Rich

Release 9.13.0

Will McGugan

Mar 06, 2021

CONTENTS:

CHAPTER

ONE

INTRODUCTION

Rich is a Python library for writing *rich* text (with color and style) to the terminal, and for displaying advanced content such as tables, markdown, and syntax highlighted code.

Use Rich to make your command line applications visually appealing and present data in a more readable way. Rich can also be a useful debugging aid by pretty printing and syntax highlighting data structures.

1.1 Requirements

Rich works with OSX, Linux and Windows.

On Windows both the (ancient) cmd.exe terminal is supported and the new Windows Terminal. The later has much improved support for color and style.

Rich requires Python 3.6.1 and above. Note that Python 3.6.0 is *not* supported due to lack of support for methods on NamedTuples.

Note: PyCharm users will need to enable "emulate terminal" in output console option in run/debug configuration to see styled output.

1.2 Installation

You can install Rich from PyPi with *pip* or your favorite package manager:

```
pip install rich
```

Add the `-U` switch to update to the current version, if Rich is already installed.

If you intend to use Rich with Jupyter then there are some additional dependencies which you can install with the following command:

```
pip install rich[jupyter]
```

1.3 Quick Start

The quickest way to get up and running with Rich is to import the alternative `print` function which takes the same arguments as the built-in `print` and may be used as a drop-in replacement. Here's how you would do that:

```
from rich import print
```

You can then print strings or objects to the terminal in the usual way. Rich will do some basic syntax highlighting and format data structures to make them easier to read.

Strings may contain *Console Markup* which can be used to insert color and styles in to the output.

The following demonstrates both console markup and pretty formatting of Python objects:

```
>>> print("[italic red]Hello[/italic red] World!", locals())
```

This writes the following output to the terminal (including all the colors and styles):

If you would rather not shadow Python's builtin print, you can import `rich.print` as `rprint` (for example):

```
from rich import print as rprint
```

Continue reading to learn about the more advanced features of Rich.

1.4 Python in the REPL

Rich may be installed in the REPL so that Python data structures are automatically pretty printed with syntax highlighting. Here's how:

```
>>> from rich import pretty
>>> pretty.install()
>>> ["Rich and pretty", True]
```

You can also use this feature to try out Rich *renderables*. Here's an example:

```
>>> from rich.panel import Panel
>>> Panel.fit("[bold yellow]Hi, I'm a Panel", border_style="red")
```

Read on to learn more about Rich renderables.

1.5 Rich Inspector

Rich has an *`inspect()`* function which can generate a report on any Python object. It is a fantastic debug aid, and a good example of the output that Rich can generate. Here is a simple example:

```
>>> from rich import inspect
>>> from rich.color import Color
>>> color = Color.parse("red")
>>> inspect(color, methods=True)
```

CHAPTER

TWO

CONSOLE API

For complete control over terminal formatting, Rich 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 rich.console import Console
console = Console()
```

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

```
from my_project.console import console
```

The console object handles the mechanics of generating ANSI escape sequences for color and style. It will auto-detect the capabilities of the terminal and convert colors if necessary.

2.1 Attributes

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

- *size* is the current dimensions of the terminal (which may change if you resize the window).
- *encoding* is the default encoding (typically "utf-8").
- *is_terminal* is a boolean that indicates if the Console instance is writing to a terminal or not.
- *color_system* is a string containing the Console color system (see below).

2.2 Color systems

There are several "standards" for writing color to the terminal which are not all universally supported. Rich will auto-detect the appropriate color system, or you can set it manually by supplying a value for `color_system` to the *Console* constructor.

You can set `color_system` to one of the following values:

- `None` Disables color entirely.
- `"auto"` Will auto-detect the color system.
- `"standard"` Can display 8 colors, with normal and bright variations, for 16 colors in total.
- `"256"` Can display the 16 colors from "standard" plus a fixed palette of 240 colors.
- `"truecolor"` Can display 16.7 million colors, which is likely all the colors your monitor can display.

- "windows" Can display 8 colors in legacy Windows terminal. New Windows terminal can display "truecolor".

Warning: Be careful when setting a color system, if you set a higher color system than your terminal supports, your text may be unreadable.

2.3 Printing

To write rich content to the terminal use the *print()* method. Rich will convert any object to a string via its (__str__) method and perform some simple syntax highlighting. It will also do pretty printing of any containers, such as dicts and lists. If you print a string it will render *Console Markup*. Here are some examples:

```
console.print([1, 2, 3])
console.print("[blue underline]Looks like a link")
console.print(locals())
console.print("FOO", style="white on blue")
```

You can also use *print()* to render objects that support the *Console Protocol*, which includes Rich's built in objects such as *Text*, *Table*, and *Syntax* – or other custom objects.

2.4 Logging

The *log()* methods offers the same capabilities as print, but adds some features useful for debugging a running application. Logging writes the current time in a column to the left, and the file and line where the method was called to a column on the right. Here's an example:

```
>>> console.log("Hello, World!")
```

To help with debugging, the log() method has a log_locals parameter. If you set this to True, Rich will display a table of local variables where the method was called.

2.5 Low level output

In additional to *print()* and *log()*, Rich has a *out()* method which provides a lower-level way of writing to the terminal. The out() method converts all the positional arguments to strings and won't pretty print, word wrap, or apply markup to the output, but can apply a basic style and will optionally do highlighting.

Here's an example:

```
>>> console.out("Locals", locals())
```

2.6 Rules

The *rule()* method will draw a horizontal line with an optional title, which is a good way of dividing your terminal output in to sections.

```
>>> console.rule("[bold red]Chapter 2")
```

The rule method also accepts a `style` parameter to set the style of the line, and an `align` parameter to align the title ("left", "center", or "right").

2.7 Status

Rich can display a status message with a 'spinner' animation that won't interfere with regular console output. Run the following command for a demo of this feature:

```
python -m rich.status
```

To display a status message, call *status()* with the status message (which may be a string, Text, or other renderable). The result is a context manager which starts and stop the status display around a block of code. Here's an example:

```
with console.status("Working..."):
    do_work()
```

You can change the spinner animation via the `spinner` parameter:

```
with console.status("Monkeying around...", spinner="monkey"):
    do_work()
```

Run the following command to see the available choices for `spinner`:

```
python -m rich.spinner
```

2.8 Justify / Alignment

Both print and log support a `justify` argument which if set must be one of "default", "left", "right", "center", or "full". If "left", any text printed (or logged) will be left aligned, if "right" text will be aligned to the right of the terminal, if "center" the text will be centered, and if "full" the text will be lined up with both the left and right edges of the terminal (like printed text in a book).

The default for `justify` is `"default"` which will generally look the same as `"left"` but with a subtle difference. Left justify will pad the right of the text with spaces, while a default justify will not. You will only notice the difference if you set a background color with the `style` argument. The following example demonstrates the difference:

```
from rich.console import Console

console = Console(width=20)

style = "bold white on blue"
console.print("Rich", style=style)
console.print("Rich", style=style, justify="left")
console.print("Rich", style=style, justify="center")
console.print("Rich", style=style, justify="right")
```

This produces the following output:

2.9 Overflow

Overflow is what happens when text you print is larger than the available space. Overflow may occur if you print long 'words' such as URLs for instance, or if you have text inside a panel or table cell with restricted space.

You can specify how Rich should handle overflow with the `overflow` argument to *`print()`* which should be one of the following strings: "fold", "crop", "ellipsis", or "ignore". The default is "fold" which will put any excess characters on the following line, creating as many new lines as required to fit the text.

The "crop" method truncates the text at the end of the line, discarding any characters that would overflow.

The "ellipsis" method is similar to "crop", but will insert an ellipsis character ("…") at the end of any text that has been truncated.

The following code demonstrates the basic overflow methods:

```
from typing import List
from rich.console import Console, OverflowMethod

console = Console(width=14)
supercali = "supercalifragilisticexpialidocious"

overflow_methods: List[OverflowMethod] = ["fold", "crop", "ellipsis"]
for overflow in overflow_methods:
    console.rule(overflow)
    console.print(supercali, overflow=overflow, style="bold blue")
    console.print()
```

This produces the following output:

You can also set overflow to "ignore" which allows text to run on to the next line. In practice this will look the same as "crop" unless you also set `crop=False` when calling *`print()`*.

2.10 Console style

The Console has a `style` attribute which you can use to apply a style to everything you print. By default `style` is None meaning no extra style is applied, but you can set it to any valid style. Here's an example of a Console with a style attribute set:

```
from rich.console import Console
blue_console = Console(style="white on blue")
blue_console.print("I'm blue. Da ba dee da ba di.")
```

2.11 Soft Wrapping

Rich word wraps text you print by inserting line breaks. You can disable this behavior by setting `soft_wrap=True` when calling *`print()`*. With *soft wrapping* enabled any text that doesn't fit will run on to the following line(s), just like the builtin `print`.

2.12 Cropping

The *`print()`* method has a boolean `crop` argument. The default value for crop is True which tells Rich to crop any content that would otherwise run on to the next line. You generally don't need to think about cropping, as Rich will resize content to fit within the available width.

Note: Cropping is automatically disabled if you print with `soft_wrap=True`.

2.13 Input

The console class has an *`input()`* which works in the same way as Python's builtin `input()` method, but can use anything that Rich can print as a prompt. For example, here's a colorful prompt with an emoji:

```
from rich.console import Console
console = Console()
console.input("What is [i]your[/i] [bold red]name[/]? :smiley: ")
```

2.14 Exporting

The Console class can export anything written to it as either text or html. To enable exporting, first set `record=True` on the constructor. This tells Rich to save a copy of any data you `print()` or `log()`. Here's an example:

```
from rich.console import Console
console = Console(record=True)
```

After you have written content, you can call *`export_text()`* or *`export_html()`* to get the console output as a string. You can also call *`save_text()`* or *`save_html()`* to write the contents directly to disk.

For examples of the html output generated by Rich Console, see *Standard Colors*.

2.15 Error console

The Console object will write to `sys.stdout` by default (so that you see output in the terminal). If you construct the Console with `stderr=True` Rich will write to `sys.stderr`. You may want to use this to create an *error console* so you can split error messages from regular output. Here's an example:

```
from rich.console import Console
error_console = Console(stderr=True)
```

You might also want to set the `style` parameter on the Console to make error messages visually distinct. Here's how you might do that:

```
error_console = Console(stderr=True, style="bold red")
```

2.16 File output

You can also tell the Console object to write to a file by setting the `file` argument on the constructor – which should be a file-like object opened for writing text. You could use this to write to a file without the output ever appearing on the terminal. Here's an example:

```
import sys
from rich.console import Console
from datetime import datetime

with open("report.txt", "wt") as report_file:
    console = Console(file=report_file)
    console.rule(f"Report Generated {datetime.now().ctime()}")
```

Note that when writing to a file you may want to explicitly the `width` argument if you don't want to wrap the output to the current console width.

2.17 Capturing output

There may be situations where you want to *capture* the output from a Console rather than writing it directly to the terminal. You can do this with the *`capture()`* method which returns a context manager. On exit from this context manager, call *`get()`* to return the string that would have been written to the terminal. Here's an example:

```
from rich.console import Console
console = Console()
with console.capture() as capture:
    console.print("[bold red]Hello[/] World")
str_output = capture.get()
```

An alternative way of capturing output is to set the Console file to a `io.StringIO`. This is the recommended method if you are testing console output in unit tests. Here's an example:

```
from io import StringIO
from rich.console import Console
console = Console(file=StringIO())
console.print("[bold red]Hello[/] World")
str_output = console.file.getvalue()
```

2.18 Paging

If you have some long output to present to the user you can use a *pager* to display it. A pager is typically an application on your operating system which will at least support pressing a key to scroll, but will often support scrolling up and down through the text and other features.

You can page output from a Console by calling *pager()* which returns a context manger. When the pager exits, anything that was printed will be sent to the pager. Here's an example:

```
from rich.__main__ import make_test_card
from rich.console import Console

console = Console()
with console.pager():
    console.print(make_test_card())
```

Since the default pager on most platforms don't support color, Rich will strip color from the output. If you know that your pager supports color, you can set `styles=True` when calling the *pager()* method.

Note: Rich will use the `PAGER` environment variable to get the pager command. On Linux and macOS you can set this to `less -r` to enable paging with ANSI styles.

2.19 Alternate screen

Warning: This feature is currently experimental. You might want to wait before using it in production.

Terminals support an 'alternate screen' mode which is separate from the regular terminal and allows for full-screen applications that leave your stream of input and commands intact. Rich supports this mode via the *set_alt_screen()* method, although it is recommended that you use *screen()* which returns a context manager that disables alternate mode on exit.

Here's an example of an alternate screen:

```
from time import sleep
from rich.console import Console

console = Console()
with console.screen():
    console.print(locals())
    sleep(5)
```

The above code will display a pretty printed dictionary on the alternate screen before returning to the command prompt after 5 seconds.

You can also provide a renderable to *screen()* which will be displayed in the alternate screen when you call `update()`.

Here's an example:

```
from time import sleep
```

(continues on next page)

(continued from previous page)

```
from rich.console import Console
from rich.align import Align
from rich.text import Text
from rich.panel import Panel

console = Console()

with console.screen(style="bold white on red") as screen:
    for count in range(5, 0, -1):
        text = Align.center(
            Text.from_markup(f"[blink]Don't Panic![/blink]\n{count}", justify="center
↪"),
            vertical="middle",
        )
        screen.update(Panel(text))
        sleep(1)
```

Updating the screen with a renderable allows Rich to crop the contents to fit the screen without scrolling.

For a more powerful way of building full screen interfaces with Rich, see *Live Display*.

Note: If you ever find yourself stuck in alternate mode after exiting Python code, type `reset` in the terminal

2.20 Terminal detection

If Rich detects that it is not writing to a terminal it will strip control codes from the output. If you want to write control codes to a regular file then set `force_terminal=True` on the constructor.

Letting Rich auto-detect terminals is useful as it will write plain text when you pipe output to a file or other application.

2.20.1 Interactive mode

Rich will remove animations such as progress bars and status indicators when not writing to a terminal as you probably don't want to write these out to a text file (for example). You can override this behavior by setting the `force_interactive` argument on the constructor. Set it to True to enable animations or False to disable them.

Note: Some CI systems support ANSI color and style but not anything that moves the cursor or selectively refreshes parts of the terminal. For these you might want to set `force_terminal` to `True` and `force_interactve` to `False`.

2.21 Environment variables

Rich respects some standard environment variables.

Setting the environment variable `TERM` to `"dumb"` or `"unknown"` will disable color/style and some features that require moving the cursor, such as progress bars.

If the environment variable `NO_COLOR` is set, Rich will disable all color in the output.

CHAPTER

THREE

STYLES

In various places in the Rich API you can set a "style" which defines the color of the text and various attributes such as bold, italic etc. A style may be given as a string containing a *style definition* or as an instance of a *Style* class.

3.1 Defining Styles

A style definition is a string containing one or more words to set colors and attributes.

To specify a foreground color use one of the 256 *Standard Colors*. For example, to print "Hello" in magenta:

```
console.print("Hello", style="magenta")
```

You may also use the color's number (an integer between 0 and 255) with the syntax `"color(<number>)"`. The following will give the equivalent output:

```
console.print("Hello", style="color(5)")
```

Alteratively you can use a CSS-like syntax to specify a color with a "#" followed by three pairs of hex characters, or in RGB form with three decimal integers. The following two lines both print "Hello" in the same color (purple):

```
console.print("Hello", style="#af00ff")
console.print("Hello", style="rgb(175,0,255)")
```

The hex and rgb forms allow you to select from the full *truecolor* set of 16.7 million colors.

Note: Some terminals only support 256 colors. Rich will attempt to pick the closest color it can if your color isn't available.

By itself, a color will change the *foreground* color. To specify a *background* color, precede the color with the word "on". For example, the following prints text in red on a white background:

```
console.print("DANGER!", style="red on white")
```

You can also set a color with the word `"default"` which will reset the color to a default managed by your terminal software. This works for backgrounds as well, so the style of `"default on default"` is what your terminal starts with.

You can set a style attribute by adding one or more of the following words:

- `"bold"` or `"b"` for bold text.
- `"blink"` for text that flashes (use this one sparingly).

- "blink2" for text that flashes rapidly (not supported by most terminals).
- "conceal" for *concealed* text (not supported by most terminals).
- "italic" or "i" for italic text (not supported on Windows).
- "reverse" or "r" for text with foreground and background colors reversed.
- "strike" or "s" for text with a line through it.
- "underline" or "u" for underlined text.

Rich also supports the following styles, which are not well supported and may not display in your terminal:

- "underline2" or "uu" for doubly underlined text.
- "frame" for framed text.
- "encircle" for encircled text.
- "overline" or "o" for overlined text.

Style attributes and colors may be used in combination with each other. For example:

```
console.print("Danger, Will Robinson!", style="blink bold red underline on white")
```

Styles may be negated by prefixing the attribute with the word "not". This can be used to turn off styles if they overlap. For example:

```
console.print("foo [not bold]bar[/not bold] baz", style="bold")
```

This will print "foo" and "baz" in bold, but "bar" will be in normal text.

Styles may also have a "link" attribute, which will turn any styled text in to a *hyperlink* (if supported by your terminal software).

To add a link to a style, the definition should contain the word "link" followed by a URL. The following example will make a clickable link:

```
console.print("Google", style="link https://google.com")
```

Note: If you are familiar with HTML you may find applying links in this way a little odd, but the terminal considers a link to be another attribute just like bold, italic etc.

3.2 Style Class

Ultimately the style definition is parsed and an instance of a *Style* class is created. If you prefer, you can use the Style class in place of the style definition. Here's an example:

```
from rich.style import Style
danger_style = Style(color="red", blink=True, bold=True)
console.print("Danger, Will Robinson!", style=danger_style)
```

It is slightly quicker to construct a Style class like this, since a style definition takes a little time to parse – but only on the first call, as Rich will cache parsed style definitions.

Styles may be combined by adding them together, which is useful if you want to modify attributes of an existing style. Here's an example:

```
from rich.console import Console
from rich.style import Style
console = Console()

base_style = Style.parse("cyan")
console.print("Hello, World", style = base_style + Style(underline=True))
```

You can parse a style definition explicitly with the *parse()* method, which accepts the style definition and returns a Style instance. For example, the following two lines are equivalent:

```
style = Style(color="magenta", bgcolor="yellow", italic=True)
style = Style.parse("italic magenta on yellow")
```

3.3 Style Themes

If you re-use styles it can be a maintenance headache if you ever want to modify an attribute or color – you would have to change every line where the style is used. Rich provides a *Theme* class which you can use to define custom styles that you can refer to by name. That way you only need update your styles in one place.

Style themes can make your code more semantic, for instance a style called `"warning"` better expresses intent that `"italic magenta underline"`.

To use a style theme, construct a *Theme* instance and pass it to the *Console* constructor. Here's an example:

```
from rich.console import Console
from rich.theme import Theme
custom_theme = Theme({
    "info" : "dim cyan",
    "warning": "magenta",
    "danger": "bold red"
})
console = Console(theme=custom_theme)
console.print("This is information", style="info")
console.print("[warning]The pod bay doors are locked[/warning]")
console.print("Something terrible happened!", style="danger")
```

Note: style names must be lower case, start with a letter, and only contain letters or the characters `"."`, `"-"`, `"_"`.

3.3.1 Customizing Defaults

The Theme class will inherit the default styles builtin to Rich. If your custom theme contains the name of an existing style, it will replace it. This allows you to customize the defaults as easily as you can create your own styles. For instance, here's how you can change how Rich highlights numbers:

```
from rich.console import Console
from rich.theme import Theme
console = Console(theme=Theme({"repr.number": "bold green blink"}))
console.print("The total is 128")
```

You can disable inheriting the default theme by setting `inherit=False` on the *rich.theme.Theme* constructor.

To see the default theme, run the following command:

```
python -m rich.theme
```

3.3.2 Loading Themes

If you prefer, you can write your styles in an external config file rather than in Python. Here's an example of the format:

```
[styles]
info = dim cyan
warning = magenta
danger = bold red
```

You can read these files with the *read()* method.

CHAPTER

FOUR

CONSOLE MARKUP

Rich supports a simple markup which you can use to insert color and styles virtually everywhere Rich would accept a string (e.g. *print()* and *log()*).

4.1 Syntax

Console markup uses a syntax inspired by bbcode. If you write the style (see *Styles*) in square brackets, e.g. `[bold red]`, that style will apply until it is *closed* with a corresponding `[/bold red]`.

Here's a simple example:

```
from rich import print
print("[bold red]alert![/bold red] Something happened")
```

If you don't close a style, it will apply until the end of the string. Which is sometimes convenient if you want to style a single line. For example:

```
print("[bold italic yellow on red blink]This text is impossible to read")
```

There is a shorthand for closing a style. If you omit the style name from the closing tag, Rich will close the last style. For example:

```
print("[bold red]Bold and red[/] not bold or red")
```

These markup tags may be use in combination with each other and don't need to be strictly nested. The following examples demonstrates overlapping of markup tags:

```
print("[bold]Bold[italic] bold and italic [/bold]italic[/italic]")
```

4.1.1 Errors

Rich will raise `MarkupError` if the markup contains one of the following errors:

- Mismatched tags, e.g. `"[bold]Hello[/red]"`
- No matching tag for implicit close, e.g. `"no tags[/]"`

4.1.2 Links

Console markup can output hyperlinks with the following syntax: `[link=URL]text[/link]`. Here's an example:

```
print("Visit my [link=https://www.willmcgugan.com]blog[/link]!")
```

If your terminal software supports hyperlinks, you will be able to click the word "blog" which will typically open a browser. If your terminal doesn't support hyperlinks, you will see the text but it won't be clickable.

4.1.3 Escaping

Occasionally you may want to print something that Rich would interpret as markup. You can *escape* a tag by preceding it with a backslash. Here's an example:

```
>>> from rich import print
>>> print(r"foo\[bar]")
foo[bar]
```

Without the backslash, Rich will assume that `[bar]` is a tag and remove it from the output if there is no "bar" style.

Note: If you want to prevent the backslash from escaping the tag and output a literal backslash before a tag you can enter two backslashes.

The function *escape()* will handle escaping of text for you.

Escaping is important if you construct console markup dynamically, with `str.format` or f strings (for example). Without escaping it may be possible to inject tags where you don't want them. Consider the following function:

```
def greet(name):
    console.print(f"Hello {name}!")
```

Calling `greet("Will")` will print a greeting, but if you were to call `greet("[blink]Gotcha![/blink]")` then you will also get blinking text, which may not be desirable. The solution is to escape the arguments:

```
from rich.markup import escape
def greet(name):
    console.print(f"Hello {escape(name)}!")
```

4.2 Rendering Markup

By default, Rich will render console markup when you explicitly pass a string to `print()` or implicitly when you embed a string in another renderable object such as *Table* or *Panel*.

Console markup is convenient, but you may wish to disable it if the syntax clashes with the string you want to print. You can do this by setting `markup=False` on the `print()` method or on the *Console* constructor.

4.3 Markup API

You can convert a string to styled text by calling *from_markup()*, which returns a *Text* instance you can print or add more styles to.

CHAPTER

FIVE

RICH TEXT

Rich has a *Text* class you can use to mark up strings with color and style attributes. You can use a Text instance anywhere a string is accepted, which gives you a lot of control over presentation.

You can consider this class to be like a string with marked up regions of text. Unlike a builtin `str`, a Text instance is mutable, and most methods operate in-place rather than returning a new instance.

One way to add a style to Text is the *stylize()* method which applies a style to a start and end offset. Here is an example:

```
from rich.console import Console
from rich.text import Text

console = Console()
text = Text("Hello, World!")
text.stylize("bold magenta", 0, 6)
console.print(text)
```

This will print "Hello, World!" to the terminal, with the first word in bold magenta.

Alternatively, you can construct styled text by calling *append()* to add a string and style to the end of the Text. Here's an example:

```
text = Text()
text.append("Hello", style="bold magenta")
text.append(" World!")
console.print(text)
```

Since building Text instances from parts is a common requirement, Rich offers *assemble()* which will combine strings or pairs of string and Style, and return a Text instance. The follow example is equivalent to the code above:

```
text = Text.assemble(("Hello", "bold magenta"), " World!")
console.print(text)
```

You can apply a style to given words in the text with *highlight_words()* or for ultimate control call *highlight_regex()* to highlight text matching a *regular expression*.

5.1 Text attributes

The Text class has a number of parameters you can set on the constructor to modify how the text is displayed.

- `justify` should be "left", "center", "right", or "full", and will override default justify behavior.
- `overflow` should be "fold", "crop", or "ellipsis", and will override default overflow.
- `no_wrap` prevents wrapping if the text is longer then the available width.
- `tab_size` Sets the number of characters in a tab.

A Text instance may be used in place of a plain string virtually everywhere in the Rich API, which gives you a lot of control in how text renders within other Rich renderables. For instance, the following example right aligns text within a *rich.panel.Panel*:

```
from rich import print
from rich.panel import Panel
from rich.text import Text
panel = Panel(Text("Hello", justify="right"))
print(panel)
```

CHAPTER

SIX

HIGHLIGHTING

Rich can apply styles to patterns in text which you *print()* or *log()*. With the default settings, Rich will highlight things such as numbers, strings, collections, booleans, None, and a few more exotic patterns such as file paths, URLs and UUIDs.

You can disable highlighting either by setting `highlight=False` on *print()* or *log()*, or by setting `highlight=False` on the *Console* constructor which disables it everywhere. If you disable highlighting on the constructor, you can still selectively *enable* highlighting with `highlight=True` on print/log.

6.1 Custom Highlighters

If the default highlighting doesn't fit your needs, you can define a custom highlighter. The easiest way to do this is to extend the *RegexHighlighter* class which applies a style to any text matching a list of regular expressions.

Here's an example which highlights text that looks like an email address:

```
from rich.console import Console
from rich.highlighter import RegexHighlighter
from rich.theme import Theme

class EmailHighlighter(RegexHighlighter):
    """Apply style to anything that looks like an email."""

    base_style = "example."
    highlights = [r"(?P<email>[\w-]+@([\w-]+\.)+[\w-]+)"]


theme = Theme({"example.email": "bold magenta"})
console = Console(highlighter=EmailHighlighter(), theme=theme)
console.print("Send funds to money@example.org")
```

The `highlights` class variable should contain a list of regular expressions. The group names of any matching expressions are prefixed with the `base_style` attribute and used as styles for matching text. In the example above, any email addresses will have the style "example.email" applied, which we've defined in a custom *Theme*.

Setting the highlighter on the Console will apply highlighting to all text you print (if enabled). You can also use a highlighter on a more granular level by using the instance as a callable and printing the result. For example, we could use the email highlighter class like this:

```
console = Console(theme=theme)
highlight_emails = EmailHighlighter()
console.print(highlight_emails("Send funds to money@example.org"))
```

While *RegexHighlighter* is quite powerful, you can also extend its base class *Highlighter* to implement a custom scheme for highlighting. It contains a single method *highlight* which is passed the *Text* to highlight.

Here's a silly example that highlights every character with a different color:

```
from random import randint

from rich import print
from rich.highlighter import Highlighter


class RainbowHighlighter(Highlighter):
    def highlight(self, text):
        for index in range(len(text)):
            text.stylize(f"color({randint(16, 255)})", index, index + 1)


rainbow = RainbowHighlighter()
print(rainbow("I must not fear. Fear is the mind-killer."))
```

CHAPTER

SEVEN

LOGGING HANDLER

Rich supplies a *logging handler* which will format and colorize text written by Python's logging module.

Here's an example of how to set up a rich logger:

```
import logging
from rich.logging import RichHandler

FORMAT = "%(message)s"
logging.basicConfig(
    level="NOTSET", format=FORMAT, datefmt="[%X]", handlers=[RichHandler()]
)

log = logging.getLogger("rich")
log.info("Hello, World!")
```

Rich logs won't render *Console Markup* in logging by default as most libraries won't be aware of the need to escape literal square brackets, but you can enable it by setting `markup=True` on the handler. Alternatively you can enable it per log message by supplying the `extra` argument as follows:

```
log.error("[bold red blink]Server is shutting down![/]", extra={"markup": True})
```

7.1 Handle exceptions

The *RichHandler* class may be configured to use Rich's *Traceback* class to format exceptions, which provides more context than a builtin exception. To get beautiful exceptions in your logs set `rich_tracebacks=True` on the handler constructor:

```
import logging
from rich.logging import RichHandler

logging.basicConfig(
    level="NOTSET",
    format="%(message)s",
    datefmt="[%X]",
    handlers=[RichHandler(rich_tracebacks=True)]
)

log = logging.getLogger("rich")
try:
    print(1 / 0)
except Exception:
    log.exception("unable print!")
```

There are a number of other options you can use to configure logging output, see the *RichHandler* reference for details.

CHAPTER

EIGHT

TRACEBACK

Rich can render Python tracebacks with syntax highlighting and formatting. Rich tracebacks are easier to read, and show more code, than standard Python tracebacks.

8.1 Printing tracebacks

The *print_exception()* method will print a traceback for the current exception being handled. Here's an example:

```
try:
    do_something()
except:
    console.print_exception()
```

8.2 Traceback handler

Rich can be installed as the default traceback handler so that all uncaught exceptions will be rendered with highlighting. Here's how:

```
from rich.traceback import install
install()
```

There are a few options to configure the traceback handler, see *install()* for details.

CHAPTER

NINE

PROMPT

Rich has a number of *Prompt* classes which ask a user for input and loop until a valid response is received. Here's a simple example:

```
>>> from rich.prompt import Prompt
>>> name = Prompt.ask("Enter your name")
```

The prompt may be given as a string (which may contain *Console Markup* and emoji code) or as a *Text* instance.

You can set a default value which will be returned if the user presses return without entering any text:

```
>>> from rich.prompt import Prompt
>>> name = Prompt.ask("Enter your name", default="Paul Atreides")
```

If you supply a list of choices, the prompt will loop until the user enters one of the choices:

```
>>> from rich.prompt import Prompt
>>> name = Prompt.ask("Enter your name", choices=["Paul", "Jessica", "Duncan"],␣
↪default="Paul")
```

In addition to *Prompt* which returns strings, you can also use *IntPrompt* which asks the user for an integer, and *FloatPrompt* for floats.

The *Confirm* class is a specialized prompt which may be used to ask the user a simple yes / no question. Here's an example:

```
>>> from rich.prompt import Confirm
>>> is_rich_great = Confirm.ask("Do you like rich?")
>>> assert is_rich_great
```

The Prompt class was designed to be customizable via inheritance. See prompt.py for examples.

To see some of the prompts in action, run the following command from the command line:

```
python -m rich.prompt
```

CHAPTER

TEN

COLUMNS

Rich can render text or other Rich renderables in neat columns with the *Columns* class. To use, construct a Columns instance with an iterable of renderables and print it to the Console.

The following example is a very basic clone of the `ls` command in OSX / Linux to list directory contents:

```
import os
import sys

from rich import print
from rich.columns import Columns

if len(sys.argv) < 2:
    print("Usage: python columns.py DIRECTORY")
else:
    directory = os.listdir(sys.argv[1])
    columns = Columns(directory, equal=True, expand=True)
    print(columns)
```

See columns.py for an example which outputs columns containing more than just text.

CHAPTER

ELEVEN

RENDER GROUPS

The *RenderGroup* class allows you to group several renderables together so they may be rendered in a context where only a single renderable may be supplied. For instance, you might want to display several renderables within a *Panel*.

To render two panels within a third panel, you would construct a RenderGroup with the *child* renderables as positional arguments then wrap the result in another Panel:

```
from rich import print
from rich.console import RenderGroup
from rich.panel import Panel

panel_group = RenderGroup(
    Panel("Hello", style="on blue"),
    Panel("World", style="on red"),
)
print(Panel(panel_group))
```

This pattern is nice when you know in advance what renderables will be in a group, put can get awkward if you have a larger number of renderables, especially if they are dynamic. Rich provides a *render_group()* decorator to help with these situations. The decorator builds a render group from an iterator of renderables. The following is the equivalent of the previous example using the decorator:

```
from rich import print
from rich.console import render_group
from rich.panel import Panel

@render_group()
def get_panels():
    yield Panel("Hello", style="on blue")
    yield Panel("World", style="on red")

print(Panel(get_panels()))
```

CHAPTER

TWELVE

MARKDOWN

Rich can render Markdown to the console. To render markdown, construct a *Markdown* object then print it to the console. Markdown is a great way of adding rich content to your command line applications. Here's an example of use:

```
MARKDOWN = """
# This is an h1

Rich can do a pretty *decent* job of rendering markdown.

1. This is a list item
2. This is another list item
"""
from rich.console import Console
from rich.markdown import Markdown

console = Console()
md = Markdown(MARKDOWN)
console.print(md)
```

Note that code blocks are rendered with full syntax highlighting!

You can also use the Markdown class from the command line. The following example displays a readme in the terminal:

```
python -m rich.markdown README.md
```

Run the following to see the full list of arguments for the markdown command:

```
python -m rich.markdown -h
```

CHAPTER

THIRTEEN

PADDING

The *Padding* class may be used to add whitespace around text or other renderable. The following example will print the word "Hello" with a padding of 1 character, so there will be a blank line above and below, and a space on the left and right edges:

```
from rich import print
from rich.padding import Padding
test = Padding("Hello", 1)
print(test)
```

You can specify the padding on a more granular level by using a tuple of values rather than a single value. A tuple of 2 values sets the top/bottom and left/right padding, whereas a tuple of 4 values sets the padding for top, right, bottom, and left sides. You may recognize this scheme if you are familiar with CSS.

For example, the following displays 2 blank lines above and below the text, and a padding of 4 spaces on the left and right sides:

```
from rich import print
from rich.padding import Padding
test = Padding("Hello", (2, 4))
print(test)
```

The Padding class can also accept a `style` argument which applies a style to the padding and contents, and an `expand` switch which can be set to False to prevent the padding from extending to the full with of the terminal. Here's an example which demonstrates both these arguments:

```
from rich import print
from rich.padding import Padding
test = Padding("Hello", (2, 4), style="on blue", expand=False)
print(test)
```

Note that, as with all Rich renderables, you can use Padding any context. For instance, if you want to emphasize an item in a *Table* you could add a Padding object to a row with a padding of 1 and a style of "on red".

CHAPTER

FOURTEEN

PANEL

To draw a border around text or other renderable, construct a *Panel* with the renderable as the first positional argument. Here's an example:

```
from rich import print
from rich.panel import Panel
print(Panel("Hello, [red]World!"))
```

You can change the style of the panel by setting the `box` argument to the Panel constructor. See *Box* for a list of available box styles.

Panels will extend to the full width of the terminal. You can make panel *fit* the content by setting `expand=False` on the constructor, or by creating the Panel with *fit()*. For example:

```
from rich import print
from rich.panel import Panel
print(Panel.fit("Hello, [red]World!"))
```

The Panel constructor accepts a `title` argument which will draw a title within the panel:

```
from rich import print
from rich.panel import Panel
print(Panel("Hello, [red]World!", title="Welcome"))
```

See *Panel* for details how to customize Panels.

CHAPTER

FIFTEEN

PROGRESS DISPLAY

Rich can display continuously updated information regarding the progress of long running tasks / file copies etc. The information displayed is configurable, the default will display a description of the 'task', a progress bar, percentage complete, and estimated time remaining.

Rich progress display supports multiple tasks, each with a bar and progress information. You can use this to track concurrent tasks where the work is happening in threads or processes.

To see how the progress display looks, try this from the command line:

```
python -m rich.progress
```

Note: Progress works with Jupyter notebooks, with the caveat that auto-refresh is disabled. You will need to explicitly call *refresh()* or set `refresh=True` when calling *update()*. Or use the *track()* function which does a refresh automatically on each loop.

15.1 Basic Usage

For basic usage call the *track()* function, which accepts a sequence (such as a list or range object) and an optional description of the job you are working on. The track method will yield values from the sequence and update the progress information on each iteration. Here's an example:

```
from rich.progress import track

for n in track(range(n), description="Processing..."):
    do_work(n)
```

15.2 Advanced usage

If you require multiple tasks in the display, or wish to configure the columns in the progress display, you can work directly with the *Progress* class. Once you have constructed a Progress object, add task(s) with (*add_task()*) and update progress with *update()*.

The Progress class is designed to be used as a *context manager* which will start and stop the progress display automatically.

Here's a simple example:

```
import time

from rich.progress import Progress

with Progress() as progress:

    task1 = progress.add_task("[red]Downloading...", total=1000)
    task2 = progress.add_task("[green]Processing...", total=1000)
    task3 = progress.add_task("[cyan]Cooking...", total=1000)

    while not progress.finished:
        progress.update(task1, advance=0.5)
        progress.update(task2, advance=0.3)
        progress.update(task3, advance=0.9)
        time.sleep(0.02)
```

The `total` value associated with a task is the number of steps that must be completed for the progress to reach 100%. A *step* in this context is whatever makes sense for your application; it could be number of bytes of a file read, or number of images processed, etc.

15.2.1 Updating tasks

When you call *`add_task()`* you get back a *Task ID*. Use this ID to call *`update()`* whenever you have completed some work, or any information has changed. Typically you will need to update `completed` every time you have completed a step. You can do this by updated `completed` directly or by setting `advance` which will add to the current `completed` value.

The *`update()`* method collects keyword arguments which are also associated with the task. Use this to supply any additional information you would like to render in the progress display. The additional arguments are stored in `task.fields` and may be referenced in *Column classes*.

15.2.2 Hiding tasks

You can show or hide tasks by updating the tasks `visible` value. Tasks are visible by default, but you can also add a invisible task by calling *`add_task()`* with `visible=False`.

15.2.3 Transient progress

Normally when you exit the progress context manager (or call *`stop()`*) the last refreshed display remains in the terminal with the cursor on the following line. You can also make the progress display disappear on exit by setting `transient=True` on the Progress constructor. Here's an example:

```
with Progress(transient=True) as progress:
    task = progress.add_task("Working", total=100)
    do_work(task)
```

Transient progress displays are useful if you want more minimal output in the terminal when tasks are complete.

15.2.4 Indeterminate progress

When you add a task it is automatically *started*, which means it will show a progress bar at 0% and the time remaining will be calculated from the current time. This may not work well if there is a long delay before you can start updating progress; you may need to wait for a response from a server or count files in a directory (for example). In these cases you can call *add_task()* with `start=False` which will display a pulsing animation that lets the user know something is working. This is know as an *indeterminate* progress bar. When you have the number of steps you can call *start_task()* which will display the progress bar at 0%, then *update()* as normal.

15.2.5 Auto refresh

By default, the progress information will refresh 10 times a second. You can set the refresh rate with the `refresh_per_second` argument on the *Progress* constructor. You should set this to something lower than 10 if you know your updates will not be that frequent.

You might want to disable auto-refresh entirely if your updates are not very frequent, which you can do by setting `auto_refresh=False` on the constructor. If you disable auto-refresh you will need to call *refresh()* manually after updating your task(s).

15.2.6 Expand

The progress bar(s) will use only as much of the width of the terminal as required to show the task information. If you set the `expand` argument on the Progress constructor, then Rich will stretch the progress display to the full available width.

15.2.7 Columns

You may customize the columns in the progress display with the positional arguments to the *Progress* constructor. The columns are specified as either a format string or a *ProgressColumn* object.

Format strings will be rendered with a single value *"task"* which will be a *Task* instance. For example `"{task.description}"` would display the task description in the column, and `"{task.completed} of {task.total}"` would display how many of the total steps have been completed.

The defaults are roughly equivalent to the following:

```
progress = Progress(
    "[progress.description]{task.description}",
    BarColumn(),
    "[progress.percentage]{task.percentage:>3.0f}%",
    TimeRemainingColumn(),
)
```

The following column objects are available:

- *BarColumn* Displays the bar.
- *TextColumn* Displays text.
- *TimeElapsedColumn* Displays the time elapsed.
- *TimeRemainingColumn* Displays the estimated time remaining.
- *FileSizeColumn* Displays progress as file size (assumes the steps are bytes).
- *TotalFileSizeColumn* Displays total file size (assumes the steps are bytes).

- *DownloadColumn* Displays download progress (assumes the steps are bytes).
- *TransferSpeedColumn* Displays transfer speed (assumes the steps are bytes.
- *SpinnerColumn* Displays a "spinner" animation.
- *RenderableColumn* Displays an arbitrary Rich renderable in the column.

To implement your own columns, extend the *ProgressColumn* class and use it as you would the other columns.

15.2.8 Table Columns

Rich builds a :class:~rich.table.Table` for the tasks in the Progress instance. You can customize how the columns of this *tasks table* are created by specifying the *table_column* argument in the Column constructor, which should be a *Column* instance.

The following example demonstrates a progress bar where the description takes one third of the width of the terminal, and the bar takes up the remaining third.

from time import sleep

from rich.table import Column from rich.progress import Progress, BarColumn, TextColumn

text_column = TextColumn("{task.description}", table_column=Column(ratio=1)) bar_column = BarColumn(bar_width=None, table_column=Column(ratio=2)) progress = Progress(text_column, bar_column, expand=True)

with progress:

for n in progress.track(range(100)): progress.print(n) sleep(0.1)

15.2.9 Print / log

The Progress class will create an internal Console object which you can access via `progress.console`. If you print or log to this console, the output will be displayed *above* the progress display. Here's an example:

```
with Progress() as progress:
    task = progress.add_task("twiddling thumbs", total=10)
    for job in range(10):
        progress.console.print(f"Working on job #{job}")
        run_job(job)
        progress.advance(task)
```

If you have another Console object you want to use, pass it in to the *Progress* constructor. Here's an example:

```
from my_project import my_console

with Progress(console=my_console) as progress:
    my_console.print("[bold blue]Starting work!")
    do_work(progress)
```

15.2.10 Redirecting stdout / stderr

To avoid breaking the progress display visuals, Rich will redirect `stdout` and `stderr` so that you can use the builtin `print` statement. This feature is enabled by default, but you can disable by setting `redirect_stdout` or `redirect_stderr` to `False`

15.2.11 Customizing

If the *Progress* class doesn't offer exactly what you need in terms of a progress display, you can override the *get_renderables* method. For example, the following class will render a *Panel* around the progress display:

```
from rich.panel import Panel
from rich.progress import Progress

class MyProgress(Progress):
    def get_renderables(self):
        yield Panel(self.make_tasks_table(self.tasks))
```

15.3 Multiple Progress

You can't have different columns per task with a single Progress instance. However, you can have as many Progress instance as you like in a *Live Display*. See live_progress.py for an example of using mutiple Progress instances.

15.4 Example

See downloader.py for a realistic application of a progress display. This script can download multiple concurrent files with a progress bar, transfer speed and file size.

CHAPTER

SIXTEEN

SYNTAX

Rich can syntax highlight various programming languages with line numbers.

To syntax highlight code, construct a *Syntax* object and print it to the console. Here's an example:

```
from rich.console import Console
from rich.syntax import Syntax

console = Console()
with open("syntax.py", "rt") as code_file:
    syntax = Syntax(code_file.read(), "python")
console.print(syntax)
```

You may also use the *from_path()* alternative constructor which will load the code from disk and auto-detect the file type. The example above could be re-written as follows:

```
from rich.console import Console
from rich.syntax import Syntax

console = Console()
syntax = Syntax.from_path("syntax.py")
console.print(syntax)
```

16.1 Line numbers

If you set `line_numbers=True`, Rich will render a column for line numbers:

```
syntax = Syntax.from_path("syntax.py", line_numbers=True)
```

16.2 Theme

The Syntax constructor (and *from_path()*) accept a `theme` attribute which should be the name of a Pygments theme. It may also be one of the special case theme names "ansi_dark" or "ansi_light" which will use the color theme configured by the terminal.

16.3 Background color

You can override the background color from the theme by supplying a `background_color` argument to the constructor. This should be a string in the same format a style definition accepts, .e.g "red", "#ff0000", "rgb(255,0,0)" etc. You may also set the special value "default" which will use the default background color set in the terminal.

16.4 Syntax CLI

You can use this class from the command line. Here's how you would syntax highlight a file called "syntax.py":

```
python -m rich.syntax syntax.py
```

For the full list of arguments, run the following:

```
python -m rich.syntax -h
```

CHAPTER

SEVENTEEN

TABLES

Rich's *Table* class offers a variety of ways to render tabular data to the terminal.

To render a table, construct a *Table* object, add columns with *add_column()*, and rows with *add_row()* – then print it to the console.

Here's an example:

```
from rich.console import Console
from rich.table import Table

table = Table(title="Star Wars Movies")

table.add_column("Released", justify="right", style="cyan", no_wrap=True)
table.add_column("Title", style="magenta")
table.add_column("Box Office", justify="right", style="green")

table.add_row("Dec 20, 2019", "Star Wars: The Rise of Skywalker", "$952,110,690")
table.add_row("May 25, 2018", "Solo: A Star Wars Story", "$393,151,347")
table.add_row("Dec 15, 2017", "Star Wars Ep. V111: The Last Jedi", "$1,332,539,889")
table.add_row("Dec 16, 2016", "Rogue One: A Star Wars Story", "$1,332,439,889")

console = Console()
console.print(table)
```

This produces the following output:

Rich is quite smart about rendering the table. It will adjust the column widths to fit the contents and will wrap text if it doesn't fit. You can also add anything that Rich knows how to render as a title or row cell (even another table)!

You can set the border style by importing one of the preset `Box` objects and setting the `box` argument in the table constructor. Here's an example that modifies the look of the Star Wars table:

```
from rich import box
table = Table(title="Star Wars Movies", box=box.MINIMAL_DOUBLE_HEAD)
```

See *Box* for other box styles.

The *Table* class 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.

17.1 Empty Tables

Printing a table with no columns results in a blank line. If you are building a table dynamically and the data source has no columns, you might want to print something different. Here's how you might do that:

```
if table.columns:
    print(table)
else:
    print("[i]No data...[/i]")
```

17.2 Adding Columns

You may also add columns by specifying them in the positional arguments of the *Table* constructor. For example, we could construct a table with three columns like this:

```
table = Table("Released", "Title", "Box Office", title="Star Wars Movies")
```

This allows you to specify the text of the column only. If you want to set other attributes, such as width and style, you can add an *Column* class. Here's an example:

```
from rich.table import Column
table = Table(
    "Released",
    "Title",
    Column(header="Box Office", justify="right"),
    title="Star Wars Movies"
)
```

17.3 Lines

By default, Tables will show a line under the header only. If you want to show lines between all rows add `show_lines=True` to the constructor.

17.4 Grids

The Table class can also make a great layout tool. If you disable headers and borders you can use it to position content within the terminal. The alternative constructor *grid()* can create such a table for you.

For instance, the following code displays two pieces of text aligned to both the left and right edges of the terminal on a single line:

```
from rich import print
from rich.table import Table

grid = Table.grid(expand=True)
grid.add_column()
grid.add_column(justify="right")
grid.add_row("Raising shields", "[bold magenta]COMPLETED [green]:heavy_check_mark:")

print(grid)
```

CHAPTER

EIGHTEEN

TREE

Rich has a *Tree* class which can generate a tree view in the terminal. A tree view is a great way of presenting the contents of a filesystem or any other hierarchical data. Each branch of the tree can have a label which may be text or any other Rich renderable.

Run the following command to see a demonstration of a Rich tree:

```
python -m rich.tree
```

The following code creates and prints a tree with a simple text label:

```
from rich.tree import Tree
from rich import print

tree = Tree("Rich Tree")
print(tree)
```

With only a single `Tree` instance this will output nothing more than the text "Rich Tree". Things get more interesting when we call *add()* to add more branches to the Tree. The following code adds two more branches:

```
tree.add("foo")
tree.add("bar")
print(tree)
```

The tree will now have two branches connected to the original tree with guide lines.

When you call *add()* a new Tree instance is returned. You can use this instance to add more branches to, and build up a more complex tree. Let's add a few more levels to the tree:

```
baz_tree = tree.add("baz")
baz_tree.add("[red]Red").add("[green]Green").add("[blue]Blue")
print(tree)
```

18.1 Tree Styles

The Tree constructor and *add()* method allows you to specify a `style` argument which sets a style for the entire branch, and `guide_style` which sets the style for the guide lines. These styles are inherited by the branches and will apply to any sub-trees as well.

If you set `guide_style` to bold, Rich will select the thicker variations of unicode line characters. Similarly, if you select the "underline2" style you will get double line style of unicode characters.

18.2 Examples

For a more practical demonstration, see tree.py which can generate a tree view of a directory in your hard drive.

CHAPTER

NINETEEN

LIVE DISPLAY

Progress bars and status indicators use a *live* display to animate parts of the terminal. You can build custom live displays with the `Live` class.

For a demonstration of a live display, running the following command:

    python -m rich.live

Note: If you see ellipsis "...", this indicates that the terminal is not tall enough to show the full table.

19.1 Basic usage

To create a live display, construct a `Live` object with a renderable and use it has a context manager. The live display will persist for the duration of the context. You can update the renderable to update the display:

```
import time

from rich.live import Live
from rich.table import Table

table = Table()
table.add_column("Row ID")
table.add_column("Description")
table.add_column("Level")

with Live(table, refresh_per_second=4):  # update 4 times a second to feel fluid
    for row in range(12):
        time.sleep(0.4)  # arbitrary delay
        # update the renderable internally
        table.add_row(f"{row}", f"description {row}", "[red]ERROR")
```

19.2 Updating the renderable

You can also change the renderable on-the-fly by calling the *update()* method. This may be useful if the information you wish to display is too dynamic to generate by updating a single renderable. Here is an example:

```
import random
import time

from rich.live import Live
from rich.table import Table


def generate_table() -> Table:
    """Make a new table."""
    table = Table()
    table.add_column("ID")
    table.add_column("Value")
    table.add_column("Status")

    for row in range(random.randint(2, 6)):
        value = random.random() * 100
        table.add_row(
            f"{row}", f"{value:3.2f}", "[red]ERROR" if value < 50 else "[green]SUCCESS
↪"
        )
    return table


with Live(generate_table(), refresh_per_second=4) as live:
    for _ in range(40):
        time.sleep(0.4)
        live.update(generate_table())
```

19.3 Alternate screen

You can opt to show a Live display in the "alternate screen" by setting `screen=True` on the constructor. This will allow your live display to go full screen and restore the command prompt on exit.

You can use this feature in combination with *Layout* to display sophisticated terminal "applications".

19.4 Transient display

Normally when you exit live context manager (or call *stop()*) the last refreshed item remains in the terminal with the cursor on the following line. You can also make the live display disappear on exit by setting `transient=True` on the Live constructor.

19.5 Auto refresh

By default, the live display will refresh 4 times a second. You can set the refresh rate with the `refresh_per_second` argument on the *Live* constructor. You should set this to something lower than 4 if you know your updates will not be that frequent or higher for a smoother feeling.

You might want to disable auto-refresh entirely if your updates are not very frequent, which you can do by setting `auto_refresh=False` on the constructor. If you disable auto-refresh you will need to call *refresh()* manually or *update()* with `refresh=True`.

19.6 Vertical overflow

By default, the live display will display ellipsis if the renderable is too large for the terminal. You can adjust this by setting the `vertical_overflow` argument on the *Live* constructor.

- "crop" Show renderable up to the terminal height. The rest is hidden.
- "ellipsis" Similar to crop except last line of the terminal is replaced with "...". This is the default behavior.
- "visible" Will allow the whole renderable to be shown. Note that the display cannot be properly cleared in this mode.

Note: Once the live display stops on a non-transient renderable, the last frame will render as **visible** since it doesn't have to be cleared.

19.7 Print / log

The Live class will create an internal Console object which you can access via `live.console`. If you print or log to this console, the output will be displayed *above* the live display. Here's an example:

```
import time

from rich.live import Live
from rich.table import Table

table = Table()
table.add_column("Row ID")
table.add_column("Description")
table.add_column("Level")

with Live(table, refresh_per_second=4) as live:  # update 4 times a second to feel␣
↪fluid
    for row in range(12):
        live.console.print("Working on row #{row}")
        time.sleep(0.4)
        table.add_row(f"{row}", f"description {row}", "[red]ERROR")
```

If you have another Console object you want to use, pass it in to the *Live* constructor. Here's an example:

```
from my_project import my_console

with Live(console=my_console) as live:
```

(continues on next page)

(continued from previous page)

```
    my_console.print("[bold blue]Starting work!")
    ...
```

Note: If you are passing in a file console, the live display only show the last item once the live context is left.

19.8 Redirecting stdout / stderr

To avoid breaking the live display visuals, Rich will redirect `stdout` and `stderr` so that you can use the builtin `print` statement. This feature is enabled by default, but you can disable by setting `redirect_stdout` or `redirect_stderr` to `False`.

19.8.1 Nesting Lives

Note that only a single live context may be active at any one time. The following will raise a `LiveError` because status also uses Live:

```
with Live(table, console=console):
    with console.status("working"):  # Will not work
        do_work()
```

In practice this is rarely a problem because you can display any combination of renderables in a Live context.

19.8.2 Examples

See table_movie.py and top_lite_simulator.py for deeper examples of live displaying.

CHAPTER

TWENTY

LAYOUT

Rich offers a *Layout* class which can be used to divide the screen area in to parts, where each part may contain independent content. It can be used with *Live Display* to create full-screen "applications" but may be used standalone.

To see an example of a Layout, run the following from the command line:

```
python -m rich.layout
```

20.1 Creating layouts

To define a layout, construct a Layout object and print it:

```
from rich import print
from rich.layout import Layout

layout = Layout()
print(layout)
```

This will draw a box the size of the terminal with some information regarding the layout. The box is a "placeholder" because we have yet to add any content to it. Before we do that, let's create a more interesting layout by calling the *split()* method to divide the layout in to two sub-layouts:

```
layout.split(
    Layout(name="upper"),
    Layout(name="lower")
)
print(layout)
```

This will divide the terminal screen in to two equal sized portions, one on top of the other. The `name` attribute is an internal identifier we can use to look up the sub-layout later. Let's use that to create another split:

```
layout["lower"].split(
    Layout(name="left"),
    Layout(name="right"),
    direction="horizontal"
)
print(layout)
```

The addition of the `direction="horizontal"` tells the Layout class to split left-to-right, rather than the default of top-to-bottom.

You should now see the screen area divided in to 3 portions; an upper half and a lower half that is split in to two quarters.

You can continue to call split() in this way to create as many parts to the screen as you wish.

20.2 Setting renderables

The first position argument to `Layout` can be any Rich renderable, which will be sized to fit within the layout's area. Here's how we might divide the "right" layout in to two panels:

```
layout["right"].split(
    Layout(Panel("Hello")),
    Layout(Panel("World!"))
)
```

You can also call *`update()`* to set or replace the current renderable:

```
layout["left"].update(
    "The mystery of life isn't a problem to solve, but a reality to experience."
)
print(layout)
```

20.3 Fixed size

You can set a layout to use a fixed size by setting the `size` argument on the Layout constructor or by setting the attribute. Here's an example:

```
layout["upper"].size = 10
print(layout)
```

This will set the upper portion to be exactly 10 rows, no matter the size of the terminal. If the parent layout is horizontal rather than vertical, then the size applies to the number of characters rather that rows.

20.4 Ratio

In addition to a fixed size, you can also make a flexible layout setting the `ratio` argument on the constructor or by assigning to the attribute. The ratio defines how much of the screen the layout should occupy in relation to other layouts. For example, lets reset the size and set the ratio of the upper layout to 2:

```
layout["upper"].size = None
layout["upper"].ratio = 2
print(layout)
```

This makes the top layout take up two thirds of the space. This is because the default ratio is 1, giving the upper and lower layouts a combined total of 3. As the upper layout has a ratio of 2, it takes up two thirds of the space, leaving the remaining third for the lower layout.

A layout with a ratio set may also have a minimum size to prevent it from getting too small. For instance, here's how we could set the minimum size of the lower sub-layout so that it won't shrink beyond 10 rows:

```
layout["lower"].minimum_size = 10
```

20.5 Visibility

You can make a layout invisible by setting the `visible` attribute to False. Here's an example:

```
layout["upper"].visible = False
print(layout)
```

The top layout is now invisible, and the "lower" layout will expand to fill the available space. Set `visible` to True to bring it back:

```
layout["upper"].visible = True
print(layout)
```

You could use this to toggle parts of your interface based on your applications configuration.

20.6 Tree

To help visualize complex layouts you can print the `tree` attribute which will display a summary of the layout as a tree:

```
print(layout.tree)
```

20.7 Example

See fullscreen.py for an example that combines *Layout* and *Live* to create a fullscreen "application".

CHAPTER

TWENTYONE

CONSOLE PROTOCOL

Rich supports a simple protocol to add rich formatting capabilities to custom objects, so you can *print()* your object with color, styles and formatting.

Use this for presentation or to display additional debugging information that might be hard to parse from a typical __repr__ string.

21.1 Console Customization

The easiest way to customize console output for your object is to implement a __rich__ method. This method accepts no arguments, and should return an object that Rich knows how to render, such as a *Text* or *Table*. If you return a plain string it will be rendered as *Console Markup*. Here's an example:

```
class MyObject:
    def __rich__(self) -> str:
        return "[bold cyan]MyObject()"
```

If you were to print or log an instance of `MyObject` it would render as `MyObject()` in bold cyan. Naturally, you would want to put this to better use, perhaps by adding specialized syntax highlighting.

21.2 Console Render

The __rich__ method is limited to a single renderable object. For more advanced rendering, add a __rich_console__ method to your class.

The __rich_console__ method should accept a *Console* and a *ConsoleOptions* instance. It should return an iterable of other renderable objects. Although that means it *could* return a container such as a list, it generally easier implemented by using the `yield` statement (making the method a generator).

Here's an example of a __rich_console__ method:

```
from dataclasses import dataclass
from rich.console import Console, ConsoleOptions, RenderResult
from rich.table import Table

@dataclass
class Student:
    id: int
    name: str
    age: int
    def __rich_console__(self, console: Console, options: ConsoleOptions) ->␣
↪RenderResult:
```

(continues on next page)

(continued from previous page)

```
        yield f"[b]Student:[/b] #{self.id}"
        my_table = Table("Attribute", "Value")
        my_table.add_row("name", self.name)
        my_table.add_row("age", str(self.age))
        yield my_table
```

If you were to print a `Student` instance, it would render a simple table to the terminal.

21.2.1 Low Level Render

For complete control over how a custom object is rendered to the terminal, you can yield *`Segment`* objects. A Segment consists of a piece of text and an optional Style. The following example writes multi-colored text when rendering a `MyObject` instance:

```
class MyObject:
    def __rich_console__(self, console: Console, options: ConsoleOptions) ->␣
↪RenderResult:
        yield Segment("My", Style(color="magenta"))
        yield Segment("Object", Style(color="green"))
        yield Segment("()", Style(color="cyan"))
```

21.2.2 Measuring Renderables

Sometimes Rich needs to know how many characters an object will take up when rendering. The *`Table`* class, for instance, will use this information to calculate the optimal dimensions for the columns. If you aren't using one of the renderable objects in the Rich module, you will need to supply a `__rich_measure__` method which accepts a *`Console`* and the maximum width and returns a *`Measurement`* object. The Measurement object should contain the *minimum* and *maximum* number of characters required to render.

For example, if we are rendering a chess board, it would require a minimum of 8 characters to render. The maximum can be left as the maximum available width (assuming a centered board):

```
class ChessBoard:
    def __rich_measure__(self, console: Console, max_width: int) -> Measurement:
        return Measurement(8, max_width)
```

CHAPTER

TWENTYTWO

REFERENCE

22.1 rich.align

class rich.align.**Align**(*renderable: RenderableType*, *align: typing_extensions.Literal[left, center, right] = 'left'*, *style: Union[str, Style] = None*, ***, *vertical: typing_extensions.Literal[top, middle, bottom] = None*, *pad: bool = True*, *width: int = None*, *height: int = None*)

Align a renderable by adding spaces if necessary.

Parameters

- **renderable** (*RenderableType*) – A console renderable.
- **align** (*AlignMethod*) – One of “left”, “center”, or “right””
- **style** (*StyleType, optional*) – An optional style to apply to the background.
- **vertical** (*Optional[VerticalAlginMethod], optional*) – Optional vertical align, one of “top”, “middle”, or “bottom”. Defaults to None.
- **pad** (*bool, optional*) – Pad the right with spaces. Defaults to True.
- **width** (*int, optional*) – Restrict contents to given width, or None to use default width. Defaults to None.
- **height** (*int, optional*) – Set height of align renderable, or None to fit to contents. Defaults to None.

Raises **ValueError** – if `align` is not one of the expected values.

classmethod center(*renderable: RenderableType*, *style: Union[str, Style] = None*, ***, *vertical: typing_extensions.Literal[top, middle, bottom] = None*, *pad: bool = True*, *width: int = None*, *height: int = None*) → *Align*

Align a renderable to the center.

classmethod left(*renderable: RenderableType*, *style: Union[str, Style] = None*, ***, *vertical: typing_extensions.Literal[top, middle, bottom] = None*, *pad: bool = True*, *width: int = None*, *height: int = None*) → *Align*

Align a renderable to the left.

classmethod right(*renderable: RenderableType*, *style: Union[str, Style] = None*, ***, *vertical: typing_extensions.Literal[top, middle, bottom] = None*, *pad: bool = True*, *width: int = None*, *height: int = None*) → *Align*

Align a renderable to the right.

class rich.align.**VerticalCenter**(*renderable: RenderableType*, *style: Union[str, Style] = None*)

Vertically aligns a renderable.

Warns

- **This class is deprecated and may be removed in a future version. Use Align class with**
- **`vertical="middle"`.**

Parameters **renderable** (*RenderableType*) – A renderable object.

22.2 rich.bar

class rich.bar.**Bar**(*size: float*, *begin: float*, *end: float*, ***, *width: Optional[int] = None*, *color: Union[rich.color.Color, str] = 'default'*, *bgcolor: Union[rich.color.Color, str] = 'default'*)

Renders a solid block bar.

Parameters

- **size** (*float*) – Value for the end of the bar.
- **begin** (*float*) – Begin point (between 0 and size, inclusive).
- **end** (*float*) – End point (between 0 and size, inclusive).
- **width** (*int, optional*) – Width of the bar, or None for maximum width. Defaults to None.
- **color** (*Union[Color, str], optional*) – Color of the bar. Defaults to "default".
- **bgcolor** (*Union[Color, str], optional*) – Color of bar background. Defaults to "default".

22.3 rich.color

class rich.color.**Color**(*name: str*, *type: rich.color.ColorType*, *number: Optional[int] = None*, *triplet: Optional[rich.color_triplet.ColorTriplet] = None*)

Terminal color definition.

classmethod **default**() → *rich.color.Color*

Get a Color instance representing the default color.

Returns Default color.

Return type *Color*

downgrade(*system: rich.color.ColorSystem*) → *rich.color.Color*

Downgrade a color system to a system with fewer colors.

classmethod **from_ansi**(*number: int*) → *rich.color.Color*

Create a Color number from it's 8-bit ansi number.

Parameters **number** (*int*) – A number between 0-255 inclusive.

Returns A new Color instance.

Return type *Color*

classmethod **from_rgb**(*red: float*, *green: float*, *blue: float*) → *rich.color.Color*

Create a truecolor from three color components in the range(0->255).

Parameters

- **red** (*float*) – Red component in range 0-255.

- **green** (*float*) – Green component in range 0-255.
- **blue** (*float*) – Blue component in range 0-255.

Returns A new color object.

Return type *Color*

classmethod from_triplet(*triplet: rich.color_triplet.ColorTriplet*) → *rich.color.Color*
Create a truecolor RGB color from a triplet of values.

Parameters **triplet** (*ColorTriplet*) – A color triplet containing red, green and blue components.

Returns A new color object.

Return type *Color*

get_ansi_codes(*foreground: bool = True*) → Tuple[str, ...]
Get the ANSI escape codes for this color.

get_truecolor(*theme: TerminalTheme = None*, *foreground=True*) → rich.color_triplet.ColorTriplet
Get an equivalent color triplet for this color.

Parameters

- **theme** (*TerminalTheme, optional*) – Optional terminal theme, or None to use default. Defaults to None.
- **foreground** (*bool, optional*) – True for a foreground color, or False for background. Defaults to True.

Returns A color triplet containing RGB components.

Return type ColorTriplet

property is_default
Check if the color is a default color.

property is_system_defined
Check if the color is ultimately defined by the system.

property name
The name of the color (typically the input to Color.parse).

property number
The color number, if a standard color, or None.

classmethod parse(*color: str*) → *rich.color.Color*
Parse a color definition.

property system
Get the native color system for this color.

property triplet
A triplet of color components, if an RGB color.

property type
The type of the color.

exception rich.color.**ColorParseError**
The color could not be parsed.

class rich.color.**ColorSystem**(*value*)
One of the 3 color system supported by terminals.

class rich.color.**ColorType**(*value*)
Type of color stored in Color class.

rich.color.**blend_rgb**(*color1: rich.color_triplet.ColorTriplet*, *color2: rich.color_triplet.ColorTriplet*, *cross_fade: float = 0.5*) → rich.color_triplet.ColorTriplet
Blend one RGB color in to another.

rich.color.**parse_rgb_hex**(*hex_color: str*) → rich.color_triplet.ColorTriplet
Parse six hex characters in to RGB triplet.

22.4 rich.columns

class rich.columns.**Columns**(*renderables: Iterable[Union[*rich.console.ConsoleRenderable, rich.console.RichCast*, str]] = None*, *padding: Union[int, Tuple[int], Tuple[int, int], Tuple[int, int, int, int]] = (0, 1)*, ***, *width: int = None*, *expand: bool = False*, *equal: bool = False*, *column_first: bool = False*, *right_to_left: bool = False*, *align: typing_extensions.Literal[left, center, right] = None*, *title: Union[str, Text] = None*)
Display renderables in neat columns.

Parameters

- **renderables** (*Iterable[RenderableType]*) – Any number of Rich renderables (including str).
- **width** (*int, optional*) – The desired width of the columns, or None to auto detect. Defaults to None.
- **padding** (*PaddingDimensions, optional*) – Optional padding around cells. Defaults to (0, 1).
- **expand** (*bool, optional*) – Expand columns to full width. Defaults to False.
- **equal** (*bool, optional*) – Arrange in to equal sized columns. Defaults to False.
- **column_first** (*bool, optional*) – Align items from top to bottom (rather than left to right). Defaults to False.
- **right_to_left** (*bool, optional*) – Start column from right hand side. Defaults to False.
- **align** (*str, optional*) – Align value (“left”, “right”, or “center”) or None for default. Defaults to None.
- **title** (*TextType, optional*) – Optional title for Columns.

add_renderable(*renderable: Union[*rich.console.ConsoleRenderable, rich.console.RichCast, *str]*) → None
Add a renderable to the columns.

Parameters **renderable** (*RenderableType*) – Any renderable object.

22.5 rich.console

class rich.console.**Capture**(*console:* rich.console.Console)

Context manager to capture the result of printing to the console. See *capture()* for how to use.

Parameters **console** (Console) – A console instance to capture output.

get() → str

Get the result of the capture.

exception rich.console.**CaptureError**

An error in the Capture context manager.

class rich.console.**Console**(**, color_system: Optional[typing_extensions.Literal[auto, standard, 256, truecolor, windows]] = 'auto', force_terminal: Optional[bool] = None, force_jupyter: Optional[bool] = None, force_interactive: Optional[bool] = None, soft_wrap: bool = False, theme: Optional[rich.theme.Theme] = None, stderr: bool = False, file: Optional[IO[str]] = None, quiet: bool = False, width: Optional[int] = None, height: Optional[int] = None, style: Optional[Union[str, rich.style.Style]] = None, no_color: Optional[bool] = None, tab_size: int = 8, record: bool = False, markup: bool = True, emoji: bool = True, highlight: bool = True, log_time: bool = True, log_path: bool = True, log_time_format: Union[str, Callable[[datetime.datetime], rich.text.Text]] = '[%X]', highlighter: Optional[Callable[[Union[str, Text]], Text]] = <rich.highlighter.ReprHighlighter object>, legacy_windows: Optional[bool] = None, safe_box: bool = True, get_datetime: Optional[Callable[[], datetime.datetime]] = None, get_time: Optional[Callable[[], float]] = None, _environ: Optional[Mapping[str, str]] = None*)

A high level console interface.

Parameters

- **color_system** (*str, optional*) – The color system supported by your terminal, either "standard", "256" or "truecolor". Leave as "auto" to autodetect.
- **force_terminal** (*Optional[bool], optional*) – Enable/disable terminal control codes, or None to auto-detect terminal. Defaults to None.
- **force_jupyter** (*Optional[bool], optional*) – Enable/disable Jupyter rendering, or None to auto-detect Jupyter. Defaults to None.
- **force_interactive** (*Optional[bool], optional*) – Enable/disable interactive mode, or None to auto detect. Defaults to None.
- **soft_wrap** (*Optional[bool], optional*) – Set soft wrap default on print method. Defaults to False.
- **theme** (Theme, *optional*) – An optional style theme object, or None for default theme.
- **stderr** (*bool, optional*) – Use stderr rather than stdout if file is not specified. Defaults to False.
- **file** (*IO, optional*) – A file object where the console should write to. Defaults to stdout.
- **quiet** (*bool, Optional*) – Boolean to suppress all output. Defaults to False.

- **width** (*int, optional*) – The width of the terminal. Leave as default to auto-detect width.
- **height** (*int, optional*) – The height of the terminal. Leave as default to auto-detect height.
- **style** (*StyleType, optional*) – Style to apply to all output, or None for no style. Defaults to None.
- **no_color** (*Optional[bool], optional*) – Enabled no color mode, or None to auto detect. Defaults to None.
- **tab_size** (*int, optional*) – Number of spaces used to replace a tab character. Defaults to 8.
- **record** (*bool, optional*) – Boolean to enable recording of terminal output, required to call *export_html()* and *export_text()*. Defaults to False.
- **markup** (*bool, optional*) – Boolean to enable *Console Markup*. Defaults to True.
- **emoji** (*bool, optional*) – Enable emoji code. Defaults to True.
- **highlight** (*bool, optional*) – Enable automatic highlighting. Defaults to True.
- **log_time** (*bool, optional*) – Boolean to enable logging of time by *log()* methods. Defaults to True.
- **log_path** (*bool, optional*) – Boolean to enable the logging of the caller by *log()*. Defaults to True.
- **log_time_format** (*Union[str, TimeFormatterCallable], optional*) – If `log_time` is enabled, either string for strftime or callable that formats the time. Defaults to "[%X] ".
- **highlighter** (*HighlighterType, optional*) – Default highlighter.
- **legacy_windows** (*bool, optional*) – Enable legacy Windows mode, or `None` to auto detect. Defaults to `None`.
- **safe_box** (*bool, optional*) – Restrict box options that don't render on legacy Windows.
- **get_datetime** (*Callable[[], datetime], optional*) – Callable that gets the current time as a datetime.datetime object (used by Console.log), or None for datetime.now.
- **get_time** (*Callable[[], time], optional*) – Callable that gets the current time in seconds, default uses time.monotonic.

begin_capture() → None
Begin capturing console output. Call *end_capture()* to exit capture mode and return output.

bell() → None
Play a 'bell' sound (if supported by the terminal).

capture() → *rich.console.Capture*
A context manager to *capture* the result of print() or log() in a string, rather than writing it to the console.

Example

```
>>> from rich.console import Console
>>> console = Console()
>>> with console.capture() as capture:
...     console.print("[bold magenta]Hello World[/]")
>>> print(capture.get())
```

Returns Context manager with disables writing to the terminal.

Return type *Capture*

clear(*home: bool = True*) → None
Clear the screen.

Parameters **home** (*bool, optional*) – Also move the cursor to 'home' position. Defaults to True.

clear_live() → None
Clear the Live instance.

property color_system
Get color system string.

Returns "standard", "256" or "truecolor".

Return type Optional[str]

control(*control_codes: Union[rich.control.Control, str]*) → None
Insert non-printing control codes.

Parameters **control_codes** (*str*) – Control codes, such as those that may move the cursor.

property encoding
Get the encoding of the console file, e.g. `"utf-8"`.

Returns A standard encoding string.

Return type str

end_capture() → str
End capture mode and return captured string.

Returns Console output.

Return type str

export_html(**, theme: Optional[rich.terminal_theme.TerminalTheme] = None, clear: bool = True, code_format: Optional[str] = None, inline_styles: bool = False*) → str
Generate HTML from console contents (requires record=True argument in constructor).

Parameters

- **theme** (*TerminalTheme, optional*) – TerminalTheme object containing console colors.
- **clear** (*bool, optional*) – Clear record buffer after exporting. Defaults to `True`.
- **code_format** (*str, optional*) – Format string to render HTML, should contain {foreground} {background} and {code}.
- **inline_styles** (*bool, optional*) – If `True` styles will be inlined in to spans, which makes files larger but easier to cut and paste markup. If `False`, styles will be embedded in a style tag. Defaults to False.

Returns String containing console contents as HTML.

Return type str

export_text(*, *clear: bool = True*, *styles: bool = False*) → str
Generate text from console contents (requires record=True argument in constructor).

Parameters

- **clear** (*bool, optional*) – Clear record buffer after exporting. Defaults to `True`.
- **styles** (*bool, optional*) – If `True`, ansi escape codes will be included. `False` for plain text. Defaults to `False`.

Returns String containing console contents.

Return type str

property file
Get the file object to write to.

get_style(*name: Union[str, rich.style.Style]*, *, *default: Optional[Union[str, rich.style.Style]] = None*) → *rich.style.Style*
Get a Style instance by it's theme name or parse a definition.

Parameters **name** (*str*) – The name of a style or a style definition.

Returns A Style object.

Return type *Style*

Raises **MissingStyle** – If no style could be parsed from name.

property height
Get the height of the console.

Returns The height (in lines) of the console.

Return type int

input(*prompt: Union[str, rich.text.Text] = ''*, *, *markup: bool = True*, *emoji: bool = True*, *password: bool = False*, *stream: Optional[TextIO] = None*) → str
Displays a prompt and waits for input from the user. The prompt may contain color / style.

Parameters

- **prompt** (*Union[str, Text]*) – Text to render in the prompt.
- **markup** (*bool, optional*) – Enable console markup (requires a str prompt). Defaults to True.
- **emoji** (*bool, optional*) – Enable emoji (requires a str prompt). Defaults to True.
- **password** – (bool, optional): Hide typed text. Defaults to False.
- **stream** – (TextIO, optional): Optional file to read input from (rather than stdin). Defaults to None.

Returns Text read from stdin.

Return type str

property is_dumb_terminal
Detect dumb terminal.

Returns True if writing to a dumb terminal, otherwise False.

Return type bool

property is_terminal
Check if the console is writing to a terminal.

Returns True if the console writing to a device capable of understanding terminal codes, otherwise False.

Return type bool

line(*count: int = 1*) → None
Write new line(s).

Parameters **count** (*int, optional*) – Number of new lines. Defaults to 1.

log(**objects: Any*, *sep=' '*, *end='\n'*, *style: Optional[Union[str, rich.style.Style]] = None*, *justify: Optional[typing_extensions.Literal[default, left, center, right, full]] = None*, *emoji: Optional[bool] = None*, *markup: Optional[bool] = None*, *highlight: Optional[bool] = None*, *log_locals: bool = False*, *_stack_offset=1*) → None
Log rich content to the terminal.

Parameters

- **objects** (*positional args*) – Objects to log to the terminal.
- **sep** (*str, optional*) – String to write between print data. Defaults to " ".
- **end** (*str, optional*) – String to write at end of print data. Defaults to "\n".
- **style** (*Union[str, Style], optional*) – A style to apply to output. Defaults to None.
- **justify** (*str, optional*) – One of "left", "right", "center", or "full". Defaults to None.
- **overflow** (*str, optional*) – Overflow method: "crop", "fold", or "ellipsis". Defaults to None.
- **emoji** (*Optional[bool], optional*) – Enable emoji code, or None to use console default. Defaults to None.
- **markup** (*Optional[bool], optional*) – Enable markup, or None to use console default. Defaults to None.
- **highlight** (*Optional[bool], optional*) – Enable automatic highlighting, or None to use console default. Defaults to None.
- **log_locals** (*bool, optional*) – Boolean to enable logging of locals where log() was called. Defaults to False.
- **_stack_offset** (*int, optional*) – Offset of caller from end of call stack. Defaults to 1.

property options
Get default console options.

out(**objects: Any*, *sep=' '*, *end='\n'*, *style: Optional[Union[str, rich.style.Style]] = None*, *highlight: Optional[bool] = None*) → None
Output to the terminal. This is a low-level way of writing to the terminal which unlike *print()* won't pretty print, wrap text, or apply markup, but will optionally apply highlighting and a basic style.

Parameters

- **sep** (*str, optional*) – String to write between print data. Defaults to " ".
- **end** (*str, optional*) – String to write at end of print data. Defaults to "\n".

- **style** (*Union[str, Style], optional*) – A style to apply to output. Defaults to None.
- **highlight** (*Optional[bool], optional*) – Enable automatic highlighting, or None to use console default. Defaults to None.

pager(*pager: Optional[rich.pager.Pager] = None*, *styles: bool = False*, *links: bool = False*) → *rich.console.PagerContext*

A context manager to display anything printed within a "pager". The pager application is defined by the system and will typically support at least pressing a key to scroll.

Parameters

- **pager** (*Pager, optional*) – A pager object, or None to use :class:~rich.pager.SystemPager`. Defaults to None.
- **styles** (*bool, optional*) – Show styles in pager. Defaults to False.
- **links** (*bool, optional*) – Show links in pager. Defaults to False.

Example

```
>>> from rich.console import Console
>>> from rich.__main__ import make_test_card
>>> console = Console()
>>> with console.pager():
        console.print(make_test_card())
```

Returns A context manager.

Return type *PagerContext*

pop_render_hook() → None

Pop the last renderhook from the stack.

pop_theme() → None

Remove theme from top of stack, restoring previous theme.

print(**objects: Any*, *sep=' '*, *end='\n'*, *style: Optional[Union[str, rich.style.Style]] = None*, *justify: Optional[typing_extensions.Literal[default, left, center, right, full]] = None*, *overflow: Optional[typing_extensions.Literal[fold, crop, ellipsis, ignore]] = None*, *no_wrap: Optional[bool] = None*, *emoji: Optional[bool] = None*, *markup: Optional[bool] = None*, *highlight: Optional[bool] = None*, *width: Optional[int] = None*, *height: Optional[int] = None*, *crop: bool = True*, *soft_wrap: Optional[bool] = None*) → None

Print to the console.

Parameters

- **objects** (*positional args*) – Objects to log to the terminal.
- **sep** (*str, optional*) – String to write between print data. Defaults to " ".
- **end** (*str, optional*) – String to write at end of print data. Defaults to "\n".
- **style** (*Union[str, Style], optional*) – A style to apply to output. Defaults to None.
- **justify** (*str, optional*) – Justify method: "default", "left", "right", "center", or "full". Defaults to None.

- **overflow** (*str, optional*) – Overflow method: “ignore”, “crop”, “fold”, or “ellipsis”. Defaults to None.
- **no_wrap** (*Optional[bool], optional*) – Disable word wrapping. Defaults to None.
- **emoji** (*Optional[bool], optional*) – Enable emoji code, or `None` to use console default. Defaults to `None`.
- **markup** (*Optional[bool], optional*) – Enable markup, or `None` to use console default. Defaults to `None`.
- **highlight** (*Optional[bool], optional*) – Enable automatic highlighting, or `None` to use console default. Defaults to `None`.
- **width** (*Optional[int], optional*) – Width of output, or `None` to auto-detect. Defaults to `None`.
- **crop** (*Optional[bool], optional*) – Crop output to width of terminal. Defaults to True.
- **soft_wrap** (*bool, optional*) – Enable soft wrap mode which disables word wrapping and cropping of text or None for Console default. Defaults to `None`.

print_exception(*, *width: Optional[int] = 100*, *extra_lines: int = 3*, *theme: Optional[str] = None*, *word_wrap: bool = False*, *show_locals: bool = False*) → None

Prints a rich render of the last exception and traceback.

Parameters

- **width** (*Optional[int], optional*) – Number of characters used to render code. Defaults to 88.
- **extra_lines** (*int, optional*) – Additional lines of code to render. Defaults to 3.
- **theme** (*str, optional*) – Override pygments theme used in traceback
- **word_wrap** (*bool, optional*) – Enable word wrapping of long lines. Defaults to False.
- **show_locals** (*bool, optional*) – Enable display of local variables. Defaults to False.

push_render_hook(*hook:* rich.console.RenderHook) → None

Add a new render hook to the stack.

Parameters **hook** (`RenderHook`) – Render hook instance.

push_theme(*theme:* rich.theme.Theme, *, *inherit: bool = True*) → None

Push a new theme on to the top of the stack, replacing the styles from the previous theme. Generally speaking, you should call *use_theme()* to get a context manager, rather than calling this method directly.

Parameters

- **theme** (`Theme`) – A theme instance.
- **inherit** (*bool, optional*) – Inherit existing styles. Defaults to True.

render(*renderable: Union[*rich.console.ConsoleRenderable, rich.console.RichCast*, str]*, *options: Optional[*rich.console.ConsoleOptions*] = None*) → Iterable[*rich.segment.Segment*]

Render an object in to an iterable of *Segment* instances.

This method contains the logic for rendering objects with the console protocol. You are unlikely to need to use it directly, unless you are extending the library.

Parameters

- **renderable** (*RenderableType*) – An object supporting the console protocol, or an object that may be converted to a string.
- **options** (ConsoleOptions, *optional*) – An options object, or None to use self.options. Defaults to None.

Returns An iterable of segments that may be rendered.

Return type Iterable[*Segment*]

render_lines(*renderable: Union[*rich.console.ConsoleRenderable, rich.console.RichCast, *str]*, *options: Optional[*rich.console.ConsoleOptions*] = None*, **, style: Optional[*rich.style.Style*] = None*, *pad: bool = True*, *new_lines: bool = False*) → List[List[*rich.segment.Segment*]]

Render objects in to a list of lines.

The output of render_lines is useful when further formatting of rendered console text is required, such as the Panel class which draws a border around any renderable object.

Args: renderable (RenderableType): Any object renderable in the console. options (Optional[ConsoleOptions], optional): Console options, or None to use self.options. Default to None. style (Style, optional): Optional style to apply to renderables. Defaults to None. pad (bool, optional): Pad lines shorter than render width. Defaults to True. new_lines (bool, optional): Include “

” characters at end of lines.

Returns: List[List[Segment]]: A list of lines, where a line is a list of Segment objects.

render_str(*text: str*, **, style: Union[str,* rich.style.Style*] = ''*, *justify: Optional[typing_extensions.Literal[default, left, center, right, full]] = None*, *overflow: Optional[typing_extensions.Literal[fold, crop, ellipsis, ignore]] = None*, *emoji: Optional[bool] = None*, *markup: Optional[bool] = None*, *highlight: Optional[bool] = None*, *highlighter: Optional[Callable[[Union[str,* rich.text.Text*]],* rich.text.Text*]] = None*) → *rich.text.Text*

Convert a string to a Text instance. This is is called automatically if you print or log a string.

Parameters

- **text** (*str*) – Text to render.
- **style** (*Union[str,* Style*], optional*) – Style to apply to rendered text.
- **justify** (*str, optional*) – Justify method: “default”, “left”, “center”, “full”, or “right”. Defaults to None.
- **overflow** (*str, optional*) – Overflow method: “crop”, “fold”, or “ellipsis”. Defaults to None.
- **emoji** (*Optional[bool], optional*) – Enable emoji, or None to use Console default.
- **markup** (*Optional[bool], optional*) – Enable markup, or None to use Console default.
- **highlight** (*Optional[bool], optional*) – Enable highlighting, or None to use Console default.
- **highlighter** (*HighlighterType, optional*) – Optional highlighter to apply.

Returns Renderable object.

Return type *ConsoleRenderable*

rule(*title: Union[str, rich.text.Text] = ''*, ***, *characters: str = '–'*, *style: Union[str, rich.style.Style] = 'rule.line'*, *align: typing_extensions.Literal[left, center, right] = 'center'*) → None
Draw a line with optional centered title.

Parameters

- **title** (*str, optional*) – Text to render over the rule. Defaults to "".
- **characters** (*str, optional*) – Character(s) to form the line. Defaults to "–".
- **style** (*str, optional*) – Style of line. Defaults to "rule.line".
- **align** (*str, optional*) – How to align the title, one of "left", "center", or "right". Defaults to "center".

save_html(*path: str*, ***, *theme: Optional[rich.terminal_theme.TerminalTheme] = None*, *clear: bool = True*, *code_format='<!DOCTYPE html>\n<head>\n<meta charset="UTF-8">\n<style>\n{stylesheet}\nbody {{\n color: {foreground};\n background-color: {background};\n}}\n</style>\n</head>\n<html>\n<body>\n <code>\n <pre style="font-family:Menlo,\'DejaVu Sans Mono\',consolas,\'Courier New\',monospace">{code}</pre>\n </code>\n</body>\n</html>\n'*, *inline_styles: bool = False*) → None
Generate HTML from console contents and write to a file (requires record=True argument in constructor).

Parameters

- **path** (*str*) – Path to write html file.
- **theme** (*TerminalTheme, optional*) – TerminalTheme object containing console colors.
- **clear** (*bool, optional*) – Clear record buffer after exporting. Defaults to `True`.
- **code_format** (*str, optional*) – Format string to render HTML, should contain {foreground} {background} and {code}.
- **inline_styles** (*bool, optional*) – If `True` styles will be inlined in to spans, which makes files larger but easier to cut and paste markup. If `False`, styles will be embedded in a style tag. Defaults to False.

save_text(*path: str*, ***, *clear: bool = True*, *styles: bool = False*) → None
Generate text from console and save to a given location (requires record=True argument in constructor).

Parameters

- **path** (*str*) – Path to write text files.
- **clear** (*bool, optional*) – Clear record buffer after exporting. Defaults to `True`.
- **styles** (*bool, optional*) – If `True`, ansi style codes will be included. `False` for plain text. Defaults to `False`.

screen(*hide_cursor: bool = True*, *style: Optional[Union[str, rich.style.Style]] = None*) → *rich.console.ScreenContext*
Context manager to enable and disable 'alternative screen' mode.

Parameters

- **hide_cursor** (*bool, optional*) – Also hide the cursor. Defaults to False.
- **style** (*Style, optional*) – Optional style for screen. Defaults to None.

Returns Context which enables alternate screen on enter, and disables it on exit.

Return type *~ScreenContext*

set_alt_screen(*enable: bool = True*) → bool
Enables alternative screen mode.

Note, if you enable this mode, you should ensure that is disabled before the application exits. See `screen()` for a context manager that handles this for you.

Parameters **enable** (*bool, optional*) – Enable (True) or disable (False) alternate screen. Defaults to True.

Returns True if the control codes were written.

Return type bool

set_live(*live: Live*) → None
Set Live instance. Used by Live context manager.

Parameters **live** (Live) – Live instance using this Console.

Raises **errors.LiveError** – If this Console has a Live context currently active.

show_cursor(*show: bool = True*) → bool
Show or hide the cursor.

Parameters **show** (*bool, optional*) – Set visibility of the cursor.

property size
Get the size of the console.

Returns A named tuple containing the dimensions.

Return type *ConsoleDimensions*

status(*status: Union[*rich.console.ConsoleRenderable, rich.console.RichCast, *str]*, *, *spinner: str = 'dots'*, *spinner_style: str = 'status.spinner'*, *speed: float = 1.0*, *refresh_per_second: float = 12.5*) → Status
Display a status and spinner.

Parameters

- **status** (*RenderableType*) – A status renderable (str or Text typically).
- **console** (Console, *optional*) – Console instance to use, or None for global console. Defaults to None.
- **spinner** (*str, optional*) – Name of spinner animation (see python -m rich.spinner). Defaults to “dots”.
- **spinner_style** (*StyleType, optional*) – Style of spinner. Defaults to “status.spinner”.
- **speed** (*float, optional*) – Speed factor for spinner animation. Defaults to 1.0.
- **refresh_per_second** (*float, optional*) – Number of refreshes per second. Defaults to 12.5.

Returns A Status object that may be used as a context manager.

Return type *Status*

use_theme(*theme:* rich.theme.Theme, *, *inherit: bool = True*) → *rich.console.ThemeContext*
Use a different theme for the duration of the context manager.

Parameters

- **theme** (Theme) – Theme instance to user.
- **inherit** (*bool, optional*) – Inherit existing console styles. Defaults to True.

Returns [description]

Return type *ThemeContext*

property width
Get the width of the console.

Returns The width (in characters) of the console.

Return type int

class rich.console.**ConsoleDimensions**(*width: int*, *height: int*)
Size of the terminal.

property height
The height of the console in lines.

property width
The width of the console in 'cells'.

class rich.console.**ConsoleOptions**(*size: rich.console.ConsoleDimensions*, *legacy_windows: bool*, *min_width: int*, *max_width: int*, *is_terminal: bool*, *encoding: str*, *justify: Optional[typing_extensions.Literal[default, left, center, right, full]] = None*, *overflow: Optional[typing_extensions.Literal[fold, crop, ellipsis, ignore]] = None*, *no_wrap: Optional[bool] = False*, *highlight: Optional[bool] = None*, *height: Optional[int] = None*)
Options for __rich_console__ method.

property ascii_only
Check if renderables should use ascii only.

copy() → *rich.console.ConsoleOptions*
Return a copy of the options.

Returns a copy of self.

Return type *ConsoleOptions*

encoding: str
Encoding of terminal.

height: Optional[int] = None
Height available, or None for no height limit.

highlight: Optional[bool] = None
Highlight override for render_str.

is_terminal: bool
True if the target is a terminal, otherwise False.

justify: Optional[typing_extensions.Literal[default, left, center, right, full]] = No
Justify value override for renderable.

legacy_windows: bool
flag for legacy windows.

Type legacy_windows

max_width: int
Maximum width of renderable.

min_width: int
Minimum width of renderable.

no_wrap: Optional[bool] = False
Disable wrapping for text.

overflow: Optional[typing_extensions.Literal[fold, crop, ellipsis, ignore]] = None
Overflow value override for renderable.

size: *rich.console.ConsoleDimensions*
Size of console.

update(*width: Union[int, rich.console.NoChange] = <rich.console.NoChange object>, min_width: Union[int, rich.console.NoChange] = <rich.console.NoChange object>, max_width: Union[int, rich.console.NoChange] = <rich.console.NoChange object>, justify: Union[typing_extensions.Literal[default, left, center, right, full], None, rich.console.NoChange] = <rich.console.NoChange object>, overflow: Union[typing_extensions.Literal[fold, crop, ellipsis, ignore], None, rich.console.NoChange] = <rich.console.NoChange object>, no_wrap: Union[bool, None, rich.console.NoChange] = <rich.console.NoChange object>, highlight: Union[bool, None, rich.console.NoChange] = <rich.console.NoChange object>, height: Union[int, None, rich.console.NoChange] = <rich.console.NoChange object>*) → *rich.console.ConsoleOptions*
Update values, return a copy.

update_width(*width: int*) → *rich.console.ConsoleOptions*
Update just the width, return a copy.

Parameters **width** (*int*) – New width (sets both min_width and max_width)

Returns New console options instance

Return type *~ConsoleOptions*

class rich.console.**ConsoleRenderable**(**args*, ***kwds*)
An object that supports the console protocol.

class rich.console.**ConsoleThreadLocals**(*theme_stack: rich.theme.ThemeStack, buffer: List[rich.segment.Segment] = <factory>, buffer_index: int = 0*)
Thread local values for Console context.

class rich.console.**NewLine**(*count: int = 1*)
A renderable to generate new line(s)

class rich.console.**PagerContext**(*console: rich.console.Console, pager: Optional[rich.pager.Pager] = None, styles: bool = False, links: bool = False*)
A context manager that 'pages' content. See *pager()* for usage.

class rich.console.**RenderGroup**(**renderables: Union[rich.console.ConsoleRenderable, rich.console.RichCast, str], fit: bool = True*)
Takes a group of renderables and returns a renderable object that renders the group.

Parameters

- **renderables** (*Iterable[RenderableType]*) – An iterable of renderable objects.
- **fit** (*bool, optional*) – Fit dimension of group to contents, or fill available space. Defaults to True.

class rich.console.**RenderHook**
Provides hooks in to the render process.

abstract process_renderables(*renderables: List[*rich.console.ConsoleRenderable*]*) → List[*rich.console.ConsoleRenderable*]

Called with a list of objects to render.

This method can return a new list of renderables, or modify and return the same list.

Parameters **renderables** (*List[ConsoleRenderable]*) – A number of renderable objects.

Returns A replacement list of renderables.

Return type List[*ConsoleRenderable*]

rich.console.**RenderResult**

The result of calling a __rich_console__ method.

alias of Iterable[Union[rich.console.ConsoleRenderable, rich.console.RichCast, str, rich.segment.Segment]]

rich.console.**RenderableType**

A type that may be rendered by Console.

alias of Union[rich.console.ConsoleRenderable, rich.console.RichCast, str]

class rich.console.**RichCast**(**args*, ***kwds*)

An object that may be 'cast' to a console renderable.

class rich.console.**ScreenContext**(*console: rich.console.Console*, *hide_cursor: bool*, *style: Union[str, rich.style.Style] = ''*)

A context manager that enables an alternative screen. See *screen()* for usage.

update(**renderables: Union[*rich.console.ConsoleRenderable, rich.console.RichCast, *str]*, *style: Optional[Union[str, rich.style.Style]] = None*) → None

Update the screen.

Parameters

- **renderable** (*RenderableType, optional*) – Optional renderable to replace current renderable, or None for no change. Defaults to None.
- **style** – (Style, optional): Replacement style, or None for no change. Defaults to None.

class rich.console.**ThemeContext**(*console: rich.console.Console*, *theme: rich.theme.Theme*, *inherit: bool = True*)

A context manager to use a temporary theme. See *use_theme()* for usage.

rich.console.**detect_legacy_windows**() → bool

Detect legacy Windows.

rich.console.**render_group**(*fit: bool = True*) → Callable

A decorator that turns an iterable of renderables in to a group.

Parameters **fit** (*bool, optional*) – Fit dimension of group to contents, or fill available space. Defaults to True.

22.6 rich.emoji

class rich.emoji.**Emoji**(*name: str*, *style: Union[str, rich.style.Style] = 'none'*)

classmethod replace(*text: str*) → str
Replace emoji markup with corresponding unicode characters.

Parameters **text** (*str*) – A string with emojis codes, e.g. "Hello :smiley:!"

Returns A string with emoji codes replaces with actual emoji.

Return type str

22.7 rich.highlighter

class rich.highlighter.**Highlighter**
Abstract base class for highlighters.

__call__(*text: Union[str, rich.text.Text]*) → *rich.text.Text*
Highlight a str or Text instance.

Parameters **text** (*Union[str, ~Text]*) – Text to highlight.

Raises **TypeError** – If not called with text or str.

Returns A test instance with highlighting applied.

Return type *Text*

abstract highlight(*text:* rich.text.Text) → None
Apply highlighting in place to text.

Parameters **text** (*~Text*) – A text object highlight.

class rich.highlighter.**NullHighlighter**
A highlighter object that doesn't highlight.

May be used to disable highlighting entirely.

highlight(*text:* rich.text.Text) → None
Nothing to do

class rich.highlighter.**RegexHighlighter**
Applies highlighting from a list of regular expressions.

highlight(*text:* rich.text.Text) → None
Highlight *rich.text.Text* using regular expressions.

Parameters **text** (*~Text*) – Text to highlighted.

class rich.highlighter.**ReprHighlighter**
Highlights the text typically produced from __repr__ methods.

22.8 rich

Rich text and beautiful formatting in the terminal.

rich.**get_console**() → Console
Get a global *Console* instance. This function is used when Rich requires a Console, and hasn't been explicitly given one.

Returns A console instance.

Return type *Console*

rich.**inspect**(*obj: Any*, ***, *console: Console = None*, *title: str = None*, *help: bool = False*, *methods: bool = False*, *docs: bool = True*, *private: bool = False*, *dunder: bool = False*, *sort: bool = True*, *all: bool = False*, *value: bool = True*)
Inspect any Python object.

- inspect(<OBJECT>) to see summarized info.
- inspect(<OBJECT>, methods=True) to see methods.
- inspect(<OBJECT>, help=True) to see full (non-abbreviated) help.
- inspect(<OBJECT>, private=True) to see private attributes (single underscore).
- inspect(<OBJECT>, dunder=True) to see attributes beginning with double underscore.
- inspect(<OBJECT>, all=True) to see all attributes.

Parameters

- **obj** (*Any*) – An object to inspect.
- **title** (*str, optional*) – Title to display over inspect result, or None use type. Defaults to None.
- **help** (*bool, optional*) – Show full help text rather than just first paragraph. Defaults to False.
- **methods** (*bool, optional*) – Enable inspection of callables. Defaults to False.
- **docs** (*bool, optional*) – Also render doc strings. Defaults to True.
- **private** (*bool, optional*) – Show private attributes (beginning with underscore). Defaults to False.
- **dunder** (*bool, optional*) – Show attributes starting with double underscore. Defaults to False.
- **sort** (*bool, optional*) – Sort attributes alphabetically. Defaults to True.
- **all** (*bool, optional*) – Show all attributes. Defaults to False.
- **value** (*bool, optional*) – Pretty print value. Defaults to True.

rich.**print**(**objects: Any*, *sep=' '*, *end='\n'*, *file: Optional[IO[str]] = None*, *flush: bool = False*)
Print object(s) supplied via positional arguments. This function has an identical signature to the built-in print. For more advanced features, see the *Console* class.

Parameters

- **sep** (*str, optional*) – Separator between printed objects. Defaults to " ".
- **end** (*str, optional*) – Character to write at end of output. Defaults to "\n".

- **file** (*IO[str], optional*) – File to write to, or None for stdout. Defaults to None.
- **flush** (*bool, optional*) – Has no effect as Rich always flushes output. Defaults to False.

rich.**reconfigure**(**args*, ***kwargs*) → None
Reconfigures the global console bu replacing it with another.

Parameters **console** (Console) – Replacement console instance.

22.9 rich.layout

class rich.layout.**Layout**(*renderable: Optional[Union[*rich.console.ConsoleRenderable*, *rich.console.RichCast*, *str]] = None*, ***, *direction: str = 'vertical'*, *size: Optional[int] = None*, *minimum_size: int = 1*, *ratio: int = 1*, *name: Optional[str] = None*, *visible: bool = True*, *height: Optional[int] = None*)
A renderable to divide a fixed height in to rows or columns.

Parameters

- **renderable** (*RenderableType, optional*) – Renderable content, or None for placeholder. Defaults to None.
- **direction** (*str, optional*) – Direction of split, one of "vertical" or "horizontal". Defaults to "vertical".
- **size** (*int, optional*) – Optional fixed size of layout. Defaults to None.
- **minimum_size** (*int, optional*) – Minimum size of layout. Defaults to 1.
- **ratio** (*int, optional*) – Optional ratio for flexible layout. Defaults to 1.
- **name** (*str, optional*) – Optional identifier for Layout. Defaults to None.
- **visible** (*bool, optional*) – Visibility of layout. Defaults to True.

property children
Gets (visible) layout children.

get(*name: str*) → Optional[*rich.layout.Layout*]
Get a named layout, or None if it doesn't exist.

Parameters **name** (*str*) – Name of layout.

Returns Layout instance or None if no layout was found.

Return type Optional[*Layout*]

property renderable
Layout renderable.

split(**layouts*, *direction: Optional[typing_extensions.Literal[horizontal, vertical]] = None*) → None
Split the layout in to multiple sub-layours.

Parameters

- ***layouts** (Layout) – Positional arguments should be (sub) Layout instances.
- **direction** (*Direction, optional*) – One of "horizontal" or "vertical" or None for no change. Defaults to None.

property tree
Get a tree renderable to show layout structure.

update(*renderable: Union[*rich.console.ConsoleRenderable, rich.console.RichCast, *str]*) → None
Update renderable.

Parameters **renderable** (*RenderableType*) – New renderable object.

22.10 rich.live

class rich.live.**Live**(*renderable: Optional[Union[*rich.console.ConsoleRenderable, rich.console.RichCast, *str]] = None*, ***, *console: Optional[*rich.console.Console*] = None*, *screen: bool = False*, *auto_refresh: bool = True*, *refresh_per_second: float = 4*, *transient: bool = False*, *redirect_stdout: bool = True*, *redirect_stderr: bool = True*, *vertical_overflow: typing_extensions.Literal[crop, ellipsis, visible] = 'ellipsis'*, *get_renderable: Optional[Callable[], Union[*rich.console.ConsoleRenderable, rich.console.RichCast, *str]]] = None*)
Renders an auto-updating live display of any given renderable.

Parameters

- **renderable** (*RenderableType, optional*) – The renderable to live display. Defaults to displaying nothing.
- **console** (*Console, optional*) – Optional Console instance. Default will an internal Console instance writing to stdout.
- **screen** (*bool, optional*) – Enable alternate screen mode. Defaults to False.
- **auto_refresh** (*bool, optional*) – Enable auto refresh. If disabled, you will need to call *refresh()* or *update()* with refresh flag. Defaults to True
- **refresh_per_second** (*float, optional*) – Number of times per second to refresh the live display. Defaults to 1.
- **transient** (*bool, optional*) – Clear the renderable on exit (has no effect when screen=True). Defaults to False.
- **redirect_stdout** (*bool, optional*) – Enable redirection of stdout, so `print` may be used. Defaults to True.
- **redirect_stderr** (*bool, optional*) – Enable redirection of stderr. Defaults to True.
- **vertical_overflow** (*VerticalOverflowMethod, optional*) – How to handle renderable when it is too tall for the console. Defaults to "ellipsis".
- **get_renderable** (*Callable[[], RenderableType], optional*) – Optional callable to get renderable. Defaults to None.

process_renderables(*renderables: List[*rich.console.ConsoleRenderable*]*) → List[*rich.console.ConsoleRenderable*]
Process renderables to restore cursor and display progress.

refresh() → None
Update the display of the Live Render.

property renderable
Get the renderable that is being displayed

Returns Displayed renderable.

Return type RenderableType

start(*refresh=False*) → None
Start live rendering display.

Parameters **refresh** (*bool, optional*) – Also refresh. Defaults to False.

stop() → None
Stop live rendering display.

update(*renderable: Union[rich.console.ConsoleRenderable, rich.console.RichCast, str], *, refresh: bool = False*) → None
Update the renderable that is being displayed

Parameters

- **renderable** (*RenderableType*) – New renderable to use.
- **refresh** (*bool, optional*) – Refresh the display. Defaults to False.

22.11 rich.logging

class rich.logging.**RichHandler**(*level: Union[int, str] = 0, console: Optional[rich.console.Console] = None, *, show_time: bool = True, show_level: bool = True, show_path: bool = True, enable_link_path: bool = True, highlighter: Optional[rich.highlighter.Highlighter] = None, markup: bool = False, rich_tracebacks: bool = False, tracebacks_width: Optional[int] = None, tracebacks_extra_lines: int = 3, tracebacks_theme: Optional[str] = None, tracebacks_word_wrap: bool = True, tracebacks_show_locals: bool = False, locals_max_length: int = 10, locals_max_string: int = 80, log_time_format: Union[str, Callable[[datetime.datetime], rich.text.Text]] = '[%x %X]'*)
A logging handler that renders output with Rich. The time / level / message and file are displayed in columns. The level is color coded, and the message is syntax highlighted.

Note: Be careful when enabling console markup in log messages if you have configured logging for libraries not under your control. If a dependency writes messages containing square brackets, it may not produce the intended output.

Parameters

- **level** (*Union[int, str], optional*) – Log level. Defaults to logging.NOTSET.
- **console** (*Console*, optional) – Optional console instance to write logs. Default will use a global console instance writing to stdout.
- **show_time** (*bool, optional*) – Show a column for the time. Defaults to True.
- **show_level** (*bool, optional*) – Show a column for the level. Defaults to True.
- **show_path** (*bool, optional*) – Show the path to the original log call. Defaults to True.

- **enable_link_path** (*bool, optional*) – Enable terminal link of path column to file. Defaults to True.
- **highlighter** (*Highlighter, optional*) – Highlighter to style log messages, or None to use ReprHighlighter. Defaults to None.
- **markup** (*bool, optional*) – Enable console markup in log messages. Defaults to False.
- **rich_tracebacks** (*bool, optional*) – Enable rich tracebacks with syntax highlighting and formatting. Defaults to False.
- **tracebacks_width** (*Optional[int], optional*) – Number of characters used to render tracebacks, or None for full width. Defaults to None.
- **tracebacks_extra_lines** (*int, optional*) – Additional lines of code to render tracebacks, or None for full width. Defaults to None.
- **tracebacks_theme** (*str, optional*) – Override pygments theme used in traceback.
- **tracebacks_word_wrap** (*bool, optional*) – Enable word wrapping of long tracebacks lines. Defaults to True.
- **tracebacks_show_locals** (*bool, optional*) – Enable display of locals in tracebacks. Defaults to False.
- **locals_max_length** (*int, optional*) – Maximum length of containers before abbreviating, or None for no abbreviation. Defaults to 10.
- **locals_max_string** (*int, optional*) – Maximum length of string before truncating, or None to disable. Defaults to 80.
- **log_time_format** (*Union[str, TimeFormatterCallable], optional*) – If `log_time` is enabled, either string for strftime or callable that formats the time. Defaults to "[%x %X] ".

HIGHLIGHTER_CLASS
alias of *rich.highlighter.ReprHighlighter*

emit(*record: logging.LogRecord*) → None
Invoked by logging.

get_level_text(*record: logging.LogRecord*) → *rich.text.Text*
Get the level name from the record.

Parameters **record** (*LogRecord*) – LogRecord instance.

Returns A tuple of the style and level name.

Return type *Text*

render(*, *record: logging.LogRecord*, *traceback: Optional[*rich.traceback.Traceback*]*, *message_renderable:* rich.console.ConsoleRenderable) → *rich.console.ConsoleRenderable*
Render log for display.

Parameters

- **record** (*LogRecord*) – logging Record.
- **traceback** (*Optional[Traceback]*) – Traceback instance or None for no Traceback.
- **message_renderable** (*ConsoleRenderable*) – Renderable (typically Text) containing log message contents.

Returns Renderable to display log.

Return type *ConsoleRenderable*

render_message(*record: logging.LogRecord*, *message: str*) → *rich.console.ConsoleRenderable*
Render message text in to Text.

record (LogRecord): logging Record. message (str): String cotaining log message.

Returns Renderable to display log message.

Return type *ConsoleRenderable*

22.12 rich.markdown

class rich.markdown.**BlockQuote**
A block quote.

on_child_close(*context: rich.markdown.MarkdownContext*, *child: rich.markdown.MarkdownElement*) → bool
Called when a child element is closed.

This method allows a parent element to take over rendering of its children.

Parameters

- **context** (MarkdownContext) – The markdown context.
- **child** (*MarkdownElement*) – The child markdown element.

Returns Return True to render the element, or False to not render the element.

Return type bool

class rich.markdown.**CodeBlock**(*lexer_name: str*, *theme: str*)
A code block with syntax highlighting.

classmethod create(*markdown: rich.markdown.Markdown*, *node: Any*) → *rich.markdown.CodeBlock*
Factory to create markdown element,

Parameters

- **markdown** (Markdown) – THe parent Markdown object.
- **node** (*Any*) – A node from Pygments.

Returns A new markdown element

Return type MarkdownElement

class rich.markdown.**Heading**(*level: int*)
A heading.

classmethod create(*markdown: rich.markdown.Markdown*, *node: Any*) → *rich.markdown.Heading*
Factory to create markdown element,

Parameters

- **markdown** (Markdown) – THe parent Markdown object.
- **node** (*Any*) – A node from Pygments.

Returns A new markdown element

Return type MarkdownElement

on_enter(*context:* rich.markdown.MarkdownContext) → None
Called when the node is entered.

Parameters **context** (MarkdownContext) – The markdown context.

class rich.markdown.**HorizontalRule**
A horizontal rule to divide sections.

class rich.markdown.**ImageItem**(*destination: str*, *hyperlinks: bool*)
Renders a placeholder for an image.

classmethod create(*markdown:* rich.markdown.Markdown, *node: Any*) → rich.markdown.MarkdownElement
Factory to create markdown element,

Parameters

- **markdown** (Markdown) – THe parent Markdown object.
- **node** (*Any*) – A node from Pygments.

Returns A new markdown element

Return type MarkdownElement

on_enter(*context:* rich.markdown.MarkdownContext) → None
Called when the node is entered.

Parameters **context** (MarkdownContext) – The markdown context.

class rich.markdown.**ListElement**(*list_type: str*, *list_start: Optional[int]*)
A list element.

classmethod create(*markdown:* rich.markdown.Markdown, *node: Any*) → *rich.markdown.ListElement*
Factory to create markdown element,

Parameters

- **markdown** (Markdown) – THe parent Markdown object.
- **node** (*Any*) – A node from Pygments.

Returns A new markdown element

Return type MarkdownElement

on_child_close(*context:* rich.markdown.MarkdownContext, *child: rich.markdown.MarkdownElement*) → bool
Called when a child element is closed.

This method allows a parent element to take over rendering of its children.

Parameters

- **context** (MarkdownContext) – The markdown context.
- **child** (*MarkdownElement*) – The child markdown element.

Returns Return True to render the element, or False to not render the element.

Return type bool

class rich.markdown.**ListItem**
An item in a list.

on_child_close(*context: rich.markdown.MarkdownContext*, *child: rich.markdown.MarkdownElement*) → bool
Called when a child element is closed.

This method allows a parent element to take over rendering of its children.

Parameters

- **context** (MarkdownContext) – The markdown context.
- **child** (*MarkdownElement*) – The child markdown element.

Returns Return True to render the element, or False to not render the element.

Return type bool

class rich.markdown.**Markdown**(*markup: str*, *code_theme: str = 'monokai'*, *justify: Optional[typing_extensions.Literal[default, left, center, right, full]] = None*, *style: Union[str, rich.style.Style] = 'none'*, *hyperlinks: bool = True*, *inline_code_lexer: Optional[str] = None*, *inline_code_theme: Optional[str] = None*)
A Markdown renderable.

Parameters

- **markup** (*str*) – A string containing markdown.
- **code_theme** (*str, optional*) – Pygments theme for code blocks. Defaults to "monokai".
- **justify** (*JustifyMethod, optional*) – Justify value for paragraphs. Defaults to None.
- **style** (*Union[str, Style], optional*) – Optional style to apply to markdown.
- **hyperlinks** (*bool, optional*) – Enable hyperlinks. Defaults to True.
- **inline_code_lexer** – (str, optional): Lexer to use if inline code highlighting is enabled. Defaults to "python".
- **inline_code_theme** – (Optional[str], optional): Pygments theme for inline code highlighting, or None for no highlighting. Defaults to None.

class rich.markdown.**MarkdownContext**(*console: rich.console.Console*, *options: rich.console.ConsoleOptions*, *style: rich.style.Style*, *inline_code_lexer: Optional[str] = None*, *inline_code_theme: str = 'monokai'*)
Manages the console render state.

property current_style
Current style which is the product of all styles on the stack.

enter_style(*style_name: Union[str, rich.style.Style]*) → *rich.style.Style*
Enter a style context.

leave_style() → *rich.style.Style*
Leave a style context.

on_text(*text: str*, *node_type: str*) → None
Called when the parser visits text.

class rich.markdown.**Paragraph**(*justify: typing_extensions.Literal[default, left, center, right, full]*)
A Paragraph.

classmethod create(*markdown: rich.markdown.Markdown*, *node*) → *rich.markdown.Paragraph*
Factory to create markdown element,

Parameters

- **markdown** (Markdown) – THe parent Markdown object.
- **node** (*Any*) – A node from Pygments.

Returns A new markdown element

Return type MarkdownElement

class rich.markdown.**TextElement**

Base class for elements that render text.

on_enter(*context:* rich.markdown.MarkdownContext) → None

Called when the node is entered.

Parameters **context** (MarkdownContext) – The markdown context.

on_leave(*context:* rich.markdown.MarkdownContext) → None

Called when the parser leaves the element.

Parameters **context** (MarkdownContext) – [description]

on_text(*context:* rich.markdown.MarkdownContext, *text: Union[str,* rich.text.Text*]*) → None

Called when text is parsed.

Parameters **context** (MarkdownContext) – The markdown context.

class rich.markdown.**UnknownElement**

An unknown element.

Hopefully there will be no unknown elements, and we will have a MarkdownElement for everything in the document.

22.13 rich.markup

class rich.markup.**Tag**(*name: str*, *parameters: Optional[str]*)

A tag in console markup.

property markup

Get the string representation of this tag.

property name

The tag name. e.g. 'bold'.

property parameters

Any additional parameters after the name.

rich.markup.**escape**(*markup: str*, *_escape=<built-in method sub of re.Pattern object>*) → str

Escapes text so that it won't be interpreted as markup.

Parameters **markup** (*str*) – Content to be inserted in to markup.

Returns Markup with square brackets escaped.

Return type str

rich.markup.**render**(*markup: str*, *style: Union[str,* rich.style.Style*] = ''*, *emoji: bool = True*) → *rich.text.Text*

Render console markup in to a Text instance.

Parameters

- **markup** (*str*) – A string containing console markup.

- **emoji** (*bool, optional*) – Also render emoji code. Defaults to True.

Raises **MarkupError** – If there is a syntax error in the markup.

Returns A test instance.

Return type *Text*

22.14 rich.measure

class rich.measure.**Measurement**(*minimum: int*, *maximum: int*)

Stores the minimum and maximum widths (in characters) required to render an object.

clamp(*min_width: Optional[int] = None*, *max_width: Optional[int] = None*) → *rich.measure.Measurement*

Clamp a measurement within the specified range.

Parameters

- **min_width** (*int*) – Minimum desired width, or None for no minimum. Defaults to None.
- **max_width** (*int*) – Maximum desired width, or None for no maximum. Defaults to None.

Returns New Measurement object.

Return type *Measurement*

classmethod get(*console: Console*, *renderable: RenderableType*, *max_width: int = None*) → *Measurement*

Get a measurement for a renderable.

Parameters

- **console** (Console) – Console instance.
- **renderable** (*RenderableType*) – An object that may be rendered with Rich.
- **max_width** (*int, optional*) – The maximum width available, or None to use console.width. Defaults to None.

Raises **errors.NotRenderableError** – If the object is not renderable.

Returns Measurement object containing range of character widths required to render the object.

Return type *Measurement*

property maximum

Maximum number of cells required to render.

property minimum

Minimum number of cells required to render.

normalize() → *rich.measure.Measurement*

Get measurement that ensures that minimum <= maximum and minimum >= 0

Returns A normalized measurement.

Return type *Measurement*

property span

Get difference between maximum and minimum.

with_maximum(*width: int*) → *rich.measure.Measurement*
Get a RenderableWith where the widths are <= width.

Parameters **width** (*int*) – Maximum desired width.

Returns New Measurement object.

Return type *Measurement*

with_minimum(*width: int*) → *rich.measure.Measurement*
Get a RenderableWith where the widths are >= width.

Parameters **width** (*int*) – Minimum desired width.

Returns New Measurement object.

Return type *Measurement*

rich.measure.**measure_renderables**(*console: Console*, *renderables: Iterable[RenderableType]*, *max_width: int*) → *Measurement*
Get a measurement that would fit a number of renderables.

Parameters

- **console** (Console) – Console instance.
- **renderables** (*Iterable[RenderableType]*) – One or more renderable objects.
- **max_width** (*int*) – The maximum width available.

Returns Measurement object containing range of character widths required to contain all given renderables.

Return type *Measurement*

22.15 rich.padding

class rich.padding.**Padding**(*renderable: RenderableType*, *pad: PaddingDimensions = (0, 0, 0, 0)*, **, style: Union[str, rich.style.Style] = 'none'*, *expand: bool = True*)
Draw space around content.

Example

```
>>> print(Padding("Hello", (2, 4), style="on blue"))
```

Parameters

- **renderable** (*RenderableType*) – String or other renderable.
- **pad** (*Union[int, Tuple[int]]*) – Padding for top, right, bottom, and left borders. May be specified with 1, 2, or 4 integers (CSS style).
- **style** (*Union[str, Style], optional*) – Style for padding characters. Defaults to “none”.
- **expand** (*bool, optional*) – Expand padding to fit available width. Defaults to True.

classmethod indent(*renderable: RenderableType*, *level: int*) → *Padding*
Make padding instance to render an indent.

Parameters

- **renderable** (*RenderableType*) – String or other renderable.
- **level** (*int*) – Number of characters to indent.

Returns A Padding instance.

Return type *Padding*

static unpack(*pad: Union[int, Tuple[int], Tuple[int, int], Tuple[int, int, int, int]]*) → Tuple[int, int, int, int]

Unpack padding specified in CSS style.

22.16 rich.panel

class rich.panel.**Panel**(*renderable: RenderableType*, *box: rich.box.Box = Box(...)*, ***, *title: Union[str, Text] = None*, *title_align: typing_extensions.Literal[left, center, right] = 'center'*, *safe_box: Optional[bool] = None*, *expand: bool = True*, *style: Union[str, Style] = 'none'*, *border_style: Union[str, Style] = 'none'*, *width: Optional[int] = None*, *height: Optional[int] = None*, *padding: Union[int, Tuple[int], Tuple[int, int], Tuple[int, int, int, int]] = (0, 1)*, *highlight: bool = False*)

A console renderable that draws a border around its contents.

Example

```
>>> console.print(Panel("Hello, World!"))
```

Parameters

- **renderable** (*RenderableType*) – A console renderable object.
- **box** (*Box, optional*) – A Box instance that defines the look of the border (see *Box*. Defaults to box.ROUNDED.
- **safe_box** (*bool, optional*) – Disable box characters that don't display on windows legacy terminal with *raster* fonts. Defaults to True.
- **expand** (*bool, optional*) – If True the panel will stretch to fill the console width, otherwise it will be sized to fit the contents. Defaults to True.
- **style** (*str, optional*) – The style of the panel (border and contents). Defaults to "none".
- **border_style** (*str, optional*) – The style of the border. Defaults to "none".
- **width** (*Optional[int], optional*) – Optional width of panel. Defaults to None to auto-detect.
- **height** (*Optional[int], optional*) – Optional height of panel. Defaults to None to auto-detect.
- **padding** (*Optional[PaddingDimensions]*) – Optional padding around renderable. Defaults to 0.
- **highlight** (*bool, optional*) – Enable automatic highlighting of panel title (if str). Defaults to False.

classmethod fit(*renderable: RenderableType*, *box: rich.box.Box = Box(...)*, ***, *title: Union[str, Text] = None*, *title_align: typing_extensions.Literal[left, center, right] = 'center'*, *safe_box: Optional[bool] = None*, *style: Union[str, Style] = 'none'*, *border_style: Union[str, Style] = 'none'*, *width: Optional[int] = None*, *padding: Union[int, Tuple[int], Tuple[int, int], Tuple[int, int, int, int]] = (0, 1)*)
An alternative constructor that sets expand=False.

22.17 rich.pretty

class rich.pretty.**Node**(*key_repr: str = ''*, *value_repr: str = ''*, *open_brace: str = ''*, *close_brace: str = ''*, *empty: str = ''*, *last: bool = False*, *is_tuple: bool = False*, *children: Optional[List[rich.pretty.Node]] = None*)
A node in a repr tree. May be atomic or a container.

check_length(*start_length: int*, *max_length: int*) → bool
Check the length fits within a limit.

Parameters

- **start_length** (*int*) – Starting length of the line (indent, prefix, suffix).
- **max_length** (*int*) – Maximum length.

Returns True if the node can be rendered within max length, otherwise False.

Return type bool

iter_tokens() → Iterable[str]
Generate tokens for this node.

render(*max_width: int = 80*, *indent_size: int = 4*, *expand_all: bool = False*) → str
Render the node to a pretty repr.

Parameters

- **max_width** (*int, optional*) – Maximum width of the repr. Defaults to 80.
- **indent_size** (*int, optional*) – Size of indents. Defaults to 4.
- **expand_all** (*bool, optional*) – Expand all levels. Defaults to False.

Returns A repr string of the original object.

Return type str

property separator
Get separator between items.

class rich.pretty.**Pretty**(*_object: Any*, *highlighter: HighlighterType = None*, ***, *indent_size: int = 4*, *justify: JustifyMethod = None*, *overflow: Optional[OverflowMethod] = None*, *no_wrap: Optional[bool] = False*, *indent_guides: bool = False*, *max_length: int = None*, *max_string: int = None*, *expand_all: bool = False*, *margin: int = 0*, *insert_line: bool = False*)
A rich renderable that pretty prints an object.

Parameters

- **_object** (*Any*) – An object to pretty print.
- **highlighter** (*HighlighterType, optional*) – Highlighter object to apply to result, or None for ReprHighlighter. Defaults to None.
- **indent_size** (*int, optional*) – Number of spaces in indent. Defaults to 4.

- **justify** (*JustifyMethod, optional*) – Justify method, or None for default. Defaults to None.
- **overflow** (*OverflowMethod, optional*) – Overflow method, or None for default. Defaults to None.
- **no_wrap** (*Optional[bool], optional*) – Disable word wrapping. Defaults to False.
- **indent_guides** (*bool, optional*) – Enable indentation guides. Defaults to False.
- **max_length** (*int, optional*) – Maximum length of containers before abbreviating, or None for no abbreviation. Defaults to None.
- **max_string** (*int, optional*) – Maximum length of string before truncating, or None to disable. Defaults to None.
- **expand_all** (*bool, optional*) – Expand all containers. Defaults to False.
- **margin** (*int, optional*) – Subtrace a margin from width to force containers to expand earlier. Defaults to 0.
- **insert_line** (*bool, optional*) – Insert a new line if the output has multiple new lines. Defaults to False.

rich.pretty.**install**(*console: Console = None*, *overflow: OverflowMethod = 'ignore'*, *crop: bool = False*, *indent_guides: bool = False*, *max_length: int = None*, *max_string: int = None*, *expand_all: bool = False*) → None

Install automatic pretty printing in the Python REPL.

Parameters

- **console** (Console, *optional*) – Console instance or None to use global console. Defaults to None.
- **overflow** (*Optional[OverflowMethod], optional*) – Overflow method. Defaults to "ignore".
- **crop** (*Optional[bool], optional*) – Enable cropping of long lines. Defaults to False.
- **indent_guides** (*bool, optional*) – Enable indentation guides. Defaults to False.
- **max_length** (*int, optional*) – Maximum length of containers before abbreviating, or None for no abbreviation. Defaults to None.
- **max_string** (*int, optional*) – Maximum length of string before truncating, or None to disable. Defaults to None.
- **expand_all** (*bool, optional*) – Expand all containers. Defaults to False

rich.pretty.**is_expandable**(*obj: Any*) → bool

Check if an object may be expanded by pretty print.

rich.pretty.**pprint**(*_object: Any*, ***, *console: Console = None*, *indent_guides: bool = True*, *max_length: int = None*, *max_string: int = None*, *expand_all: bool = False*)

A convenience function for pretty printing.

Parameters

- **_object** (*Any*) – Object to pretty print.
- **console** (Console, *optional*) – Console instance, or None to use default. Defaults to None.

- **max_length** (*int, optional*) – Maximum length of containers before abbreviating, or None for no abbreviation. Defaults to None.
- **max_string** (*int, optional*) – Maximum length of strings before truncating, or None to disable. Defaults to None.
- **indent_guides** (*bool, optional*) – Enable indentation guides. Defaults to True.
- **expand_all** (*bool, optional*) – Expand all containers. Defaults to False.

rich.pretty.**pretty_repr**(*_object: Any*, ***, *max_width: int = 80*, *indent_size: int = 4*, *max_length: Optional[int] = None*, *max_string: Optional[int] = None*, *expand_all: bool = False*) → str

Prettify repr string by expanding on to new lines to fit within a given width.

Parameters

- **_object** (*Any*) – Object to repr.
- **max_width** (*int, optional*) – Desired maximum width of repr string. Defaults to 80.
- **indent_size** (*int, optional*) – Number of spaces to indent. Defaults to 4.
- **max_length** (*int, optional*) – Maximum length of containers before abbreviating, or None for no abbreviation. Defaults to None.
- **max_string** (*int, optional*) – Maximum length of string before truncating, or None to disable truncating. Defaults to None.
- **expand_all** (*bool, optional*) – Expand all containers regardless of available width. Defaults to False.

Returns A possibly multi-line representation of the object.

Return type str

rich.pretty.**traverse**(*_object: Any*, *max_length: Optional[int] = None*, *max_string: Optional[int] = None*) → *rich.pretty.Node*

Traverse object and generate a tree.

Parameters

- **_object** (*Any*) – Object to be traversed.
- **max_length** (*int, optional*) – Maximum length of containers before abbreviating, or None for no abbreviation. Defaults to None.
- **max_string** (*int, optional*) – Maximum length of string before truncating, or None to disable truncating. Defaults to None.

Returns The root of a tree structure which can be used to render a pretty repr.

Return type *Node*

22.18 rich.progress_bar

class rich.progress_bar.**ProgressBar**(*total: float = 100*, *completed: float = 0*, *width: Optional[int] = None*, *pulse: bool = False*, *style: Union[str, rich.style.Style] = 'bar.back'*, *complete_style: Union[str, rich.style.Style] = 'bar.complete'*, *finished_style: Union[str, rich.style.Style] = 'bar.finished'*, *pulse_style: Union[str, rich.style.Style] = 'bar.pulse'*, *animation_time: Optional[float] = None*)

Renders a (progress) bar. Used by rich.progress.

Parameters

- **total** (*float, optional*) – Number of steps in the bar. Defaults to 100.
- **completed** (*float, optional*) – Number of steps completed. Defaults to 0.
- **width** (*int, optional*) – Width of the bar, or None for maximum width. Defaults to None.
- **pulse** (*bool, optional*) – Enable pulse effect. Defaults to False.
- **style** (*StyleType, optional*) – Style for the bar background. Defaults to "bar.back".
- **complete_style** (*StyleType, optional*) – Style for the completed bar. Defaults to "bar.complete".
- **finished_style** (*StyleType, optional*) – Style for a finished bar. Defaults to "bar.done".
- **pulse_style** (*StyleType, optional*) – Style for pulsing bars. Defaults to "bar.pulse".
- **animation_time** (*Optional[float], optional*) – Time in seconds to use for animation, or None to use system time.

property percentage_completed

Calculate percentage complete.

update(*completed: float*, *total: Optional[float] = None*) → None

Update progress with new values.

Parameters

- **completed** (*float*) – Number of steps completed.
- **total** (*float, optional*) – Total number of steps, or None to not change. Defaults to None.

22.19 rich.progress

class rich.progress.**BarColumn**(*bar_width: Optional[int] = 40*, *style: Union[str, Style] = 'bar.back'*, *complete_style: Union[str, Style] = 'bar.complete'*, *finished_style: Union[str, Style] = 'bar.finished'*, *pulse_style: Union[str, Style] = 'bar.pulse'*, *table_column: rich.table.Column = None*)

Renders a visual progress bar.

Parameters

- **bar_width** (*Optional[int], optional*) – Width of bar or None for full width. Defaults to 40.
- **style** (*StyleType, optional*) – Style for the bar background. Defaults to "bar.back".
- **complete_style** (*StyleType, optional*) – Style for the completed bar. Defaults to "bar.complete".
- **finished_style** (*StyleType, optional*) – Style for a finished bar. Defaults to "bar.done".
- **pulse_style** (*StyleType, optional*) – Style for pulsing bars. Defaults to "bar.pulse".

render(*task:* rich.progress.Task) → *rich.progress_bar.ProgressBar*
Gets a progress bar widget for a task.

class rich.progress.**DownloadColumn**(*binary_units: bool = False, table_column: Optional[rich.table.Column] = None*)
Renders file size downloaded and total, e.g. '0.5/2.3 GB'.

Parameters **binary_units** (*bool, optional*) – Use binary units, KiB, MiB etc. Defaults to False.

render(*task:* rich.progress.Task) → *rich.text.Text*
Calculate common unit for completed and total.

class rich.progress.**FileSizeColumn**(*table_column: Optional[rich.table.Column] = None*)
Renders completed filesize.

render(*task:* rich.progress.Task) → *rich.text.Text*
Show data completed.

class rich.progress.**Progress**(**columns: Union[str, rich.progress.ProgressColumn], console: Optional[rich.console.Console] = None, auto_refresh: bool = True, refresh_per_second: float = 10, speed_estimate_period: float = 30.0, transient: bool = False, redirect_stdout: bool = True, redirect_stderr: bool = True, get_time: Optional[Callable[], float]] = None, disable: bool = False, expand: bool = False*)
Renders an auto-updating progress bar(s).

Parameters

- **console** (*Console, optional*) – Optional Console instance. Default will an internal Console instance writing to stdout.
- **auto_refresh** (*bool, optional*) – Enable auto refresh. If disabled, you will need to call *refresh()*.
- **refresh_per_second** (*Optional[float], optional*) – Number of times per second to refresh the progress information or None to use default (10). Defaults to None.
- **speed_estimate_period** – (float, optional): Period (in seconds) used to calculate the speed estimate. Defaults to 30.
- **transient** – (bool, optional): Clear the progress on exit. Defaults to False.
- **redirect_stdout** – (bool, optional): Enable redirection of stdout, so `print` may be used. Defaults to True.
- **redirect_stderr** – (bool, optional): Enable redirection of stderr. Defaults to True.

- **get_time** – (Callable, optional): A callable that gets the current time, or None to use Console.get_time. Defaults to None.
- **disable** (*bool, optional*) – Disable progress display. Defaults to False
- **expand** (*bool, optional*) – Expand tasks table to fit width. Defaults to False.

add_task(*description: str*, *start: bool = True*, *total: int = 100*, *completed: int = 0*, *visible: bool = True*, ***fields: Any*) → TaskID
Add a new 'task' to the Progress display.

Parameters

- **description** (*str*) – A description of the task.
- **start** (*bool, optional*) – Start the task immediately (to calculate elapsed time). If set to False, you will need to call *start* manually. Defaults to True.
- **total** (*int, optional*) – Number of total steps in the progress if know. Defaults to 100.
- **completed** (*int, optional*) – Number of steps completed so far.. Defaults to 0.
- **visible** (*bool, optional*) – Enable display of the task. Defaults to True.
- ****fields** (*str*) – Additional data fields required for rendering.

Returns An ID you can use when calling *update*.

Return type TaskID

advance(*task_id: TaskID*, *advance: float = 1*) → None
Advance task by a number of steps.

Parameters

- **task_id** (*TaskID*) – ID of task.
- **advance** (*float*) – Number of steps to advance. Default is 1.

property finished
Check if all tasks have been completed.

get_renderable() → Union[*rich.console.ConsoleRenderable*, *rich.console.RichCast*, str]
Get a renderable for the progress display.

get_renderables() → Iterable[Union[*rich.console.ConsoleRenderable*, *rich.console.RichCast*, str]]
Get a number of renderables for the progress display.

make_tasks_table(*tasks: Iterable[*rich.progress.Task*]*) → *rich.table.Table*
Get a table to render the Progress display.

Parameters **tasks** (*Iterable[Task]*) – An iterable of Task instances, one per row of the table.

Returns A table instance.

Return type *Table*

refresh() → None
Refresh (render) the progress information.

remove_task(*task_id: TaskID*) → None
Delete a task if it exists.

Parameters **task_id** (*TaskID*) – A task ID.

reset(*task_id: TaskID*, *, *start: bool = True*, *total: Optional[int] = None*, *completed: int = 0*, *visible: Optional[bool] = None*, *description: Optional[str] = None*, ***fields: Any*) → None
Reset a task so completed is 0 and the clock is reset.

Parameters

- **task_id** (*TaskID*) – ID of task.
- **start** (*bool, optional*) – Start the task after reset. Defaults to True.
- **total** (*int, optional*) – New total steps in task, or None to use current total. Defaults to None.
- **completed** (*int, optional*) – Number of steps completed. Defaults to 0.
- ****fields** (*str*) – Additional data fields required for rendering.

start() → None
Start the progress display.

start_task(*task_id: TaskID*) → None
Start a task.

Starts a task (used when calculating elapsed time). You may need to call this manually, if you called `add_task` with `start=False`.

Parameters **task_id** (*TaskID*) – ID of task.

stop() → None
Stop the progress display.

stop_task(*task_id: TaskID*) → None
Stop a task.

This will freeze the elapsed time on the task.

Parameters **task_id** (*TaskID*) – ID of task.

property task_ids
A list of task IDs.

property tasks
Get a list of Task instances.

track(*sequence: Union[Iterable[ProgressType], Sequence[ProgressType]]*, *total: Optional[int] = None*, *task_id: Optional[TaskID] = None*, *description='Working...'*, *update_period: float = 0.1*) → Iterable[ProgressType]
Track progress by iterating over a sequence.

Parameters

- **sequence** (*Sequence[ProgressType]*) – A sequence of values you want to iterate over and track progress.
- **total** – (int, optional): Total number of steps. Default is len(sequence).
- **task_id** – (TaskID): Task to track. Default is new task.
- **description** – (str, optional): Description of task, if new task is created.
- **update_period** (*float, optional*) – Minimum time (in seconds) between calls to update(). Defaults to 0.1.

Returns An iterable of values taken from the provided sequence.

Return type Iterable[ProgressType]

update(*task_id: TaskID*, *, *total: Optional[float] = None*, *completed: Optional[float] = None*, *advance: Optional[float] = None*, *description: Optional[str] = None*, *visible: Optional[bool] = None*, *refresh: bool = False*, ***fields: Any*) → None
Update information associated with a task.

Parameters

- **task_id** (*TaskID*) – Task id (returned by add_task).
- **total** (*float, optional*) – Updates task.total if not None.
- **completed** (*float, optional*) – Updates task.completed if not None.
- **advance** (*float, optional*) – Add a value to task.completed if not None.
- **description** (*str, optional*) – Change task description if not None.
- **visible** (*bool, optional*) – Set visible flag if not None.
- **refresh** (*bool*) – Force a refresh of progress information. Default is False.
- ****fields** (*Any*) – Additional data fields required for rendering.

class rich.progress.**ProgressColumn**(*table_column: Optional[rich.table.Column] = None*)
Base class for a widget to use in progress display.

get_table_column() → *rich.table.Column*
Get a table column, used to build tasks table.

abstract render(*task:* rich.progress.Task) → Union[*rich.console.ConsoleRenderable*, *rich.console.RichCast*, str]
Should return a renderable object.

class rich.progress.**ProgressSample**(*timestamp: float*, *completed: float*)
Sample of progress for a given time.

property completed
Number of steps completed.

property timestamp
Timestamp of sample.

class rich.progress.**RenderableColumn**(*renderable: Union[rich.console.ConsoleRenderable, rich.console.RichCast, str] = ''*, *, *table_column: Optional[rich.table.Column] = None*)
A column to insert an arbitrary column.

Parameters **renderable** (*RenderableType, optional*) – Any renderable. Defaults to empty string.

render(*task:* rich.progress.Task) → Union[*rich.console.ConsoleRenderable*, *rich.console.RichCast*, str]
Should return a renderable object.

class rich.progress.**SpinnerColumn**(*spinner_name: str = 'dots'*, *style: Optional[Union[str, Style]] = 'progress.spinner'*, *speed: float = 1.0*, *finished_text: Union[str, Text] = ' '*, *table_column: rich.table.Column = None*)
A column with a 'spinner' animation.

Parameters

- **spinner_name** (*str, optional*) – Name of spinner animation. Defaults to "dots".
- **style** (*StyleType, optional*) – Style of spinner. Defaults to "progress.spinner".

- **speed** (*float, optional*) – Speed factor of spinner. Defaults to 1.0.
- **finished_text** (*TextType, optional*) – Text used when task is finished. Defaults to " ".

render(*task:* rich.progress.Task) → *rich.text.Text*
Should return a renderable object.

set_spinner(*spinner_name: str*, *spinner_style: Optional[Union[str, Style]] = 'progress.spinner'*, *speed: float = 1.0*)
Set a new spinner.

Parameters

- **spinner_name** (*str*) – Spinner name, see python -m rich.spinner.
- **spinner_style** (*Optional[StyleType], optional*) – Spinner style. Defaults to "progress.spinner".
- **speed** (*float, optional*) – Speed factor of spinner. Defaults to 1.0.

class rich.progress.**Task**(*id: TaskID, description: str, total: float, completed: float, _get_time: Callable[[], float], finished_time: Optional[float] = None, visible: bool = True, fields: Dict[str, Any] = <factory>, finished_speed: Optional[float] = None, _lock: threading.RLock = <factory>*)
Information regarding a progress task.

This object should be considered read-only outside of the *Progress* class.

completed: float
Number of steps completed

Type float

description: str
Description of the task.

Type str

property elapsed
Time elapsed since task was started, or None if the task hasn't started.

Type Optional[float]

fields: Dict[str, Any]
Arbitrary fields passed in via Progress.update.

Type dict

property finished
Check if the task has finished.

finished_speed: Optional[float] = None
The last speed for a finshed task.

Type Optional[float]

finished_time: Optional[float] = None
Time task was finished.

Type float

get_time() → float
float: Get the current time, in seconds.

id: TaskID
Task ID associated with this task (used in Progress methods).

property percentage
Get progress of task as a percentage.

Type float

property remaining
Get the number of steps remaining.

Type float

property speed
Get the estimated speed in steps per second.

Type Optional[float]

start_time: Optional[float] = None
Time this task was started, or None if not started.

Type Optional[float]

property started
Check if the task as started.

Type bool

stop_time: Optional[float] = None
Time this task was stopped, or None if not stopped.

Type Optional[float]

property time_remaining
Get estimated time to completion, or `None` if no data.

Type Optional[float]

total: float
Total number of steps in this task.

Type str

visible: bool = True
Indicates if this task is visible in the progress display.

Type bool

class rich.progress.**TextColumn**(*text_format: str*, *style: Union[str, Style] = 'none'*, *justify: typing_extensions.Literal[default, left, center, right, full] = 'left'*, *markup: bool = True*, *highlighter:* rich.highlighter.Highlighter *= None*, *table_column:* rich.table.Column *= None*)
A column containing text.

render(*task:* rich.progress.Task) → *rich.text.Text*
Should return a renderable object.

class rich.progress.**TimeElapsedColumn**(*table_column: Optional[*rich.table.Column*] = None*)
Renders time elapsed.

render(*task:* rich.progress.Task) → *rich.text.Text*
Show time remaining.

class rich.progress.**TimeRemainingColumn**(*table_column: Optional[*rich.table.Column*] = None*)
Renders estimated time remaining.

render(*task:* rich.progress.Task) → *rich.text.Text*
Show time remaining.

class rich.progress.**TotalFileSizeColumn**(*table_column: Optional[rich.table.Column] = None*)
Renders total filesize.

render(*task:* rich.progress.Task) → *rich.text.Text*
Show data completed.

class rich.progress.**TransferSpeedColumn**(*table_column: Optional[rich.table.Column] = None*)
Renders human readable transfer speed.

render(*task:* rich.progress.Task) → *rich.text.Text*
Show data transfer speed.

rich.progress.**track**(*sequence: Union[Sequence[ProgressType], Iterable[ProgressType]]*, *description='Working...'*, *total: int = None*, *auto_refresh=True*, *console: Optional[rich.console.Console] = None*, *transient: bool = False*, *get_time: Callable[], float] = None*, *refresh_per_second: float = 10*, *style: Union[str, Style] = 'bar.back'*, *complete_style: Union[str, Style] = 'bar.complete'*, *finished_style: Union[str, Style] = 'bar.finished'*, *pulse_style: Union[str, Style] = 'bar.pulse'*, *update_period: float = 0.1*, *disable: bool = False*) → Iterable[ProgressType]
Track progress by iterating over a sequence.

Parameters

- **sequence** (*Iterable[ProgressType]*) – A sequence (must support "len") you wish to iterate over.
- **description** (*str, optional*) – Description of task show next to progress bar. Defaults to "Working".
- **total** – (int, optional): Total number of steps. Default is len(sequence).
- **auto_refresh** (*bool, optional*) – Automatic refresh, disable to force a refresh after each iteration. Default is True.
- **transient** – (bool, optional): Clear the progress on exit. Defaults to False.
- **console** (*Console, optional*) – Console to write to. Default creates internal Console instance.
- **refresh_per_second** (*float*) – Number of times per second to refresh the progress information. Defaults to 10.
- **style** (*StyleType, optional*) – Style for the bar background. Defaults to "bar.back".
- **complete_style** (*StyleType, optional*) – Style for the completed bar. Defaults to "bar.complete".
- **finished_style** (*StyleType, optional*) – Style for a finished bar. Defaults to "bar.done".
- **pulse_style** (*StyleType, optional*) – Style for pulsing bars. Defaults to "bar.pulse".
- **update_period** (*float, optional*) – Minimum time (in seconds) between calls to update(). Defaults to 0.1.
- **disable** (*bool, optional*) – Disable display of progress.

Returns An iterable of the values in the sequence.

Return type Iterable[ProgressType]

22.20 rich.prompt

class rich.prompt.**Confirm**(*prompt: Union[str, rich.text.Text] = ''*, ***, *console: Optional[rich.console.Console] = None*, *password: bool = False*, *choices: Optional[List[str]] = None*, *show_default: bool = True*, *show_choices: bool = True*)

A yes / no confirmation prompt.

Example

```
>>> if Confirm.ask("Continue"):
        run_job()
```

process_response(*value: str*) → bool

Convert choices to a bool.

render_default(*default: DefaultType*) → *rich.text.Text*

Render the default as (y) or (n) rather than True/False.

response_type

alias of `bool`

class rich.prompt.**FloatPrompt**(*prompt: Union[str, rich.text.Text] = ''*, ***, *console: Optional[rich.console.Console] = None*, *password: bool = False*, *choices: Optional[List[str]] = None*, *show_default: bool = True*, *show_choices: bool = True*)

A prompt that returns a float.

Example

```
>>> temperature = FloatPrompt.ask("Enter desired temperature")
```

response_type

alias of `float`

class rich.prompt.**IntPrompt**(*prompt: Union[str, rich.text.Text] = ''*, ***, *console: Optional[rich.console.Console] = None*, *password: bool = False*, *choices: Optional[List[str]] = None*, *show_default: bool = True*, *show_choices: bool = True*)

A prompt that returns an integer.

Example

```
>>> burrito_count = IntPrompt.ask("How many burritos do you want to order")
```

response_type
alias of `int`

exception rich.prompt.**InvalidResponse**(*message: Union[str, rich.text.Text]*)
Exception to indicate a response was invalid. Raise this within process_response() to indicate an error and provide an error message.

Parameters **message** (*Union[str, Text]*) – Error message.

class rich.prompt.**Prompt**(*prompt: Union[str, rich.text.Text] = ''*, ***, *console: Optional[rich.console.Console] = None*, *password: bool = False*, *choices: Optional[List[str]] = None*, *show_default: bool = True*, *show_choices: bool = True*)
A prompt that returns a str.

Example

```
>>> name = Prompt.ask("Enter your name")
```

response_type
alias of `str`

class rich.prompt.**PromptBase**(*prompt: Union[str, rich.text.Text] = ''*, ***, *console: Optional[rich.console.Console] = None*, *password: bool = False*, *choices: Optional[List[str]] = None*, *show_default: bool = True*, *show_choices: bool = True*)
Ask the user for input until a valid response is received. This is the base class, see one of the concrete classes for examples.

Parameters

- **prompt** (*TextType, optional*) – Prompt text. Defaults to "".
- **console** (*Console, optional*) – A Console instance or None to use global console. Defaults to None.
- **password** (*bool, optional*) – Enable password input. Defaults to False.
- **choices** (*List[str], optional*) – A list of valid choices. Defaults to None.
- **show_default** (*bool, optional*) – Show default in prompt. Defaults to True.
- **show_choices** (*bool, optional*) – Show choices in prompt. Defaults to True.

classmethod ask(*prompt: Union[str, Text] = ''*, ***, *console: rich.console.Console = 'None'*, *password: bool = 'False'*, *choices: List[str] = 'None'*, *show_default: bool = 'True'*, *show_choices: bool = 'True'*, *default: DefaultType*, *stream: TextIO = 'None'*) → Union[DefaultType, PromptType]
classmethod ask(*prompt: Union[str, Text] = ''*, ***, *console: rich.console.Console = 'None'*, *password: bool = 'False'*, *choices: List[str] = 'None'*, *show_default: bool = 'True'*, *show_choices: bool = 'True'*, *stream: TextIO = 'None'*) → PromptType
Shortcut to construct and run a prompt loop and return the result.

Example

```
>>> filename = Prompt.ask("Enter a filename")
```

Parameters

- **prompt** (*TextType, optional*) – Prompt text. Defaults to "".
- **console** (*Console, optional*) – A Console instance or None to use global console. Defaults to None.
- **password** (*bool, optional*) – Enable password input. Defaults to False.
- **choices** (*List[str], optional*) – A list of valid choices. Defaults to None.
- **show_default** (*bool, optional*) – Show default in prompt. Defaults to True.
- **show_choices** (*bool, optional*) – Show choices in prompt. Defaults to True.
- **stream** (*TextIO, optional*) – Optional text file open for reading to get input. Defaults to None.

check_choice(*value: str*) → bool
Check value is in the list of valid choices.

Parameters **value** (*str*) – Value entered by user.

Returns True if choice was valid, otherwise False.

Return type bool

classmethod get_input(*console: rich.console.Console, prompt: Union[str, rich.text.Text], password: bool, stream: Optional[TextIO] = None*) → str
Get input from user.

Parameters

- **console** (*Console*) – Console instance.
- **prompt** (*TextType*) – Prompt text.
- **password** (*bool*) – Enable password entry.

Returns String from user.

Return type str

make_prompt(*default: DefaultType*) → *rich.text.Text*
Make prompt text.

Parameters **default** (*DefaultType*) – Default value.

Returns Text to display in prompt.

Return type *Text*

on_validate_error(*value: str, error: rich.prompt.InvalidResponse*) → None
Called to handle validation error.

Parameters

- **value** (*str*) – String entered by user.
- **error** (*InvalidResponse*) – Exception instance the initiated the error.

pre_prompt() → None
Hook to display something before the prompt.

process_response(*value: str*) → PromptType
Process response from user, convert to prompt type.

Parameters **value** (*str*) – String typed by user.

Raises ***InvalidResponse*** – If `value` is invalid.

Returns The value to be returned from ask method.

Return type PromptType

render_default(*default: DefaultType*) → *rich.text.Text*
Turn the supplied default in to a Text instance.

Parameters **default** (*DefaultType*) – Default value.

Returns Text containing rendering of default value.

Return type *Text*

response_type
alias of `str`

exception rich.prompt.**PromptError**
Exception base class for prompt related errors.

22.21 rich.protocol

rich.protocol.**is_renderable**(*check_object: Any*) → bool
Check if an object may be rendered by Rich.

22.22 rich.rule

class rich.rule.**Rule**(*title: Union[str, rich.text.Text] = '', *, characters: str = '–', style: Union[str, rich.style.Style] = 'rule.line', end: str = '\n', align: typing_extensions.Literal[left, center, right] = 'center'*)
A console renderable to draw a horizontal rule (line).

Parameters

- **title** (*Union[str, Text], optional*) – Text to render in the rule. Defaults to "".
- **characters** (*str, optional*) – Character(s) used to draw the line. Defaults to "–".
- **style** (*StyleType, optional*) – Style of Rule. Defaults to "rule.line".
- **end** (*str, optional*) – Character at end of Rule. defaults to "\n"
- **align** (*str, optional*) – How to align the title, one of "left", "center", or "right". Defaults to "center".

22.23 rich.segment

class rich.segment.**Segment**(*text: str = ''*, *style: Optional[rich.style.Style] = None*, *is_control: bool = False*)

A piece of text with associated style. Segments are produced by the Console render process and are ultimately converted in to strings to be written to the terminal.

Parameters

- **text** (*str*) – A piece of text.
- **style** (*Style*, optional) – An optional style to apply to the text.
- **is_control** (*bool, optional*) – Boolean that marks segment as containing non-printable control codes.

classmethod adjust_line_length(*line: List[rich.segment.Segment]*, *length: int*, *style: Optional[rich.style.Style] = None*, *pad: bool = True*) → List[*rich.segment.Segment*]

Adjust a line to a given width (cropping or padding as required).

Parameters

- **segments** (*Iterable[Segment]*) – A list of segments in a single line.
- **length** (*int*) – The desired width of the line.
- **style** (*Style, optional*) – The style of padding if used (space on the end). Defaults to None.
- **pad** (*bool, optional*) – Pad lines with spaces if they are shorter than *length*. Defaults to True.

Returns A line of segments with the desired length.

Return type List[*Segment*]

classmethod apply_style(*segments: Iterable[rich.segment.Segment]*, *style: Optional[rich.style.Style] = None*, *post_style: Optional[rich.style.Style] = None*) → Iterable[*rich.segment.Segment*]

Apply style(s) to an iterable of segments.

Returns an iterable of segments where the style is replaced by `style + segment.style + post_style`.

Parameters

- **segments** (*Iterable[Segment]*) – Segments to process.
- **style** (*Style, optional*) – Base style. Defaults to None.
- **post_style** (*Style, optional*) – Style to apply on top of segment style. Defaults to None.

Returns A new iterable of segments (possibly the same iterable).

Return type Iterable[Segments]

property cell_length

Get cell length of segment.

classmethod control(*text: str*, *style: Optional[rich.style.Style] = None*) → *rich.segment.Segment*

Create a Segment with control codes.

Parameters

- **text** (*str*) – Text containing non-printable control codes.
- **style** (*Optional[style]*) – Optional style.

Returns A Segment instance with `is_control=True`.

Return type *Segment*

classmethod filter_control(*segments: Iterable[rich.segment.Segment], is_control=False*) → Iterable[*rich.segment.Segment*]

Filter segments by `is_control` attribute.

Parameters

- **segments** (*Iterable[Segment]*) – An iterable of Segment instances.
- **is_control** (*bool, optional*) – is_control flag to match in search.

Returns And iterable of Segment instances.

Return type Iterable[*Segment*]

classmethod get_line_length(*line: List[rich.segment.Segment]*) → int

Get the length of list of segments.

Parameters **line** (*List[Segment]*) – A line encoded as a list of Segments (assumes no '\n' characters),

Returns The length of the line.

Return type int

classmethod get_shape(*lines: List[List[rich.segment.Segment]]*) → Tuple[int, int]

Get the shape (enclosing rectangle) of a list of lines.

Parameters **lines** (*List[List[Segment]]*) – A list of lines (no '\n' characters).

Returns Width and height in characters.

Return type Tuple[int, int]

property is_control

True if the segment contains control codes, otherwise False.

classmethod line(*is_control: bool = False*) → *rich.segment.Segment*

Make a new line segment.

classmethod make_control(*segments: Iterable[rich.segment.Segment]*) → Iterable[*rich.segment.Segment*]

Convert all segments in to control segments.

Returns Segments with is_control=True

Return type Iterable[Segments]

classmethod remove_color(*segments: Iterable[rich.segment.Segment]*) → Iterable[*rich.segment.Segment*]

Remove all color from an iterable of segments.

Parameters **segments** (*Iterable[Segment]*) – An iterable segments.

Yields *Segment* – Segments with colorless style.

classmethod set_shape(*lines: List[List[rich.segment.Segment]], width: int, height: Optional[int] = None, style: Optional[rich.style.Style] = None, new_lines: bool = False*) → List[List[*rich.segment.Segment*]]

Set the shape of a list of lines (enclosing rectangle).

Args: lines (List[List[Segment]]): A list of lines. width (int): Desired width. height (int, optional): Desired height or None for no change. style (Style, optional): Style of any padding added. Defaults to None. new_lines (bool, optional): Padded lines should include “

“. Defaults to False.

Returns: List[List[Segment]]: New list of lines that fits width x height.

classmethod simplify(*segments: Iterable[*rich.segment.Segment*]*) → Iterable[*rich.segment.Segment*]

Simplify an iterable of segments by combining contiguous segments with the same style.

Parameters **segments** (*Iterable[Segment]*) – An iterable of segments.

Returns A possibly smaller iterable of segments that will render the same way.

Return type Iterable[*Segment*]

classmethod split_and_crop_lines(*segments: Iterable[*rich.segment.Segment*]*, *length: int*, *style: Optional[*rich.style.Style*] = None*, *pad: bool = True*, *include_new_lines: bool = True*) → Iterable[List[*rich.segment.Segment*]]

Split segments in to lines, and crop lines greater than a given length.

Parameters

- **segments** (*Iterable[Segment]*) – An iterable of segments, probably generated from console.render.
- **length** (*int*) – Desired line length.
- **style** (*Style, optional*) – Style to use for any padding.
- **pad** (*bool*) – Enable padding of lines that are less than *length*.

Returns An iterable of lines of segments.

Return type Iterable[List[*Segment*]]

classmethod split_lines(*segments: Iterable[*rich.segment.Segment*]*) → Iterable[List[*rich.segment.Segment*]]

Split a sequence of segments in to a list of lines.

Parameters **segments** (*Iterable[Segment]*) – Segments potentially containing line feeds.

Yields *Iterable[List[Segment]]* – Iterable of segment lists, one per line.

classmethod strip_links(*segments: Iterable[*rich.segment.Segment*]*) → Iterable[*rich.segment.Segment*]

Remove all links from an iterable of styles.

Parameters **segments** (*Iterable[Segment]*) – An iterable segments.

Yields *Segment* – Segments with link removed.

classmethod strip_styles(*segments: Iterable[*rich.segment.Segment*]*) → Iterable[*rich.segment.Segment*]

Remove all styles from an iterable of segments.

Parameters **segments** (*Iterable[Segment]*) – An iterable segments.

Yields *Segment* – Segments with styles replace with None

property style

An optional style.

property text
Raw text.

22.24 rich.spinner

22.25 rich.status

class rich.status.**Status**(*status: Union[rich.console.ConsoleRenderable, rich.console.RichCast, str]*, ***, *console: rich.console.Console = None*, *spinner: str = 'dots'*, *spinner_style: Union[str, Style] = 'status.spinner'*, *speed: float = 1.0*, *refresh_per_second: float = 12.5*)
Displays a status indicator with a 'spinner' animation.

Parameters

- **status** (*RenderableType*) – A status renderable (str or Text typically).
- **console** (*Console, optional*) – Console instance to use, or None for global console. Defaults to None.
- **spinner** (*str, optional*) – Name of spinner animation (see python -m rich.spinner). Defaults to "dots".
- **spinner_style** (*StyleType, optional*) – Style of spinner. Defaults to "status.spinner".
- **speed** (*float, optional*) – Speed factor for spinner animation. Defaults to 1.0.
- **refresh_per_second** (*float, optional*) – Number of refreshes per second. Defaults to 12.5.

property console
Get the Console used by the Status objects.

property renderable
Get the renderable for the status (a table with spinner and status).

start() → None
Start the status animation.

stop() → None
Stop the spinner animation.

update(*status: Optional[Union[rich.console.ConsoleRenderable, rich.console.RichCast, str]] = None*, ***, *spinner: Optional[str] = None*, *spinner_style: Optional[Union[str, Style]] = None*, *speed: Optional[float] = None*)
Update status.

Parameters

- **status** (*Optional[RenderableType], optional*) – New status renderable or None for no change. Defaults to None.
- **spinner** (*Optional[str], optional*) – New spinner or None for no change. Defaults to None.
- **spinner_style** (*Optional[StyleType], optional*) – New spinner style or None for no change. Defaults to None.

- **speed** (*Optional[float], optional*) – Speed factor for spinner animation or None for no change. Defaults to None.

22.26 rich.style

class rich.style.**Style**(*, *color: Optional[Union[rich.color.Color, str]] = None*, *bgcolor: Optional[Union[rich.color.Color, str]] = None*, *bold: Optional[bool] = None*, *dim: Optional[bool] = None*, *italic: Optional[bool] = None*, *underline: Optional[bool] = None*, *blink: Optional[bool] = None*, *blink2: Optional[bool] = None*, *reverse: Optional[bool] = None*, *conceal: Optional[bool] = None*, *strike: Optional[bool] = None*, *underline2: Optional[bool] = None*, *frame: Optional[bool] = None*, *encircle: Optional[bool] = None*, *overline: Optional[bool] = None*, *link: Optional[str] = None*)

A terminal style.

A terminal style consists of a color (*color*), a background color (*bgcolor*), and a number of attributes, such as bold, italic etc. The attributes have 3 states: they can either be on (`True`), off (`False`), or not set (`None`).

Parameters

- **color** (*Union[Color, str], optional*) – Color of terminal text. Defaults to None.
- **bgcolor** (*Union[Color, str], optional*) – Color of terminal background. Defaults to None.
- **bold** (*bool, optional*) – Enable bold text. Defaults to None.
- **dim** (*bool, optional*) – Enable dim text. Defaults to None.
- **italic** (*bool, optional*) – Enable italic text. Defaults to None.
- **underline** (*bool, optional*) – Enable underlined text. Defaults to None.
- **blink** (*bool, optional*) – Enabled blinking text. Defaults to None.
- **blink2** (*bool, optional*) – Enable fast blinking text. Defaults to None.
- **reverse** (*bool, optional*) – Enabled reverse text. Defaults to None.
- **conceal** (*bool, optional*) – Enable concealed text. Defaults to None.
- **strike** (*bool, optional*) – Enable strikethrough text. Defaults to None.
- **underline2** (*bool, optional*) – Enable doubly underlined text. Defaults to None.
- **frame** (*bool, optional*) – Enable framed text. Defaults to None.
- **encircle** (*bool, optional*) – Enable encircled text. Defaults to None.
- **overline** (*bool, optional*) – Enable overlined text. Defaults to None.
- **link** (*str, link*) – Link URL. Defaults to None.

property background_style
A Style with background only.

property bgcolor
The background color or None if it is not set.

classmethod chain(**styles:* rich.style.Style) → *rich.style.Style*
Combine styles from positional argument in to a single style.

Parameters ***styles** (*Iterable[Style]*) – Styles to combine.

Returns A new style instance.

Return type *Style*

property color
The foreground color or None if it is not set.

classmethod combine(*styles: Iterable[*rich.style.Style*]*) → *rich.style.Style*
Combine styles and get result.

Parameters **styles** (*Iterable[Style]*) – Styles to combine.

Returns A new style instance.

Return type *Style*

copy() → *rich.style.Style*
Get a copy of this style.

Returns A new Style instance with identical attributes.

Return type *Style*

classmethod from_color(*color: Optional[*rich.color.Color*] = None*, *bgcolor: Optional[*rich.color.Color*] = None*) → *rich.style.Style*
Create a new style with colors and no attributes.

Returns A (foreground) color, or None for no color. Defaults to None. bgcolor (Optional[Color]): A (background) color, or None for no color. Defaults to None.

Return type color (Optional[*Color*])

get_html_style(*theme: rich.terminal_theme.TerminalTheme = None*) → str
Get a CSS style rule.

property link
Link text, if set.

property link_id
Get a link id, used in ansi code for links.

classmethod normalize(*style: str*) → str
Normalize a style definition so that styles with the same effect have the same string representation.

Parameters **style** (*str*) – A style definition.

Returns Normal form of style definition.

Return type str

classmethod null() → *rich.style.Style*
Create an ‘null’ style, equivalent to Style(), but more performant.

classmethod parse(*style_definition: str*) → *rich.style.Style*
Parse a style definition.

Parameters **style_definition** (*str*) – A string containing a style.

Raises **errors.StyleSyntaxError** – If the style definition syntax is invalid.

Returns A Style instance.

Return type *Style*

classmethod pick_first(**values: Optional[Union[str, rich.style.Style]]*) → Union[str, *rich.style.Style*]
Pick first non-None style.

render(*text: str = ''*, ***, *color_system: Optional[rich.color.ColorSystem] = <ColorSystem.TRUECOLOR: 3>*, *legacy_windows: bool = False*) → str
Render the ANSI codes for the style.

Parameters

- **text** (*str, optional*) – A string to style. Defaults to "".
- **color_system** (*Optional[ColorSystem], optional*) – Color system to render to. Defaults to ColorSystem.TRUECOLOR.

Returns A string containing ANSI style codes.

Return type str

test(*text: Optional[str] = None*) → None
Write text with style directly to terminal.

This method is for testing purposes only.

Parameters **text** (*Optional[str], optional*) – Text to style or None for style name.

property transparent_background
Check if the style specified a transparent background.

update_link(*link: Optional[str] = None*) → *rich.style.Style*
Get a copy with a different value for link.

Parameters **link** (*str, optional*) – New value for link. Defaults to None.

Returns A new Style instance.

Return type *Style*

property without_color
Get a copy of the style with color removed.

class rich.style.**StyleStack**(*default_style:* rich.style.Style)
A stack of styles.

property current
Get the Style at the top of the stack.

pop() → *rich.style.Style*
Pop last style and discard.

Returns New current style (also available as stack.current)

Return type *Style*

push(*style:* rich.style.Style) → None
Push a new style on to the stack.

Parameters **style** (Style) – New style to combine with current style.

22.27 rich.styled

class rich.styled.**Styled**(*renderable: RenderableType*, *style: StyleType*)

Apply a style to a renderable.

Parameters

- **renderable** (*RenderableType*) – Any renderable.
- **style** (*StyleType*) – A style to apply across the entire renderable.

22.28 rich.syntax

class rich.syntax.**Syntax**(*code: str*, *lexer_name: str*, ***, *theme: Union[str, rich.syntax.SyntaxTheme] = 'monokai'*, *dedent: bool = False*, *line_numbers: bool = False*, *start_line: int = 1*, *line_range: Optional[Tuple[int, int]] = None*, *highlight_lines: Optional[Set[int]] = None*, *code_width: Optional[int] = None*, *tab_size: int = 4*, *word_wrap: bool = False*, *background_color: Optional[str] = None*, *indent_guides: bool = False*)

Construct a Syntax object to render syntax highlighted code.

Parameters

- **code** (*str*) – Code to highlight.
- **lexer_name** (*str*) – Lexer to use (see https://pygments.org/docs/lexers/)
- **theme** (*str, optional*) – Color theme, aka Pygments style (see https://pygments.org/docs/styles/#getting-a-list-of-available-styles). Defaults to "monokai".
- **dedent** (*bool, optional*) – Enable stripping of initial whitespace. Defaults to False.
- **line_numbers** (*bool, optional*) – Enable rendering of line numbers. Defaults to False.
- **start_line** (*int, optional*) – Starting number for line numbers. Defaults to 1.
- **line_range** (*Tuple[int, int], optional*) – If given should be a tuple of the start and end line to render.
- **highlight_lines** (*Set[int]*) – A set of line numbers to highlight.
- **code_width** – Width of code to render (not including line numbers), or `None` to use all available width.
- **tab_size** (*int, optional*) – Size of tabs. Defaults to 4.
- **word_wrap** (*bool, optional*) – Enable word wrapping.
- **background_color** (*str, optional*) – Optional background color, or None to use theme color. Defaults to None.
- **indent_guides** (*bool, optional*) – Show indent guides. Defaults to False.

classmethod from_path(*path: str*, *encoding: str = 'utf-8'*, *theme: Union[str, rich.syntax.SyntaxTheme] = 'monokai'*, *dedent: bool = False*, *line_numbers: bool = False*, *line_range: Optional[Tuple[int, int]] = None*, *start_line: int = 1*, *highlight_lines: Optional[Set[int]] = None*, *code_width: Optional[int] = None*, *tab_size: int = 4*, *word_wrap: bool = False*, *background_color: Optional[str] = None*, *indent_guides: bool = False*) → *rich.syntax.Syntax*

Construct a Syntax object from a file.

Parameters

- **path** (*str*) – Path to file to highlight.
- **encoding** (*str*) – Encoding of file.
- **theme** (*str, optional*) – Color theme, aka Pygments style (see https://pygments.org/docs/styles/#getting-a-list-of-available-styles). Defaults to "emacs".
- **dedent** (*bool, optional*) – Enable stripping of initial whitespace. Defaults to True.
- **line_numbers** (*bool, optional*) – Enable rendering of line numbers. Defaults to False.
- **start_line** (*int, optional*) – Starting number for line numbers. Defaults to 1.
- **line_range** (*Tuple[int, int], optional*) – If given should be a tuple of the start and end line to render.
- **highlight_lines** (*Set[int]*) – A set of line numbers to highlight.
- **code_width** – Width of code to render (not including line numbers), or `None` to use all available width.
- **tab_size** (*int, optional*) – Size of tabs. Defaults to 4.
- **word_wrap** (*bool, optional*) – Enable word wrapping of code.
- **background_color** (*str, optional*) – Optional background color, or None to use theme color. Defaults to None.
- **indent_guides** (*bool, optional*) – Show indent guides. Defaults to False.

Returns A Syntax object that may be printed to the console

Return type [*Syntax*]

classmethod get_theme(*name: Union[str, rich.syntax.SyntaxTheme]*) → rich.syntax.SyntaxTheme

Get a syntax theme instance.

highlight(*code: str*, *line_range: Optional[Tuple[int, int]] = None*) → *rich.text.Text*

Highlight code and return a Text instance.

Parameters

- **code** (*str*) – Code to highlight.
- **line_range** (*Tuple[int, int], optional*) – Optional line range to highlight.

Returns A text instance containing highlighted syntax.

Return type *Text*

22.29 rich.table

class rich.table.**Column**(*header: RenderableType = ''*, *footer: RenderableType = ''*, *header_style: Union[str, Style] = ''*, *footer_style: Union[str, Style] = ''*, *style: Union[str, Style] = ''*, *justify: JustifyMethod = 'left'*, *overflow: OverflowMethod = 'ellipsis'*, *width: Optional[int] = None*, *min_width: Optional[int] = None*, *max_width: Optional[int] = None*, *ratio: Optional[int] = None*, *no_wrap: bool = False*, *_index: int = 0*, *_cells: List[RenderableType] = <factory>*)

Defines a column in a table.

property cells

Get all cells in the column, not including header.

copy() → *rich.table.Column*

Return a copy of this Column.

property flexible

Check if this column is flexible.

footer: RenderableType = ''

Renderable for the footer (typically a string)

Type RenderableType

footer_style: Union[str, Style] = ''

The style of the footer.

Type StyleType

header: RenderableType = ''

Renderable for the header (typically a string)

Type RenderableType

header_style: Union[str, Style] = ''

The style of the header.

Type StyleType

justify: JustifyMethod = 'left'

How to justify text within the column ("left", "center", "right", or "full")

Type str

max_width: Optional[int] = None

Maximum width of column, or `None` for no maximum. Defaults to None.

Type Optional[int]

min_width: Optional[int] = None

Minimum width of column, or `None` for no minimum. Defaults to None.

Type Optional[int]

no_wrap: bool = False

Prevent wrapping of text within the column. Defaults to `False`.

Type bool

overflow: OverflowMethod = 'ellipsis'

Overflow method.

Type str

ratio: Optional[int] = None
Ratio to use when calculating column width, or None (default) to adapt to column contents.

Type Optional[int]

style: Union[str, Style] = ''
The style of the column.

Type StyleType

width: Optional[int] = None
Width of the column, or None (default) to auto calculate width.

Type Optional[int]

class rich.table.**Row**(*style: Optional[Union[str, rich.style.Style]] = None*, *end_section: bool = False*)
Information regarding a row.

end_section: bool = False
Indicated end of section, which will force a line beneath the row.

style: Optional[Union[str, *rich.style.Style*]] = None
Style to apply to row.

class rich.table.**Table**(**headers: Union[rich.table.Column, str]*, *title: Union[str, Text] = None*, *caption: Union[str, Text] = None*, *width: int = None*, *min_width: int = None*, *box: Optional[rich.box.Box] = Box(...)*, *safe_box: Optional[bool] = None*, *padding: Union[int, Tuple[int], Tuple[int, int], Tuple[int, int, int, int]] = (0, 1)*, *collapse_padding: bool = False*, *pad_edge: bool = True*, *expand: bool = False*, *show_header: bool = True*, *show_footer: bool = False*, *show_edge: bool = True*, *show_lines: bool = False*, *leading: int = 0*, *style: Union[str, Style] = 'none'*, *row_styles: Iterable[Union[str, Style]] = None*, *header_style: Optional[Union[str, Style]] = 'table.header'*, *footer_style: Optional[Union[str, Style]] = 'table.footer'*, *border_style: Union[str, Style] = None*, *title_style: Union[str, Style] = None*, *caption_style: Union[str, Style] = None*, *title_justify: JustifyMethod = 'center'*, *caption_justify: JustifyMethod = 'center'*, *highlight: bool = False*)
A console renderable to draw a table.

Parameters

- ***headers** (*Union[Column, str]*) – Column headers, either as a string, or *Column* instance.
- **title** (*Union[str, Text], optional*) – The title of the table rendered at the top. Defaults to None.
- **caption** (*Union[str, Text], optional*) – The table caption rendered below. Defaults to None.
- **width** (*int, optional*) – The width in characters of the table, or None to automatically fit. Defaults to None.
- **min_width** (*Optional[int], optional*) – The minimum width of the table, or None for no minimum. Defaults to None.
- **box** (*box.Box, optional*) – One of the constants in box.py used to draw the edges (see *Box*). Defaults to box.HEAVY_HEAD.
- **safe_box** (*Optional[bool], optional*) – Disable box characters that don't display on windows legacy terminal with *raster* fonts. Defaults to True.

- **padding** (*PaddingDimensions, optional*) – Padding for cells (top, right, bottom, left). Defaults to (0, 1).
- **collapse_padding** (*bool, optional*) – Enable collapsing of padding around cells. Defaults to False.
- **pad_edge** (*bool, optional*) – Enable padding of edge cells. Defaults to True.
- **expand** (*bool, optional*) – Expand the table to fit the available space if `True`, otherwise the table width will be auto-calculated. Defaults to False.
- **show_header** (*bool, optional*) – Show a header row. Defaults to True.
- **show_footer** (*bool, optional*) – Show a footer row. Defaults to False.
- **show_edge** (*bool, optional*) – Draw a box around the outside of the table. Defaults to True.
- **show_lines** (*bool, optional*) – Draw lines between every row. Defaults to False.
- **leading** (*bool, optional*) – Number of blank lines between rows (precludes `show_lines`). Defaults to 0.
- **style** (*Union[str, Style], optional*) – Default style for the table. Defaults to "none".
- **row_styles** (*List[Union, str], optional*) – Optional list of row styles, if more that one style is give then the styles will alternate. Defaults to None.
- **header_style** (*Union[str, Style], optional*) – Style of the header. Defaults to "table.header".
- **footer_style** (*Union[str, Style], optional*) – Style of the footer. Defaults to "table.footer".
- **border_style** (*Union[str, Style], optional*) – Style of the border. Defaults to None.
- **title_style** (*Union[str, Style], optional*) – Style of the title. Defaults to None.
- **caption_style** (*Union[str, Style], optional*) – Style of the caption. Defaults to None.
- **title_justify** (*str, optional*) – Justify method for title. Defaults to "center".
- **caption_justify** (*str, optional*) – Justify method for caption. Defaults to "center".
- **highlight** (*bool, optional*) – Highlight cell contents (if str). Defaults to False.

add_column (*header: RenderableType = ''*, *footer: RenderableType = ''*, ***, *header_style: Union[str, Style] = None*, *footer_style: Union[str, Style] = None*, *style: Union[str, Style] = None*, *justify: JustifyMethod = 'left'*, *overflow: OverflowMethod = 'ellipsis'*, *width: int = None*, *min_width: int = None*, *max_width: int = None*, *ratio: int = None*, *no_wrap: bool = False*) → None

Add a column to the table.

Parameters

- **header** (*RenderableType, optional*) – Text or renderable for the header. Defaults to "".
- **footer** (*RenderableType, optional*) – Text or renderable for the footer. Defaults to "".

- **header_style** (*Union[str, Style], optional*) – Style for the header, or None for default. Defaults to None.
- **footer_style** (*Union[str, Style], optional*) – Style for the footer, or None for default. Defaults to None.
- **style** (*Union[str, Style], optional*) – Style for the column cells, or None for default. Defaults to None.
- **justify** (*JustifyMethod, optional*) – Alignment for cells. Defaults to "left".
- **width** (*int, optional*) – Desired width of column in characters, or None to fit to contents. Defaults to None.
- **min_width** (*Optional[int], optional*) – Minimum width of column, or `None` for no minimum. Defaults to None.
- **max_width** (*Optional[int], optional*) – Maximum width of column, or `None` for no maximum. Defaults to None.
- **ratio** (*int, optional*) – Flexible ratio for the column (requires `Table.expand` or `Table.width`). Defaults to None.
- **no_wrap** (*bool, optional*) – Set to `True` to disable wrapping of this column.

add_row(**renderables: Optional[RenderableType], style: Union[str, Style] = None, end_section: bool = False*) → None

Add a row of renderables.

Parameters

- ***renderables** (*None or renderable*) – Each cell in a row must be a renderable object (including str), or `None` for a blank cell.
- **style** (*StyleType, optional*) – An optional style to apply to the entire row. Defaults to None.
- **end_section** (*bool, optional*) – End a section and draw a line. Defaults to False.

Raises **errors.NotRenderableError** – If you add something that can't be rendered.

property expand

Setting a non-None self.width implies expand.

get_row_style(*console: Console, index: int*) → Union[str, Style]

Get the current row style.

classmethod grid(**headers: Union[rich.table.Column, str], padding: Union[int, Tuple[int], Tuple[int, int], Tuple[int, int, int, int]] = 0, collapse_padding: bool = True, pad_edge: bool = False, expand: bool = False*) → *rich.table.Table*

Get a table with no lines, headers, or footer.

Parameters

- ***headers** (*Union[Column, str]*) – Column headers, either as a string, or *Column* instance.
- **padding** (*PaddingDimensions, optional*) – Get padding around cells. Defaults to 0.
- **collapse_padding** (*bool, optional*) – Enable collapsing of padding around cells. Defaults to True.
- **pad_edge** (*bool, optional*) – Enable padding around edges of table. Defaults to False.

- **expand** (*bool, optional*) – Expand the table to fit the available space if `True`, otherwise the table width will be auto-calculated. Defaults to False.

Returns A table instance.

Return type *Table*

property padding
Get cell padding.

property row_count
Get the current number of rows.

22.30 rich.text

class rich.text.**Text**(*text: str = ''*, *style: Union[str, rich.style.Style] = ''*, ***, *justify: JustifyMethod = None*, *overflow: OverflowMethod = None*, *no_wrap: bool = None*, *end: str = '\n'*, *tab_size: Optional[int] = 8*, *spans: List[rich.text.Span] = None*)
Text with color / style.

Parameters

- **text** (*str, optional*) – Default unstyled text. Defaults to "".
- **style** (*Union[str, Style], optional*) – Base style for text. Defaults to "".
- **justify** (*str, optional*) – Justify method: "left", "center", "full", "right". Defaults to None.
- **overflow** (*str, optional*) – Overflow method: "crop", "fold", "ellipsis". Defaults to None.
- **no_wrap** (*bool, optional*) – Disable text wrapping, or None for default. Defaults to None.
- **end** (*str, optional*) – Character to end text with. Defaults to "\n".
- **tab_size** (*int*) – Number of spaces per tab, or `None` to use `console.tab_size`. Defaults to 8.
- **spans** (*List[Span], optional*) –

align(*align: typing_extensions.Literal[left, center, right]*, *width: int*, *character: str = ' '*) → None
Align text to a given width.

Parameters

- **align** (*AlignMethod*) – One of "left", "center", or "right".
- **width** (*int*) – Desired width.
- **character** (*str, optional*) – Character to pad with. Defaults to " ".

append(*text: Union[rich.text.Text, str]*, *style: Optional[Union[str, rich.style.Style]] = None*) → *rich.text.Text*
Add text with an optional style.

Parameters

- **text** (*Union[Text, str]*) – A str or Text to append.
- **style** (*str, optional*) – A style name. Defaults to None.

Returns Returns self for chaining.

Return type *Text*

append_text (*text:* rich.text.Text) → *rich.text.Text*
Append another Text instance. This method is more performant that Text.append, but only works for Text.

Returns Returns self for chaining.

Return type *Text*

append_tokens (*tokens: Iterable[Tuple[str, Optional[Union[str,* rich.style.Style*]]]]*)
Append iterable of str and style. Style may be a Style instance or a str style definition.

Parameters **pairs** (*Iterable[Tuple[str, Optional[StyleType]]]*) – An iterable of tuples containing str content and style.

Returns Returns self for chaining.

Return type *Text*

classmethod assemble (**parts: Union[str,* Text*, Tuple[str, Union[str, Style]]], style: Union[str,* rich.style.Style*] = '', justify: JustifyMethod = None, overflow: OverflowMethod = None, no_wrap: bool = None, end: str = '\n', tab_size: int = 8*) → *Text*
Construct a text instance by combining a sequence of strings with optional styles. The positional arguments should be either strings, or a tuple of string + style.

Parameters

- **style** (*Union[str,* Style*], optional*) – Base style for text. Defaults to "".
- **justify** (*str, optional*) – Justify method: "left", "center", "full", "right". Defaults to None.
- **overflow** (*str, optional*) – Overflow method: "crop", "fold", "ellipsis". Defaults to None.
- **end** (*str, optional*) – Character to end text with. Defaults to "\n".
- **tab_size** (*int*) – Number of spaces per tab, or `None` to use `console.tab_size`. Defaults to 8.

Returns A new text instance.

Return type *Text*

blank_copy (*plain: str = ''*) → *rich.text.Text*
Return a new Text instance with copied meta data (but not the string or spans).

property cell_len
Get the number of cells required to render this text.

copy () → *rich.text.Text*
Return a copy of this instance.

copy_styles (*text:* rich.text.Text) → None
Copy styles from another Text instance.

Parameters **text** (`Text`) – A Text instance to copy styles from, must be the same length.

detect_indentation () → int
Auto-detect indentation of code.

Returns Number of spaces used to indent code.

Return type int

divide (*offsets: Iterable[int]*) → rich.containers.Lines
Divide text in to a number of lines at given offsets.

Parameters **offsets** (*Iterable[int]*) – Offsets used to divide text.

Returns New RichText instances between offsets.

Return type Lines

expand_tabs (*tab_size: Optional[int] = None*) → None
Converts tabs to spaces.

Parameters **tab_size** (*int, optional*) – Size of tabs. Defaults to 8.

fit (*width: int*) → rich.containers.Lines
Fit the text in to given width by chopping in to lines.

Parameters **width** (*int*) – Maximum characters in a line.

Returns List of lines.

Return type Lines

classmethod from_markup (*text: str*, ***, *style: Union[str, rich.style.Style] = ''*, *emoji: bool = True*, *justify: JustifyMethod = None*, *overflow: OverflowMethod = None*) → *Text*
Create Text instance from markup.

Parameters

- **text** (*str*) – A string containing console markup.
- **emoji** (*bool, optional*) – Also render emoji code. Defaults to True.
- **justify** (*str, optional*) – Justify method: “left”, “center”, “full”, “right”. Defaults to None.
- **overflow** (*str, optional*) – Overflow method: “crop”, “fold”, “ellipsis”. Defaults to None.

Returns A Text instance with markup rendered.

Return type *Text*

get_style_at_offset (*console: Console*, *offset: int*) → *rich.style.Style*
Get the style of a character at give offset.

Parameters

- **console** (*~Console*) – Console where text will be rendered.
- **offset** (*int*) – Offset in to text (negative indexing supported)

Returns A Style instance.

Return type *Style*

highlight_regex (*re_highlight: str*, *style: Optional[Union[Callable[[str], Optional[Union[str, rich.style.Style]]], str, rich.style.Style]] = None*, ***, *style_prefix: str = ''*) → int
Highlight text with a regular expression, where group names are translated to styles.

Parameters

- **re_highlight** (*str*) – A regular expression.
- **style** (*Union[GetStyleCallable, StyleType]*) – Optional style to apply to whole match, or a callable which accepts the matched text and returns a style. Defaults to None.

- **style_prefix** (*str, optional*) – Optional prefix to add to style group names.

Returns Number of regex matches

Return type int

highlight_words(*words: Iterable[str]*, *style: Union[str,* rich.style.Style*]*, ***, *case_sensitive: bool = True*) → int

Highlight words with a style.

Parameters

- **words** (*Iterable[str]*) – Worlds to highlight.
- **style** (*Union[str, Style]*) – Style to apply.
- **case_sensitive** (*bool, optional*) – Enable case sensitive matchings. Defaults to True.

Returns Number of words highlighted.

Return type int

join(*lines: Iterable[*rich.text.Text*]*) → *rich.text.Text*

Join text together with this instance as the separator.

Parameters **lines** (*Iterable[Text]*) – An iterable of Text instances to join.

Returns A new text instance containing join text.

Return type *Text*

pad(*count: int*, *character: str = ' '*) → None

Pad left and right with a given number of characters.

Parameters **count** (*int*) – Width of padding.

pad_left(*count: int*, *character: str = ' '*) → None

Pad the left with a given character.

Parameters

- **count** (*int*) – Number of characters to pad.
- **character** (*str, optional*) – Character to pad with. Defaults to " ".

pad_right(*count: int*, *character: str = ' '*) → None

Pad the right with a given character.

Parameters

- **count** (*int*) – Number of characters to pad.
- **character** (*str, optional*) – Character to pad with. Defaults to " ".

property plain

Get the text as a single string.

remove_suffix(*suffix: str*) → None

Remove a suffix if it exists.

Parameters **suffix** (*str*) – Suffix to remove.

render(*console: Console*, *end: str = ''*) → Iterable[Segment]

Render the text as Segments.

Parameters

- **console** (Console) – Console instance.

- **end** (*Optional[str], optional*) – Optional end character.

Returns Result of render that may be written to the console.

Return type Iterable[*Segment*]

right_crop(*amount: int = 1*) → None
Remove a number of characters from the end of the text.

rstrip() → None
Strip whitespace from end of text.

rstrip_end(*size: int*) → None
Remove whitespace beyond a certain width at the end of the text.

Parameters **size** (*int*) – The desired size of the text.

set_length(*new_length: int*) → None
Set new length of the text, clipping or padding is required.

property spans
Get a reference to the internal list of spans.

split(*separator='\n', *, include_separator: bool = False, allow_blank: bool = False*) → rich.containers.Lines
Split rich text in to lines, preserving styles.

Parameters

- **separator** (*str, optional*) – String to split on. Defaults to "\n".
- **include_separator** (*bool, optional*) – Include the separator in the lines. Defaults to False.
- **allow_blank** (*bool, optional*) – Return a blank line if the text ends with a separator. Defaults to False.

Returns A list of rich text, one per line of the original.

Return type List[RichText]

classmethod styled(*text: str, style: Union[str, Style] = '', *, justify: JustifyMethod = None, overflow: OverflowMethod = None*) → *Text*
Construct a Text instance with a pre-applied styled. A style applied in this way won't be used to pad the text when it is justified.

Parameters

- **text** (*str*) – A string containing console markup.
- **style** (*Union[str, Style]*) – Style to apply to the text. Defaults to "".
- **justify** (*str, optional*) – Justify method: "left", "center", "full", "right". Defaults to None.
- **overflow** (*str, optional*) – Overflow method: "crop", "fold", "ellipsis". Defaults to None.

Returns A text instance with a style applied to the entire string.

Return type *Text*

stylize(*style: Union[str, rich.style.Style], start: int = 0, end: Optional[int] = None*) → None
Apply a style to the text, or a portion of the text.

Parameters

- **style** (*Union[str, Style]*) – Style instance or style definition to apply.
- **start** (*int*) – Start offset (negative indexing is supported). Defaults to 0.
- **end** (*Optional[int], optional*) – End offset (negative indexing is supported), or None for end of text. Defaults to None.

truncate(*max_width: int*, ***, *overflow: Optional[OverflowMethod] = None*, *pad: bool = False*) → None

Truncate text if it is longer that a given width.

Parameters

- **max_width** (*int*) – Maximum number of characters in text.
- **overflow** (*str, optional*) – Overflow method: “crop”, “fold”, or “ellipsis”. Defaults to None, to use self.overflow.
- **pad** (*bool, optional*) – Pad with spaces if the length is less than max_width. Defaults to False.

with_indent_guides(*indent_size: Optional[int] = None*, ***, *character: str = '│'*, *style: Union[str, rich.style.Style] = 'dim green'*) → *rich.text.Text*

Adds indent guide lines to text.

Parameters

- **indent_size** (*Optional[int]*) – Size of indentation, or None to auto detect. Defaults to None.
- **character** (*str, optional*) – Character to use for indentation. Defaults to “│”.
- **style** (*Union[Style, str], optional*) – Style of indent guides.

Returns New text with indentation guides.

Return type *Text*

wrap(*console: Console*, *width: int*, ***, *justify: JustifyMethod = None*, *overflow: OverflowMethod = None*, *tab_size: int = 8*, *no_wrap: bool = None*) → rich.containers.Lines

Word wrap the text.

Parameters

- **console** (*Console*) – Console instance.
- **width** (*int*) – Number of characters per line.
- **emoji** (*bool, optional*) – Also render emoji code. Defaults to True.
- **justify** (*str, optional*) – Justify method: “default”, “left”, “center”, “full”, “right”. Defaults to “default”.
- **overflow** (*str, optional*) – Overflow method: “crop”, “fold”, or “ellipsis”. Defaults to None.
- **tab_size** (*int, optional*) – Default tab size. Defaults to 8.
- **no_wrap** (*bool, optional*) – Disable wrapping, Defaults to False.

Returns Number of lines.

Return type Lines

22.31 rich.theme

class rich.theme.**Theme**(*styles: Optional[Mapping[str, Union[str, rich.style.Style]]] = None*, *inherit: bool = True*)

A container for style information, used by *Console*.

Parameters

- **styles** (*Dict[str, Style], optional*) – A mapping of style names on to styles. Defaults to None for a theme with no styles.
- **inherit** (*bool, optional*) – Inherit default styles. Defaults to True.

property config

Get contents of a config file for this theme.

classmethod from_file(*config_file: IO[str]*, *source: Optional[str] = None*, *inherit: bool = True*) → *rich.theme.Theme*

Load a theme from a text mode file.

Parameters

- **config_file** (*IO[str]*) – An open conf file.
- **source** (*str, optional*) – The filename of the open file. Defaults to None.
- **inherit** (*bool, optional*) – Inherit default styles. Defaults to True.

Returns A New theme instance.

Return type *Theme*

classmethod read(*path: str*, *inherit: bool = True*) → *rich.theme.Theme*

Read a theme from a path.

Parameters

- **path** (*str*) – Path to a config file readable by Python configparser module.
- **inherit** (*bool, optional*) – Inherit default styles. Defaults to True.

Returns A new theme instance.

Return type *Theme*

22.32 rich.traceback

class rich.traceback.**Traceback**(*trace: Optional[rich.traceback.Trace] = None*, *width: Optional[int] = 100*, *extra_lines: int = 3*, *theme: Optional[str] = None*, *word_wrap: bool = False*, *show_locals: bool = False*, *indent_guides: bool = True*, *locals_max_length: int = 10*, *locals_max_string: int = 80*)

A Console renderable that renders a traceback.

Parameters

- **trace** (*Trace, optional*) – A *Trace* object produced from *extract*. Defaults to None, which uses the last exception.
- **width** (*Optional[int], optional*) – Number of characters used to traceback. Defaults to 100.
- **extra_lines** (*int, optional*) – Additional lines of code to render. Defaults to 3.

- **theme** (*str, optional*) – Override pygments theme used in traceback.
- **word_wrap** (*bool, optional*) – Enable word wrapping of long lines. Defaults to False.
- **show_locals** (*bool, optional*) – Enable display of local variables. Defaults to False.
- **indent_guides** (*bool, optional*) – Enable indent guides in code and locals. Defaults to True.
- **locals_max_length** (*int, optional*) – Maximum length of containers before abbreviating, or None for no abbreviation. Defaults to 10.
- **locals_max_string** (*int, optional*) – Maximum length of string before truncating, or None to disable. Defaults to 80.

classmethod extract(*exc_type: Type[BaseException]*, *exc_value: BaseException*, *traceback: Optional[traceback]*, *show_locals: bool = False*, *locals_max_length: int = 10*, *locals_max_string: int = 80*) → rich.traceback.Trace

Extract traceback information.

Parameters

- **exc_type** (*Type[BaseException]*) – Exception type.
- **exc_value** (*BaseException*) – Exception value.
- **traceback** (*TracebackType*) – Python Traceback object.
- **show_locals** (*bool, optional*) – Enable display of local variables. Defaults to False.
- **locals_max_length** (*int, optional*) – Maximum length of containers before abbreviating, or None for no abbreviation. Defaults to 10.
- **locals_max_string** (*int, optional*) – Maximum length of string before truncating, or None to disable. Defaults to 80.

Returns A Trace instance which you can use to construct a *Traceback*.

Return type Trace

classmethod from_exception(*exc_type: Type*, *exc_value: BaseException*, *traceback: Optional[traceback]*, *width: Optional[int] = 100*, *extra_lines: int = 3*, *theme: Optional[str] = None*, *word_wrap: bool = False*, *show_locals: bool = False*, *indent_guides: bool = True*, *locals_max_length: int = 10*, *locals_max_string: int = 80*) → *rich.traceback.Traceback*

Create a traceback from exception info

Parameters

- **exc_type** (*Type[BaseException]*) – Exception type.
- **exc_value** (*BaseException*) – Exception value.
- **traceback** (*TracebackType*) – Python Traceback object.
- **width** (*Optional[int], optional*) – Number of characters used to traceback. Defaults to 100.
- **extra_lines** (*int, optional*) – Additional lines of code to render. Defaults to 3.
- **theme** (*str, optional*) – Override pygments theme used in traceback.

- **word_wrap** (*bool, optional*) – Enable word wrapping of long lines. Defaults to False.
- **show_locals** (*bool, optional*) – Enable display of local variables. Defaults to False.
- **indent_guides** (*bool, optional*) – Enable indent guides in code and locals. Defaults to True.
- **locals_max_length** (*int, optional*) – Maximum length of containers before abbreviating, or None for no abbreviation. Defaults to 10.
- **locals_max_string** (*int, optional*) – Maximum length of string before truncating, or None to disable. Defaults to 80.

Returns A Traceback instance that may be printed.

Return type *Traceback*

rich.traceback.**install**(*, *console: Optional[rich.console.Console] = None*, *width: Optional[int] = 100*, *extra_lines: int = 3*, *theme: Optional[str] = None*, *word_wrap: bool = False*, *show_locals: bool = False*, *indent_guides: bool = True*) → Callable

Install a rich traceback handler.

Once installed, any tracebacks will be printed with syntax highlighting and rich formatting.

Parameters

- **console** (*Optional[Console], optional*) – Console to write exception to. Default uses internal Console instance.
- **width** (*Optional[int], optional*) – Width (in characters) of traceback. Defaults to 100.
- **extra_lines** (*int, optional*) – Extra lines of code. Defaults to 3.
- **theme** (*Optional[str], optional*) – Pygments theme to use in traceback. Defaults to `None` which will pick a theme appropriate for the platform.
- **word_wrap** (*bool, optional*) – Enable word wrapping of long lines. Defaults to False.
- **show_locals** (*bool, optional*) – Enable display of local variables. Defaults to False.
- **indent_guides** (*bool, optional*) – Enable indent guides in code and locals. Defaults to True.

Returns The previous exception handler that was replaced.

Return type Callable

22.33 rich.tree

class rich.tree.**Tree**(*label: Union[rich.console.ConsoleRenderable, rich.console.RichCast, str]*, *, *style: Union[str, rich.style.Style] = 'tree'*, *guide_style: Union[str, rich.style.Style] = 'tree.line'*, *expanded=True*, *highlight=False*)

A renderable for a tree structure.

Parameters

- **label** (*RenderableType*) – The renderable or str for the tree label.

- **style** (*StyleType, optional*) – Style of this tree. Defaults to "tree".
- **guide_style** (*StyleType, optional*) – Style of the guide lines. Defaults to "tree.line".
- **expanded** (*bool, optional*) – Also display children. Defaults to True.
- **highlight** (*bool, optional*) – Highlight renderable (if str). Defaults to False.

add(*label: Union[*rich.console.ConsoleRenderable, rich.console.RichCast, *str]*, *, *style: Optional[Union[str,* rich.style.Style*]] = None*, *guide_style: Optional[Union[str,* rich.style.Style*]] = None*, *expanded=True*, *highlight=False*) → *rich.tree.Tree*

Add a child tree.

Parameters

- **label** (*RenderableType*) – The renderable or str for the tree label.
- **style** (*StyleType, optional*) – Style of this tree. Defaults to "tree".
- **guide_style** (*StyleType, optional*) – Style of the guide lines. Defaults to "tree.line".
- **expanded** (*bool, optional*) – Also display children. Defaults to True.
- **highlight** (*Optional[bool], optional*) – Highlight renderable (if str). Defaults to False.

Returns A new child Tree, which may be further modified.

Return type *Tree*

22.34 rich.abc

class rich.abc.**RichRenderable**

An abstract base class for Rich renderables.

Note that there is no need to extend this class, the intended use is to check if an object supports the Rich renderable protocol. For example:

```
if isinstance(my_object, RichRenderable):
    console.print(my_object)
```

CHAPTER

TWENTYTHREE

APPENDIX

23.1 Box

Rich has a number of constants that set the box characters used to draw tables and panels. To select a box style import one of the constants below from `rich.box`. For example:

```
from rich import box
table = Table(box=box.SQUARE)
```

Note: Some of the box drawing characters will not display correctly on Windows legacy terminal (cmd.exe) with *raster* fonts, and are disabled by default. If you want the full range of box options on Windows legacy terminal, use a *truetype* font and set the `safe_box` parameter on the Table class to `False`.

The following table is generated with this command:

```
python -m rich.box
```

23.2 Standard Colors

The following is a list of the standard 8-bit colors supported in terminals.

Note that the first 16 colors are generally defined by the system or your terminal software, and may not display exactly as rendered here.

CHAPTER

TWENTYFOUR

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

r

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

R

S

T

U

V

W