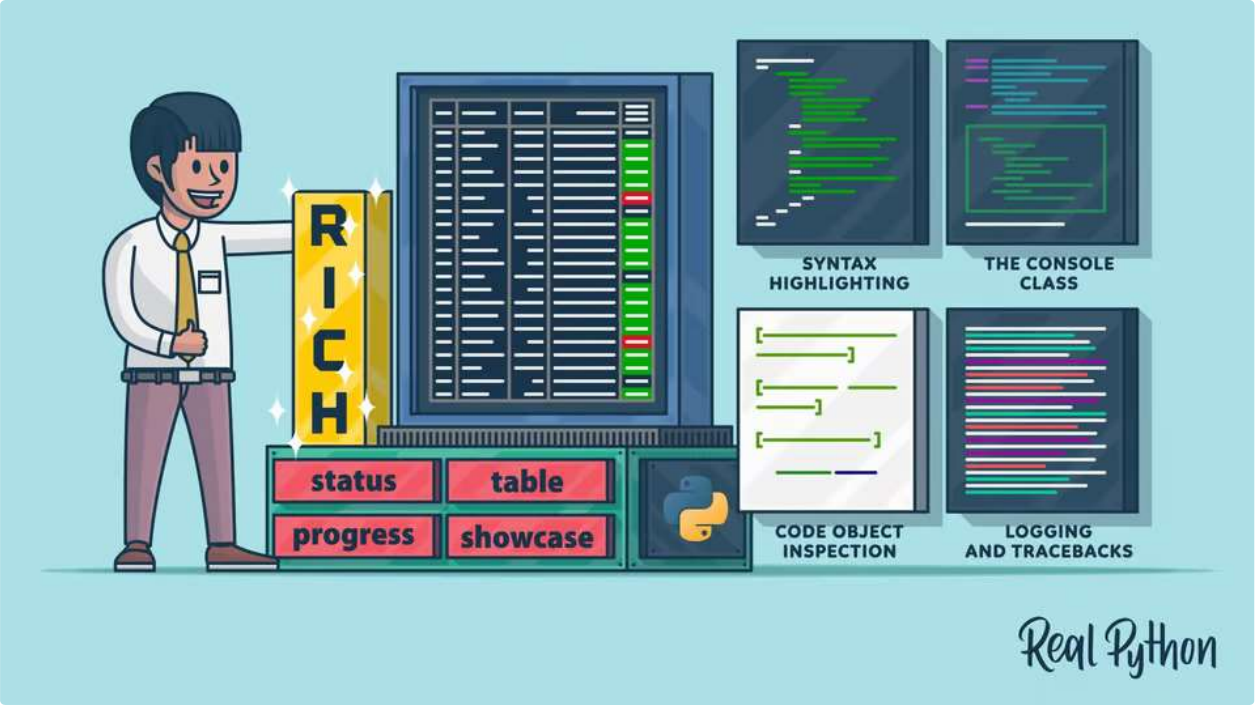



# The Python Rich Package: Unleash the Power of Console Text

by Charles de Villiers Nov 27, 2023 4 Comments front-end intermediate python



## Table of Contents


- Installing Rich
- Using Rich for Python Development
  - Syntax Highlighting
  - Code Object Inspection
  - The Console Class
  - Logging and Tracebacks
- Keeping Your User Engaged Through Animation
  - Understanding Context Managers
  - Displaying Dynamic Status With Animations
  - Animating Activities With Progress Bars
- Bringing Tables to Life
  - Building a Static Table
  - Animating a Scrolling Display
  - Accessing the Crypto Data
  - Coding the Live Table
- Conclusion
- Next Steps

Python's Rich package is a tool kit that helps you generate beautifully formatted and highlighted text in the console. More broadly, it allows you to build an attractive text-based user interface (TUI).

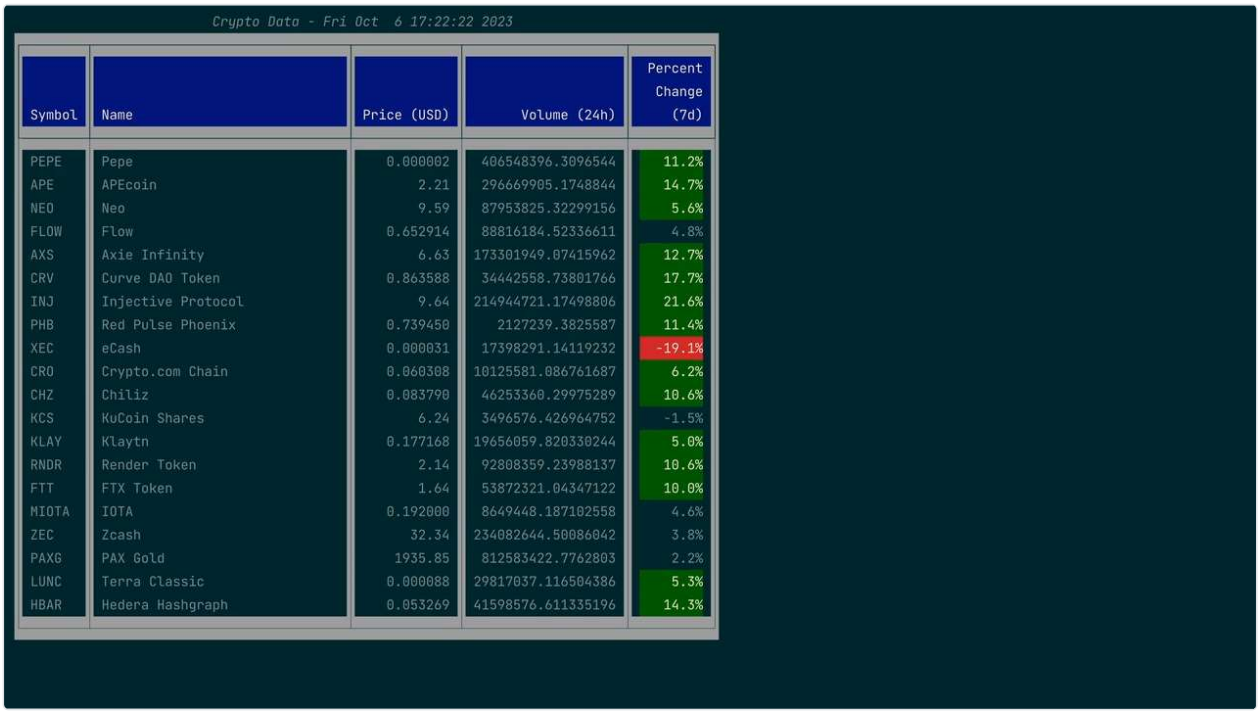
Why would you choose a TUI over a graphical user interface, or GUI? Sometimes a text display feels more appropriate. Why use a full-blown GUI for a simple application, when an elegant text interface will do? It can be refreshing to work with plain text. Text works in almost any hardware environment, even on an SSH terminal or a single-board computer display. And many applications don't need the complexity of a full graphical windowing system.

**In this tutorial, you'll learn how Rich can help you:**

- Enhance the **user interface** of **command-line tools**
- Improve the readability of **console output**
- Create attractive **dashboard displays** for real-time tabular data
- Generate **well-formatted reports**

Will McGugan, the author of Rich, has also developed the Textual package. Whereas Rich is a rich-text tool kit, Textual is a full application framework built on Rich. It provides application base classes, an event-driven architecture, and more.

There's a lot you can do with Rich on its own, and its support for engaging, dynamic displays may well be sufficient for your app. By following this tutorial, you'll experiment with many of the cool features of Rich, and you'll finish up by using your skills to build a dynamically scrolling tabular display of crypto prices:




To fully understand Rich's syntax for animations, you should have a good grasp of context managers. But if you're a bit rusty, don't worry! You'll get a quick refresher in this tutorial.

**Get Your Code: Click here to download free sample code** that shows you how to use Rich for more beautiful Python code and apps.

## Installing Rich

You can start using Rich very quickly. As always when starting a new project or investigation, it's best to create a virtual environment first, to avoid polluting your system's Python installation.

It's quite possible to install Rich and use it with the built-in Python REPL, but for a better developer experience, you may want to include support for Jupyter notebooks. Here's how you can install Rich so that it'll work with either the REPL or Jupyter:

Windows | Linux + macOS

Windows PowerShell

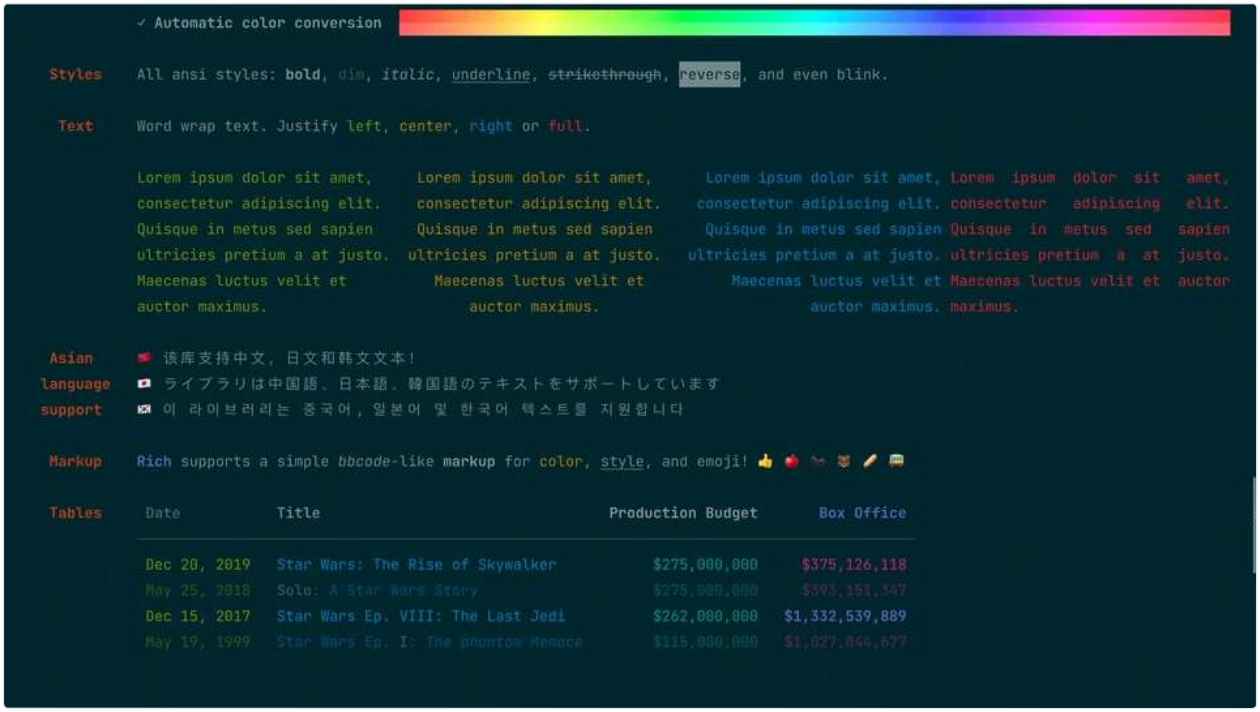
```
PS> python -m venv venv
PS> venv\Scripts\activate
(venv) PS> python -m pip install rich[jupyter]
```

Now that you've installed Rich in your new virtual environment, you can test it and also get a nice overview of its capabilities:

Shell

```
(venv) $ python -m rich
```

Running this command will make lots of magic happen. Your terminal will fill with color, and you'll see several options for customizing your text-based user interface:




Apart from displaying colorful text in a variety of styles, this demo also illustrates a few more of Rich's exciting features.

You can wrap and justify text. You can easily display any Unicode characters, as well as a wide choice of emojis. Rich renders Markdown and offers a syntax for creating elegantly formatted tables.

There's much more that you can do with Rich, as you'll discover throughout this tutorial. You can also try some of the command-line demos that Rich has thoughtfully provided for its subpackages, so you can get a flavor of what each one can do without writing any code.

Here are a few that you can try from your OS console. Execute them one by one to get a feel for the power of Rich:

Shell

```
(venv) $ python -m rich.table
(venv) $ python -m rich.progress
(venv) $ python -m rich.status
```

Most of these demos are very short, but if necessary, you can always interrupt them with `^ Ctrl` + `C`.

With the installation done, you're ready to start exploring Rich.

# Using Rich for Python Development

Rich can help to make your life as a developer a little easier. For instance, it has built-in support for formatting and syntax-highlighting Python code and data structures, and it has a very useful `inspect()` function that lets you examine deeply nested data.

You can make the most of Rich's development support by using it with IPython or Jupyter. If you don't know these tools, then it's well worth giving them a try. Rich provides special support for both of them. Here, though, you'll be exploring Rich with the built-in REPL.

## Syntax Highlighting

The first Rich feature that you'll explore is **syntax highlighting**. This helps to clarify the structure of programming statements and data. Syntax highlighting is built into Rich's `print()` function. Open the Python REPL, define a simple data structure, and print it using Python's built-in `print()` function:

Python

```
>>> student = { "person": {"name": "John Jones", "age": 30, "subscriber": True}}
>>> print(student)
```

This does just what you'd expect:

```
>>> print(student)
{'person': {'name': 'John Jones', 'age': 30, 'subscriber': True}}
```

That output has all the information that you wanted to display, but it doesn't look very exciting. Wouldn't it be nice if the different data types were color-coded to provide some visual variety and help the user keep the information straight? The Rich version does just that:

Python

```
>>> from rich import print as rprint
>>> rprint(student)
```

Now the different data types are highlighted:

```
>>> rprint(student)
{'person': {'name': 'John Jones', 'age': 30, 'subscriber': True}}
```

The syntax highlighting makes it easy to see what types the structure contains.

> Python's standard library includes a package named `pprint` that supports pretty-printing. While its formatting of complex data structures is a definite improvement over the built-in `print()`, it doesn't support colored text.

Since Rich's `print()` is a drop-in replacement for Python's built-in function, you could safely override the built-in `print()`. Here, though, to make comparisons easier, you've imported it under the alias `rprint()`.

The real benefits of pretty-printing code appear when you're dealing with more complex structures. Here's a small example. Suppose you're designing a data structure for your latest app, and you've hand-coded this dictionary:

Python

```
>>> superhero = {"person": {"name": "John Jones", "age":30,
... "address":{"street": "123 Main St", "city": "Gotham",
... "state":"NY", "zip_code": "12345"},
... "superpowers":{"leaps_buildings":True,"factorizes_polynomials":False},
... "contacts":[{"type":"email","value":"john.jones@heroes.are.us"},
... {"type":"phone","value":"555-123-4567"}],
... "hobbies": ["reading","hiking","coding","crimefighting"],
... "family":{"spouse": {"name":"Griselda Jones", "age":28},
... "children":[{"name":"Bellatrix", "age":5, "name":"Draco", "age":8}
... ]}}}
```

The Python REPL is fairly forgiving about input format. So long as your code is valid Python, as here, the REPL is happy to accept it.

You carry on coding. A little later, you decide to write the code that parses this data structure. To remind yourself of the details, you print it:

```
>>> print(superhero)
{'person': {'name': 'John Jones', 'age': 30, 'address': {'street': '123 Main St', 'city': 'Gotham', 'st
ate': 'NY', 'zip_code': '12345'}, 'superpowers': {'leaps_buildings': True, 'factorizes_polynomials': Fa
lse}, 'contacts': [{'type': 'email', 'value': 'john.jones@heroes.are.us'}, {'type': 'phone', 'value': '
555-123-4567'}], 'hobbies': ['reading', 'hiking', 'coding', 'crimefighting'], 'family': {'spouse': {'na
me': 'Griselda Jones', 'age': 28}, 'children': [{'name': 'Draco', 'age': 8}]}}}
```

The data is all there, but the REPL has its own ideas about formatting. It's certainly possible to trawl through this `print()` output and identify the nested dictionaries and lists. But it's already a bit of a tiresome task, and a much larger structure could be really annoying. Now see what happens when you pretty-print it:

Python

```
>>> rprint(superhero)
```

The code is neatly formatted and takes advantage of syntax highlighting:

```
>>> rprint(superhero)
{
    'person': {
        'name': 'John Jones',
        'age': 30,
        'address': {'street': '123 Main St', 'city': 'Gotham', 'state': 'NY', 'zip_code': '12345'},
        'superpowers': {'leaps_buildings': True, 'factorizes_polynomials': False},
        'contacts': [
            {'type': 'email', 'value': 'john.jones@heroes.are.us'},
            {'type': 'phone', 'value': '555-123-4567'}
        ],
        'hobbies': ['reading', 'hiking', 'coding', 'crimefighting'],
        'family': {
            'spouse': {'name': 'Griselda Jones', 'age': 28},
            'children': [{'name': 'Draco', 'age': 8}]
        }
    }
}
```

Not only does the layout make more sense, but the various data types are colored differently. You'll probably agree that this makes it much easier to understand the structure.

Do you always want your data structures presented like this while you're developing? As you probably know, when you simply type a variable name into the REPL and press `Enter ↩`, it echoes that variable's value to the console. By default, the format is the same as for `print()`. But wouldn't it be nice to have that value pretty-printed by default? You can install Rich pretty-printing in the REPL:

Python

```
>>> from rich import pretty
>>> pretty.install()
```

Now, simply by typing your variable's name in the REPL, you'll automatically get a pretty-printed and highlighted representation, just like the `rprint()` output above. You can even configure your REPL to always install Rich's pretty-printing at startup. Having your data structures displayed in a pleasing and readable format can really make your coding experience more pleasant and productive!



## Code Object Inspection

It's great that your data structures are now prettily color-coded and formatted, but that's not always enough in development. Sometimes you want to peek under the hood of a data structure and really get a snapshot of how it works. For that, you can use Rich's `inspect()` function. See what it can do with your example data structure:

Python

```
>>> from rich import inspect
>>> inspect(superhero, methods=True)
```

With `inspect()`, you can really examine the inner workings of an object. Since `superhero` is a `dict`, Rich shows you all the available `dict` constructors before displaying the pretty-printed data as before. Next, courtesy of the optional `methods=True` parameter, you also get a handy summary of the object's methods with their short docstrings and parameter types:


The `inspect()` function is quite powerful. You can get a comprehensive description of its parameters by typing `inspect(inspect)` as suggested in the output above.

The Python standard library has its own `inspect` module that also allows live inspection of code objects. It's much more powerful, and also much more complex, than the Rich function that you've been using. You should check it out if you need heavy-duty code introspection. Python's `inspect` module doesn't offer syntax highlighting, however. For on-the-fly investigations, you may find the Rich function more convenient.

## The `Console` Class

Rich has a `Console` class that encapsulates most of the package's capabilities. A `Console` instance can format text, generate colorized log output, or pretty-print JSON, and it can also handle indentation, horizontal rules, widgets, panels, and tables, as well as interactive prompts and animations. You can capture `Console` output and export it as text, SVG, or HTML.

> **Note:** The `print()` function that comes with Rich is a shortcut that supports some of the `Console` features. In longer programs, you should prefer `Console` and its `.print()` method over `rich.print()` as it's more powerful.

`Console` will interpret **console markup** to apply colors and attributes to text on the fly. All these features are available subject to your terminal's capabilities, but most modern terminal emulators will handle everything just fine. Console markup consists of paired tags within square brackets:

Python

```
>>> from rich.console import Console
>>> console = Console()
>>> console.print("[green underline]Green underline[/green underline] "
... "[blue italic]Blue italic[/blue italic]")
```

The specified styles are applied to the text within the tags:

```
>>> console.print("[green underline]Green underline[/green underline]
... "[blue italic]Blue italic[/blue italic]")
Green underline Blue italic
```

You can find a complete list of the names, hex codes, and RGB values of the 255 standard text colors in the Rich appendix. You can also view them from the command line:

Shell

```
(venv) $ python -m rich.colors
```

If your terminal supports true colors, then you can specify any of the sixteen million colors available by their RGB values. Along with a full range of text colors, console markup supports attributes like `bold`, `blink`, `reverse`, `underline`, and `italic`.

A combination of a color and attributes is called a `Style`, and you can put several styles together in a dictionary to create a `Theme`:

Python

```
>>> from rich.console import Console
>>> from rich.theme import Theme

>>> custom_theme = Theme(
...     {"info": "bold cyan", "warning": "magenta", "danger": "bold red"}
... )
>>> console = Console(theme=custom_theme)
```

In `custom_theme`, you've defined complementary styles that you can apply to any `Console` instance:

```
>>> console.print("The Gostak distims the doshes", style="info")
The Gostak distims the doshes
>>> console.print("All your bases are belong to us", style="warning")
All your bases are belong to us
>>> console.print("Klingons on the starboard bow!", style="danger")
Klingons on the starboard bow!
```

Using a `Console` object with a `Theme` helps to keep your formatting consistent across the application.

## Logging and Tracebacks

Creating good log statements is an important part of writing maintainable code. Logging helps you during development and testing by confirming that your code is following the expected paths. It's invaluable when things go wrong in production code, as well-placed log statements can provide insight into obscure code misbehavior.

The `Console` class supports formatted logging and uses a syntax that's very close to that of Python's standard `logging` package. It can generate nicely formatted tracebacks of any uncaught exceptions in your code while using styles from a defined `Theme`.

Python's built-in tracebacks and error messages get more informative and useful with each new release, but some extra eye appeal doesn't hurt, and there's no substitute for good log messages to give context for a crash. Continuing the previous example, you can produce some samples of logging output:

Python

```
>>> from rich.traceback import install
>>> install(show_locals=True)
```

Log messages can also use the `Theme` that you defined in the previous code snippet:

```
>>> console.log("Nothing happening here", style="info")
[10:49:54] Nothing happening here
>>> console.log("Trouble brewing...", style="warning")
[10:49:59] Trouble brewing...
>>> console.log("Bad news!", style="danger")
[10:50:03] Bad news!
```

With `console.log()`, you automatically add timestamps, source filenames, and source line numbers to your logging statements.

While things are running normally, you'll just get the clean log output, as above. But if there's a crash during development, then you'll want to log as much information as possible about the cause. Try manually throwing a `RuntimeError`, and see the difference:


When an exception happens, the traceback can optionally display all your local variables as well as the stack trace of the error. Your session may contain more local variables than this, so the output may look different.

Tools like these can really help make your development experience more pleasant and productive. Rich's colorful and nicely presented error messages are much more readable than Python's default tracebacks. You may actually look forward to getting an exception!

# Keeping Your User Engaged Through Animation

There's much more to Rich than just developer tools. Its prime purpose is to help you create an interesting and attractive interface for the user. Animations can be a powerful asset in this aim. Rich's animation tools are designed to make use of **context managers**. Here, you'll do a quick review of what context managers are and how they work. If you'd like a more in-depth discussion, you can visit Context Managers and Python's `with` Statement.

## Understanding Context Managers

A context manager is a mechanism for keeping track of resources. You can use it in any situation where a resource such as a file, a socket, or a thread must be allocated to a task and then later returned to the system. A context manager in Python is invoked with the `with` keyword, and it's active within the subsequent indented block. When the code execution leaves the block, the teardown actions are invoked automatically.

In the case of Rich animations, the resources are the timers and variables for keeping track of the changes in the display. The Rich library provides context managers to handle these resources. You, the programmer, can largely ignore the complexities and trust the context manager to do what's right.

## Displaying Dynamic Status With Animations

Rich has a `Status` class that you can use to display the status of your program. The recommended way to use it is as a context manager. Create a file named `rich_status.py` to investigate how this works:

Python dynamic_status.py

```
1  import time
2  from rich.console import Console
3
4  def do_something_important():
5      time.sleep(5.0)  # Simulates a long process
6
7  console = Console()
8  with console.status(
9      "Please wait - solving global problems...", spinner="earth"
10 ):
11     do_something_important()
12
13 console.print("All fixed! :sunglasses:")
```

The context manager scope starts at line 8. You call `console.status()` with two parameters: the message to display and the animated spinner that appears along with the message while the context block is active.

Your call to `do_something_important()` in line 11 happens within the context manager's scope, so the message and the spinner remain visible until that function returns five seconds later. Finally, in line 13, you announce success. The `status()` animation disappears, making way for a cheery announcement and an emoji:

So, the particular `spinner` parameter in this example rendered an image of the rotating planet. But how can you find out what other spinners you can use? You can see all the numerous `spinner` animations available by using another handy module-level demo:

Shell

```
(venv) $ python -m rich.spinner
```

You'll see that you have a wide choice of geometric animations, plus some more pictorial ones, like `"clock"`, `"smiley"`, and `"weather"`.

Similarly, the `:sunglasses:` syntax in line 11 shows how you can incorporate static emojis into inline text using colon-delimited names. You can insert emojis wherever Rich renders text, such as in a `log()` statement, a prompt, or a table. And of course, Rich has thousands more emojis that you can use in this way. As with the `spinner` images, you can display the full catalog of emoji names and images using yet another command-line demo:

Shell

```
(venv) $ python -m rich.emoji
```

There are far too many emojis available to fit on a single screen.

If you prefer an online reference, then you can also find spinners and emojis cataloged in the `Rich` documentation.



## Animating Activities With Progress Bars

Animated spinners are well and good if the wait is short, but for a lengthier process, you'll want to give the user some indication of how long they can expect to wait. For this case, Rich's `Progress` class has you covered. Instances of this class keep track of one or more asynchronous tasks while displaying their progress through an animated bar, a percent-completed display, and an estimated time to completion. Here's how you can code that up:

Python progress_indicator.py

```
import time
from rich.progress import Progress

with Progress() as progress:
    task1 = progress.add_task("[red]Fribbulating...[/]", total=1000)
    task2 = progress.add_task("[green]Wobbulizing...[/]", total=1000)
    task3 = progress.add_task("[cyan]Fandangling...[/]", 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(0.01)
```

The `Progress` class animates and highlights the display as your tasks progress. Just for illustrative purposes, your tasks all advance at different speeds:



In a real application, you could call the `.update()` method for a task such as a file download, with the `advance` parameter calculated from the number of bytes downloaded.

> **Note:** You can use console markup in most strings, including the progress descriptions in the example. As a shorthand, you can use `[/]` to close the last applied style.

As with most Rich classes, there are plenty of ways to customize the details of what `Progress` displays. You can check out the details in the Rich documentation

## Bringing Tables to Life

If you're working with a lot of data, then often a table is the most compact way to present it. But plain data tables can be a little dry. In this section, you'll see how you can use Rich's table-building tools along with formatting and colorization to build a table that really grabs the user's attention.

Static tables have their uses, but now you'll add some real pizzazz. You'll use Rich animation to convert your table into a simulated real-time display. A scrolling, updating table like this could be the main feature of your app, or it could be just one part of a whole data dashboard.

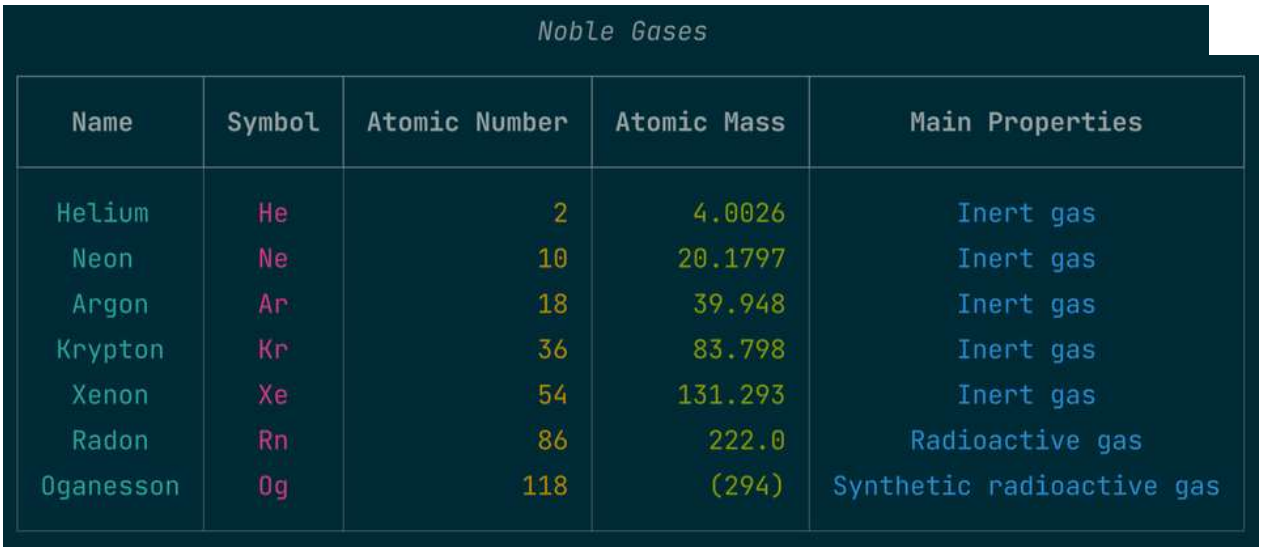
## Building a Static Table

Rich has a `Table` class that lets you build nice tabular data displays. Here's an example that shows how you can set formats and colors on a per-column basis:

Python noble_gases.py

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

console = Console()

table = Table(title="Noble Gases")
table.add_column("Name", style="cyan", justify="center")
table.add_column("Symbol", style="magenta", justify="center")
table.add_column("Atomic Number", style="yellow", justify="right")
table.add_column("Atomic Mass", style="green", justify="right")
table.add_column("Main Properties", style="blue", justify="center")

noble_gases = [
    {"name": "Helium", "symbol": "He", "atomic_number": 2,
     "atomic_mass": 4.0026, "properties": "Inert gas"},
    {"name": "Neon", "symbol": "Ne", "atomic_number": 10,
     "atomic_mass": 20.1797, "properties": "Inert gas"},
    {"name": "Argon", "symbol": "Ar", "atomic_number": 18,
     "atomic_mass": 39.948, "properties": "Inert gas"},
    {"name": "Krypton", "symbol": "Kr", "atomic_number": 36,
     "atomic_mass": 83.798, "properties": "Inert gas"},
    {"name": "Xenon", "symbol": "Xe", "atomic_number": 54,
     "atomic_mass": 131.293, "properties": "Inert gas"},
    {"name": "Radon", "symbol": "Rn", "atomic_number": 86,
     "atomic_mass": 222.0, "properties": "Radioactive gas"},
    {"name": "Oganesson", "symbol": "Og", "atomic_number": 118,
     "atomic_mass": "(294)", "properties": "Synthetic radioactive gas"},
]

for noble_gas in noble_gases:
    table.add_row(
        noble_gas["name"],
        noble_gas["symbol"],
        str(noble_gas["atomic_number"]),
        str(noble_gas["atomic_mass"]),
        noble_gas["properties"],
    )

console.print(table)
```

A table can be a very convenient way of presenting this sort of data:




The `Table` API gives you an intuitive way of building a nice-looking tabular display.

## Animating a Scrolling Display

A static table is fine for displaying static data, but what if you want to display *dynamic* data in real time? Rich has a class named `Live` that helps you do this. `Live` is a context manager that takes control of the console formatting, allowing you to update whatever fields you like without disturbing the rest of the layout. For your `Live` demo, you'll make use of some real cryptocurrency data, captured from a free crypto API.

To keep the demo focused on the display aspects, you'll use canned data to simulate real-time updates. There are one hundred entries that you'll present in a table only twenty rows deep. To do this, you'll scroll the data through the table in an endless loop, simulating the continuous arrival of new data.

In a real application, you might be getting real-time updates from a crypto API, which you'd then add to the bottom of your table while allowing the older data to scroll off the top. The overall visual effect would be very similar to your demo.



## Accessing the Crypto Data

Since the crypto data is a little bulky, you'll put it in a separate JSON-file:

crypto_data.json Show/Hide

This JSON file contains a single large data structure: a list of dictionaries, each containing the named fields of interest. The real API reports more than two thousand symbols, but these one hundred will be enough for your demo.

## Coding the Live Table

Next, you'll write the module `live_table.py` containing the code to display the dynamic table. This code will read data from `crypto_data.json`.

Your new module contains a single main function, `make_table(coin_list)`. This function uses Rich's `Table` class to generate a formatted table containing a section of your data. Then in the main module code, you'll use the `Live` object's `.update()` method to wrap a call to `make_table()` whenever your code updates the table data:

Python `live_table.py`

```
import contextlib
import json
import time
from pathlib import Path
from rich.console import Console
from rich.live import Live
from rich.table import Table

console = Console()

def make_table(coin_list):
    """Generate a Rich table from a list of coins"""
    table = Table(
        title=f"Crypto Data - {time.asctime()}",
        style="black on grey66",
        header_style="white on dark_blue",
    )
    table.add_column("Symbol")
    table.add_column("Name", width=30)
    table.add_column("Price (USD)", justify="right")
    table.add_column("Volume (24h)", justify="right", width=16)
    table.add_column("Percent Change (7d)", justify="right", width=8)
    for coin in coin_list:
        symbol, name, price, volume, pct_change = (
            coin["symbol"],
            coin["name"],
            coin["price_usd"],
            f"{coin['volume24']:.2f}",
            float(coin["percent_change_7d"]),
        )
        pct_change_str = f"{pct_change:2.1f}%"
        if pct_change > 5.0:
            pct_change_str = f"[white on dark_green]{pct_change_str:>8}[/]"
        elif pct_change < -5.0:
            pct_change_str = f"[white on red]{pct_change_str:>8}[/]"
        table.add_row(symbol, name, price, volume, pct_change_str)
    return table

# Load the coins data
raw_data = json.loads(Path("crypto_data.json").read_text(encoding="utf-8"))
num_coins = len(raw_data)
coins = raw_data + raw_data
num_lines = 20

with Live(make_table(coins[:num_lines]), screen=True) as live:
    index = 0
    with contextlib.suppress(KeyboardInterrupt):
        while True:
            live.update(make_table(coins[index : index + num_lines]))
            time.sleep(0.5)
            index = (index + 1) % num_coins
```

The `make_table()` function starting at line 11 is similar to the code that you wrote previously for the static table of noble gases. There's just one embellishment. Lines 31 to 35 provide different formatting for the `pct_change` field depending on whether it's greater than 5 percent, less than -5 percent, or between these two values.

The `num_lines` variable in line 43 determines the number of lines in the displayed table. The animation magic happens when you invoke the `Live` context manager starting at line 45.

The optional `screen=True` parameter enables a nifty feature of `Live`. The original text display is saved, and the `Live` text appears on an alternate screen. The effect of this is that your program will seamlessly restore the original display when the function returns and exits the `Live` context.

The first parameter passed to `Live` is the table created by `make_table()`. Your program will call the same function each time it updates the display. On its first call in line 45, `make_table()` receives the first `num_lines` rows of coin data. In the subsequent calls wrapped in `live.update()` in line 49, the data progressively scrolls, using the `index` value as the starting point.

To simulate streaming data, you slice your static data. In line 42, you repeat the data twice in order to avoid complicated logic at the end of your dataset. You use the modulo operator (`%`) to restart `index` at `0` when you've shown all available data in the table.

Notice that the live-update code occurs within an infinite loop. When you're tired of admiring your scrolling table, you can interrupt the code by pressing `^ Ctrl` + `C`. That interruption is caught by the `suppress()` context manager, which exits the loop and the `Live` context manager, stops the animation, and returns cleanly to your previous display.

You can invoke the table demo from the OS console:

Shell

```
(venv) $ python live_table.py
```

This will display a scrolling table of twenty lines:



If you'd like to see a different table height, then you can modify `num_lines` in the code, or even make it a parameter for your script. You can probably think of plenty of ways to improve this table and make it suit your use case. But regardless of what you do to tweak your table, you have an attractive animation that should delight your user.

## Conclusion

Congratulations on completing this whirlwind introduction to Python's Rich library! You've built an animated, highlighted table display that could be the main component in a real-time data dashboard. Along the way, you've learned about using Rich in your development tool kit, and you've styled attractive end-user displays and lively status and progress widgets.

**In this tutorial, you've learned how to:**

- Use Rich for **stylish, helpful detail** during coding, debugging and logging
- Generate entertaining and informative **animations** to keep your user engaged during **lengthy processing**
- Create a **scrolling dashboard display** that you could use to monitor a remote process

Now you're equipped to start using Rich to enhance your development experience or to create an engaging TUI for your command-line application!

Do you see a role for Rich in your next project? Have you found more ways to exploit its power to create engaging user experiences? Leave a note in the comments below to share your insights!