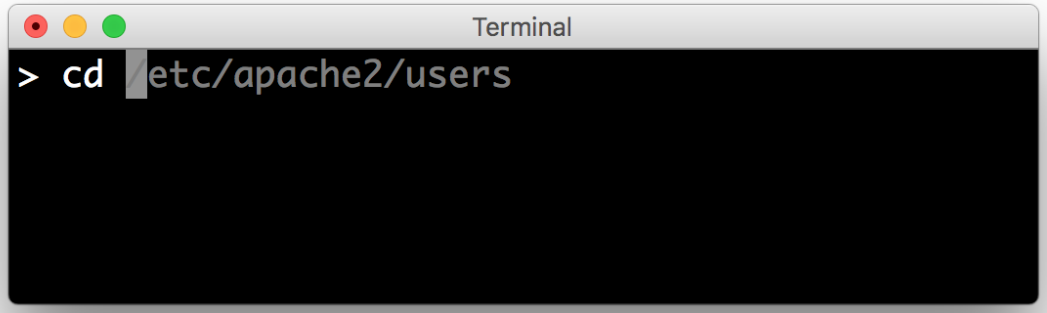
Usually, the input is compared to the history and when there is another entry starting with the given text, the completion will be shown as gray text behind the current input. Pressing the right arrow → or `ctrl-e` will insert this suggestion, `alt-f` will insert the first word of the suggestion.

Added :param: `suggest` on v2022.5

Note: When suggestions are based on the history, don't forget to share one `History` object between consecutive prompt calls. Using a `Prompt`

Example:

```
from quo.prompt import Prompt
from quo.history import MemoryHistory

MemoryHistory.append("import os")
MemoryHistory.append('print("hello")')
MemoryHistory.append('print("world")')
MemoryHistory.append("import path")

session = Prompt(history=MemoryHistory, suggest="history")
while True:
      text = session.prompt('> ')
      print(f"You said: {text}")
```



A suggestion does not have to come from the history. Any implementation of the `AutoSuggest` abstract base class can be passed as a string i.e *history*, *dynamic* or *conditional*

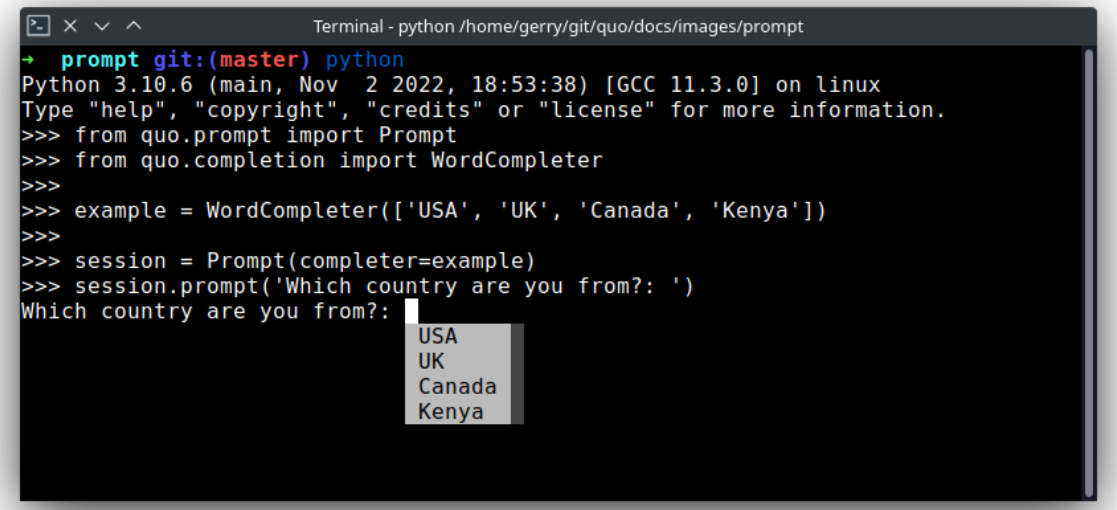
11.14.2 Autocompletion

Autocompletion can be added by passing a completer parameter.

Press [Tab] to autocomplete

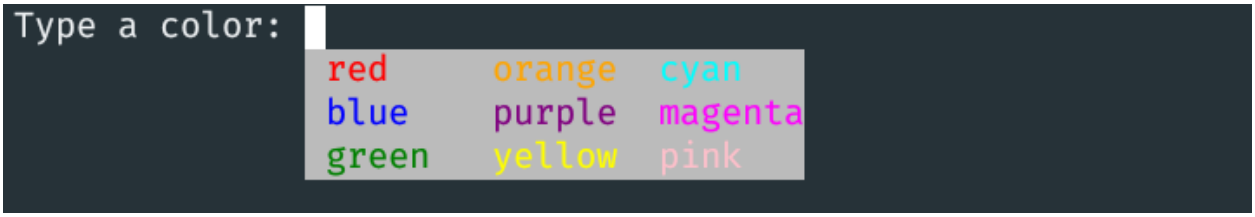
```
from quo.prompt import Prompt
from quo.completion import WordCompleter

example = WordCompleter(['USA', 'UK', 'Canada', 'Kenya'])
session = Prompt(completer=example)
session.prompt('Which country are you from?: ')
```

`WordCompleter` is a simple completer that completes the last word before the cursor with any of the given words.



Demonstration of a custom completer class and the possibility of styling completions independently.

```
from quo.completion import Completer, Completion
from quo.prompt import Prompt

colors = [
    "red",
    "blue",
    "green",
    "orange",
    "purple",
    "yellow",
    "cyan",
    "magenta",
    "pink",
    ]

class ColorCompleter(Completer):
    def get_completions(self, document, complete_event):
         word = document.get_word_before_cursor()
         for color in colors:
             if color.startswith(word):
                 yield Completion(
                          color,
                          start_position=-len(word),
                          style="fg:" + color,
                          selected_style="fg:white bg:" + color,
                          )

session = Prompt(completer=ColorCompleter(), complete_style="multi_column")
session.prompt("Type a color: ")
```



11.14.3 Nested completion

Sometimes you have a command line interface where the completion depends on the previous words from the input. Examples are the CLIs from routers and switches. A simple `WordCompleter` is not enough in that case. We want to to be able to define completions at multiple hierarchical levels. `NestedCompleter` solves this issue:

```
from quo.prompt import Prompt
from quo.completion import NestedCompleter

completer = NestedCompleter.add({
    'show': {
        'version': None,
        'clock': None,
        'ip': {
            'interface': {'brief'}
        }
    },
    'exit': None
 })
 session = Prompt(completer=completer)
 session.prompt('# ')
```

Whenever there is a `None` value in the dictionary, it means that there is no further nested completion at that point. When all values of a dictionary would be `None`, it can also be replaced with a set.

11.14.4 Complete while typing

Autcompletions can be generated automatically while typing or when the user presses the tab key. This can be configured with the `complete_while_typing` option:

```
session.prompt('Enter HTML: ', completer=completer, complete_while_typing=True)
```

Notice that this setting is incompatible with the `enable_history_search` option. The reason for this is that the up and down key bindings would conflict otherwise. So, make sure to disable history search for this.

11.15 History

A `History` object keeps track of all the previously entered strings, so that the up-arrow can reveal previously entered items.

11.15.1 MemoryHistory

The recommended way is to use a `Prompt`, which uses an `MemoryHistory` which has ^ (up) arrow partial string matching enabled by default.

```
from quo.history import MemoryHistory
from quo.prompt import Prompt

MemoryHistory.append("import os")
MemoryHistory.append('print("hello")')
MemoryHistory.append('print("world")')
MemoryHistory.append("import path")

session = Prompt(history=MemoryHistory)

while True:
    session.prompt()
```

11.15.2 FileHistory

To persist a history to disk, use a `FileHistory` instead of the default `MemoryHistory`. This history object can be passed to a `Prompt`. For instance:

```
from quo.history import FileHistory
from quo.prompt import Prompt

history = FileHistory("~/.myhistory")
session = Prompt(history=history)

while True:
    session.prompt()
```

11.16 Adding custom key bindings

By default, every prompt already has a set of key bindings which implements the usual Vi or Emacs behaviour. We can extend this by passing `quo.keys.bind()` which is an instance of `Bind`.

Note: `quo.prompt()` function does not support key bindings but `quo.prompt.Prompt` does

---

An example of a prompt that prints `'hello world'` when `Control-T` is pressed.

```
from quo import print
from quo.keys import bind
from quo.prompt import Prompt

@bind.add('ctrl-t')
def _(event):
# Print `Hello, World!` when `ctrl-t` is pressed."
    print("Hello, World!")

   @bind.add('ctrl-x')
def _(event):
  #Exit when `ctrl-x` is pressed. "
    event.app.exit()

session = Prompt()

session.prompt('> ')
```

11.16.1 Conditional Key bindings

Often, some key bindings can be enabled or disabled according to a certain condition. For instance, the Emacs and Vi bindings will never be active at the same time, but it is possible to switch between Emacs and Vi bindings at run time.

In order to enable a key binding according to a certain condition, we have to pass it a `Condition` instance. (Read more about filters.)

```
import datetime
from quo.filters import Condition
from quo.keys import bind
from quo.prompt import Prompt

@Condition
def second_half():
    " Only activate key binding on the second half of each minute. "
    return datetime.datetime.now().second > 30

@bind.add('ctrl-t', filter=second_half)
def _(event):
    # ...
    pass
session = Prompt()
session.prompt('> ')
```

11.16.2 Toggle visibility of input

Display asterisks instead of the actual characters with the addition of a ControlT shortcut to hide/show the input.

```
from quo.filters import Condition
from quo.keys import bind
from quo.prompt import Prompt

hidden = [True]  # Nonlocal

@bind.add("ctrl-t")
def _(event):
    "When ControlT has been pressed, toggle visibility."
    hidden[0] = not hidden[0]

session = Prompt(hide=Condition(lambda : hidden[0]))
session.prompt( "Password: ")
```

11.17 Mouse support

There is limited mouse support for positioning the cursor, for scrolling (in case of large multiline inputs) and for clicking in the autocompletion menu.

Enabling this can be done by passing the `mouse_support=True` option.

```
from quo.prompt import Prompt

session = Prompt(mouse_support=True)
session.prompt('What is your name: ')
```

11.18 Line wrapping

Line wrapping is enabled by default. This is what most people are used to and this is what GNU Readline does. When it is disabled, the input string will scroll horizontally.

```
from quo.prompt import Prompt

session = Prompt(wrap_lines=False)
session.prompt('What is your name: ')
```

» Check out more examples here

CHAPTER

TWELVE

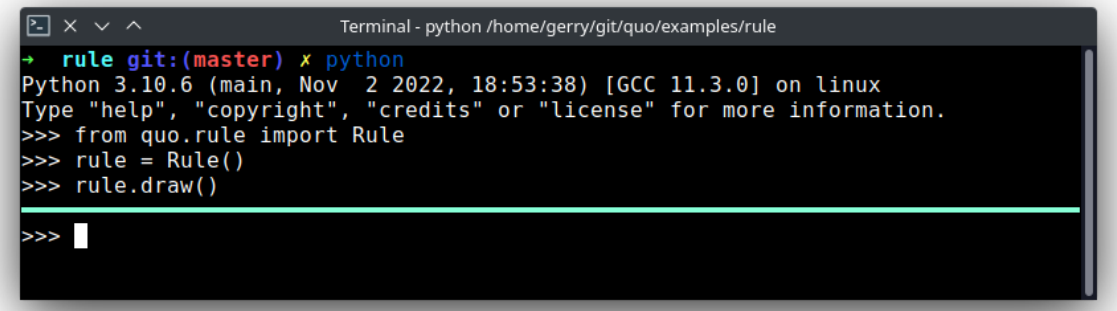
RULE

The `Rule` method will draw a horizontal line. *Added on v2023.1*

Parameters

- `char` Optional[*(str)*] - Character to be used to draw out the border.
- `lines` Optional[*(int)*] - Indicates the number of lines to be drawn.
- `multicolored` Optional[*(bool)*] - If True, a multicolored border will be applied.
- `color` Optional[*(str)*] - Color to be applied.

```
from quo.rule import Rule

rule = Rule()

rule.draw()
```

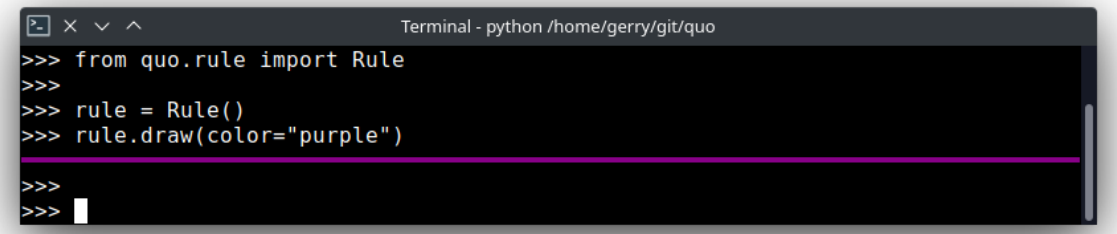


- Multicolored

```
from quo.rule import Rule

rule = Rule()

rule.draw(multicolored=True)
```

- Styled

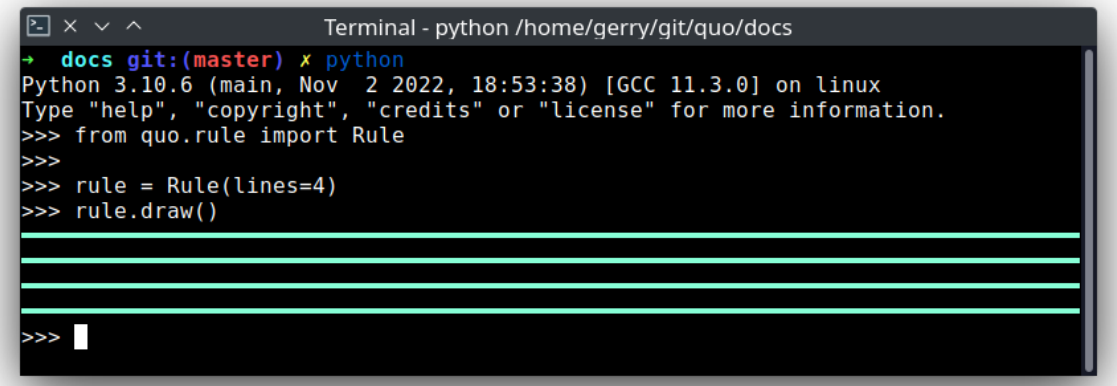
```
from quo.rule import Rule

rule = Rule()
rule.draw(color="purple")
```



- Multiline

Added on v2023.3

```
from quo.rule import Rule

rule = Rule(lines=4)

rule.draw()
```


Terminal - python /home/gerry/git/quo/docs
→ docs git:(master) ✗ python
Python 3.10.6 (main, Nov 2 2022, 18:53:38) [GCC 11.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from quo.rule import Rule
>>>
>>> rule = Rule(lines=4)
>>> rule.draw()
>>>

CHAPTER

THIRTEEN

TABLE

13.1 Printing tabular data

`quo.table.Table` function offers a number of configuration options to set the look and feel of the table, including how borders are rendered and the style and alignment of the columns.

Parameters

- `data` - The first required argument. Can be a list-of-lists *(or another iterable of iterables)*, a list of named tuples, a dictionary of iterables, an iterable of dictionaries, a two-dimensional NumPy array, NumPy record array, or a Pandas' dataframe.
- `align` - `WindowAlign` value or callable that return an `WindowAlign` value. alignment of content. i.e `left`, `centre` or `right`. `centre` is the default value.
- `style` - A style string.
- **`theme` - plain - Separates columns with a double space.**
  - **simple** - like Pandoc simple_tables.
  - **grid** - similar to tables produced by Emacs table.el package.
  - **fancy_grid** - *(Default theme)* draws a grid using box-drawing characters.
  - **pipe** - Like tables in PHP Markdown Extra extension.
  - **orgtbl** - Like tables in Emacs org-mode and orgtbl-mode.
  - **latex** - Produces a tabular environment of LaTeX document markup.
  - **presto** - Like tables produce by the Presto CLI.
  - **mediawiki** - Produces a table markup used in Wikipedia and on other MediaWiki-based sites.
  - **rst** - Like a simple table format from reStructuredText.

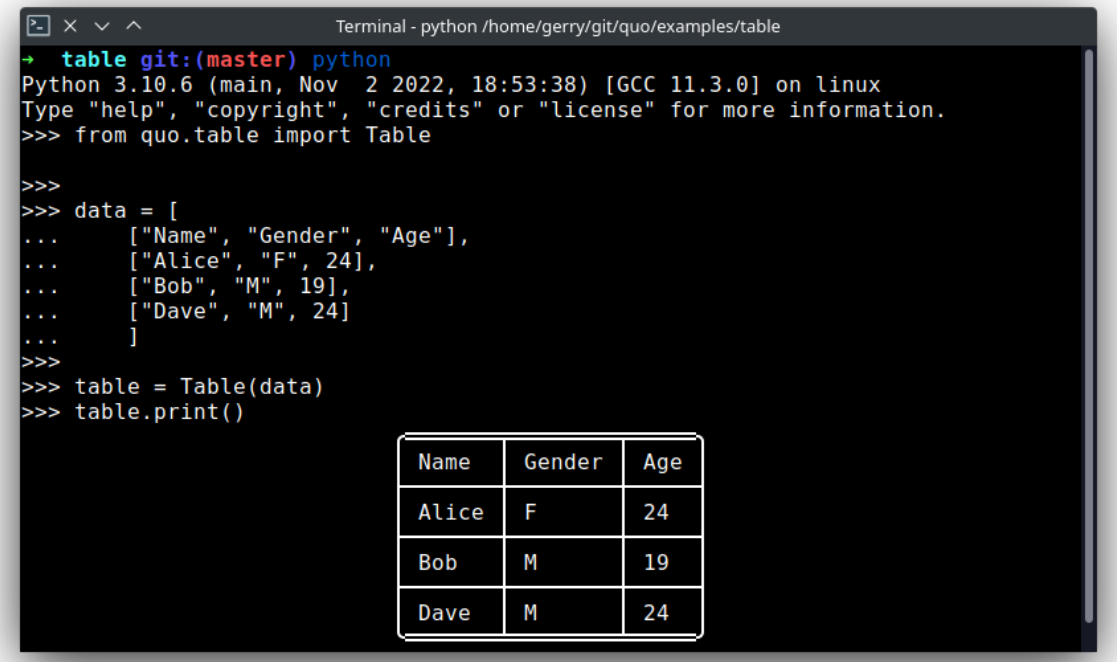
Changed on *v2022.4.3*

```
from quo.table import Table

data = [
["Name", "Gender", "Age"],
["Alice", "F", 24],
["Bob", "M", 19],
["Dave", "M", 24]
]
```

(continues on next page)

(continued from previous page)

```
table = Table(data)

table.print()
```



13.2 Table headers

To print nice column headers, supply the `headers` argument.

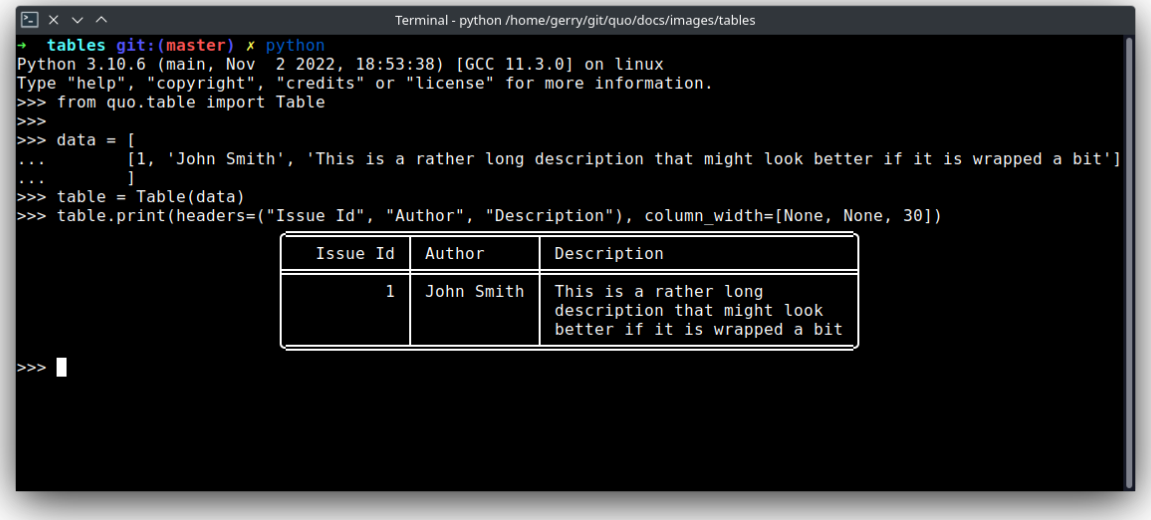
- *headers* can be an explicit list of column headers.
- if *headers="firstrow"*, then the first row of data is used
- if *headers="keys"*, then dictionary keys or column indices are used otherwise a headerless table is produced.

```
from quo.table import Table

data = [
   ["Name", "Gender", "Age"],
   ["Alice", "F", 24],
   ["Bob", "M", 19],
   ["Dave", "M", 24]
   ]

table = Table(data)
table.print(headers="firstrow")
```

13.3 Column Widths and Line Wrapping

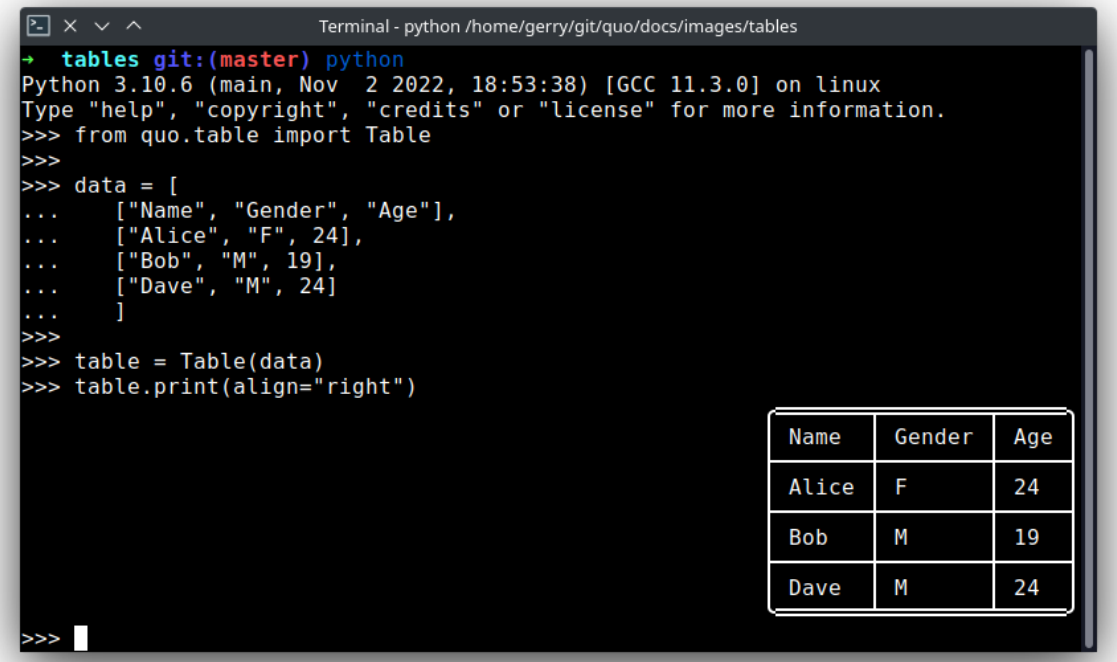
`Table()` will, by default, set the width of each column to the length of the longest element in that column. However, in situations where fields are expected to reasonably be too long to look good as a single line, **:param:`column_width`** can help automate word wrapping long fields.

```
from quo.table import Table

data = [
      [1, 'John Smith', 'This is a rather long description that might look better if it␣
↪is wrapped a bit']
      ]

table = Table(data)
table.print(headers=("Issue Id", "Author", "Description"), column_width=[None, None, 30])
```



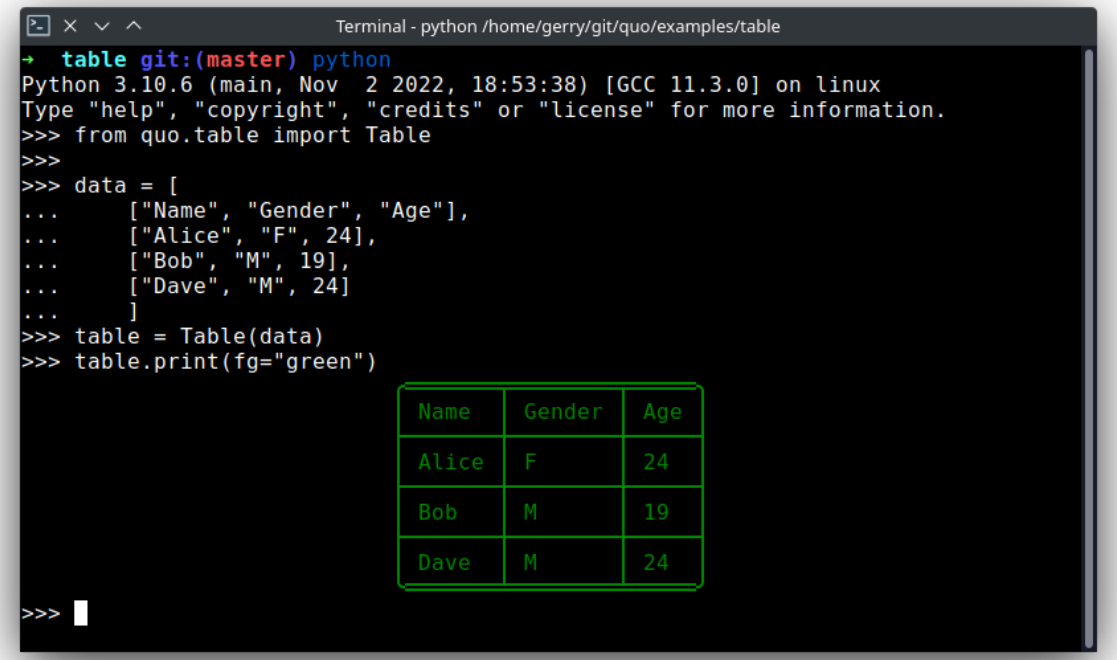
Right aligned table

```
from quo.table import Table

data = [
   ["Name", "Gender", "Age"],
   ["Alice", "F", 24],
   ["Bob", "M", 19],
   ["Dave", "M", 24]
   ]

table = Table(data)
table.print(align="right")
```



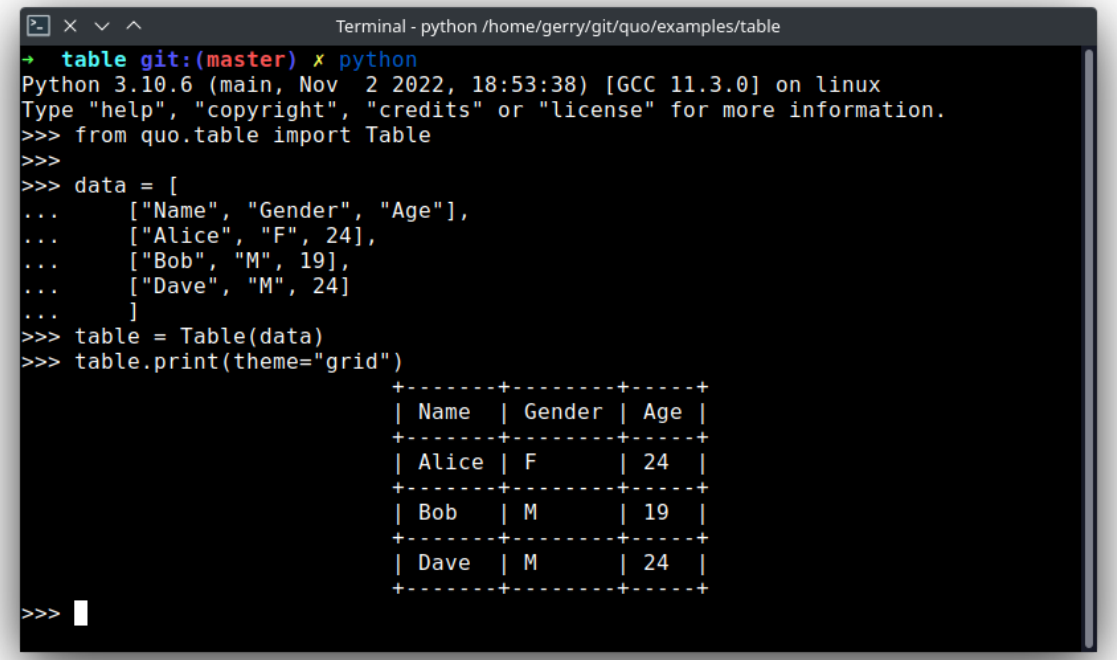
Colored table

```
from quo.table import Table

data = [
   ["Name", "Gender", "Age"],
   ["Alice", "F", 24],
   ["Bob", "M", 19],
   ["Dave", "M", 24]
   ]

table = Table(data)
table.print(fg="green")
```



Grid table

```
from quo.table import Table

data = [
   ["Name", "Gender", "Age"],
   ["Alice", "F", 24],
   ["Bob", "M", 19],
   ["Dave", "M", 24]
   ]

table = Table(data)
table.print(theme="grid")
```


Terminal - python /home/gerry/git/quo/examples/table
→ table git:(master) ✗ python
Python 3.10.6 (main, Nov 2 2022, 18:53:38) [GCC 11.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from quo.table import Table
>>>
>>> data = [
... ["Name", "Gender", "Age"],
... ["Alice", "F", 24],
... ["Bob", "M", 19],
... ["Dave", "M", 24]
...]
>>> table = Table(data)
>>> table.print(theme="grid")
+-------+--------+-----+
| Name | Gender | Age |
+-------+--------+-----+
| Alice | F | 24 |
+-------+--------+-----+
| Bob | M | 19 |
+-------+--------+-----+
| Dave | M | 24 |
+-------+--------+-----+
>>>

CHAPTER

FOURTEEN

WIDGETS

A collection of reusable components for building full screen applications.

When in `full_screen` mode, the default key binder to exit the application is `Ctrl-C`, however you can set your own.

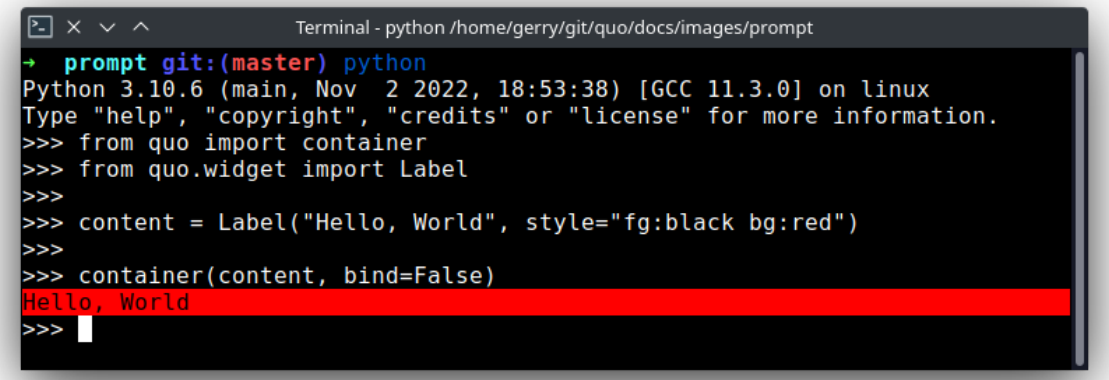
14.1 Frame

Draw a border around any container, optionally with a title text. Changing the title and body of the frame is possible at runtime by assigning to the *body* and *title* attributes of this class.

Parameters

- `body` - Another container object.
- `title` - Text to be displayed in the top of the frame *(can be formatted text)*
- `style` - Style string to be applied to this widget.
- `width` - Frame width
- `height` - Frame height.

```
from quo import container
from quo.widget import Frame, Label


root = Frame(
          Label("Hello, World!"),
          title="Quo: python")

@bind.add("ctrl-c")
def _(event):
       event.app.exit()

container(root, bind=True, full_screen=True)
```



14.2 Box

Add padding around a container. This also makes sure that the parent can provide more space than required by the child. This is very useful when wrapping a small element with a fixed size into a `VSplit` or `HSplit` object.

Parameters

- `body` - Another container object.
- `padding` - The margin to be used around the body. This can be overridden by **:param:`padding_left`**, **:param:`padding_right`**, **:param:`padding_top`** and **:param:`padding_bottom`** parameters.
- `fg` *(Optional[str])* - A foregound color string.
- `bg` *(Optional[str])* - A background color string.
- `char` *(Optional[str])* - Character to be used for filling the space around the body. *(This is supposed to be a character with a terminal width of 1.)*

```
from quo import container
from quo.box import Box
from quo.keys import bind
from quo.label import Label

label = Label("<fg='black' bg='red'>Hello, World</style>")

content = Box(label, padding=5)

# Press `q` to cancel
@bind.add("q")
def _(event):
  event.app.exit()

container(content, bind=True, full_screen=True)
```

14.3 Label

Widget that displays the given text. It is not editable or focusable.

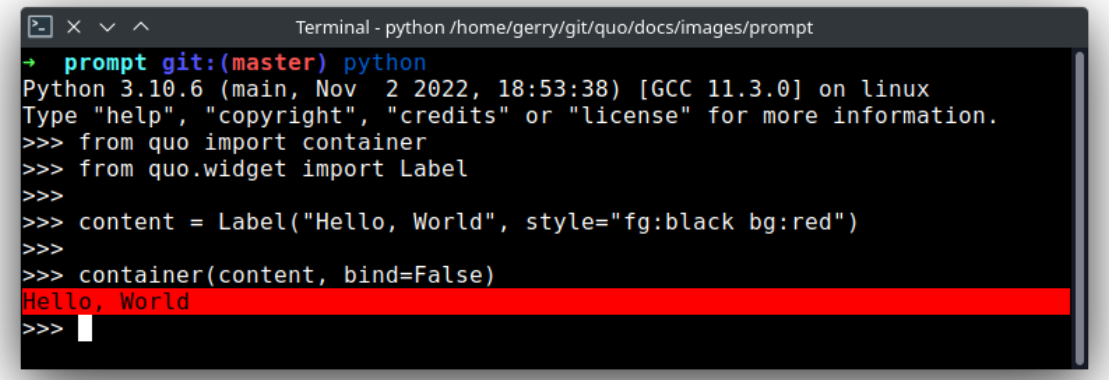
Parameters

- `text` *(str)* - Text to display. Can be multiline.
- `width` *(int)*- When given, use this width, rather than calculating it from the text size.
- `bold` *(bool)* - Bold text.
- `italic` *(bool)*- Italic text.
- `underline` *(bool)* - Underline text.
- `fg` *(str)* - Foreground text color.
- `bg` *(str)* - Background text color.
- `fixed_width` *(bool)* - When *False*, don't take up more width than preferred, i.e. the length of the longest line of the text, or value of *width* parameter, if given. *True* by default
- `fixed_height` *(bool)*- When *False*, don't take up more width than the preferred height, i.e. the number of lines of the text. *True* by default.

(Changed on v2023.3)

You can print the layout to the output in a non-interactive way like so:

```
from quo import container
from quo.label import Label

content = Label("Hello, World", fg='black', bg='red')
container(content)
```



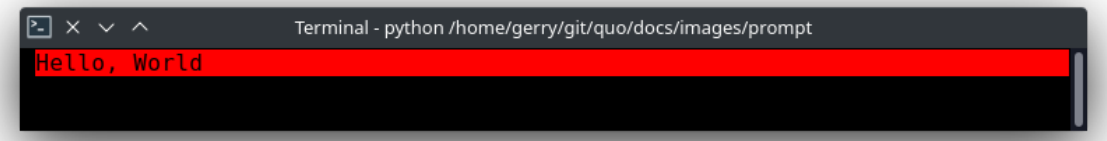
To make it fullscreen set **:param:`bind`** and **:param:`full_screen`** to `True` Press `Ctrl-C` to quit

```
from quo import container
from quo.label import Label
```

(continues on next page)

(continued from previous page)

```
content = Label("<fg='black' bg='red'>Hello, World</style>")
container(content, bind=True, full_screen=True)
```



Adding a custom key binder

```
from quo import container
from quo.keys import bind
from quo.label import Label

content = Label("<fg='black' bg='red'>Hello, World</style>")

#Press Ctrl-a to exit
@bind.add("ctrl-a")
def _(event):
   event.app.exit()

container(content, bind=True, full_screen=True)
```

14.4 TextField

A simple input field. This is a higher level abstraction on top of several other classes with sane defaults.

This widget does have the most common options, but it does not intend to cover every single use case.

Parameters - `text` *(str)* - The initial text. - `prompt` *(Optional[TextFieldFormattedText, str])* - Prompt. ie *<blue>What is your name?</blue>* - `multiline` *(bool)* - If True, allow multiline input. - `completer` - `Completer` instance for auto completion. - `complete_while_typing` - Boolean. - `accept_handler` - Called when *Enter* is pressed *(This should be a callable that takes a buffer as input)*. - `history` - `History` instance. - `auto_suggest` - `AutoSuggest` instance for input suggestions. - `hide` *(bool)* - When *True*, display using asterisks. - `focusable` *(bool)* - When *True*, allow this widget to receive the focus. - `focus_on_click` *(bool)* - When *True*, focus after mouse click. - `input_processors` - *None* or a list of `Processor` objects. - `type` - *None* or a `Validator` object. - `highlighter` - `Lexer` instance for syntax highlighting. - `wrap_lines` *(bool)* - When *True*, don't scroll horizontally, but wrap lines. - `width` - Window width. (`Dimension` object.) - `height` - Window height. (`Dimension` object.) - `scrollbar` *(bool)* - When *True*, display a scroll bar. - `fg` *(Optional[str])* - A foregound color string. - `bg` *(Optional[str])* - A background color string. - `fixed_width` *(bool)* - When *True*, don't take up more width than the preferred width reported by the control. - `fixed_height` *(bool)* - When *True*, don't take up more width than the preferred height reported by the control. - `get_line_prefix` - None or a callable that returns formatted text to be inserted before a line. It takes a line number *(int)* and a wrap_count and returns formatted text. This can be used for implementation of line continuations, things like Vim "breakindent" and so on.

14.4.1 Other attributes

- `search_field` - An optional *SearchToolbar* object.

14.5 Button

Clickable button.

Parameters

- `text` - The caption for the button.
- `handler` - *None* or callable. Called when the button is clicked. No parameters are passed to this callable. Use for instance Python's *functools.partial* to pass parameters to this callable if needed.
- `width` - Width of the button.

14.6 Shadow

Draw a shadow underneath/behind this container. *(This applies `class:shadow` the the cells under the shadow. The Style should define the colors for the shadow.)*

Parameters

- `body` - Another container object.

CHAPTER

FIFTEEN

UTILITIES

15.1 Screen Clearing

To clear the terminal screen, you can use the `quo.clear()` function. It does what the name suggests: it clears the entire visible screen in a platform-agnostic way:

```
from quo import clear

clear()
```

15.2 Getting Characters from Terminal(getchar)

Normally, when reading input from the terminal, you would read from standard input. However, this is buffered input and will not show up until the line has been terminated. In certain circumstances, you might not want to do that and instead read individual characters as they are being written.

For this, Quo provides the `getchar()` function which reads a single character from the terminal buffer and returns it as a Unicode character.

Note that this function will always read from the terminal, even if stdin is instead a pipe.

```
from quo import getchar

gc = getchar()

if gc == 'y':
    print('We will go on')
elif gc == 'n':
    print('Abort!')
```

Note that this reads raw input, which means that things like arrow keys will show up in the platform's native escape format. The only characters translated are `^C` and `^D` which are converted into keyboard interrupts and end of file exceptions respectively. This is done because otherwise, it's too easy to forget about that and to create scripts that cannot be properly exited.

15.3 Exitting

Quo has a low-level exit that skips Python's cleanup and speeds up exit by about 10ms for things like shell completion.
Parmameters

- `code` *(str)* - Exit code.

```
from quo import exit

exit(1)
```

15.4 Waiting for Key Press(pause)

Sometimes, it's useful to pause until the user presses any key on the keyboard.

In quo, this can be accomplished with the `quo.pause()` function. This function will print a quick message to the terminal (which can be customized) and wait for the user to press a key. In addition to that, it will also become a NOP (no operation instruction) if the script is not run interactively.

Parameters

- `info` *(Optional[str])* – The message to print before pausing. Defaults to "Press any key to proceed >> ..".

```
from quo import pause

pause()
```

CHAPTER

SIXTEEN

EXCEPTION(ERROR) HANDLING

Quo internally uses exceptions to signal various error conditions that the user of the application might have caused. Primarily this is things like incorrect usage.

16.1 Where are Errors Handled?

Quo's main error handling is happening in `BaseCommand.main()`. In there it handles all subclasses of `Outlier` as well as the standard `EOFError` and `KeyboardInterrupt` exceptions. The latter are internally translated into a `Abort`.

The logic applied is the following:

1. If an `EOFError` or `KeyboardInterrupt` happens, reraise it as `Abort`.
2. If an `Outlier` is raised, invoke the `Outlier.show()` method on it to display it and then exit the program with `Outlier.exit_code`.
3. If an `Abort` exception is raised print the string `Aborted!` to standard error and exit the program with exit code `1`.
4. if it goes through well, exit the program with exit code `0`.

16.2 Which Exceptions Exist?

Quo has two exception bases: `Outlier` which is raised for all exceptions that quo wants to signal to the user and `Abort` which is used to instruct quo to abort the execution.

A `Outlier` has a `show()` method which can render an error message to stderr or the given file object. If you want to use the exception yourself for doing something check the API docs about what else they provide.

The following common subclasses exist:

- `UsageError` to inform the user that something went wrong.
- `BadParameter` to inform the user that something went wrong with a specific parameter. These are often handled internally in quo and augmented with extra information if possible. For instance if those are raised from a callback quo will automatically augment it with the parameter name if possible.
- `FileError` this is an error that is raised by the `FileType` if quo encounters issues opening the file.
- `ValidationError` if quo encounters issues validating an input.

CHAPTER

SEVENTEEN

TEXT USER INTERFACE (FULL SCREEN APPLICATIONS)

quo can be used to create complex full screen terminal applications. Typically, an application consists of a layout (to describe the graphical part) and a set of key bindings.

The sections below describe the components required for full screen applications (or custom, non full screen applications), and how to assemble them together.

Note: Also remember that the `examples` directory of the quo repository contains plenty of examples. Each example is supposed to explain one idea. So, this as well should help you get started.

Don't hesitate to open a GitHub issue if you feel that a certain example is missing.

17.1 A simple application

Almost every quo application is an instance of an `container()`. The simplest full screen example would look like this:

```
from quo import container
from quo.label import Label

content = Label("Hello, world")

container(content)
```

This will only consume the least amount of space required.

Note:

If we set the `full_screen` option, the application will run in an alternate screen buffer, in full screen mode.

Starting with v2022.4.5, `ctrl-c` will be the default key binder for to exit the app, you will still be able to define your own set of key bindings.

```
from quo import container
from quo.textfield import TextField

content = TextField("Hello, world")
container(content, bind=True, full_screen=True)
```

An application consists of several components. The most important are:

- I/O objects: the input and output device.
- The layout: this defines the graphical structure of the application. For instance, a text box on the left side, and a button on the right side.
- A style: this defines what colors and underline/bold/italic styles are used everywhere.
- A set of key bindings.

We will discuss all of these in more detail below.

17.2 The layout

Under the hood, class `Layout` is the layout for function `container()`.

- Here's a simple example of a a text area displaying *Hello World!*

```
from quo import container
from quo.box imort Box
from quo.textfield TextField

# Layout for displaying hello world.
# (The box takes care of the margin/padding.)

textfield = TextField("Hello, world!!")

content = Box(textfield)

container(content, bind=True, full_screen=True)
```

In the example above, the Layout consists of `Box` and `TextField` for displaying hello world.

The class `Box` takes care of the margin/padding and class `TextField` takes care of the text to be printed. `quo.container()` prints the layout.

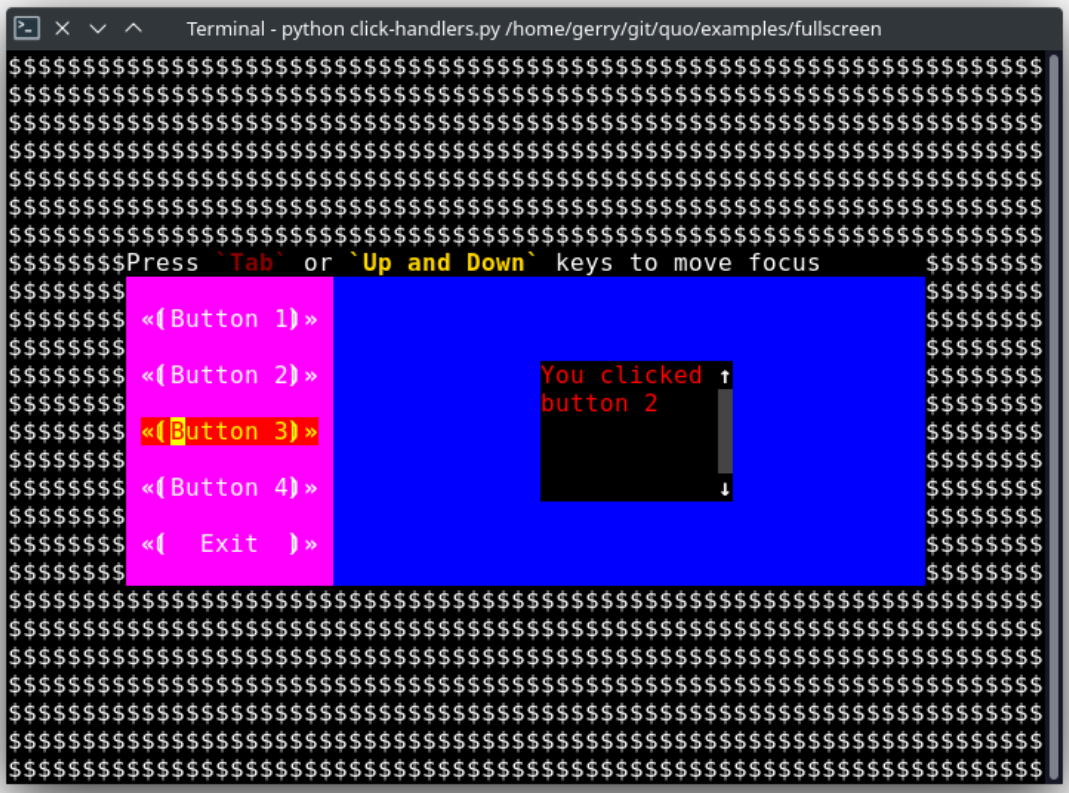
17.2.1 container

Print the layout to the output

Parameters

- `container` - AnyContainer
- `bind` *(bool)* - When True, initiate a `Bind` instance for the key bindings.
- `full_screen` *(bool)* - When True, run the application on the alternate screen buffer.
- `focused_element` - element to be focused initially. *(Can be anything the `focus` function accepts.)*
- `mouse_support` - `Filter` or boolean. When True, enable mouse support.
- `style` - A style string.

Here's a simple example of a few buttons and click handlers.



» Source code here

17.2.2 A layered layout architecture

There are several ways to create a layout, depending on how customizable you want things to be.

> Examples of `Container` objects are `VSplit` (vertical split), `HSplit` (horizontal split)
>
> `Window` object is a special kind of container that can contain objects responsible for the generation of content. The `Window` object acts as an adaptor between the `UIControl` and other containers, but it's also responsible for the scrolling and line wrapping of the content.

Quo contains several widgets like: `Button`, `Frame`, `Label`, `TextField`,

- The highest level abstractions can be found in the `dialog` module.

More complex layouts can be achieved by nesting multiple `VSplit`, `HSplit`

17.3 HSplit

Several layouts, one stacked above/under the other. like so:

```
+--------------------+
|                    |
+--------------------+
|                    |
+--------------------+
```

By default, this doesn't display a horizontal line between the children, but if this is something you need, then create a HSplit as follows:

```
HSplit(subset=[ ... ], padding_char='-', padding=1, padding_style='fg:red')
```

Parameters

- `subset` - List of child `Container` objects.
- `window_too_small` - A `Container` object that is displayed if there is not enough space for all the subsets. By default, this is a "Window too small" message.
- `align` - A *VerticalAlign* value. i.e `top`, `center`, `bottom` or `justify`
- `width` - When given, use this width instead of looking at the subsets.
- `height` - When given, use this height instead of looking at the subsets.
- `z_index`- (int or None) When specified, this can be used to bring element in front of floating elements. *None* means: inherit from parent.
- `style` - A style string.
- `modal` *(bool)* - Setting `modal=True` makes what is called a **modal** container. Normally, a subset container would inherit its parent key bindings. This does not apply to **modal** containers.
- `bind` - `None` or a `Bind` object.
- `padding` - (*Dimension* or int), size to be used for the padding. - `padding_char` - Character to be used for filling in the padding.
- `padding_style` - Style to applied to the padding.

```
from quo import container
from quo.layout import HSplit
from quo.window import Window
from quo.label import Label

# 1. The layout
content = HSplit([
       Label("\n\n(Top pane)"),
       Window(height=1, char="-"),  # Horizontal line in the middle.
       Label("\n\n(Bottom pane)")
       ])

# 2. The `Application`
# Press `ctrl-c` to exit
container(content, bind=True)
```

17.4 VSplit

Several layouts, one stacked left/right of the other like so:

```
+---------+----------+
|         |          |
|         |          |
+---------+----------+
```

By default, this doesn't display a vertical line between the children, but if this is something you need, then create a VSplit as follows:

```
VSplipt([ ... ], padding_char='|', padding=1, padding_style='fg:blue')
```

Parameters

- `subset` - List of subsets `Container` objects.
- `window_too_small` - A `Container` object that is displayed if there is not enough space for all the children. By default, this is a "Window too small" message.
- `align`- A *HorizontalAlign* value. i.e `left`, `centre`, `right` or `justify`
- `width` - When given, use this width instead of looking at the subsets.
- `height` - When given, use this height instead of looking at the subsets.
- `z_index` - (int or None) When specified, this can be used to bring element in front of floating elements. *None* means: inherit from parent.
- `style` - A style string.
- `modal` *(bool)* - Setting `modal=True` makes what is called a **modal** container. Normally, a subset container would inherit its parent key bindings. This does not apply to **modal** containers.
- `bind` - `None` or a `Bind` object.
- `padding` - (*Dimension* or int), size to be used for the padding.
- `padding_char` - Character to be used for filling in the padding.
- `padding_style` - Style to applied to the padding.

```
# Press `ctrl-c` to exit
from quo import container
from quo.label import Label
from quo.layout import VSplit
from quo,window import Window


# 1. The layout
content = VSplit([
         Label("(Left pane)"),
         Window(width=1, char="|"), # Vertical line in the middle.
         Label("(Right pane)")
         ])

container(content, bind=True, full_screen=True)
```

17.5 Key bindings

17.5.1 Global key bindings

Key bindings can be passed to the application as follows:

```
from quo import container
from quo.keys import bind

container(bind=True)
```

17.5.2 Registering Key bindings

To register a new keyboard shortcut, we can use the `add()` method as a decorator of the key handler:

```
from quo import container
from quo.keys import bind
from quo.textfield import TextField

content = TextField("Hello, world")

# A custom Key binder to exit the application
@bind.add("ctrl-q")
def exit_(event):
      """
      Pressing "ctrl-q" will exit the user interface
      """
     event.app.exit()

container(content, bind=True, full_screen=True)
```

The callback function is named `exit_` for clarity, but it could have been named _ (underscore) as well, or anything you see fit

Read more about key bindings

VSplit and HSplit take a modal argument.

Setting modal=True makes what is called a **modal** container. Normally, a child container would inherit its parent key bindings. This does not apply to **modal** containers.

Consider a **modal** container (e.g. VSplit) is child of another container, its parent. Any key bindings from the parent are not taken into account if the **modal** container (subset) has the focus.

This is useful in a complex layout, where many controls have their own key bindings, but you only want to enable the key bindings for a certain region of the layout.

The global key bindings are always active.

17.5.3 Window

Window is a Container that wraps a UIControl, like a BufferControl or FormattedTextControl.

Parameters

- content - UIControl instance.
- width - Dimension instance or callable.
- height - Dimension instance or callable.
- z_index - When specified, this can be used to bring element in front of floating elements.
- fixed_width *(bool)* - When *True*, don't take up more width then the preferred width reported by the control.
- fixed_height *(bool)* - When *True*, don't take up more width then the preferred height reported by the control.
- ignore_content_width *(bool)* - A *bool* or Filter instance. Ignore the UIContent width when calculating the dimensions.
- ignore_content_height *(bool)* - A *bool* or Filter instance. Ignore the UIContent height when calculating the dimensions.
- left_margins - A list of Margin instance to be displayed on the left. For instance: NumberedMargin can be one of them in order to show line numbers.
- right_margins - Like *left_margins*, but on the other side.
- scroll_offsets - ScrollOffsets instance, representing the preferred amount of lines/columns to be always visible before/after the cursor. When both top and bottom are a very high number, the cursor will be centered vertically most of the time.
- allow_scroll_beyond_bottom *(bool)* - A *bool* or Filter instance. When True, allow scrolling so far, that the top part of the content is not visible anymore, while there is still empty space available at the bottom of the window. In the Vi editor for instance, this is possible. You will see tildes while the top part of the body is hidden.
- wrap_lines *(bool)** - A *bool* or Filter instance. When True, don't scroll horizontally, but wrap lines instead.
- get_vertical_scroll - Callable that takes this window instance as input and returns a preferred vertical scroll. *(When this is `None`, the scroll is only determined by the last and current cursor position.)*
- get_horizontal_scroll - Callable that takes this window instance as input and returns a preferred vertical scroll.
- always_hide_cursor *(bool)* - A *bool* or Filter instance. When True, never display the cursor, even when the user control specifies a cursor position.

- `cursorline` *(bool)* - A *bool* or `Filter` instance. When True, display a cursorline.
- `cursorcolumn` *(bool)* - A *bool* or `Filter` instance When True, display a cursorcolumn.
- `colorcolumns` - A list of `ColorColumn` instances that describe the columns to be highlighted, or a callable that returns such a list.
- `align` - `WindowAlign` value or callable that returns an `WindowAlign` value. alignment of content. i.e `left`, `centre` or `right`
- `style` - A style string. Style to be applied to all the cells in this window. *(This can be a callable that returns a string.)*
- `char` *(str)* - Character to be used for filling the background. This can also be a callable that returns a character.
- `get_line_prefix` - None or a callable that returns formatted text to atted text to be inserted before a line. It takes a line number (int) and a wrap_count and returns formatted text. This can be used for implementation of line continuations, things like Vim "breakindent".

» Check out more examples here

CHAPTER

EIGHTEEN

KEY BINDING

A key binding is an association between a physical key on a keyboard and a parameter. A parameter can have any number of key bindings associated with it, and a particular key binding can control any number of parameters.

Note: This page contains a couple of extra notes about key bindings.

Key bindings can be defined by importing `quo.keys.bind()` which is an instance of `Bind`

```
from quo.keys import bind

@bind.add('a')
def _(event):
    " Do something if 'a' has been pressed. "
    ...


@bind.add('ctrl-t')
def _(event):
    " Do something if Control-T has been pressed. "
    ...
```

Note: `ctrl-q` (control-q) and `ctrl-s` (control-s) are often captured by the terminal, because they were used traditionally for software flow control. When this is enabled, the application will automatically freeze when `ctrl-s` is pressed, until `ctrl-q` is pressed. It won't be possible to bind these keys.

In order to disable this, execute the following command in your shell, or even add it to your *.bashrc*.

```
stty -ixon
```

Key bindings can even consist of a sequence of multiple keys. The binding is only triggered when all the keys in this sequence are pressed.

```
@bind.add('q', 'u', 'o')
def _(start):
    " Do something if 'q' is pressed, then 'u' and then 'o' is pressed. "
    ...
```

If the user presses only *q*, then nothing will happen until either a second key (like *u* or *o*) has been pressed or until the timeout expires.

18.1 List of special keys

Besides literal characters, any of the following keys can be used in a key binding:

Name	Possible keys
Escape Shift + escape	`escape` `s-escape`
Arrows	`left`, `right`, `up`, `down`
Navigation	`home`, `end`, `delete`, `pageup`, `pagedown`, `insert`
Control+letter	`ctrl-a`, `ctrl-b`, `ctrl-c`, `ctrl-d`, `ctrl-e`, `ctrl-f`, `ctrl-g`, `ctrl-h`, `cttl-i`, `ctrl-j`, `ctrl-k`, `ctrl-l`, `ctrl-m`, `ctrl-n`, `ctrl-o`, `ctrl-p`, `ctrl-q`, `ctrl-r`, `ctrl-s`, `ctrl-t`, `ctrl-u`, `ctrl-v`, `ctrl-w`, `ctrl-x`, `ctrl-y`, `ctrl-z`
Control + number	`ctrl-1`, `ctrl-2`, `ctrl-3`, `ctrl-4`, `ctrl-5`, `ctrl-6`, `ctrl-7`, `ctrl-8`, `ctrl-9`, `ctrl-0`
Control + arrow	`ctrl-left`, `ctrl-right`, `ctrl-up`, `ctrl-down`
Other control keys	`ctrl-@`, `ctrl-\`, `ctrl-]`, `ctrl-^`, `ctrl-_`, `ctrl-delete`
Shift + arrow	`s-left`, `s-right`, `s-up`, `s-down`
Control + Shift + arrow	`c-s-left`, `c-s-right`, `c-s-up`, `c-s-down`
Other shift keys	`s-delete`, `s-tab`
F-keys	`f1`, `f2`, `f3`, `f4`, `f5`, `f6`, `f7`, `f8`, `f9`, `f10`, `f11`, `f12`, `f13`, `f14`, `f15`, `f16`, `f17`, `f18`, `f19`, `f20`, `f21`, `f22`, `f23`, `f24`

There are a couple of useful aliases as well:

`ctrl-h|backspace`
`ctrl-@|ctrl-space`
`ctrl-m|enter`
`ctrl-i|tab`

Note: Note that the supported keys are limited to what typical VT100 terminals offer. Binding `ctrl-7` (control + number 7) for instance is not supported.

18.2 Binding alt+something, option+something or meta+something

Vt100 terminals translate the alt key into a leading `escape` key. For instance, in order to handle `alt-f`, we have to handle `escape + f`. Notice that we receive this as two individual keys. This means that it's exactly the same as first typing `escape` and then typing `f`. Something this alt-key is also known as option or meta.

In code that looks as follows:

```
@bind.add('escape', 'f')
def _(event):
    " Do something if alt-f or meta-f have been pressed. "
```

18.3 Wildcards

Sometimes you want to catch any key that follows after a certain key stroke. This is possible by binding the '<any>' key:

```
@bind.add('a', '<any>')
def _(start):
    ...
```

This will handle *aa*, *ab*, *ac*, etcetera. The key binding can check the *event* object for which keys exactly have been pressed.

18.4 Attaching a Condition to key bindings

In order to enable a key binding according to a certain condition, we have to pass it to `Condition` instance. (Read more about filters.)

```
import datetime
from quo.filters import Condition
from quo.keys import bind

@Condition
def is_active():
    " Only activate key binding on the second half of each minute. "
    return datetime.datetime.now().second > 30

@bind.add('ctrl-t', filter=is_active)
def _(event):
    # ...
    pass
```

The key binding will be ignored when this condition is not satisfied.

18.5 ConditionalKeyBindings: Disabling a set of key bindings

Sometimes you want to enable or disable a whole set of key bindings according to a certain condition. This is possible by wrapping it in a `ConditionalKeyBindings` object.

```
from quo.filters import Condition
from quo.keys ConditionalKeyBindings

@Condition
def is_active():
    " Only activate key binding on the second half of each minute. "
    return datetime.datetime.now().second > 30

 bindings = ConditionalKeyBindings(
     bind=my_bindings,
     filter=is_active)
```

If the condition is not satisfied, all the key bindings in *my_bindings* above will be ignored.

18.6 Merging key bindings

Sometimes you have different parts of your application generate a collection of key bindings. It is possible to merge them together through the `merge_key_bindings()` function. This is preferred above passing a `Bind` object around and having everyone populate it.

```
from quo.keys import merge_key_bindings

bindings = merge_key_bindings([
    bindings1,
    bindings2,
])
```

18.7 Eager

Usually not required, but if ever you have to override an existing key binding, the *eager* flag can be useful.

Suppose that there is already an active binding for *ab* and you'd like to add a second binding that only handles *a*. When the user presses only *a*, quo has to wait for the next key press in order to know which handler to call.

By passing the *eager* flag to this second binding, we are actually saying that quo shouldn't wait for longer matches when all the keys in this key binding are matched. So, if *a* has been pressed, this second binding will be called, even if there's an active *ab* binding.

```
@bind.add('a', 'b')
def binding_1(event):
    ...

@bind.add('a', eager=True)
def binding_2(event):
    ...
```

This is mainly useful in order to conditionally override another binding.

18.8 Asyncio coroutines

Key binders handlers can be asyncio coroutines.

```
@bind.add('x')
async def print_hello(event):
    """
    Pressing 'x' will print 5 times "hello" in the background above the
    prompt.
    """
    for i in range(5):
        # Print hello above the current prompt.
        print("Hello")

        # Sleep, but allow further input editing in the meantime.
        await asyncio.sleep(1)
```

If the user accepts the input on the prompt, while this coroutine is not yet finished , an *asyncio.CancelledError* exception will be thrown in this coroutine.

18.9 Timeouts

There are two timeout settings that effect the handling of keys.

- `Application.ttimeoutlen`: Like Vim's *ttimeoutlen* option. When to flush the input (For flushing escape keys.) This is important on terminals that use vt100 input. We can't distinguish the escape key from for instance the left-arrow key, if we don't know what follows after "x1b". This little timer will consider "x1b" to be escape if nothing did follow in this time span. This seems to work like the *ttimeoutlen* option in Vim.
- `KeyProcessor.timeoutlen`: like Vim's *timeoutlen* option. This can be *None* or a float. For instance, suppose that we have a key binding AB and a second key binding A. If the uses presses A and then waits, we don't handle this binding yet (unless it was marked 'eager'), because we don't know what will follow. This timeout is the maximum amount of time that we wait until we call the handlers anyway. Pass *None* to disable this timeout.

18.10 Recording macros

Both Emacs and Vi mode allow macro recording. By default, all key presses are recorded during a macro, but it is possible to exclude certain keys by setting the *record_in_macro* parameter to *False*:

```
@bind.add('ctrl-t', record_in_macro=False)
def _(event):
    # ...
    pass
```

18.11 Creating new Vi text objects and operators

We tried very hard to ship prompt_toolkit with as many as possible Vi text objects and operators, so that text editing feels as natural as possible to Vi users.

If you wish to create a new text object or key binding, that is actually possible. Check the *custom-vi-operator-and-text-object.py* example for more information.

CHAPTER

NINETEEN

LICENSE

19.1 MIT License

CHAPTER

TWENTY

CHANGELOG

`Version 2023.5`

Realeased On 2023-04-28

20.1 Added

- Added **:param:`bold`** to `Label`.
- Added **:param:`italic`** to `Label`.
- Added **:param:`underline`** to `Label`.
- Added **:param:`fg`** to `Label`.
- Added **:param:`bg`** to `Label`.
- Added **:param:`bg`** to `Box`.

`Version 2023.4`

Realeased On 2023-04-27

20.2 Added

- Added **:param:`fixed_height`** and **:param:`fixed_width`** to `Label`
- Added `Progressbar` as an alias of `quo.progress.ProgressBar`

20.2.1 Version 2023.3

Released On 2023-03-30

20.3 Added

- Added **:param:`lines`** to `quo.rule.Rule`
- Added :param:``spinner` to :obj:``quo.progress.ProgressBar`

20.4 Fixed

- Fixed `Label`
- Fixed `Window`

20.4.1 Version 2023.2

Released On 2023-03-23

20.5 Added

- Added **:param:`fg`** and *bg* to `quo.color.Color()`

20.5.1 Version 2023.1

Released On 2023-03-22

20.6 Added

- Added `Bar`
- Added `Rule`
- Added `Table`

20.7 Changed

- Deprecated `quo.console.Console.bar()`
- Deprecated `quo.console.Console.rule()`
- Deprecated **:param:`fmt`** in `quo.print()`
- Deprecated `quo.table.Table()`

20.7.1 Version 2022.9

Released 0n 2022-10-17

20.8 Added

- Added `Highlight` for syntax highlighting
- Added `Parser` for parsing arguments

20.9 Changed

- Deprecated `quo.console.app()`, `quo.console.arg()` and `quo.console.command()` in favor of `quo.parse.Parser`
- Deprecated several syntax highlighters in favor of `Highlight`

20.9.1 Version 2022.8.1

Released on 2022-08-21

20.10 Changed

- Renamed **:param:`animated`** in `quo.console.Console.rule` to **:param:`multicolored`**

20.10.1 Version 2022.8

Released on 2022-08-20

20.11 Added

- Added **:param:`animated`** to `quo.console.Console.rule`

20.11.1 Version 2022.7

Released on 2022-08-19

20.12 Added

- Added **:param:`fmt`** to `quo.console.Console.bar`

20.12.1 Version 2022.6.1

Released on 2022-06-12

- Optimised `quo.console.Console.bell`

20.12.2 Version 2022.6

Released on 2022-06-12

- Deprecated clipboard
- Optimized help paramater

20.12.3 Version 2022.5.3

Released on 2022-05-14

- Under the hood optimizations.

20.12.4 Version 2022.5.2

Released on 2022-05-08

20.13 Added

- Added `quo.color.Color()`

20.13.1 Version 2022.5.1

Released on 2022-05-07

20.14 Fixed

- Fixed `SpinningWheel` attribute error.

20.14.1 Version 2022.5

Released on 2022-05-01

20.15 Added

- Added `quo.console.Console.spin()`
- Added **:param:`column_width`** and **:param:`headers`** to `quo.table.Table()`
- Added **:param:`suggest`** to `quo.prompt.Prompt`

20.15.1 *Version 2022.4.5`*

Released on 2022-04-23

20.16 Added

- Added **:param:`case_sensitive`** to `quo.completion.WordCompleter`

  Changed

- Renamed `quo.console.Console.openfile()` to `open()`

20.16.1 Version 2022.4.4

Released on 2022-04-21

20.17 Added

- Added **:param:`int`** to `quo.prompt.Prompt`
- Added `continuation()` to `quo.prompt.Prompt`

20.17.1 Version 2022.4.3

Released on 2022-04-18

20.18 Added

- Added **:param:`style`** to `quo.table.Table()`

20.18.1 Version 2022.4.2

Released on 2022-04-16

20.19 Changed

- Under the hood optimization of class `quo.progress.ProgressBar`

20.19.1 Version 2022.4.1

Released on 2022-04-14

20.20 Fixed

20.20.1 Version 2022.4

Released on 2022-04-01

20.21 Added

- Added `quo.console.Console.pager()`
- Added **:param:`fmt`** to `quo.print()`
- Added **:param:`bg`** to all dialog boxes.
- Added **:param:`multiline`** to `quo.dialog.InputBox()`
- Added *TextField* as an aliase to `TextArea`

20.21.1 Version 2022.3.5

Released on 2022-03-19

20.22 Changed

- Optimized `quo.print()`

20.22.1 Version 2022.3.4

Released on 2022-03-18

20.23 Added

- Added **:param:`bind`** to `quo.container()`
- Added **:param:`focused_element`** to `quo.container()`
- Added **:param:`full_screen`** to `quo.container()`
- Added **:param:`mouse_support`** to `quo.container()`
- Added **:param:`refresh`** to `quo.container()`
- Added `quo.keys.bind()` as an instance of `quo.keys.Bind`
- Added `quo.console.console()` as an instance of `quo.console.Console`

20.23.1 Version 2022.3.3

Released on 2022-03-16

20.24 Changed

- Optimized **:param:`align`** in `quo.layout.Window`, `quo.layout.HSplit` and `quo.layout.VSplit`

20.24.1 Version 2022.3.2

Released on 2022-3-14

20.25 Added

- Added `quo.console.Console.bar()`
- Added `qquo.console.Console.rule()`

20.26 Changed

- Deprecated **:param:`.run()`** in the Dialog UI.

20.26.1 Version 2022.3.1

Released on 2022-3-12

20.27 Added

- Added **:param:`ul`** as an alias of **:param:`underline`** for `Style`.

20.27.1 Version 2022.3

Released on 2022-3-6

20.28 Added

- Added key binder <any> enabling the user to press any key to exit the help page.
- Introduced `quo.keys.Bind` as an alias of `quo.keys.KeyBinder`

20.29 Changed

- Changed **:param:`enable_system_elicit`** in favor of **:param:`system_prompt`**.
- Changed **:param:`enable_suspend`** in favor of **:param:`suspend`**.

20.30 Fixed

- Optimized the help page.
- Fixed Deprecated notice *TypeError*

20.30.1 Version 2022.2.2

Released on 2022-2-2

20.31 Added

- Added `quo.console.command()`
- Added `quo.console.app()`
- Added `quo.console.arg()`
- Added `quo.console.tether()`
- Added highlighters : *Actionscript*, *Arrow*, *Bibtex*, *Cpp*, *Css*, *Email, Fortran, Go, Haskell, HTML, Javascript, Julia, Perl, Php, Python, Ruby, Rust, Shell, Solidity, Sql*

20.32 Fixed

- Under the hood optimizations.

20.32.1 Version 2022.2.1

Released on 2022-2-25

20.33 Changed

- Deprecated **:param:`is_password`** in favor of **:param:`hide`**

20.34 Fixed

- Fixed `quo.Console.edit()`, `quo.Console.openfile()`, `quo.Console.encoding()`

`Version 2022.2`

Released on 2022-2-16

20.35 Added

- Added `quo.Console.edit()`
- Added `quo.Console.launch()`
- Added `quo.Console.size()`
- Added `quo.Console.encoding()`
- Added `quo.Console.bell()` - Added `quo.Console.rule()`
- Added `quo.Console.openfile()`
- Added `quo.types.integer()`

20.36 Changed

- Deprecated **:param:`password`** in favor of **:param:`hide`**
- Deprecated `quo.text.HTML` in favor of `quo.text.Text`
- Deprecated **:param:`r_elicit`** in favor of **:param:`rprompt`**
- Deprecated `quo.Suite` in favor of `quo.console.Console`
- Deprecated **:param:`validator`** in favor of **:param:`type`**
- Dropped support for *python < 3.8*

20.37 Fixed

- Full support for Windows

20.37.1 Version 2022.1.6

Released on 2022-1-17

- Under the hood optimizations
- Introduced `quo.dialog.MessageBox()`, `quo.dialog.PromptBox()`, `quo.dialog.RadiolistBox()`, `quo.dialog.ConfirmBox()`, `quo.dialog.CheckBox()`, `quo.dialog.ChoiceBox()` widgets for displaying formatted text in a window.

20.37.2 Version 2022.1.5

Released on 2022-1-11

20.38 Fixed

- ImportError: **:issue:`37`** affecting Windows OS

20.38.1 Version 2022.1

Released on 2022-1-11

20.39 Changed

- Dependency update **:issue:`32`** to **:issue:`35`**

20.40 Fixed

- Unexpected argument in `quo.prompt()` **:issue:`36`**

20.40.1 Version 2021.7

Released on 2021-12-25

20.41 Changed

- Deprecated **:param:`foreground`** and **:param:`background``** in favor of **:param:`fg`** and **:param:`bg`**

20.42 Fixed

- Fixed broken placeholder() issue **:issue:`30`**

20.42.1 Version 2021.6

Released on 2021-11-20

20.43 Added

- Added Support of a placeholder text that is displayed as long as no input is given.

20.43.1 Version 2021.5.5.2

Released on 2021-09-28

20.44 Fixed

- Pypi README fix

20.44.1 Version 2021.5.5

Released on 2021-09-2

20.45 Added

- Added support for tabular presentation of data.
- Added support for colorful error messages.
- Added **:param:`ul`** to `quo.echo()`. Can be used as a substitute for **:param:`underline`** parameter.
- Added **:param:`strike``** to `quo.echo()`
- Added a **:param:`hidded`** to `quo.echo()`

20.45.1 Version 2021.4.5

Released on 2021-08-22

20.46 Added

- Intoduced `quo.clipboard.InMemoryClipboard` class to copy and paste data flawlessly.

20.46.1 Version 2021.3.5

Released on 2021-07-19

20.47 Added

- Added **:param:`fg`** and **:param:`bg`** as an alias of **:param:`foreground`** and **:param:`background`**.
- Added `quo.progress.ProgressBar` class.
- Added `quo.text.HTML` for easy text formating.

20.48 Changed

- Fixed changelong link on PyPI.
- Fixed wcwidth dependancy issue **:issue:`18`**

20.48.1 Version 2021.2

Released on 2021-06-28

- Under the hood optimizations.

20.48.2 Version 2021.1

Released on 2021-06-18

20.49 Added

- Added support for `ANSI colors` for better coloring of the terminal
- Added support for RGB tuples of three integers

20.49.1 Version 2021.1.dev0

Released on 2021-01-10

- Proof of concept/Initial release.

CHAPTER

TWENTYONE

APPENDIX

21.1 ECHO ANSI COLORS

List of ANSI colors supported by `quo.echo()`

- `black (might be a gray)`
- `red`
- `green`
- `yellow` *(might be an orange)*
- `blue`
- `magenta`
- `cyan`
- `white` *(might be light gray)*
- `vblack` *vibrant black*
- `vblue`
- `vmagenta`
- `vwhite`
- `vcyan`
- `vred`
- `vgreen`
- `vyellow`

21.2 Available Syntax highlighters

- CSS
- Email
- Fortran
- Go
- Haskell
- HTML

- Python
- Ruby
- Rust
- Shell
- Solidity
- Sql

CHAPTER

TWENTYTWO

DONATE

In order to for us to maintain this project and grow our community of contributors. Donate

CHAPTER

TWENTYTHREE

GETTING HELP

23.1 Community

For discussions about the usage, development, and the future of quo, please join our Google community

- Community

CHAPTER

TWENTYFOUR

RESOURCES

24.1 Bug tracker

If you have any suggestions, bug reports, or annoyances please report them to our issue tracker at Bug tracker or send an email to:

scalabli@proton.me | scalabli@googlegroups.com