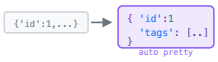
Rich

A field guide to Rich, which brings color and structure to terminal output with tables, progress bars, highlighting, and clean tracebacks.

Print, markup & styles

A drop-in print that understands [bold red]...[/] markup and emoji.

PURPOSE	COMMAND	RESULT
Pretty drop-in for print.	`from rich import print` `print({"id": 1, "tags": ["a", "b"]})`	
Inline style markup.	`print("[bold magenta]Hello` `[/bold magenta] World")`	
Emoji and shorthand close.	`print(":rocket: launch [green]ready[/]")`	

Style via the Console.	`from rich.console import Console`
	`Console().print("alert",`
	`style="bold white on red")`
		

Build styled spans by index.	`from rich.text import Text ; t = Text("OK done")` `t.stylize("bold green", 0, 2)`	
Escape literal brackets.	`from rich.markup import escape` `print(escape("[not a tag]"))`	

from rich import print is a markup-aware, container-pretty drop-in; Console().print(style=...) and Text.stylize apply styles programmatically

Tables

Add columns, add rows, print; rich measures and draws the grid.

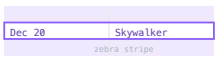
PURPOSE	COMMAND	RESULT
Make a titled table.	`table = Table(title="Movies")`	
Add styled columns.	`add_column("Released", justify="right",` `style="cyan", no_wrap=True)`	
Add a data row.	`add_row("Dec 20, 2019", "Skywalker")`	
Render to the console.	`console.print(table)`	
Quick key/value grid (no borders).	`grid = Table.grid(padding=1)` `grid.add_row("name:", "ada")`	
Per-cell color and footers.	`add_row("2021", "[red]flop[/red]")` `Table(show_footer=True)`	

Table = add_column + add_row + console.print; Table.grid() is the same engine with borders off for aligned layouts

Trees & Columns

Nest .add(...) for hierarchies; flow renderables into columns.

PURPOSE	COMMAND	RESULT
Start a tree from a root.	`tree = Tree("project")`	
Add a branch (returns it).	`branch = tree.add("src")`	
Nest deeper on the branch.	`branch.add("app.py"); branch.add(...)`	
Render the tree.	`console.print(tree)`	
Lay renderables side by side.	`console.print(Columns(panels, equal=True))`	
Style guides and labels.	`Tree("[bold]root[/bold]", guide_style="dim")`	

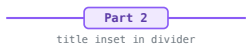
every .add(...) returns the new node, so nest by adding onto the returned branch; wrap renderables in Columns for side-by-side layout

Panels & Layout

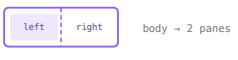
Box anything in a Panel; split the screen with Layout.

PURPOSE	COMMAND	RESULT

Box a renderable.	`from rich.panel importPanel`
	`console.print(Panel("Hello",`
	`  title="greeting"))`
		

Shrink the box to fit.	`console.print(Panel.fit("snug",` `  subtitle="v15"))`	
Center / align content.	`from rich.align importAlign` `console.print(Align.center("centered"))`	
Add a horizontal rule.	`console.rule("[bold]Part 2[/bold]")`	

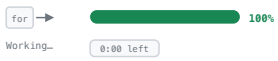
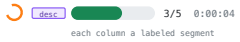
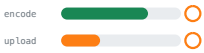
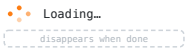
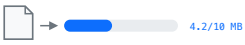
Split the screen.	`from rich.layout importLayout; layout =Layout()`
	`layout.split_column(Layout(name="top"),`
	`  Layout(name="body"))`
		

Address a named region.	`layout["body"].split_row(left, right)`	

Panel boxes any renderable with a title; Layout carves the terminal into named regions you split and address by name

Progress Bars & Spinners

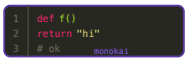
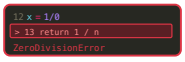
Wrap any loop with track; build custom bars with Progress.

PURPOSE	COMMAND	RESULT
Progress over any iterable.	`from rich.progress import track` `for item in track(items, description=…):`	
A custom multi-column bar.	`from rich.progress import Progress, SpinnerColumn, BarColumn, MofNCompleteColumn`	
Drive tasks manually.	`with Progress() as p: t = p.add_task(…)` `p.update(t, advance=10)`	
Multiple concurrent tasks.	`p.add_task("encode", total=50)` `p.add_task("upload", total=50)`	
A transient spinner / status.	`with console.status("Loading…",` `spinner="dots"):`	
Wrap a file read with a bar.	`from rich.progress import wrap_file` `# or progress.open(path, "rb")`	

track(iterable) is the one-liner; Progress with explicit columns and add_task / update drives bars by hand; status shows a transient spinner

Syntax & Pretty Tracebacks

Pygments-highlighted code, JSON, and gorgeous error frames.

PURPOSE	COMMAND	RESULT
Highlight a code string.	`Syntax(code, "python",` `theme="monokai", line_numbers=True)`	
Highlight a whole file.	`Syntax.from_path("app.py", ...)`	
Pretty-print any object.	`pprint({"a": [1, 2, 3], "b": {"x": 1}})`	
Highlight JSON.	`print_json('{"id": 1, "ok": true}')`	
Install pretty tracebacks.	`install(show_locals=True)`	
Render a caught exception.	`console.print_exception(show_locals=True)`	

Syntax and print_json colorize code and data via Pygments; traceback.install() makes every error a rich, locals-aware frame

Live Updating Displays

Refresh one region in place; great for dashboards and status.

PURPOSE	COMMAND	RESULT
Live-update one renderable.	`with Live(table, refresh_per_second=4) as live:`	
Swap the displayed renderable.	`live.update(new_table)`	
Mutate then auto-refresh.	`table.add_row("new")`	
Live progress + log together.	`Group(Panel(...), Progress(...))`	
Throttle redraw rate.	`Live(renderable, refresh_per_second=10, transient=True)`	
Build the frame each tick.	`live.update(make_table())`	

Live claims one region and redraws it in place; call live.update(...) or mutate the shown renderable, throttle with refresh_per_second

Console & Logging Handler

One Console object, plus a drop-in handler for stdlib logging.

PURPOSE	COMMAND	RESULT
Make a configured console.	`console = Console()`	
Print with a timestamp + caller.	`console.log("saved", {"rows": 42})`	
Name your own styles.	`Console(theme=Theme({ "ok": "bold green", "err": "bold red"}))`	
Wire into stdlib logging.	`logging.basicConfig(format="%(message)s", handlers=[RichHandler()])`	
Leveled, colored log output.	`log.info("up");.warning/.error`	
Record output to export.	`Console(record=True).export_svg("o.svg")`	

One Console is the single sink; add a RichHandler to logging.basicConfig and ordinary log calls come out colored, aligned, and timestamped