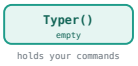
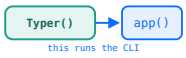
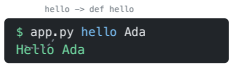
Typer

A field guide to Typer, building command-line apps from type hints where function arguments become options, help, and validation.

A command from a function

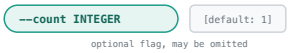
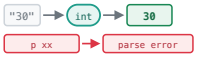
Decorate a typed function and it becomes a CLI command.

PURPOSE	COMMAND	RESULT
Create the Typer app.	`app = typer.Typer()`	
Turn a function into a command.	`@app.command()` `def hello(name: str): ...`	
Print output.	`typer.echo(f"Hello {name}")`	
Make the file runnable.	`if __name__ == "__main__":` `  app()`	
Invoke a command by name.	`python app.py hello Ada`	
One command collapses to root.	`@app.command()` (only one) `python app.py Ada`	

typer.Typer() holds your @app.command() functions; the function signature is the interface and app() runs the CLI

Arguments vs Options

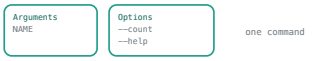
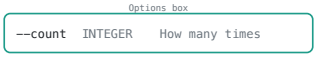
No default means a positional ARGUMENT; a default means a --OPTION; the type hint decides parsing.

PURPOSE	COMMAND	RESULT
Positional argument (required).	`name: Annotated[str, typer.Argument()]`	
Option with a default.	`count: Annotated[int, typer.Option()] = 1`	
Boolean becomes a flag pair.	`force: Annotated[bool, typer.Option()] = False`	
Short alias and option name.	`count: Annotated[int, typer.Option("--count", "-c")]`	
Type hint drives conversion.	`age: Annotated[int, typer.Argument()]`	
Legacy spelling (avoid).	`count: int = typer.Option(1)`	

No default to a required ARGUMENT, a default to an optional --OPTION; the type hint converts and validates the shell string

The auto-generated --help

Typer builds a Rich help screen from your signature, docstrings, and help strings.

PURPOSE	COMMAND	RESULT
Help comes free.	`python app.py --help`	
Per-command help.	`python app.py hello --help`	
Docstring becomes the help text.	`def hello(...):` `"""Greet NAME count times."""`	
Per-parameter help text.	`typer.Option(help="How many times")`	
App-level help and name.	`typer.Typer(help="Demo CLI")`	
Auto exit code.	`app.py --help ; echo $?`	

Typer reads names, types, defaults, docstrings, and help= strings to build a Rich --help at both app and command level, exiting 0

Multiple commands and subcommands

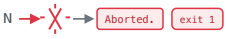
Many functions make a command set; add_typer nests one app under another for groups like git remote add.

PURPOSE	COMMAND	RESULT
Several commands, one app.	`@app.command() def create(): ...` `@app.command() def delete(): ...`	
Name a command explicitly.	`@app.command("rm") def delete(): ...`	
Build a subcommand group.	`items_app = typer.Typer()`	
Mount the group.	`app.add_typer(items_app, name="items")`	
Nested help and usage.	`python app.py items --help`	
Group help text.	`typer.Typer(help="Manage items")`	

more @app.command() functions build a command set; add_typer(child, name="items") nests a group with its own help and usage

Prompts & Confirms

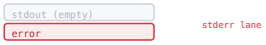
Ask for missing input interactively, hide passwords, and gate destructive actions.

PURPOSE	COMMAND	RESULT
Prompt for a missing value.	`Annotated[str, typer.Option(prompt=True)]`	
Confirm a password (twice).	`typer.Option(prompt=True, confirmation_prompt=True,hide_input=True)`	
Ask inline anywhere.	`value = typer.prompt("Age", type=int)`	
Yes/no gate.	`typer.confirm("Delete it?")`	
Abort on decline.	`if not typer.confirm("Sure?"): raisetyper.Abort()`	
Auto-abort shortcut.	`typer.confirm("Proceed?",abort=True)`	

prompt=True asks when a value is omitted; typer.confirm gates destructive actions; Abort() (or abort=True) prints Aborted. and exits nonzero

Rich output and colors

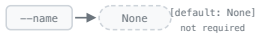
Color text with secho/style, or print Rich tables and markup since Typer ships Rich.

PURPOSE	COMMAND	RESULT
Plain output.	`typer.echo("done")`	
Colored output in one call.	`typer.secho("saved", fg=typer.colors.GREEN , bold=True)`	
Style a string, print later.	`msg = typer.style("warn", fg=YELLOW)`	
Print to stderr.	`typer.echo("error", err=True)`	
Rich table (ships with Typer).	`from rich.console import Console Console().print(table)`	
Rich markup in help.	`typer.Typer(rich_segment_mode="rich")`	

echo prints plainly, secho/style add color, err=True routes to stderr, and Rich (a Typer dependency) draws tables and markup

Validation & Defaults

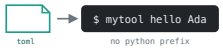
Constrain inputs with type hints, ranges, choices, and callbacks; set defaults and env fallbacks.

PURPOSE	COMMAND	RESULT
Choices from an Enum.	`class Color(str, Enum): red; blue` `color: Annotated[Color, typer.Option()]`	
Numeric range bounds.	`count: Annotated[int, typer.Option(` `min=1, max=10)]`	
Custom validator callback.	`typer.Argument(callback=validate_name)` `raise typer.BadParameter("...")`	
Path that must exist.	`path: Annotated[Path, typer.Argument(` `exists=True)]`	
Default from the environment.	`typer.Option(envvar="APP_TOKEN")`	
Optional value (can be None).	`name: Annotated[Optional[str], typer.Option(` `)] = None`	

type hints, Enum choices, min/max, callback, exists=True, and envvar constrain input; Optional[T] = None makes a flag truly optional

Run the app

Wire an entry point, run a single-function script, install completion, and test it.

PURPOSE	COMMAND	RESULT
Standard entry point.	`if __name__ == "__main__": app()`	
One-off single command.	`typer.run(main)`	
Install as a console script.	`# pyproject.toml[project.scripts]` `mytool= "pkg.cli:app"`	
Show help when run with no args.	`typer.Typer(no_args_is_help=True)`	
Control the exit code.	`raise typer.Exit(code=2)`	
Test commands in-process.	`CliRunner().invoke(app, ["hello", "Ada"])`	

app() under __main__ for the usual case, typer.run(main) for one function, [project.scripts] to install, CliRunner to test