# cyclopts

*Release 4.5.0*

**Brian Pugh**

**Jan 16, 2026**

# USAGE


1 Why Cyclopts? 3

2 Installation 5

3 Quick Start 7

4 Compared to Typer 9

5 Installation 13

6 Getting Started 15

7 Commands 21

8 Parameters 27

9 Default Parameter 37

10 Groups 41

11 Parameter Validators 47

12 Group Validators 49

13 Help 53

14 Version 61

15 Shell Completion 63

16 Coercion Rules 67

17 Text Editor 85

18 API 87

19 CLI Reference 163

20 Known Issues 165

21 Lazy Loading 167

22 Help Customization 171

23 User Classes 181

24 Args & Kwargs 187

25 Config Files 189

26 Sphinx Integration 195

27 MkDocs Integration 203

28 Packaging 213

29 App Calling & Return Values 215

30 Meta App 219

31 Command Chaining 223

32 AutoRegistry 225

33 App Upgrade 227

34 Dataclass Commands 229

35 Interactive Shell & Help 231

36 Rich Formatted Exceptions 233

37 Sharing Parameters 235

38 Unit Testing 239

39 Reading/Writing From File or Stdin/Stdout 245

40 Migrating From Typer 249

41 Typer Comparison 251

42 Fire Comparison 267

43 Arguably Comparison 269

Index 273

**Documentation:** https://cyclopts.readthedocs.io

**Source Code:** https://github.com/BrianPugh/cyclopts

Cyclopts is a modern, easy-to-use command-line interface (CLI) framework that aims to provide an intuitive & efficient developer experience.

CHAPTER

ONE

# WHY CYCLOPTS?

- **Intuitive API**: Quickly write CLI applications using a terse, intuitive syntax.
- **Advanced Type Hinting**: Full support of all builtin types and even user-specified (yes, including Pydantic, Dataclasses, and Attrs).
- **Rich Help Generation**: Automatically generates beautiful help pages from **docstrings** and other contextual data.
- **Extendable**: Easily customize converters, validators, token parsing, and application launching.

CHAPTER

TWO

# INSTALLATION

Cyclopts requires Python >=3.10; to install Cyclopts, run:

```
pip install cyclopts
```

CHAPTER

# THREE

# QUICK START

- Import `cyclopts.run()` and give it a function to run.

```python
from cyclopts import run

def foo(loops: int):
    for i in range(loops):
        print(f"Looping! {i}")

run(foo)
```

Execute the script from the command line:

```
$ python start.py 3
Looping! 0
Looping! 1
Looping! 2
```

When you need more control:

- Create an application using `cyclopts.App`.
- Register commands with the `command` decorator.
- Register a default function with the `default` decorator.

```python
from cyclopts import App

app = App()

@app.command
def foo(loops: int):
    for i in range(loops):
        print(f"Looping! {i}")

@app.default
def default_action():
    print("Hello world! This runs when no command is specified.")

app()
```

Execute the script from the command line:

```
$ python demo.py
Hello world! This runs when no command is specified.

$ python demo.py foo 3
Looping! 0
Looping! 1
Looping! 2
```

With just a few additional lines of code, we have a full-featured CLI app. See the docs for more advanced usage.

CHAPTER

FOUR

# COMPARED TO TYPER

Cyclopts is what you thought Typer was. Cyclopts's includes information from docstrings, support more complex types (even Unions and Literals!), and include proper validation support. See the documentation for a complete Typer comparison.

Consider the following short 29-line Cyclopts application:

```python
import cyclopts
from typing import Literal

app = cyclopts.App()

@app.command
def deploy(
    env: Literal["dev", "staging", "prod"],
    replicas: int | Literal["default", "performance"] = "default",
):
    """Deploy code to an environment.

    Parameters
    ----------
    env
        Environment to deploy to.
    replicas
        Number of workers to spin up.
    """
    if replicas == "default":
        replicas = 10
    elif replicas == "performance":
        replicas = 20

    print(f"Deploying to {env} with {replicas} replicas.")


if __name__ == "__main__":
    app()
```

```
$ my-script deploy --help
Usage: my-script.py deploy [ARGS] [OPTIONS]

Deploy code to an environment.
```

(continues on next page)

(continued from previous page)

```
╭─ Parameters ─────────────────────────────────────────────────────────────────────────────────────╮
│ *  ENV --env            Environment to deploy to. [choices: dev, staging, prod] [required]   │
│    REPLICAS --replicas  Number of workers to spin up. [choices: default, performance] [default:  │
│                         default]                                                             │
╰──────────────────────────────────────────────────────────────────────────────────────────────╯

$ my-script deploy staging
Deploying to staging with 10 replicas.

$ my-script deploy staging 7
Deploying to staging with 7 replicas.

$ my-script deploy staging performance
Deploying to staging with 20 replicas.

$ my-script deploy nonexistent-env
╭─ Error ──────────────────────────────────────────────────────────────────────────────────────────╮
│ Error converting value "nonexistent-env" to typing.Literal['dev', 'staging', 'prod'] for "--env".  │
╰──────────────────────────────────────────────────────────────────────────────────────────────╯

$ my-script --version
0.0.0
```

In its current state, this application would be impossible to implement in Typer. However, lets see how close we can get with Typer (47-lines):

```python
import typer
from typing import Annotated, Literal
from enum import Enum

app = typer.Typer()

class Environment(str, Enum):
    dev = "dev"
    staging = "staging"
    prod = "prod"

def replica_parser(value: str):
    if value == "default":
        return 10
    elif value == "performance":
        return 20
    else:
        return int(value)

def _version_callback(value: bool):
```

(continues on next page)

(continued from previous page)

```
    if value:
        print("0.0.0")
        raise typer.Exit()

@app.callback()
def callback(
    version: Annotated[
        bool | None, typer.Option("--version", callback=_version_callback)
    ] = None,
):
    pass

@app.command(help="Deploy code to an environment.")
def deploy(
    env: Annotated[Environment, typer.Argument(help="Environment to deploy to.")],
    replicas: Annotated[
        int,
        typer.Argument(
            parser=replica_parser,
            help="Number of workers to spin up.",
        ),
    ] = replica_parser("default"),
):
    print(f"Deploying to {env.name} with {replicas} replicas.")

if __name__ == "__main__":
    app()
```

```
$ my-script deploy --help

Usage: my-script deploy [OPTIONS] ENV:{dev|staging|prod} [REPLICAS]

 Deploy code to an environment.

╭─ Arguments ──────────────────────────────────────────────────────────────────╮
│ *    env           ENV:{dev|staging|prod}  Environment to deploy to. [default: None] │
[required] │
│      replicas      [REPLICAS]              Number of workers to spin up. [default: 10] │
│
╰──────────────────────────────────────────────────────────────────────────────╯
╭─ Options ────────────────────────────────────────────────────────────────────╮
│ --help          Show this message and exit.                                  │
│
╰──────────────────────────────────────────────────────────────────────────────╯

$ my-script deploy staging
Deploying to staging with 10 replicas.

$ my-script deploy staging 7
Deploying to staging with 7 replicas.
```

(continues on next page)

(continued from previous page)

```
$ my-script deploy staging performance
Deploying to staging with 20 replicas.

$ my-script deploy nonexistent-env
Usage: my-script.py deploy [OPTIONS] ENV:{dev|staging|prod} [REPLICAS]
Try 'my-script.py deploy --help' for help.
╭─ Error ──────────────────────────────────────────────────────────────────────╮
│ Invalid value for '[REPLICAS]': nonexistent-env                              │
╰──────────────────────────────────────────────────────────────────────────────╯

$ my-script --version
0.0.0
```

The Typer implementation is 47 lines long, while the Cyclopts implementation is just 29 (38% shorter!). Not only is the Cyclopts implementation significantly shorter, but the code is easier to read. Since Typer does not support Unions, the choices for `replica` could not be displayed on the help page. Cyclopts is much more terse, much more readable, and much more intuitive to use.

For extensive documentation on all the features Cyclopts has to offer, checkout the *API* page.

CHAPTER

FIVE

# INSTALLATION

Cyclopts requires Python >=3.10 and can be installed from PyPI via:

```
python -m pip install cyclopts
```

To install directly from github, you can run:

```
python -m pip install git+https://github.com/BrianPugh/cyclopts.git
```

For Cyclopts development, its recommended to use uv:

```
git clone https://github.com/BrianPugh/cyclopts.git
cd cyclopts
uv sync --all-extras
```

CHAPTER

SIX

# GETTING STARTED

Cyclopts relies heavily on function parameter type hints. If you are new to type hints or need a refresher, checkout the mypy cheatsheet.

## 6.1 A Basic Cyclopts Application

The most basic Cyclopts application is as follows:

```python
from cyclopts import App

app = App()

@app.default
def main():
    print("Hello World!")

if __name__ == "__main__":
    app()
```

Save this as `main.py` and execute it to see:

```
$ python main.py
Hello World!
```

The *App* class offers various configuration options that we'll explore in more detail later. The `app` object has a decorator method, *default*, which registers a function as the **default action**. In this example, the `main` function is our default action, and is executed when no CLI command is provided.

## 6.2 Function Arguments

Let's add some arguments to make this program a little more interesting.

```python
from cyclopts import App

app = App()

@app.default
def main(name):
    print(f"Hello {name}!")
```

(continues on next page)

(continued from previous page)

```
if __name__ == "__main__":
    app()
```

Executing the script with the argument `Alice` produces the following:

```
$ python main.py Alice
Hello Alice!
```

Code explanation:

1. The function `main()` was registered to `app` as the **default** action.
2. Calling `app()` at the bottom triggers the app to begin parsing CLI inputs.
3. Cyclopts identifies `"Alice"` as a positional argument and matches it to the parameter `name`. In the absence of an explicit type hint, Cyclopts defaults to parsing the value as a `str`.

   > **Note**
   >
   > Without a type annotation, Cyclopts will actually first attempt to use the **type** of the parameter's **default value**. If the parameter doesn't have a default value, it will then fallback to `str`. See *Coercion Rules*.

4. Cyclopts calls the registered **default** function `main("Alice")`, and the greeting is printed.

## 6.3 Multiple Arguments

Extending the example, lets add more arguments and type hints:

```
from cyclopts import App

app = App()

@app.default
def main(name: str, count: int, formal: bool = False):
    for _ in range(count):
        if formal:
            print(f"Hello {name}!")
        else:
            print(f"Hey {name}!")

if __name__ == "__main__":
    app()
```

```
$ python main.py Alice 3
Hey Alice!
Hey Alice!
Hey Alice!

$ python main.py Alice 3 --formal
Hello Alice!
Hello Alice!
Hello Alice!
```

The command line input "3" is converted to an integer because the parameter `count` has the type hint `int`. Boolean parameters (e.g., `--formal` in this example) are interpreted as flags. Cyclopts natively handles all python builtin types (*and more!*). Cyclopts adheres to Python's argument binding rules, allowing for both positional and keyword arguments. All of the following CLI invocations are equivalent:

```
$ python main.py Alice 3                    # Supplying arguments positionally.
$ python main.py --name Alice --count 3     # Supplying arguments via keywords.
$ python main.py --name=Alice --count=3     # Using = for matching keywords to values is␣
↪allowed.
$ python main.py --count 3 --name=Alice     # Keyword order does not matter.
$ python main.py Alice --count 3            # Positional followed by keyword
$ python main.py --count 3 Alice            # Keywords can come before positional if the␣
↪keyword is later in the function signature.
$ python main.py --count 3 -- Alice         # Using the POSIX convention to indicate the␣
↪end of keywords
```

Like calling functions in python, positional arguments cannot be specified after a **prior** argument in the function signature was specified via keyword. For example, you cannot supply the count value "3" positionally while the value for `name` is specified via keyword:

```
# The following are NOT allowed.
$ python main.py --name=Alice 3  # invalid python: main(name="Alice", 3)
$ python main.py 3 --name=Alice  # invalid python: main(3, name="Alice")
```

## 6.4 Adding a Help Page

All CLI apps need to have a help page explaining how to use the application. By default, Cyclopts adds the `--help` (and the shortform `-h`) commands to your CLI. We can add application-level help documentation when creating our app:

```python
from cyclopts import App

app = App(help="Help string for this demo application.")

@app.default
def main(name: str, count: int):
    for _ in range(count):
        print(f"Hello {name}!")

if __name__ == "__main__":
    app()
```

```
$ python main.py --help
Usage: main COMMAND [ARGS] [OPTIONS]

Help string for this demo application.

╭─ Commands ─────────────────────────────────────────────────────────────────╮
│ --help -h  Display this message and exit.                                  │
│ --version  Display application version.                                    │
╰────────────────────────────────────────────────────────────────────────────╯
╭─ Parameters ───────────────────────────────────────────────────────────────╮
```

(continues on next page)

(continued from previous page)

```
│ *  NAME --name    [required]                                                  │
│ *  COUNT --count  [required]                                                  │
──────────────────────────────────────────────────────────────────────────────
```

> **Note**
>
> Help flags can be changed with *help_flags*.

Let's add some help documentation for our parameters. Cyclopts uses the function's docstring and can interpret ReST, Google, Numpydoc-style and Epydoc docstrings (shoutout to docstring_parser).

```python
from cyclopts import App

app = App()

@app.default
def main(name: str, count: int):
    """Help string for this demo application.

    Parameters
    ----------
    name: str
        Name of the user to be greeted.
    count: int
        Number of times to greet.
    """
    for _ in range(count):
        print(f"Hello {name}!")

if __name__ == "__main__":
    app()
```

```
$ python main.py --help
Usage: main COMMAND [ARGS] [OPTIONS]

Help string for this demo application.

─ Commands ─────────────────────────────────────────────────────────────────
│ --help -h  Display this message and exit.                                 │
│ --version  Display application version.                                   │
──────────────────────────────────────────────────────────────────────────────
─ Parameters ───────────────────────────────────────────────────────────────
│ *  NAME --name    Name of the user to be greeted. [required]              │
│ *  COUNT --count  Number of times to greet. [required]                    │
──────────────────────────────────────────────────────────────────────────────
```

> **Note**
>
> If *App.help* is not explicitly set, Cyclopts will fallback to the first line (short description) of the registered `@app.default` function's docstring.

## 6.5 Run

An alternative, terser API is available for simple applications with a single command. The *run()* function takes in a single callable (usually a function) and runs it as a Cyclopts application.

```
import cyclopts

def main(name: str, count: int):
    for _ in range(count):
        print(f"Hello {name}!")

if __name__ == "__main__":
    cyclopts.run(main)
```

The *run()* function is intentionally simple. If greater control is required, then use the conventional *App* interface.

CHAPTER

# SEVEN

# COMMANDS

There are two different ways of registering functions:

1. *app.default* - Registers an action for when no registered command is provided. This was previously demonstrated in *Getting Started*.

   A sub-app **cannot** be registered with *app.default*. If no `default` command is registered, Cyclopts will display the help-page.

2. *app.command* - Registers a function or *App* as a command.

This section will detail how to use the *@app.command* decorator.

## 7.1 Registering a Command

The *@app.command* decorator adds a **command** to a Cyclopts application.

```python
from cyclopts import App

app = App()

@app.command
def fizz(n: int):
    print(f"FIZZ: {n}")

@app.command
def buzz(n: int):
    print(f"BUZZ: {n}")

app()
```

We can now control which command runs from the CLI:

```
$ my-script fizz 3
FIZZ: 3

$ my-script buzz 4
BUZZ: 4

$ my-script fuzz
╭─ Error ──────────────────────────────────────────────────────────────────────╮
│ Unknown command "fuzz". Did you mean "fizz"?                                 │
╰──────────────────────────────────────────────────────────────────────────────╯
```

## 7.2 Registering a SubCommand

The *app.command* method can also register another Cyclopts *App* as a command.

```python
from cyclopts import App

app = App()
sub_app = App(name="foo")  # "foo" would be a better variable name than "sub_app".
# "sub_app" in this example emphasizes the name comes from name="foo".
app.command(sub_app)  # Registers sub_app to command "foo"
# Or, as a one-liner:  sub_app = app.command(App(name="foo"))


@sub_app.command
def bar(n: int):
    print(f"BAR: {n}")


# Alternatively, access subapps from app like a dictionary.
@app["foo"].command
def baz(n: int):
    print(f"BAZ: {n}")


app()
```

```
$ my-script foo bar 3
BAR: 3

$ my-script foo baz 4
BAZ: 4
```

The subcommand may have their own registered `default` action. Cyclopts's command structure is fully recursive.

## 7.3 Flattening SubCommands

Sometimes you want to make all commands from a sub-app directly accessible from the parent app, without requiring users to type the intermediate subcommand name.

You can flatten a sub-app by registering it with the special `name="*"`:

```python
from cyclopts import App

app = App()
tools_app = App(name="tools")

@tools_app.command
def compress(file: str):
    print(f"Compressing {file}")

@tools_app.command
def extract(file: str):
    print(f"Extracting {file}")
```

(continues on next page)

(continued from previous page)

```
# Flatten: make all tools_app commands directly accessible
app.command(tools_app, name="*")

app()
```

```
$ my-script compress data.txt
Compressing data.txt

$ my-script extract archive.zip
Extracting archive.zip
```

Caveats of flattening:

- Parent app commands take precedence over flattened commands if there are name collisions.
- Multiple sub-apps can be flattened into the same parent app.
- You cannot supply additional configuration kwargs when using `name="*"`.
- Only *App* instances can be flattened (not functions or import paths).

Flattening is useful for organizing related commands into logical groups in your code while keeping the CLI interface simple and flat.

## 7.4 SubCommand Configuration

Subcommands inherit configuration from their parent apps.

```
from cyclopts import App

# Root app with specific error handling
root_app = App(
    exit_on_error=False,
    print_error=False,
)

# Child app inherits parent's settings
child_app = root_app.command(App(name="child"))

@child_app.default
def child_action():
    return "Child executed successfully"

# Child can override parent settings if needed
grandchild_app = child_app.command(App(name="grandchild", exit_on_error=True))
```

When `parent_app("child ...")` is called, the child command will use the parent's error handling settings unless explicitly overridden.

## 7.5 Changing Command Name

By default, commands are registered to the python function's name with underscores replaced with hyphens. Any leading or trailing underscores will be stripped. For example, the function `_foo_bar()` will become the command `foo-bar`. This renaming is done because CLI programs generally tend to use hyphens instead of underscores. The name transform can be configured by *App.name_transform*. For example, to make CLI command names be identical to their python function name counterparts, we can configure *App* as follows:

```python
from cyclopts import App

app = App(name_transform=lambda s: s)

@app.command
def foo_bar():  # will now be "foo_bar" instead of "foo-bar"
    print("running function foo_bar")

app()
```

```
$ my-script foo_bar
running function foo_bar
```

Alternatively, the name can be **manually** changed in the *@app.command* decorator. Manually set names are **not** subject to *App.name_transform*.

```python
from cyclopts import App

app = App()

@app.command(name="bar")
def foo():  # function name will NOT be used.
    print("Hello World!")

app()
```

```
$ my-script bar
Hello World!
```

Finally, if you would like to register an **additional** name to the Cyclopts-derived names, you can set an *alias*:

```python
from cyclopts import App

app = App()

@app.command(alias="bar")
def foo():  # both "foo" and "bar" will trigger this function.
    print("Running foo.")

app()
```

```
$ my-script foo
Running bar.

$ my-script bar
```

(continues on next page)

(continued from previous page)

```
Running bar.
```

## 7.6 Adding Help

There are a few ways to add a help string to a command:

1. If the function has a docstring, the **short description** will be used as the help string for the command. This is generally the preferred method of providing help strings.
2. If the registered command is a sub app, the sub app's *help* field will be used.

```python
sub_app = App(name="foo", help="Help text for foo.")
app.command(sub_app)
```

3. The *help* field of *@app.command*. If provided, the docstring or subapp help field will **not** be used.

```python
from cyclopts import App

app = App()

@app.command
def foo():
    """Help string for foo."""
    pass

@app.command(help="Help string for bar.")
def bar():
    """This got overridden."""

app()
```

```
$ my-script --help
─ Commands ──────────────────────────────────────────────────────────────────
│ bar         Help string for bar.                                          │
│ foo         Help string for foo.                                          │
│ --help,-h   Display this message and exit.                                │
│ --version   Display application version.                                  │
─────────────────────────────────────────────────────────────────────────────
```

## 7.7 Async

Cyclopts also works with **async** commands; when an async command is encountered, an event loop will be automatically created using the specified `backend` parameter (default `asyncio`).

```python
import asyncio
from cyclopts import App

app = App()

@app.command
async def foo():
```

(continues on next page)

(continued from previous page)

```
    await asyncio.sleep(10)

app()
```

When calling from within an existing async context, `await` the async method *run_async()*:

```
async def main():
    result = await app.run_async(["foo"])
    # Instead of: app(["foo"]) which would raise RuntimeError
```

## 7.8 Decorated Function Details

Cyclopts **does not modify the decorated function in any way**. The returned function is the **exact same function** being decorated and can be used exactly as if it were not decorated by Cyclopts.

## 7.9 See Also

For improved CLI startup performance with large applications, see *Lazy Loading*.

CHAPTER

# EIGHT

# PARAMETERS

Typically, Cyclopts gets all the information it needs from object names, type hints, and the function docstring:

```python
from cyclopts import App

app = App(help="This is help for the root application.")

@app.command
def foo(value: int):  # Cyclopts uses the ``value`` name and ``int`` type hint
    """Cyclopts uses this short description for help.

    Parameters
    ----------
    value: int
        Cyclopts uses this description for ``value``'s help.
    """

app()
```

Running the example:

```
$ my-script --help
Usage: my-script COMMAND

This is help for the root application.

╭─ Commands ─────────────────────────────────────────────────────────────────╮
│ foo         Cyclopts uses this short description for help.                 │
│ --help,-h   Display this message and exit.                                 │
│ --version   Display application version.                                   │
╰────────────────────────────────────────────────────────────────────────────╯

$ my-script foo --help
Usage: my-script [ARGS] [OPTIONS]

Cyclopts uses this short description for help.

╭─ Parameters ─────────────────────────────────────────────────────────────────────────╮
│ *  VALUE --value  Cyclopts uses this description for value's help. [required]        │
╰──────────────────────────────────────────────────────────────────────────────────────╯
```

This keeps the code as clean and terse as possible. However, if more control is required, we can provide additional

information by annotating type hints with *Parameter*.

```
from cyclopts import App, Parameter
from typing import Annotated

app = App()

@app.command
def foo(bar: Annotated[int, Parameter(...)]):
    pass

app()
```

*Parameter* gives complete control on how Cyclopts processes the annotated parameter. See the *API* page for all configurable options. This page will investigate some of the more common use-cases.

**Note**

*Parameter* can also be used as a decorator. This is *particularly useful for class definitions*.

## 8.1 Naming

Like *command names*, CLI parameter names are derived from their python counterparts. However, sometimes customization is needed.

### 8.1.1 Manual Naming

Parameter names (and their short forms) can be manually specified:

```
from cyclopts import App, Parameter
from typing import Annotated

app = App()

@app.default
def main(
    *,
    foo: Annotated[str, Parameter(name=["--foo", "-f"])],  # Adding a short-form
    # Equivalently, you could have done Parameter(alias="-f")
    bar: Annotated[str, Parameter(name="--something-else")],
):
    pass

app()
```

```
$ my-script --help

Usage: main COMMAND [OPTIONS]
╭─ Commands ─────────────────────────────────────────────────╮
│ --help -h  Display this message and exit.                  │
│ --version  Display application version.                    │
╰────────────────────────────────────────────────────────────╯
```

(continues on next page)

(continued from previous page)

```
╭─ Parameters ─────────────────────────────────────────────────╮
│ *  --foo              -f  [required]                         │
│ *  --something-else       [required]                         │
╰──────────────────────────────────────────────────────────────╯
```

Manually set names via *Parameter.name* are not subject to *Parameter.name_transform*. Alternatively, additional names can be added to the Cyclopts-derived names (instead of completely overriding them) with *Parameter.alias*.

**Note**

Docstrings should always use the **Python variable name** from the function signature.

```
@app.default
def main(internal_name: Annotated[str, Parameter(name="external-name")]):
    """Command description.

    Parameters
    ----------
    internal_name:            # Use the Python variable name
        Help text here.
    """
```

This follows standard Python documentation conventions; the parameter will still appear as `--external-name` on the CLI.

### 8.1.2 Name Transform

The name transform function that converts the python variable name to it's CLI counterpart can be configured by setting *Parameter.name_transform* (defaults to *default_name_transform()*).

```
from cyclopts import App, Parameter
from typing import Annotated

app = App()

def name_transform(s: str) -> str:
    return s.upper()

@app.default
def main(
    *,
    foo: Annotated[str, Parameter(name_transform=name_transform)],
    bar: Annotated[str, Parameter(name_transform=name_transform)],
):
    pass

app()
```

```
$ my-script --help
Usage: main COMMAND [OPTIONS]

╭─ Commands ─────────────────────────────────────────────────╮
```

(continues on next page)

(continued from previous page)

```
│ --help -h  Display this message and exit.                    │
│ --version  Display application version.                      │
──────────────────────────────────────────────────────────────
─ Parameters ─────────────────────────────────────────────────
│ *  --FOO  [required]                                         │
│ *  --BAR  [required]                                         │
──────────────────────────────────────────────────────────────
```

Notice how the parameter is now `--FOO` instead of the standard `--foo`.

**Note**

The returned string is **before** the standard `--` is prepended.

Generally, it is not very useful to set the name transform on **individual** parameters; it would be easier/clearer *to manually specify the name*. However, we can change the default name transform for the **entire app** by configuring the app's *default_parameter*.

To change the *name_transform* across your entire app, add the following to your *App* configuration:

```
app = App(
    default_parameter=Parameter(name_transform=my_custom_name_transform),
)
```

## 8.2 Help

It is recommended to use docstrings for your parameter help, but if necessary, you can explicitly set a help string:

```
@app.command
def foo(value: Annotated[int, Parameter(help="THIS IS USED.")]):
    """
    Parameters
    ----------
    value: int
        This description is not used; got overridden.
    """
```

```
$ my-script foo --help
─ Parameters ─────────────────────────────────────────────────
│ *  VALUE,--value  THIS IS USED. [required]                  │
──────────────────────────────────────────────────────────────
```

## 8.3 Converters

Cyclopts has a powerful coercion engine that automatically converts CLI string tokens to the types hinted in a function signature. However, sometimes a custom *converter* is required to transform the input string tokens into the desired type.

Lets consider a case where we want the user to specify a file size, and we want to allows suffixes like *"MB"*.

```python
from cyclopts import App, Parameter, Token
from typing import Annotated, Sequence
from pathlib import Path

app = App()

mapping = {
    "kb": 1024,
    "mb": 1024 * 1024,
    "gb": 1024 * 1024 * 1024,
}

def byte_units(type_, tokens: Sequence[Token]) -> int:
    # type_ is ``int``,
    value = tokens[0].value.lower()
    try:
        return type_(value)  # If this works, it didn't have a suffix.
    except ValueError:
        pass
    number, suffix = value[:-2], value[-2:]
    return int(number) * mapping[suffix]

@app.command
def zero(file: Path, size: Annotated[int, Parameter(converter=byte_units)]):
    """Creates a file of all-zeros."""
    print(f"Writing {size} zeros to {file}.")
    file.write_bytes(bytes(size))

app()
```

```
$ my-script zero out.bin 100
Writing 100 zeros to out.bin.

$ my-script zero out.bin 1kb
Writing 1024 zeros to out.bin.

$ my-script zero out.bin 3mb
Writing 3145728 zeros to out.bin.
```

The converter function gets the annotated type, and the *Token* s parsed for this argument. Tokens are Cyclopt's way of bookkeeping user inputs; in the last command the `tokens` object would look like:

```
 # tokens is a length-1 tuple. The variable "size" only takes in 1 token:
 tuple(
   Token(
      keyword=None,  # "3mb" was provided positionally, not by keyword
      value='3mb',   # The string from the command line
      source='cli',  # The value came from the command line, as opposed to other␣
↪Cyclopts mechanisms.
      index=0,       # For the variable "size", this is the first (0th) token.
   ),
)
```

### 8.3.1 Controlling Token Count

By default, Cyclopts infers how many tokens a parameter should consume from its type hint. For example, `int` consumes 1 token, `tuple[int, int]` consumes 2, and `list[int]` consumes all remaining tokens. When using custom converters, you may need to override this inference with *Parameter.n_tokens*:

```python
from cyclopts import App, Parameter
from typing import Annotated

class Point:
    def __init__(self, x: int, y: int):
        self.x = x
        self.y = y

def parse_point(type_, tokens):
    """Parse a coordinate string like '10,20' into a Point."""
    x, y = tokens[0].value.split(",")
    return Point(int(x), int(y))

app = App()

@app.default
def main(pos: Annotated[Point, Parameter(n_tokens=1, converter=parse_point, accepts_
↪keys=False)]):
    """Without n_tokens=1, Cyclopts would expect 2 tokens based on Point's __init__␣
↪signature."""
    print(f"Position: ({pos.x}, {pos.y})")

app()
```

```
$ my-script --pos 10,20
Position: (10, 20)
```

The *Parameter.accepts_keys* parameter prevents Cyclopts from generating nested options like `--pos.x` and `--pos.y`.

Alternative to the above syntax, you can directly **decorate the converter function** itself with *Parameter* to define its behavior. This keeps all the information organized in a single location.

```python
from cyclopts import App, Parameter
from typing import Annotated

class Point:
    def __init__(self, x: int, y: int):
        self.x = x
        self.y = y

@Parameter(n_tokens=1, accepts_keys=False)
def parse_point(type_, tokens):
    """Parse a coordinate string like '10,20' into a Point."""
    x, y = tokens[0].value.split(",")
    return Point(int(x), int(y))

app = App()
```

(continues on next page)

(continued from previous page)

```python

@app.default
def main(pos: Annotated[Point, Parameter(converter=parse_point)]):
    """The converter's n_tokens and accepts_keys are automatically inherited."""
    print(f"Position: ({pos.x}, {pos.y})")

app()
```

```
$ my-script --pos 10,20
Position: (10, 20)
```

You can also decorate classes directly with the converter:

```python
@Parameter(converter=parse_point)
class Point:
    def __init__(self, x: int, y: int):
        self.x = x
        self.y = y

@app.default
def main(pos: Point):
    """No Annotated wrapper needed - converter is part of the class definition."""
    print(f"Position: ({pos.x}, {pos.y})")
```

### 8.3.2 Using Classmethods as Converters

Converter functions are often closely associated with the class they create, making classmethods a natural choice. Cyclopts supports using classmethods as converters through forward string references (for class decoration) or direct references (for function annotations):

```python
from cyclopts import App, Parameter
from typing import Annotated

# Decorate the classmethod to configure n_tokens and accepts_keys
# (decorator must go above @classmethod)
@Parameter(converter="parse")
class Point:
    def __init__(self, x: int, y: int):
        self.x = x
        self.y = y

    @Parameter(n_tokens=1, accepts_keys=False)
    @classmethod
    def parse(cls, tokens):
        """Parse a coordinate string like '10,20' into a Point.

        Note: classmethod signature is (cls, tokens), not (type_, tokens)
        """
        x, y = tokens[0].value.split(",")
        return cls(int(x), int(y))

app = App()
```

(continues on next page)

(continued from previous page)

```python

@app.default
def main(pos: Point):
    """The classmethod's n_tokens and accepts_keys are automatically inherited."""
    print(f"Position: ({pos.x}, {pos.y})")

app()
```

```
$ my-script --pos 10,20
Position: (10, 20)
```

Alternatively, you can reference the classmethod directly in function annotations:

```python
class Point:
    def __init__(self, x: int, y: int):
        self.x = x
        self.y = y

    @classmethod
    def parse(cls, tokens):
        x, y = tokens[0].value.split(",")
        return cls(int(x), int(y))

@app.default
def main(pos: Annotated[Point, Parameter(converter=Point.parse, n_tokens=1, accepts_
↪keys=False)]):
    print(f"Position: ({pos.x}, {pos.y})")
```

**Note on classmethod signatures**: Classmethods used as converters should have the signature `(cls, tokens)` rather than `(type_, tokens)`. Cyclopts automatically detects bound methods and calls them with just the `tokens` parameter, since `cls` is already bound.

## 8.4 Validating Input

Just because data is of the correct type, doesn't mean it's valid. If we had a program that accepts integer user age as an input, `-1` is an integer, but not a valid age.

```python
from cyclopts import App, Parameter
from typing import Annotated

app = App()

def validate_age(type_, value):
    if value < 0:
        raise ValueError("Negative ages not allowed.")
    if value > 150:
        raise ValueError("You are too old to be using this application.")

@app.default
def allowed_to_buy_alcohol(age: Annotated[int, Parameter(validator=validate_age)]):
    print("Under 21: prohibited." if age < 21 else "Good to go!")
```

(continues on next page)

(continued from previous page)

```
app()
```

```
$ my-script 30
Good to go!

$ my-script 10
Under 21: prohibited.

$ my-script -1
╭─ Error ──────────────────────────────────────────────────────────────────────╮
│ Invalid value "-1" for "AGE". Negative ages not allowed.                     │
╰──────────────────────────────────────────────────────────────────────────────╯

$ my-script 200
╭─ Error ──────────────────────────────────────────────────────────────────────╮
│ Invalid value "200" for "AGE". You are too old to be using this application. │
╰──────────────────────────────────────────────────────────────────────────────╯
```

Certain builtin error types (`ValueError`, `TypeError`, `AssertionError`) will be re-interpreted by Cyclopts and formatted into a prettier message for the application user.

Cyclopts has some *builtin validators* for common situations We can create a similar app as above:

```python
from cyclopts import App, Parameter, validators
from typing import Annotated

app = App()

@app.default
def allowed_to_buy_alcohol(age: Annotated[int, Parameter(validator=validators.
↪Number(gte=0, lte=150))]):
    # gte - greater than or equal to
    # lte - less than or equal to
    print("Under 21: prohibited." if age < 21 else "Good to go!")

app()
```

Taking this one step further, Cyclopts has some *builtin convenience types*. If we didn't care about the upper age bound, we could simplify the application to:

```python
from cyclopts import App
from cyclopts.types import NonNegativeInt

app = App()

@app.default
def allowed_to_buy_alcohol(age: NonNegativeInt):
    print("Under 21: prohibited." if age < 21 else "Good to go!")

app()
```

## 8.5 Parameter Resolution

Cyclopts can combine multiple *Parameter* annotations together. Say you want to define a new `int` type that uses the *byte-centric converter from above*.

We can define the type:

```
ByteSize = Annotated[int, Parameter(converter=byte_units)]
```

We can then either directly annotate a function parameter with this:

```
@app.command
def zero(size: ByteSize):
    pass
```

or even stack annotations to add additional features, like a validator:

```
def must_be_multiple_of_4096(type_, value):
    assert value % 4096 == 0, "Size must be a multiple of 4096"


@app.command
def zero(size: Annotated[ByteSize, Parameter(validator=must_be_multiple_of_4096)]):
    pass
```

Python automatically flattens out annotations, so this is interpreted as:

```
Annotated[ByteSize, Parameter(converter=byte_units), Parameter(validator=must_be_
↪multiple_of_4096)]
```

Cyclopts will search **right-to-left** for **set** parameter attributes until one is found. I.e. right-most parameter attributes have the highest priority.

```
$ my-script 1234
╭─ Error ──────────────────────────────────────────────────────────────────────╮
│ Invalid value "1234" for "SIZE". Size must be a multiple of 4096             │
╰──────────────────────────────────────────────────────────────────────────────╯
```

See *Parameter Resolution Order* for more details.

CHAPTER

NINE

# DEFAULT PARAMETER

The default values of *Parameter* for an app can be configured via *App.default_parameter*.

For example, to disable the *negative* flag feature across your entire app:

```
from cyclopts import App, Parameter

app = App(default_parameter=Parameter(negative=()))

@app.command
def foo(*, flag: bool):
    pass

app()
```

Consequently, `--no-flag` is no longer an allowed flag:

```
$ my-script foo --help
Usage: my-script foo [ARGS] [OPTIONS]

╭─ Parameters ─────────────────────────────────────────────────────────────────╮
│ *  --flag  [required]                                                        │
╰──────────────────────────────────────────────────────────────────────────────╯
```

Explicitly annotating the parameter with *negative* overrides this configuration and works as expected:

```
from cyclopts import App, Parameter
from typing import Annotated

app = App(default_parameter=Parameter(negative=()))

@app.command
def foo(*, flag: Annotated[bool, Parameter(negative="--anti-flag")]):
    pass

app()
```

```
$ my-script foo --help
Usage: my-script foo [ARGS] [OPTIONS]

╭─ Parameters ─────────────────────────────────────────────────────────────────╮
```

(continues on next page)

(continued from previous page)

```
| *  --flag --anti-flag  [required]                                    |
```

## 9.1 Resolution Order

When resolving what the *Parameter* values for an individual function parameter should be, explicitly set attributes of higher priority *Parameter* s override lower priority *Parameter* s. The resolution order is as follows:

1. **Highest Priority:** Parameter-annotated command function signature `Annotated[..., Parameter()]`.
2. *Group.default_parameter* that the **parameter** belongs to.
3. *App.default_parameter* of the **app** that registered the command.
4. *Group.default_parameter* of the **app** that the function belongs to.
5. **Lowest Priority:** (2-4) recursively of the parenting app call-chain.

Any of Parameter's fields can be set to *None* to revert back to the true-original Cyclopts default.

## 9.2 Skipping Private Parameters

The *Parameter.parse* attribute can accept a **regex pattern** to selectively skip parameters based on their name. This is useful for defining "private" parameters that are externally injected (e.g. a *Meta App*, dependency-injection framework, etc) rather than parsed from the CLI.

For example, to skip all underscore-prefixed parameters:

```python
from typing import Annotated
from cyclopts import App, Parameter

# The regex "^(?!_)" matches names that do NOT start with underscore.
app = App(default_parameter=Parameter(parse="^(?!_)"))

@app.command
def greet(name: str, *, _db: Database):
    user = _db.get_user(name)
    print(f"Hello {user.full_name}!")

@app.meta.default
def launcher(*tokens: Annotated[str, Parameter(show=False, allow_leading_hyphen=True)]):
    # Create shared resources
    db = Database("myapp.db")

    # Parse CLI and get ignored (non-parsed) parameters
    command, bound, ignored = app.parse_args(tokens)

    # Inject ignored parameters
    for name, type_ in ignored.items():
        if type_ is Database:
            bound.kwargs[name] = db

    return command(*bound.args, **bound.kwargs)
```

(continues on next page)

(continued from previous page)

```python
if __name__ == "__main__":
    app.meta()
```

```
$ my-script --help
Usage: my-script COMMAND

╭─ Commands ─────────────────────────────────────────────────────╮
│ greet                                                          │
│ --help,-h  Display this message and exit.                      │
│ --version  Display application version.                        │
╰────────────────────────────────────────────────────────────────╯

$ my-script greet --help
Usage: my-script greet [ARGS] [OPTIONS]

╭─ Parameters ───────────────────────────────────────────────────╮
│ *  NAME,--name  [required]                                     │
╰────────────────────────────────────────────────────────────────╯
```

Notice that `_db` does not appear in the help screen. Parameters that don't match the regex pattern are added to the `ignored` dictionary returned by *App.parse_args()*, making them available for meta-app injection.

Like all other `Parameter` configurations, explicitly annotating with `parse=True` overrides the app-level regex:

```python
from typing import Annotated
from cyclopts import App, Parameter

app = App(default_parameter=Parameter(parse="^(?!_)"))

@app.default
def main(name: str, *, _verbose: Annotated[bool, Parameter(parse=True)] = False):
    """_verbose IS parsed despite the underscore prefix"""
```

**Important**

Parameters that are not parsed (either via `parse=False` or a non-matching regex pattern) **must** be either:

- Keyword-only (defined after `*` in the function signature), or
- Have a default value

```python
# Valid: keyword-only parameter
def main(*, _context: dict): ...

# Valid: has default value
def main(_context: dict = None): ...

# Invalid: positional without default - raises ValueError
def main(_context: dict): ...
```

CHAPTER

# TEN

# GROUPS

Groups offer a way of organizing parameters and commands on the help-page; for example:

```
Usage: my-script.py create [OPTIONS]

╭─ Vehicle (choose one) ──────────────────────────────────────────────────────╮
│ --car    [default: False]                                                   │
│ --truck  [default: False]                                                   │
╰─────────────────────────────────────────────────────────────────────────────╯
╭─ Engine ────────────────────────────────────────────────────────────────────╮
│ --hp         [default: 200]                                                 │
│ --cylinders  [default: 6]                                                   │
╰─────────────────────────────────────────────────────────────────────────────╯
╭─ Wheels ────────────────────────────────────────────────────────────────────╮
│ --wheel-diameter  [default: 18]                                             │
│ --rims,--no-rims  [default: False]                                          │
╰─────────────────────────────────────────────────────────────────────────────╯
```

They also provide an additional abstraction layer that *validators* can operate on.

Groups can be created in two ways:

1. Explicitly creating a *Group* object.
2. Implicitly with a **string**. This will implicitly create a group, `Group(my_str_group_name)`, if it doesn't exist. If there exists a *Group* object with the same name within the command/parameter context, it will join that group.

> **Warning**
>
> While convenient and terse, mistyping a group name will unintentionally create a new group!

Every command and parameter belongs to at least one group.

Group(s) can be provided to the `group` keyword argument of *app.command* and *Parameter*. Like *Parameter*, the *Group* class itself only marks objects with metadata; the group does **not** contain direct references to it's members. This means that groups can be reused across commands.

## 10.1 Command Groups

An example of using groups to organize commands:

```python
from cyclopts import App

app = App()

# Change the group of "--help" and "--version" to the implicitly created "Admin" group.
app["--help"].group = "Admin"
app["--version"].group = "Admin"

@app.command(group="Admin")
def info():
    """Print debugging system information."""
    print("Displaying system info.")

@app.command
def download(path, url):
    """Download a file."""
    print(f"Downloading {url} to {path}.")

@app.command
def upload(path, url):
    """Upload a file."""
    print(f"Downloading {url} to {path}.")

app()
```

```
$ python my-script.py --help
Usage: my-script.py COMMAND

─ Admin ──────────────────────────────────────────────────────────────────────
│ info       Print debugging system information.                             │
│ --help,-h  Display this message and exit.                                  │
│ --version  Display application version.                                    │
──────────────────────────────────────────────────────────────────────────────
─ Commands ───────────────────────────────────────────────────────────────────
│ download  Download a file.                                                 │
│ upload    Upload a file.                                                   │
──────────────────────────────────────────────────────────────────────────────
```

The default group is defined by the registering app's *App.group_commands*, which defaults to a group named "Commands".

## 10.2 Parameter Groups

Like commands above, parameter groups allow us to organize parameters on the help page. They also allow us to add additional inter-parameter validators (e.g. mutually-exclusive parameters). An example of using groups with parameters:

```python
from cyclopts import App, Group, Parameter, validators
from typing import Annotated

app = App()
```

(continues on next page)

(continued from previous page)

```python
vehicle_type_group = Group(
    "Vehicle (choose one)",
    default_parameter=Parameter(negative=""),  # Disable "--no-" flags
    validator=validators.MutuallyExclusive(),  # Only one option is allowed to be␣
↪selected.
)

@app.command
def create(
    *,  # force all subsequent variables to be keyword-only
    # Using an explicitly created group object.
    car: Annotated[bool, Parameter(group=vehicle_type_group)] = False,
    truck: Annotated[bool, Parameter(group=vehicle_type_group)] = False,
    # Implicitly creating an "Engine" group.
    hp: Annotated[float, Parameter(group="Engine")] = 200,
    cylinders: Annotated[int, Parameter(group="Engine")] = 6,
    # You can explicitly create groups in-line.
    wheel_diameter: Annotated[float, Parameter(group=Group("Wheels"))] = 18,
    # Groups within the function signature can always be referenced with a string.
    rims: Annotated[bool, Parameter(group="Wheels")] = False,
):
    pass

app()
```

```
$ python my-script.py create --help
Usage: my-script.py create [OPTIONS]

- Engine ------------------------------------------------------------
| --hp         [default: 200]                                       |
| --cylinders  [default: 6]                                         |
--------------------------------------------------------------------
- Vehicle (choose one) ----------------------------------------------
| --car    [default: False]                                         |
| --truck  [default: False]                                         |
--------------------------------------------------------------------
- Wheels ------------------------------------------------------------
| --wheel-diameter  [default: 18]                                   |
| --rims --no-rims  [default: False]                                |
--------------------------------------------------------------------

$ python my-script.py create --car --truck
- Error -------------------------------------------------------------
| Invalid values for group "Vehicle (choose one)". Mutually         |
| exclusive arguments: {--car, --truck}                             |
--------------------------------------------------------------------
```

In this example, we use the *MutuallyExclusive* validator to make it so the user can only specify `--car` or `--truck`.

The default groups are defined by the registering app:

- *App.group_arguments* for positional-only arguments, which defaults to a group named `"Arguments"`.
- *App.group_parameters* for all other parameters, which defaults to a group named `"Parameters"`.

## 10.3 Validators

Group validators offer a way of jointly validating group parameter members of CLI-provided values. Groups with an empty name, or with `show=False`, are a way of using group validators without impacting the help-page.

```python
from cyclopts import App, Group, Parameter, validators
from typing import Annotated

app = App()

mutually_exclusive = Group(
   # This Group has no name, so it won't impact the help page.
   validator=validators.MutuallyExclusive(),
   # show_default=False - Showing "[default: False]" isn't too meaningful for mutually-exclusive options.
   # negative="" - Don't create a "--no-" flag
   default_parameter=Parameter(show_default=False, negative=""),
)

@app.command
def foo(
    car: Annotated[bool, Parameter(group=(app.group_parameters, mutually_exclusive))] = False,
    truck: Annotated[bool, Parameter(group=(app.group_parameters, mutually_exclusive))] = False,
):
    print(f"{car=} {truck=}")

app()
```

```
$ python demo.py foo --help
Usage: demo.py foo [ARGS] [OPTIONS]

─ Parameters ──────────────────────────────────────────────────────────────────
│ CAR,--car                                                                   │
│ TRUCK,--truck                                                               │
───────────────────────────────────────────────────────────────────────────────

$ python demo.py foo --car
car=True truck=False

$ python demo.py foo --truck
car=False truck=True

$ python demo.py foo --car --truck
─ Error ───────────────────────────────────────────────────────────────────────
│  Mutually exclusive arguments: {--car, --truck}                             │
───────────────────────────────────────────────────────────────────────────────
```

See *Group.validator* for details.

Cyclopts has some *builtin group-validators for common use-cases.*

## 10.4 Help Page

Groups form titled panels on the help-page.

Groups with an empty name, or with *show=False*, are **not** shown on the help-page. This is useful for applying additional grouping logic (such as applying a *LimitedChoice* validator) without impacting the help-page.

By default, the ordering of panels is **alphabetical**. However, the sorting can be manipulated by *Group.sort_key*. See it's documentation for usage.

The *Group.create_ordered()* convenience classmethod creates a *Group* with a *sort_key* value drawn drawn from a global monotonically increasing counter. This means that the order in the help-page will match the order that the groups were instantiated.

```python
from cyclopts import App, Group

app = App()

plants = Group.create_ordered("Plants")
animals = Group.create_ordered("Animals")
fungi = Group.create_ordered("Fungi")

@app.command(group=animals)
def zebra():
    pass

@app.command(group=plants)
def daisy():
    pass

@app.command(group=fungi)
def portobello():
    pass

app()
```

```
$ my-script --help

Usage: scratch.py COMMAND

─ Plants ──────────────────────────────────────────────────────────────────────
│ daisy                                                                       │
──────────────────────────────────────────────────────────────────────────────
─ Animals ─────────────────────────────────────────────────────────────────────
│ zebra                                                                       │
──────────────────────────────────────────────────────────────────────────────
─ Fungi ───────────────────────────────────────────────────────────────────────
│ portobello                                                                  │
──────────────────────────────────────────────────────────────────────────────
─ Commands ────────────────────────────────────────────────────────────────────
│ --help -h  Display this message and exit.                                   │
│ --version  Display application version.                                     │
──────────────────────────────────────────────────────────────────────────────
```

Even when using *Group.create_ordered()*, a *sort_key* can still be supplied; the global counter will only be used

to break sorting ties.

CHAPTER

# ELEVEN

# PARAMETER VALIDATORS

In CLI applications, users have the freedom to input a wide range of data. This flexibility can lead to inputs the application does not expect. By coercing the input into a data type (like an `int`), we are already limiting the input to a certain degree (e.g. "foo" cannot be coerced into an integer). To further restrict the user input, you can populate the *`validator`* field of *`Parameter`*.

A validator is any callable object (such as a function) that has the signature:

```
def validator(type_, value: Any) -> None:
    pass  # Raise any exception here if ``value`` is invalid.
```

Validation happens **after** the data converter runs. Any of `AssertionError`, `TypeError` or `ValidationError` will be promoted to a *`cyclopts.ValidationError`* so that the exception gets presented to the end-user in a nicer way. More than one validator can be supplied as a list to the *`validator`* field.

Cyclopts has some builtin common validators in the *cyclopts.validators* module. See *Types* for common specific definitions provided as convenient pre-annotated types.

## 11.1 Path

The *`Path`* validator ensures certain properties of the parsed `pathlib.Path` object, such as asserting the file must exist.

```
from cyclopts import App, Parameter, validators
from typing import Annotated
from pathlib import Path

app = App()

@app.default()
def foo(path: Annotated[Path, Parameter(validator=validators.Path(exists=True))]):
    print(f"File contents:\n{path.read_text()}")

app()
```

```
$ echo Hello World > my_file.txt

$ my-script my_file.txt
File contents:
Hello World

$ my-script this_file_does_not_exist.txt
╭─ Error ──────────────────────────────────────────────────────────────────────╮
```

(continues on next page)

(continued from previous page)

```
╭─ Error ──────────────────────────────────────────────────────────────────────╮
│ Invalid value "this_file_does_not_exist.txt" for "PATH".                     │
│ "this_file_does_not_exist.txt" does not exist.                               │
╰──────────────────────────────────────────────────────────────────────────────╯
```

See *Annotated Path Types* for Annotated-Type equivalents of common Path converter/validators.

## 11.2 Number

The *Number* validator can set minimum and maximum input values.

```python
from cyclopts import App, Parameter, validators
from typing import Annotated

app = App()

@app.default()
def foo(n: Annotated[int, Parameter(validator=validators.Number(gte=0, lt=16))]):
    print(f"Your number in hex is {str(hex(n))[2]}.")

app()
```

```
$ my-script 0
Your number in hex is 0.

$ my-script 15
Your number in hex is f.

$ my-script 16
╭─ Error ──────────────────────────────────────────────────────────────────────╮
│ Invalid value "16" for "N". Must be < 16.                                    │
╰──────────────────────────────────────────────────────────────────────────────╯
```

See *Annotated Number Types* for Annotated-Type equivalents of common Number converter/validators.

CHAPTER

# TWELVE

# GROUP VALIDATORS

Group validators operate on a set of parameters, *ensuring that their values are mutually compatible*. Validator(s) for a group can be set via the *Group.validator* attribute. An individual validator is a callable object/function with signature:

```
def validator(argument_collection: ArgumentCollection):
    "Raise an exception if something is invalid."
```

Cyclopts has some builtin common group validators in the *cyclopts.validators* module.

## 12.1 LimitedChoice

Limits the number of specified arguments within the group. Most commonly used for mutually-exclusive arguments (default behavior).

```
from cyclopts import App, Group, Parameter, validators
from typing import Annotated

app = App()

vehicle = Group(
    "Vehicle (choose one)",
    default_parameter=Parameter(negative=""),  # Disable "--no-" flags
    validator=validators.LimitedChoice(),  # Mutually Exclusive Options
)

@app.default
def main(
    *,
    car: Annotated[bool, Parameter(group=vehicle)] = False,
    truck: Annotated[bool, Parameter(group=vehicle)] = False,
):
    if car:
        print("I'm driving a car.")
    if truck:
        print("I'm driving a truck.")

app()
```

```
$ python drive.py --help
Usage: main COMMAND [OPTIONS]
```

(continues on next page)

(continued from previous page)

```
╭─ Commands ─────────────────────────────────────────────────────────────────╮
│ --help -h  Display this message and exit.                                  │
│ --version  Display application version.                                    │
╰────────────────────────────────────────────────────────────────────────────╯
╭─ Vehicle (choose one) ─────────────────────────────────────────────────────╮
│ --car    [default: False]                                                  │
│ --truck  [default: False]                                                  │
╰────────────────────────────────────────────────────────────────────────────╯

$ python drive.py --car
I'm driving a car.

$ python drive.py --car --truck
╭─ Error ────────────────────────────────────────────────────────────────────╮
│ Invalid values for group "Vehicle (choose one)". Mutually                  │
│ exclusive arguments: {--car, --truck}                                      │
╰────────────────────────────────────────────────────────────────────────────╯
```

See the *LimitedChoice* docs for more info.

## 12.2 MutuallyExclusive

Alias for *LimitedChoice* with default arguments. Exists primarily because the usage/implication will be more directly obvious and searchable to developers than *LimitedChoice*. Since this class takes no arguments, an already instantiated version *mutually_exclusive* is also provided for convenience.

## 12.3 all_or_none

Group validator that enforces that either **all** parameters in the group must be supplied an argument, or **none** of them.

```python
from typing import Annotated

from cyclopts import App, Group, Parameter
from cyclopts.validators import all_or_none

app = App()

group_1 = Group(validator=all_or_none)
group_2 = Group(validator=all_or_none)


@app.default
def default(
    foo: Annotated[bool, Parameter(group=group_1)] = False,
    bar: Annotated[bool, Parameter(group=group_1)] = False,
    fizz: Annotated[bool, Parameter(group=group_2)] = False,
    buzz: Annotated[bool, Parameter(group=group_2)] = False,
):
    print(f"{foo=} {bar=}")
    print(f"{fizz=} {buzz=}")
```

(continues on next page)

(continued from previous page)

```
if __name__ == "__main__":
    app()
```

```
$ python all_or_none.py
foo=False bar=False
fizz=False buzz=False

$ python all_or_none.py --foo
╭─ Error ──────────────────────────────────────────────────────────────╮
│ Missing argument: --bar                                              │
╰──────────────────────────────────────────────────────────────────────╯

$ python all_or_none.py --foo --bar
foo=True bar=True
fizz=False buzz=False

$ python all_or_none.py --foo --bar --fizz
╭─ Error ──────────────────────────────────────────────────────────────╮
│ Missing argument: --buzz                                             │
╰──────────────────────────────────────────────────────────────────────╯

$ python all_or_none.py --foo --bar --fizz --buzz
foo=True bar=True
fizz=True buzz=True
```

See the *all_or_none* docs for more info.

CHAPTER

# THIRTEEN

# HELP

A help screen is standard for every CLI application. Cyclopts by-default adds `--help` and `-h` flags to the application:

```
$ my-application --help
Usage: my-application COMMAND

My application short description.

╭─ Commands ─────────────────────────────────────────────────────────────────╮
│ foo         Foo help string.                                               │
│ bar         Bar help string.                                               │
│ --help -h   Display this message and exit.                                 │
│ --version   Display application version.                                   │
╰────────────────────────────────────────────────────────────────────────────╯
```

Cyclopts derives the components of the help string from a variety of sources. The source resolution order is as follows (as applicable):

1. The `help` field in the *`@app.command`* decorator.

   ```python
   app = cyclopts.App()


   @app.command(help="This is the highest precedence help-string for 'bar'.")
   def bar():
       pass
   ```

   When registering an *`App`* object, supplying `help` via the *`@app.command`* decorator is forbidden to reduce ambiguity and will raise a `ValueError`. See (2).

2. Via *`App.help`*.

   ```python
   app = cyclopts.App(help="This help string has highest precedence at the app-level.")

   sub_app = cyclopts.App(help="This is the help string for the 'foo' subcommand.")
   app.command(sub_app, name="foo")
   app.command(sub_app, name="foo", help="This is illegal and raises a ValueError.")
   ```

3. The `__doc__` docstring of the registered *`@app.default`* command. Cyclopts parses the docstring to populate short-descriptions and long-descriptions at the command-level, as well as at the parameter-level.

   ```python
   app = cyclopts.App()
   app.command(cyclopts.App(), name="foo")
   ```

(continues on next page)

(continued from previous page)

```
@app.default
def bar(val1: str):
    """This is the primary application docstring.

    Parameters
    ----------
    val1: str
        This will be parsed for val1 help-string.
    """


@app["foo"].default  # You can access sub-apps like a dictionary.
def foo_handler():
    """This will be shown for the "foo" command."""
```

**Note**

Docstrings should always use the **Python variable name** from the function signature.

```
@app.default
def main(internal_name: Annotated[str, Parameter(name="external-name")]):
    """Command description.

    Parameters
    ----------
    internal_name:            # Use the Python variable name
        Help text here.
    """
```

This follows standard Python documentation conventions; the parameter will still appear as `--external-name` on the CLI.

4. This resolution order, but of the *Meta App*.

```
app = cyclopts.App()


@app.meta.default
def bar():
    """This is the primary application docstring."""
```

## 13.1 Markup Format

While the standard markup language for docstrings in Python is reStructuredText (see PEP-0287), Cyclopts defaults to Markdown for better readability and simplicity. Cyclopts mostly respects PEP-0257, but has some slight differences for developer ergonomics:

1. The "summary line" (AKA short-description) may actually be multiple lines. Cyclopts will unwrap the first block of text and interpret it as the short description. The first block of text ends at the first double-newline (i.e. a single blank line) is reached.

```python
def my_command():
    """
    This entire sentence
    is part of the short description and will
    have all the newlines removed.

    This is the beginning of the long description.
    """
```

2. If a docstring is provided with a long description, it **must** also have a short description.

By default, Cyclopts parses docstring descriptions as markdown and renders it appropriately. To change the markup format, set the *App.help_format* field accordingly. The different options are described below.

Subapps inherit their parent's *App.help_format* unless explicitly overridden. I.e. you only need to set *App.help_format* in your main root application for all docstrings to be parsed appropriately.

### 13.1.1 PlainText

Do not perform any additional parsing, display supplied text as-is.

```python
from cyclopts import App

app = App(help_format="plaintext")

@app.default
def default():
    """My application summary.

    This is a pretty standard docstring; if there's a really long sentence
    I should probably wrap it because people don't like code that is more
    than 80 columns long.

    In this new paragraph, I would like to discuss the benefits of relaxing 80 cols to␣
↪120 cols.
    More text in this paragraph.

    Some new paragraph.
    """

app()
```

```
Usage: default COMMAND

My application summary.

This is a pretty standard docstring; if there's a really long
sentence
I should probably wrap it because people don't like code that is
more
than 80 columns long.

In this new paragraph, I would like to discuss the benefits of
relaxing 80 cols to 120 cols.
```

(continues on next page)

(continued from previous page)

```
More text in this paragraph.

Some new paragraph.

╭─ Commands ─────────────────────────────────────────────────────╮
│ --help,-h  Display this message and exit.                      │
│ --version  Display application version.                        │
╰────────────────────────────────────────────────────────────────╯
```

Most noteworthy, is **no additional text reflow is performed**; newlines are presented as-is.

### 13.1.2 Rich

Displays text as Rich Markup.

> **Note**
>
> Newlines are interpreted literally.

```python
from cyclopts import App

app = App(help_format="rich")

@app.default
def default():
    """Rich can display colors like [red]red[/red] easily.

    However, I cannot be bothered to figure out how to show that in documentation.
    """

app()
```

### 13.1.3 ReStructuredText

ReStructuredText can be enabled by setting *help_format* to "restructuredtext" or "rst".

```python
app = App(help_format="restructuredtext")  # or "rst"

@app.default
def default():
    """My application summary.

    We can do RST things like have **bold text**.
    More words in this paragraph.

    This is a new paragraph with some bulletpoints below:

    * bullet point 1.
    * bullet point 2.
    """
```

(continues on next page)

(continued from previous page)

```
app()
```

Resulting help:

Under most circumstances, plaintext (without any additional markup) looks prettier and reflows better when interpreted as restructuredtext (or markdown, for that matter).

### 13.1.4 Markdown

Markdown is the default parsing behavior of Cyclopts, so *help_format* won't need to be explicitly set. It's another popular markup language that Cyclopts can render.

```
app = App(help_format="markdown")  # or "md"
# or don't supply help_format at all; markdown is default.


@app.default
def default():
    """My application summary.

    We can do markdown things like have **bold text**.
    [Hyperlinks work as well.](https://cyclopts.readthedocs.io)
    """
```

Resulting help:

## 13.2 Help Flags

The default `--help` flags can be changed to different name(s) via the `help_flags` parameter.

```
app = cyclopts.App(help_flags="--show-help")
app = cyclopts.App(help_flags=["--send-help", "--send-help-plz", "-h"])
```

To disable the help-page entirely, set `help_flags` to an empty string or iterable.

```
app = cyclopts.App(help_flags="")
app = cyclopts.App(help_flags=[])
```

## 13.3 Help Epilogue

An epilogue is text displayed at the end of the help screen, after all command and parameter panels. This is commonly used for version information, support contact details, or additional notes.

The epilogue is set via the *App.help_epilogue* attribute:

```
from cyclopts import App

app = App(
    name="myapp",
    help="My application description.",
    help_epilogue="Support: support@example.com"
```

(continues on next page)

(continued from previous page)

```
)

@app.default
def main():
    """Main command."""
    pass

app()
```

```
$ myapp --help
Usage: myapp [ARGS]

My application description.

─ Commands ──────────────────────────────────────────────────────────────────
│ --help -h  Display this message and exit.                                 │
│ --version  Display application version.                                   │
─────────────────────────────────────────────────────────────────────────────

Support: support@example.com
```

Like *App.help_format*, epilogues inherit from parent to child apps. This allows you to set a single epilogue that applies across your entire application:

```
parent = App(
    name="myapp",
    help_epilogue="Version 1.0.0 | support@example.com"
)

# Child inherits parent's epilogue
child = App(name="process", help="Process data files.")
parent.command(child)

# Another child overrides with its own epilogue
admin = App(
    name="admin",
    help="Admin commands.",
    help_epilogue="Admin Tools v2.0 | USE WITH CAUTION"
)
parent.command(admin)

parent()
```

```
$ myapp process --help
Usage: myapp process

Process data files.

Version 1.0.0 | support@example.com    # Inherited from parent

$ myapp admin --help
Usage: myapp admin
```

(continues on next page)

(continued from previous page)

```
Admin commands.

Admin Tools v2.0 | USE WITH CAUTION    # Overridden by child
```

To disable the epilogue for a specific subcommand, set it to an empty string:

```
no_epilogue = App(name="internal", help_epilogue="")
parent.command(no_epilogue)
```

## 13.4 Help Customization

For advanced customization of help screen appearance, including custom formatters, styled panels, and dynamic column layouts, see *Help Customization*.

CHAPTER

# FOURTEEN

# VERSION

All CLI applications should have the basic ability to check the installed version; i.e.:

```
$ my-application --version
7.5.8
```

By default, Cyclopts adds a command, *--version*:, that does exactly this. Cyclopts try's to reasonably figure out your package's version by itself. The resolution order for determining the version string is as follows:

1. An explicitly supplied version string or callable to the root Cyclopts application:

   ```python
   from cyclopts import App

   app = App(version="7.5.8")

   app()
   ```

   If a callable is provided, it will be invoked when running the `--version` command:

   ```python
   from cyclopts import App

   def get_my_application_version() -> str:
       return "7.5.8"

   app = App(version=get_my_application_version)
   app()
   ```

2. The invoking-package's Distribution Package's Version Number via importlib.metadata.version. Cyclopts attempts to derive the package module that instantiated the *App* object by traversing the call stack.

3. The invoking-package's defacto PEP8 standard `__version__` string. Cyclopts attempts to derive the package module that instantiated the *App* object by traversing the call stack.

   ```python
   # mypackage/__init__.py
   __version__ = "7.5.8"

   # mypackage/__main__.py
   # ``App`` will use ``mypackage.__version__``.
   app = cyclopts.App()
   ```

4. The default version string `"0.0.0"` will be displayed.

In short, if your CLI application is a properly structured python package, Cyclopts will automatically derive the correct version.

The `--version` flag can be changed to a different name(s) via the `version_flags` parameter.

```
app = cyclopts.App(version_flags="--show-version")
app = cyclopts.App(version_flags=["--version", "-v"])
```

To disable the `--version` flag, set `version_flags` to an empty string or iterable.

```
app = cyclopts.App(version_flags="")
app = cyclopts.App(version_flags=[])
```

CHAPTER

FIFTEEN

# SHELL COMPLETION

Cyclopts provides shell completion (tab completion) for bash, zsh, and fish shells.

## 15.1 Development & Standalone Scripts

Shell completion systems (bash, zsh, fish) can only provide completion for **installed commands** (executables in your `$PATH`), not for arbitrary Python scripts like `python myapp.py`. This is a fundamental limitation of how shells work.

To work around this during development, Cyclopts provides a `cyclopts run` command that acts as a wrapper:

```
$ cyclopts run myapp.py --help
$ cyclopts run myapp.py:app --verbose
```

Since `cyclopts` itself is an installed command, the shell can provide completion for it. The `cyclopts run` command then loads and executes your script, giving you completion for your development scripts without needing to package and install them.

**Script Path Format:**

- `cyclopts run script.py` - Auto-detects the App object. If an App object cannot be determined, it will raise an error.
- `cyclopts run script.py:app` - Explicitly specifies the App object to run

This is particularly useful during development before packaging your application.

**Virtual Environment Behavior:**

`cyclopts run` imports your script directly into the **same Python process** (no subprocess is created). This means:

- It uses whatever Python interpreter is currently running `cyclopts`
- Your script has access to all packages installed in the current environment
- You must install `cyclopts` in your project's virtual environment
- To use: activate your venv, then run `cyclopts run script.py`

```
$ source .venv/bin/activate  # or your venv activation method
$ cyclopts run myapp.py
```

> **Note**
>
> Completion for your script's commands comes through the `cyclopts` CLI completion. Install it once with: `cyclopts --install-completion`

> **Warning**
>
> **Performance:** `cyclopts run` uses **dynamic completion**, which imports your script and calls Python on **every tab press**. This can be slow if your script has heavy imports.
>
> To mitigate slow imports during development, consider using *Lazy Loading* for your commands. For production or frequent use, install **static completion** using the methods below. Static completion is pre-generated and does not call Python, making it instantaneous.

To install completion specifically for your standalone script (without using `cyclopts run`), you can use the Manual Installation approach below with your script's App object.

## 15.2 Installation

### 15.2.1 Programmatic Installation (Recommended)

Add completion installation to your CLI application using *App.register_install_completion_command*:

```python
from cyclopts import App

app = App(name="myapp")
app.register_install_completion_command()

# Your commands here...

if __name__ == "__main__":
    app()
```

Users can then install completion by running:

```
myapp --install-completion
```

### 15.2.2 Manual Installation

For programmatic control, use *App.install_completion* directly:

```python
from cyclopts import App
from pathlib import Path

app = App(name="myapp")

# Install for current shell
install_path = app.install_completion()
print(f"Installed completion to {install_path}")

# Install for specific shell
install_path = app.install_completion(shell="zsh")

# Install to custom location
install_path = app.install_completion(
    shell="bash",
    output=Path("/custom/path/completion.sh"),
)
```

### 15.2.3 Default Installation Paths

- **Zsh**: `~/.zsh/completions/_<app_name>`
- **Bash**: `~/.local/share/bash-completion/completions/<app_name>`
- **Fish**: `~/.config/fish/completions/<app_name>.fish`

## 15.3 Script Generation

To generate a completion script without installing it, use *App.generate_completion*:

```
from cyclopts import App

app = App(name="myapp")
script = app.generate_completion(shell="zsh")
print(script)
```

## 15.4 Shell Configuration

By default, Cyclopts modifies your shell RC file to enable completion:

- **Zsh**: Adds to `~/.zshrc`
- **Bash**: Adds to `~/.bashrc`
- **Fish**: No modification needed (automatically loads from `~/.config/fish/completions/`)

After installation, restart your shell or source the RC file.

To install without modifying shell RC files, use:

```
app.register_install_completion_command(add_to_startup=False)
```

CHAPTER

SIXTEEN

# COERCION RULES

This page intends to serve as a terse set of type coercion rules that Cyclopts follows.

Automatic coercion can always be overridden by the *`Parameter.converter`* field. Typically, the *`converter`* function will receive a single token, but it may receive multiple tokens if the annotated type is iterable (e.g. `list`, `set`). The number of tokens can be explicitly controlled with *`n_tokens`*, which is useful when the type signature doesn't match the desired CLI token consumption.

## 16.1 No Hint

If no explicit type hint is provided:

- If the parameter has a **non-None** default value, interpret the type as `type(default_value)`.

```python
from cyclopts import App

app = App()

@app.default
def default(value=5):
    print(f"{value=} {type(value)=}")

app()
```

```
$ my-program 3
value=3 type(value)=<class 'int'>
```

- Otherwise, *interpret the type as string*.

```python
from cyclopts import App

app = App()

@app.default
def default(value):
    print(f"{value=} {type(value)=}")

app()
```

```
$ my-program foo
value='foo' type(value)=<class 'str'>
```

## 16.2 Any

A standalone `Any` type hint is equivalent to *No Hint*

## 16.3 Str

No operation is performed, CLI tokens are natively strings.

```
from cyclopts import App

app = App()

@app.default
def default(value: str):
    print(f"{value=} {type(value)=}")

app()
```

```
$ my-program foo
value='foo' type(value)=<class 'str'>
```

## 16.4 Int

For convenience, Cyclopts provides a richer feature-set of parsing integers than just naively calling `int`.

- Accepts vanilla decimal values (e.g. `123`, `3.1415`). Floating-point values will be rounded prior to casting to an `int`.
- Accepts binary values (strings starting with `0b`)
- Accepts octal values (strings starting with `0o`)
- Accepts hexadecimal values (strings starting with `0x`).

### 16.4.1 Counting Flags

For parameters that need to track the number of times a flag appears (e.g., verbosity levels like `-vvv`), use *Parameter.count* with an `int` type hint.

```
from cyclopts import App, Parameter
from typing import Annotated

app = App()

@app.default
def main(verbose: Annotated[int, Parameter(alias="-v", count=True)] = 0):
    print(f"Verbosity level: {verbose}")

app()
```

```
$ my-program
Verbosity level: 0
```

(continues on next page)

(continued from previous page)

```
$ my-program -v
Verbosity level: 1

$ my-program -vvv
Verbosity level: 3

$ my-program --verbose --verbose
Verbosity level: 2

$ my-program -v --verbose -vv
Verbosity level: 4
```

## 16.5 Float

Token gets cast as `float(token)`. For example, `float("3.14")`.

## 16.6 Complex

Token gets cast as `complex(token)`. For example, `complex("3+5j")`

## 16.7 Bool

1. If specified as a **keyword**, booleans are interpreted flags that take no parameter. The default **false-like** flag are `--no-FLAG-NAME`. See *Parameter.negative* for more about this feature.

   Example:

   ```
   from cyclopts import App

   app = App()

   @app.command
   def foo(my_flag: bool):
       print(my_flag)

   app()
   ```

   ```
   $ my-program foo --my-flag
   True

   $ my-program foo --no-my-flag
   False
   ```

2. If specified as a **positional** argument, a case-insensitive lookup is performed:

   - If the token is a **true-like value** `{"yes", "y", "1", "true", "t"}`, then it is parsed as `True`.
   - If the token is a **false-like value** `{"no", "n", "0", "false", "f"}`, then it is parsed as `False`.
   - Otherwise, a *CoercionError* will be raised.

   Cyclopts is stricter than traditional `bool` casting; the provided value **must** be one of the above. For example, `2` is **not** considered a true-like value and will raise an error.

```
$ my-program foo 1
True

$ my-program foo 0
False

$ my-program foo 2
╭─ Error ──────────────────────────────────────╮
│ Invalid value for "--my-flag": unable to     │
│ convert "2" into bool.                       │
╰──────────────────────────────────────────────╯

$ my-program foo not-a-true-or-false-value
╭─ Error ──────────────────────────────────────────────────╮
│ Invalid value for "--my-flag": unable to convert         │
│ "not-a-true-or-false-value" into bool.                   │
╰──────────────────────────────────────────────────────────╯
```

3. If specified as a keyword with a value attached with an =, then the provided value will be parsed according to positional argument rules above (2).

```python
from cyclopts import App

app = App()

@app.command
def foo(my_flag: bool):
    print(my_flag)

 app()
```

```
$ my-program foo --my-flag=true
True

$ my-program foo --my-flag=false
False

$ my-program foo --no-my-flag=true
False

$ my-program foo --no-my-flag=false
True
```

## 16.8 List

Unlike more simple types like `str` and `int`, lists use different parsing rules depending on whether the values are provided positionally or by keyword.

### 16.8.1 Positional

When arguments are provided positionally:

- If *Parameter.allow_leading_hyphen* is `False` (default behavior), reaching an option-like token will stop parsing for this parameter. If the number of consumed tokens is not a multiple of the required number of tokens to create an element of the list, a *MissingArgumentError* will be raised.

```python
from cyclopts import App

app = App()

@app.command
def foo(values: list[int]):  # 1 CLI token per element
    print(values)

@app.command
def bar(values: list[tuple[int, str]]):  # 2 CLI tokens per element
    print(values)

app()
```

```
$ my-program foo 1 2 3
[1, 2, 3]

$ my-program bar 1 one 2 two
[(1, 'one'), (2, 'two')]

$ my-program bar 1 one 2
╭─ Error ──────────────────────────────────────────────────────╮
│ Command "bar" parameter "--values" requires 2 arguments.     │
│ Only got 1.                                                  │
╰──────────────────────────────────────────────────────────────╯
```

- If *Parameter.allow_leading_hyphen* is `True`, CLI tokens will be consumed unconditionally until exhausted.

```python
from cyclopts import App, Parameter
from pathlib import Path
from typing import Annotated

app = App()

@app.default
def main(
    files: Annotated[list[Path], Parameter(allow_leading_hyphen=True)],
    some_flag: bool = False,
):
    print(f"{some_flag=}")
    print(f"Analyzing files {files}")

app()
```

```
$ my-program foo.bin bar.bin --fizz.bin buzz.bin --some-flag
some_flag=True
```

(continues on next page)

(continued from previous page)

```
Analyzing files [PosixPath('foo.bin'), PosixPath('bar.bin'), PosixPath('--fizz.bin
↪'), PosixPath('buzz.bin')]
```

Known keyword arguments are parsed first (in this case, `--some-flag`). To unambiguously pass in values positionally, provide them after a bare `--`:

```
$ my-program -- foo.bin bar.bin --fizz.bin buzz.bin --some-flag
some_flag=False
Analyzing files [PosixPath('foo.bin'), PosixPath('bar.bin'), PosixPath('--fizz.bin
↪'), PosixPath('buzz.bin'), PosixPath('--some-flag')]
```

### 16.8.2 Keyword

When arguments are provided by keyword:

- Tokens will be consumed until enough data is collected to form the type-hinted object.
- The keyword can be specified multiple times.
- If *Parameter.allow_leading_hyphen* is `False` (default behavior), reaching an option-like token will raise *MissingArgumentError* if insufficient tokens have been parsed.

```python
from cyclopts import App

app = App()

@app.command
def foo(values: list[int]):  # 1 CLI token per element
    print(values)

@app.command
def bar(values: list[tuple[int, str]]):  # 2 CLI tokens per element
    print(values)

app()
```

```
$ my-program foo --values 1 --values 2 --values 3
[1, 2, 3]

$ my-program bar --values 1 one --values 2 two
[(1, 'one'), (2, 'two')]

$ my-program bar --values 1 --values 2
╭─ Error ──────────────────────────────────────────────────────╮
│ Command "bar" parameter "--values" requires 2 arguments.     │
│ Only got 1.                                                  │
╰──────────────────────────────────────────────────────────────╯
```

- If *Parameter.consume_multiple* is `True`, all remaining tokens will be consumed (until an option-like token is reached if *Parameter.allow_leading_hyphen* is `False`)

```python
from cyclopts import App, Parameter
from typing import Annotated
```

(continues on next page)

(continued from previous page)

```python
app = App()

@app.default
def foo(values: Annotated[list[int], Parameter(consume_multiple=True)]):  # 1 CLI␣
↪token per element
   print(values)

app()
```

```
$ my-program foo --values 1 2 3
[1, 2, 3]
```

### 16.8.3 Empty List

Commonly, if we want a default list for a parameter in a function, we set the default value to `None` in the signature and then set it to the actual list in the function body:

```python
def foo(extensions: Optional[list] = None):
   if extensions is None:
      extensions = [".png", ".jpg"]
```

We do this because mutable defaults is a common unexpected source of bugs in python.

However, sometimes we actually want to specify an empty list. To get an empty list pass in the flag `--empty-MY-LIST-NAME`.

```python
from cyclopts import App

app = App()

@app.default
def main(extensions: list | None = None):
   if extensions is None:
      extensions = [".png", ".jpg"]
   print(f"{extensions=}")

app()
```

```
$ my-program
extensions=['.png', '.jpg']

$ my-program --empty-extensions
extensions=[]
```

See *Parameter.negative* for more about this feature.

### 16.8.4 Positional Only With Subsequent Parameters

When a list is **positional-only**, it will consume tokens such that it leaves enough tokens for subsequent positional-only parameters.

```python
from pathlib import Path
from cyclopts import App

app = App()

@app.default
def main(srcs: list[Path], dst: Path, /):  # "/" makes all prior parameters POSITIONAL_
↪ONLY
    print(f"Processing files {srcs!r} to {dst!r}.")

app()
```

```
$ my-program foo.bin bar.bin output.bin
Processing files [PosixPath('foo.bin'), PosixPath('bar.bin')] to PosixPath('output.bin').
```

The console wildcard * is expanded by the console, so this example will naturally work with wildcards.

```
$ ls foo
buzz.bin fizz.bin

$ my-program foo/*.bin output.bin
Processing files [PosixPath('foo/buzz.bin'), PosixPath('foo/fizz.bin')] to PosixPath(
↪'output.bin').
```

## 16.9 Iterable

Follows the same rules as *List*. The passed in data will be a `list`.

## 16.10 Sequence

Follows the same rules as *List*. The passed in data will be a `list`.

## 16.11 Set

Follows the same rules as *List*, but the resulting datatype is a `set`.

## 16.12 Frozenset

Follows the same rules as *Set*, but the resulting datatype is a `frozenset`.

## 16.13 Tuple

- The inner type hint(s) will be applied independently to each element. Enough CLI tokens will be consumed to populate the inner types.
- Nested fixed-length tuples are allowed: E.g. `tuple[tuple[int, str], str]` will consume 3 CLI tokens.
- Indeterminite-size tuples `tuple[type, ...]` are only supported at the root-annotation level and behave similarly to *List*.

```python
from cyclopts import App

app = App()

@app.default
def default(coordinates: tuple[float, float, str]):
   print(f"{coordinates=}")

app()
```

And invoke our script:

```
$ my-program --coordinates 3.14 2.718 my-coord-name
coordinates=(3.14, 2.718, 'my-coord-name')
```

## 16.14 Dict

Cyclopts can populate dictionaries using keyword dot-notation:

```python
from cyclopts import App

app = App()

@app.default
def default(message: str, *, mapping: dict[str, str] | None = None):
    if mapping:
        for find, replace in mapping.items():
            message = message.replace(find, replace)
    print(message)

app()
```

```
$ my_program 'Hello Cyclopts users!'
Hello Cyclopts users!

$ my_program 'Hello Cyclopts users!' --mapping.Hello Hey
Hey Cyclopts users!

$ my_program 'Hello Cyclopts users!' --mapping.Hello Hey --mapping.users developers
Hey Cyclopts developers!
```

Due to the way of specifying keys, it is recommended to make dict parameters keyword-only; dicts **cannot** be populated positionally. If you do not wish for the user to be able to specify arbitrary keys, see *User-Defined Classes*. For specifying arbitrary keywords at the root level, see *kwargs*.

## 16.15 Union

The unioned types will be iterated **left-to-right** until a successful coercion is performed. `None` type hints are ignored.

```python
from cyclopts import App
from typing import Union
```

(continues on next page)

(continued from previous page)

```python
app = App()

@app.default
def default(a: Union[None, int, str]):
    print(type(a))

app()
```

```
$ my-program 10
<class 'int'>

$ my-program bar
<class 'str'>
```

## 16.16 Optional

`Optional[...]` is syntactic sugar for `Union[..., None]`. See *Union* rules.

## 16.17 Literal

The `Literal` type is a good option for limiting user input to a set of choices. Like *Union*, the `Literal` options will be iterated **left-to-right** until a successful coercion is performed. Cyclopts attempts to coerce the input token into the **type** of each `Literal` option.

```python
from cyclopts import App
from typing import Literal

app = App()

@app.default
def default(value: Literal["foo", "bar", 3]):
    print(f"{value=} {type(value)=}")

app()
```

```
$ my-program foo
value='foo' type(value)=<class 'str'>

$ my-program bar
value='bar' type(value)=<class 'str'>

$ my-program 3
value=3 type(value)=<class 'int'>

$ my-program fizz
╭─ Error ──────────────────────────────────────────────╮
│ Invalid value for "VALUE": unable to convert "fizz"  │
│ into one of {'foo', 'bar', 3}.                       │
╰──────────────────────────────────────────────────────╯
```

## 16.18 Enum

While *Literal* is the recommended way of providing the user a set of choices, another method is using `Enum`.

The *`Parameter.name_transform`* gets applied to all `Enum` names, as well as the CLI provided token. By default,this means that a **case-insensitive name** lookup is performed. If an enum name contains an underscore, the CLI parameter **may** instead contain a hyphen, `-`. Leading/Trailing underscores will be stripped.

If coming from Typer, **Cyclopts Enum handling is the reverse of Typer**. Typer attempts to match the token to an Enum **value**; Cyclopts attempts to match the token to an Enum **name**. This is done because generally the **name** of the enum is meant to be human readable, while the **value** has some program/machine significance.

As a real-world example, the PNG image format supports 5 different color-types, which gets encoded into a 1-byte int in the image header.

```python
from cyclopts import App
from enum import IntEnum

app = App()

class ColorType(IntEnum):
    GRAYSCALE = 0
    RGB = 2
    PALETTE = 3
    GRAYSCALE_ALPHA = 4
    RGBA = 6

@app.default
def default(color_type: ColorType = ColorType.RGB):
    print(f"Writing color-type value: {color_type} to the image header.")

app()
```

```
$ my-program
Writing color-type value: 2 to the image header.

$ my-program grayscale-alpha
Writing color-type value: 4 to the image header.
```

## 16.19 Flag

`Flag` enums (and by extension, `IntFlag`) are treated as a collection of boolean flags.

The *`Parameter.name_transform`* gets applied to all `Flag` names, as well as the CLI provided token. By default, this means that a **case-insensitive name** lookup is performed. If an enum name contains an underscore, the CLI parameter **may** instead contain a hyphen, `-`. Leading/Trailing underscores will be stripped.

```python
from cyclopts import App
from enum import Flag, auto

app = App()

class Permission(Flag):
    READ = auto()
```

(continues on next page)

(continued from previous page)

```
    WRITE = auto()
    EXECUTE = auto()

@app.default
def default(permissions: Permission = Permission.READ):
    print(f"Permissions: {permissions}")

app()
```

```
$ my-program
Permissions: Permission.READ

$ my-program write
Permissions: Permission.WRITE

$ my-program read write
Permissions: Permission.READ|WRITE

$ my-program --permissions.write
Permissions: Permission.WRITE

$ my-program --permissions.write --permissions.read
Permissions: Permission.READ|WRITE
```

**Note**

If you want to directly expose the flags as booleans (e.g. `--read`), then see *Namespace Flattening*.

## 16.20 date

Cyclopts supports parsing dates into a `date` object. It uses `fromisoformat()` under the hood, so the only supported format is `%Y-%m-%d` (e.g. 1956-01-31). However, if you use newer Python (>= 3.11), it also supports other formats such as `%Y%m%d` (e.g., 20191204), 2021-W01-1, etc, defined by ISO 8601.

## 16.21 datetime

Cyclopts supports parsing timestamps into a `datetime` object. The supplied time must be in one of the following formats:

- `%Y-%m-%d` (e.g. 1956-01-31)
- `%Y-%m-%dT%H:%M:%S` (e.g. 1956-01-31T10:00:00)
- `%Y-%m-%d %H:%M:%S` (e.g. 1956-01-31 10:00:00)
- `%Y-%m-%dT%H:%M:%S%z` (e.g. 1956-01-31T10:00:00+0000)
- `%Y-%m-%dT%H:%M:%S.%f` (e.g. 1956-01-31T10:00:00.123456)
- `%Y-%m-%dT%H:%M:%S.%f%z` (e.g. 1956-01-31T10:00:00.123456+0000)

## 16.22 timedelta

Cyclopts supports parsing time durations into a `timedelta` object. The supplied time must be in one of the following formats:

- `30s` - 30 seconds
- `5m` - 5 minutes
- `2h` - 2 hours
- `1d` - 1 day
- `3w` - 3 weeks
- `6M` - 6 months (approximate)
- `1y` - 1 year (approximate)

Combining durations is also supported:

- "1h30m" - 1 hour and 30 minutes
- "1d12h" - 1 day and 12 hours

## 16.23 User-Defined Classes

Cyclopts supports classically defined user classes, as well as classes defined by the following dataclass-like libraries:

- attrs
- dataclass
- NamedTuple
- pydantic
- TypedDict

> **Note**
>
> For `pydantic` classes, Cyclopts will *not* internally perform type conversions and instead relies on pydantic's coercion engine.

Subkey parsing allows for assigning values positionally and by keyword with a dot-separator.

```
from cyclopts import App
from dataclasses import dataclass
from typing import Literal

app = App()

@dataclass
class User:
   name: str
   age: int
   region: Literal["us", "ca"] = "us"

@app.default
```

(continues on next page)

(continued from previous page)

```python
def main(user: User):
    print(user)

app()
```

```
$ my-program --help
Usage: main COMMAND [ARGS] [OPTIONS]

╭─ Commands ─────────────────────────────────────────────────────────────────────╮
│ --help -h  Display this message and exit.                                      │
│ --version  Display application version.                                        │
╰────────────────────────────────────────────────────────────────────────────────╯
╭─ Parameters ───────────────────────────────────────────────────────────────────╮
│ *  USER.NAME --user.name      [required]                                       │
│ *  USER.AGE --user.age        [required]                                       │
│    USER.REGION --user.region  [choices: us, ca] [default: us]                  │
╰────────────────────────────────────────────────────────────────────────────────╯

$ my-program 'Bob Smith' 30
User(name='Bob Smith', age=30, region='us')

$ my-program --user.name 'Bob Smith' --user.age 30
User(name='Bob Smith', age=30, region='us')

$ my-program --user.name 'Bob Smith' 30 --user.region=ca
User(name='Bob Smith', age=30, region='ca')
```

Cyclopts will recursively search for *Parameter* annotations and respect them:

```python
from cyclopts import App, Parameter
from dataclasses import dataclass
from typing import Annotated

app = App()

@dataclass
class User:
    # Beginning with "--" will completely override the parenting parameter name.
    name: Annotated[str, Parameter(name="--nickname")]
    # Not beginning with "--" will tack it on to the parenting parameter name.
    age: Annotated[int, Parameter(name="years-young")]

@app.default
def main(user: Annotated[User, Parameter(name="player")]):
    print(user)

app()
```

```
$ my-program --help
Usage: main COMMAND [ARGS] [OPTIONS]

╭─ Commands ─────────────────────────────────────────────────────────────────────╮
```

(continues on next page)

(continued from previous page)

```
│ --help -h  Display this message and exit.                    │
│ --version  Display application version.                      │
──────────────────────────────────────────────────────────────
─ Parameters ─────────────────────────────────────────────────
│ *  NICKNAME --nickname     [required]                        │
│ *  PLAYER.YEARS-YOUNG      [required]                        │
│      --player.years-young                                    │
──────────────────────────────────────────────────────────────
```

### 16.23.1 Namespace Flattening

The special parameter name "*" will remove the immediate parameter's name from the dotted-hierarchal name:

```python
from cyclopts import App, Parameter
from dataclasses import dataclass
from typing import Annotated

app = App()

@dataclass
class User:
   name: str
   age: int

@app.default
def main(user: Annotated[User, Parameter(name="*")]):
   print(user)

app()
```

```
$ my-program --help
Usage: main COMMAND [ARGS] [OPTIONS]

─ Commands ──────────────────────────────────────────────────
│ --help -h  Display this message and exit.                  │
│ --version  Display application version.                    │
────────────────────────────────────────────────────────────
─ Parameters ────────────────────────────────────────────────
│ *  NAME --name  [required]                                 │
│ *  AGE --age    [required]                                 │
────────────────────────────────────────────────────────────
```

This can be used to conveniently share parameters between commands, and to create a global config object. See *Sharing Parameters*.

### 16.23.2 Docstrings

Docstrings from the class are used for the help page. Docstrings from the command have priority over class docstrings, if supplied:

```python
from cyclopts import App
from dataclasses import dataclass
```

(continues on next page)

(continued from previous page)

```python
app = App()

@dataclass
class User:
   name: str
   "First and last name of the user."

   age: int
   "Age in years of the user."

@app.default
def main(user: User):
   """A short summary of what this program does.

   Parameters
   ----------
   user.age: int
      User's age docstring from the command docstring.
   """
   print(user)

app()
```

```
$ my-program --help
Usage: main COMMAND [ARGS] [OPTIONS]

A short summary of what this program does.

╭─ Commands ─────────────────────────────────────────────────────────────────────╮
│ --help -h  Display this message and exit.                                      │
│ --version  Display application version.                                        │
╰────────────────────────────────────────────────────────────────────────────────╯
╭─ Parameters ───────────────────────────────────────────────────────────────────╮
│ *  USER.NAME --user.name  First and last name of the user. [required]          │
│ *  USER.AGE --user.age    User's age docstring from the command docstring.     │
│                           [required]                                           │
╰────────────────────────────────────────────────────────────────────────────────╯
```

### 16.23.3 Parameter(accepts_keys=False)

If the class is annotated with `Parameter(accepts_keys=False)`, then no dot-notation subkeys are exported. The class parameter will consume enough tokens to populate the **required positional** arguments.

```python
from cyclopts import App, Parameter
from dataclasses import dataclass
from typing import Annotated, Literal

app = App()

@dataclass
class User:
```

(continues on next page)

(continued from previous page)

```python
    name: str
    age: int
    region: Literal["us", "ca"] = "us"

@app.default
def main(user: Annotated[User, Parameter(accepts_keys=False)]):
    print(user)

app()
```

```
$ my-program --help
Usage: main COMMAND [ARGS] [OPTIONS]

╭─ Commands ─────────────────────────────────────────────────────────────────────╮
│ --help -h  Display this message and exit.                                      │
│ --version  Display application version.                                        │
╰────────────────────────────────────────────────────────────────────────────────╯
╭─ Parameters ───────────────────────────────────────────────────────────────────╮
│ *  USER --user  [required]                                                     │
╰────────────────────────────────────────────────────────────────────────────────╯

$ my-program 'Bob Smith' 27
User(name='Bob Smith', age=27, region='us')

$ my-program 'Bob Smith'
╭─ Error ────────────────────────────────────────────────────────────────────────╮
│ Parameter "--user" requires 2 arguments. Only got 1.                           │
╰────────────────────────────────────────────────────────────────────────────────╯
```

In this example, we are unable to change the `region` parameter of `User` from the CLI.

CHAPTER

# SEVENTEEN

# TEXT EDITOR

Some CLI programs require users to edit more complex fields in a text editor. For example, `git` may open a text editor for the user when rebasing or editing a commit message. While not directly related to CLI command parsing, Cyclopts provides *cyclopts.edit()* to satisfy this common need.

Here is an example application that mimics `git commit` functionality.

```python
# git.py
import cyclopts
from textwrap import dedent
import sys

app = cyclopts.App(name="git")

@app.command
def commit():
    try:
        response = cyclopts.edit(  # blocks until text editor is closed.
            dedent(  # removes the  leading 4-tab indentation.
                """\


                # Please enter the commit message for your changes.Lines starting
                # with '#' will be ignored, and an empty message aborts the commit.
                """
            )
        )
    except (cyclopts.EditorDidNotSaveError, cyclopts.EditorDidNotChangeError):
        print("Aborting commit due to empty commit message.")
        sys.exit(1)
    filtered = "\n".join(x for x in response.split("\n") if not x.startswith("#"))
    filtered = filtered.strip()  # remove leading/trailing whitespace.
    print(f"Your commit message: {filtered}")

if __name__ == "__main__":
    app()
```

Running `python git.py commit` will bring up a text editor with the pre-defined text, and then return the contents of the file. For more interactive CLI prompting, we recommend using the questionary package. See *edit()* API page for more advanced usage.

CHAPTER

# EIGHTEEN

# API

class cyclopts.**App**(*name: None | Any | Iterable[Any] = None*, *help: str | None = None*, *usage: str | None = None*, ***, *alias: None | Any | Iterable[Any] = None*, *default_command=None*, *default_parameter: Parameter | None = None*, *config: None | Any | Iterable[Any] = None*, *version: None | str | Callable[[...], str] | Callable[[...], Coroutine[Any, Any, str]] = None*, *version_flags: None | Any | Iterable[Any] = ['--version']*, *show: bool = True*, *console: Console | None = None*, *error_console: Console | None = None*, *help_flags: None | Any | Iterable[Any] = ['--help', '-h']*, *help_format: Literal['markdown', 'md', 'plaintext', 'restructuredtext', 'rst', 'rich'] | None = None*, *help_on_error: bool | None = None*, *help_epilogue: str | None = None*, *version_format: Literal['markdown', 'md', 'plaintext', 'restructuredtext', 'rst', 'rich'] | None = None*, *group: None | Any | Iterable[Any] = None*, *group_arguments: None | str | Group = None*, *group_parameters: None | str | Group = None*, *group_commands: None | str | Group = None*, *validator: None | Any | Iterable[Any] = None*, *name_transform: Callable[[str], str] | None = None*, *sort_key: Any = None*, *end_of_options_delimiter: str | None = None*, *print_error: bool | None = None*, *exit_on_error: bool | None = None*, *verbose: bool | None = None*, *suppress_keyboard_interrupt: bool = True*, *backend: Literal['asyncio', 'trio'] | None = None*, *help_formatter: None | Literal['default', 'plain'] | Any = None*, *result_action: None | Any | Iterable[Any] = None*)

Cyclopts Application.

**name:** `str | Iterable[str] | None = None`

Name of application, or subcommand if registering to another application. Name resolution order:

1. User specified *name* parameter.
2. If a *default* function has been registered, the name of that function.
3. If the module name is `__main__.py`, the name of the encompassing package.
4. The value of `sys.argv[0]`; i.e. the name of the python script.

Multiple names can be provided in the case of a subcommand, but this is relatively unusual.

Special value `"*"` can be used when registering sub-apps with *command()* to flatten all commands from the sub-app into the parent app. See *Flattening SubCommands* for details.

Example:

```
from cyclopts import App

app = App()
app.command(App(name="foo"))

@app["foo"].command
```

(continues on next page)

(continued from previous page)

```
def bar():
    print("Running bar.")

app()
```

```
$ my-script foo bar
Running bar.
```

**alias:** `str | Iterable[str] | None = None`

Extends *name* with additional names. Unlike *name*, this does not override Cyclopts-derived names.

```
from cyclopts import App

app = App()

@app.command(alias="bar")
def foo():
    print("Running foo.")

app()
```

```
$ my-script foo
Running bar.

$ my-script bar
Running bar.
```

**help:** `str | None = None`

Text to display on help screen. If not supplied, fallbacks to parsing the docstring of function registered with *App.default()*.

```
from cyclopts import App

app = App(help="This is my help string.")
app()
```

```
$ my-script --help
Usage: scratch.py COMMAND

This is my help string.

╭─ Commands ─────────────────────────────────────────────────────────────────╮
│ --help -h  Display this message and exit.                                  │
│ --version  Display application version.                                    │
╰────────────────────────────────────────────────────────────────────────────╯
```

**help_flags:** `str | Iterable[str] = ("--help", "-h")`

CLI flags that trigger *help_print()*. Set to an empty list to disable this feature. Defaults to `["--help", "-h"]`.

**help_format:** `Literal['plaintext', 'markdown', 'md', 'restructuredtext', 'rst'] | None = None`

The markup language used in function docstring. If None, fallback to parenting *help_format*. If no *help_format* is defined, falls back to "markdown".

**help_formatter: None | Literal['default', 'plain'] | *HelpFormatter* = None**

Help formatter to use for rendering help panels.

- If None (default), inherits from parent *App*, eventually defaulting to *DefaultFormatter*.
- If "default", uses *DefaultFormatter*.
- If "plain", uses *PlainFormatter* for no-frills plain text output.
- If a callable (see *HelpFormatter* protocol), uses the provided formatter.

Example:

```
from cyclopts import App
from cyclopts.help import DefaultFormatter, PlainFormatter, PanelSpec

# Use plain text formatter
app = App(help_formatter="plain")

# Use default formatter with customization
app = App(help_formatter=DefaultFormatter(
    panel_spec=PanelSpec(border_style="blue")
))
```

See *Help Customization* for detailed examples and advanced usage.

**help_epilogue: str | None = None**

Text to display at the end of the help screen, after all help panels. Commonly used for version information, contact details, or additional notes. If None, no epilogue is displayed. If not set, attempts to inherit from parenting *App*.

The epilogue supports the same formatting as *help* based on *help_format* (markdown, plaintext, restructuredtext, or rich).

Example:

```
from cyclopts import App

app = App(
    name="myapp",
    help="My application help.",
    help_epilogue="Support: support@example.com"
)
app()
```

```
$ my-script --help
Usage: myapp COMMAND

My application help.

╭─ Commands ─────────────────────────────────────────────────────────────────╮
│ --help -h  Display this message and exit.                                  │
│ --version  Display application version.                                    │
```

(continues on next page)

(continued from previous page)

```
──────────────────────────────────────────────────────────────────────────────


Support: support@example.com
```

**help_on_error:** `bool | None = None`

Prints the help-page before printing an error. If not set, attempts to inherit from parenting *App*, eventually defaulting to `False`.

**print_error:** `bool | None = None`

Print a rich-formatted error on error. If not set, attempts to inherit from parenting *App*, eventually defaulting to `True`.

**exit_on_error:** `bool | None = None`

If there is an error parsing the CLI tokens, invoke `sys.exit(1)`. Otherwise, continue to raise the exception. If not set, attempts to inherit from parenting *App*, eventually defaulting to `True`.

**verbose:** `bool | None = None`

Populate exception strings with more information intended for developers. If not set, attempts to inherit from parenting *App*, eventually defaulting to `False`.

**version_format:** `Literal['plaintext', 'markdown', 'md', 'restructuredtext', 'rst'] | None = None`

The markup language used in the version string. If `None`, fallback to parenting *version_format*. If no *version_format* is defined, falls back to resolved *help_format*.

**usage:** `str | None = None`

Text to be displayed in lieue of the default `Usage: app COMMAND ...` at the beginning of the help-page. Set to an empty-string "" to disable showing the default usage.

**show:** `bool = True`

Show this **command** on the help screen. Hidden commands (`show=False`) are still executable.

```python
from cyclopts import App
app = App()

@app.command
def foo():
   print("Running foo.")

@app.command(show=False)
def bar():
   print("Running bar.")

app()
```

```
$ my-script foo
Running foo.

$ my-script bar
Running bar.

$ my-script --help
Usage: scratch.py COMMAND
```

(continues on next page)

(continued from previous page)

```
╭─ Commands ─────────────────────────────────────────────────────╮
│ foo                                                            │
│ --help -h  Display this message and exit.                      │
│ --version  Display application version.                        │
╰────────────────────────────────────────────────────────────────╯
```

**sort_key: Any = None**

Modifies command display order on the help-page.

1. If *sort_key* is a generator, it will be consumed immediately with `next()` to get the actual value.
2. If *sort_key*, or any of it's contents, are `Callable`, then invoke it `sort_key(app)` and apply the returned value to (3) if `None`, (4) otherwise.
3. For all commands with `sort_key==None` (default value), sort them alphabetically. These sorted commands will be displayed **after** `sort_key != None` list (see 4).
4. For all commands with `sort_key!=None`, sort them by `(sort_key, app.name)`. It is the user's responsibility that `sort_key` s are comparable.

Example usage:

```python
from cyclopts import App

app = App()

@app.command  # sort_key not specified; will be sorted AFTER bob/charlie.
def alice():
    """Alice help description."""

@app.command(sort_key=2)
def bob():
    """Bob help description."""

@app.command(sort_key=1)
def charlie():
    """Charlie help description."""

app()
```

Resulting help-page:

```
Usage: demo.py COMMAND

╭─ Commands ─────────────────────────────────────────────────────╮
│ charlie    Charlie help description.                           │
│ bob        Bob help description.                               │
│ alice      Alice help description.                             │
│ --help -h  Display this message and exit.                      │
│ --version  Display application version.                        │
╰────────────────────────────────────────────────────────────────╯
```

Using generators (e.g., `itertools.count()`):

```python
import itertools
from cyclopts import App

app = App()
counter = itertools.count()

@app.command(sort_key=counter)
def beta():
    """Beta help description."""

@app.command(sort_key=counter)
def alpha():
    """Alpha help description."""

app()
```

```
Usage: demo.py COMMAND

╭─ Commands ─────────────────────────────────────────────────────╮
│ beta       Beta help description.                              │
│ alpha      Alpha help description.                             │
│ --help -h  Display this message and exit.                      │
│ --version  Display application version.                        │
╰────────────────────────────────────────────────────────────────╯
```

**version: None | str | Callable = None**

Version to be displayed when a *version_flags* is parsed. Defaults to the version of the package instantiating *App*. If a Callable, it will be invoked with no arguments when version is queried.

**version_flags: str | Iterable[str] = ("--version",)**

Token(s) that trigger *version_print()*. Set to an empty list to disable version feature. Defaults to ["--version"].

**console: Console = None**

Default Console to use when displaying runtime messages. Cyclopts console resolution is as follows:

1. Any explicitly passed in console to methods like *App.__call__()*, *App.parse_args()*, etc.
2. The relevant subcommand's *App.console* attribute, if not None.
3. The parenting *App.console* (and so on), if not None.
4. If all values are None, then the default Console is used.

**error_console: Console = None**

Default Console to use when displaying error messages. Cyclopts error_console resolution is as follows:

1. Any explicitly passed in error_console to methods like *App.__call__()*, *App.parse_args()*, etc.
2. The relevant subcommand's *App.error_console* attribute, if not None.
3. The parenting *App.error_console* (and so on), if not None.
4. If all values are None, then a default Console with stderr=True is used.

This separation of error output from normal output follows Unix conventions, allowing users to redirect error messages independently from normal output (e.g., program > output.txt 2> errors.txt).

**default_parameter: *Parameter* = None**

Default *Parameter* configuration. Unspecified values of command-annotated *Parameter* will inherit these values. See *Default Parameter* for more details.

**group: None | str | *Group* | Iterable[str | *Group*] = None**

The group(s) that `default_command` belongs to.

- If `None`, defaults to the `"Commands"` group.
- If `str`, use an existing *Group* (from neighboring sub-commands) with name, **or** create a *Group* with provided name if it does not exist.
- If *Group*, directly use it.

**group_commands: *Group* = Group("Commands")**

The default *Group* that sub-commands are assigned to.

**group_arguments: *Group* = Group("Arguments")**

The default *Group* that positional-only parameters are assigned to.

**group_parameters: *Group* = Group("Parameters")**

The default *Group* that non-positional-only parameters are assigned to.

**validator: None | Callable | list[Callable] = []**

A function (or list of functions) where all the converted CLI-provided variables will be **keyword-unpacked**, regardless of their positional/keyword-type in the command function signature. The python variable names will be used, which may differ from their CLI names.

Example usage:

```
def validator(**kwargs):
    "Raise an exception if something is invalid."
```

This validator runs **after** *Parameter* and *Group* validators.

**name_transform: Callable[[str], str] | None = None**

A function that converts function names to their CLI command counterparts.

The function must have signature:

```
def name_transform(s: str) -> str:
    ...
```

The returned string should be **without** a leading `--`. If `None` (default value), uses *default_name_transform()*. Subapps inherit from the first non-`None` parent *name_transform*.

**config: None | Callable | Iterable[Callable] = None**

A function or list of functions that are consecutively executed after parsing CLI tokens and environment variables. These function(s) are called **before** any conversion and validation. Each config function must have signature:

```
def config(app: "App", commands: Tuple[str, ...], arguments:␣
↪ArgumentCollection):
    """Modifies given mapping inplace with some injected values.

    Parameters
    ----------
    app: App
```

(continues on next page)

(continued from previous page)

```
        The current command app being executed.
    commands: Tuple[str, ...]
        The CLI strings that led to the current command function.
    arguments: ArgumentCollection
        Complete ArgumentCollection for the app.
        Modify this collection inplace to influence values provided to the␣
↪function.
    """
```

The intended use-case of this feature is to allow users to specify functions that can load defaults from some external configuration. See *cyclopts.config* for useful builtins and *Config Files* for examples.

**end_of_options_delimiter: str | None = None**

All tokens after this delimiter will be force-interpreted as positional arguments. If not set, attempts to inherit from parenting *App*, eventually defaulting to POSIX-standard `"--"`. Set to an empty string to disable.

**suppress_keyboard_interrupt: bool = True**

If the application receives a keyboard interrupt (Ctrl-C), suppress the error message and exit gracefully. Set to `False` to let `KeyboardInterrupt` propagate normally.

**backend: Literal['asyncio', 'trio'] | None = None**

The async backend to use when executing async commands. If not set, attempts to inherit from parenting *App*, eventually defaulting to `"asyncio"`.

Example:

```
from cyclopts import App

app = App(backend="asyncio")

@app.default
async def main():
    await some_async_operation()

app()
```

The backend can also be overridden on a per-call basis:

```
app(backend="trio")  # Override the app's backend for this call
```

**result_action: Literal['return_value', 'call_if_callable', 'print_non_int_return_int_as_exit_code', 'print_str_return_int_as_exit_code', 'print_str_return_zero', 'print_non_none_return_int_as_exit_code', 'print_non_none_return_zero', 'return_int_as_exit_code_else_zero', 'print_non_int_sys_exit', 'sys_exit', 'return_none', 'return_zero', 'print_return_zero', 'sys_exit_zero', 'print_sys_exit_zero'] | Callable[[Any], Any] | Iterable[Literal[...] | Callable[[Any], Any]] | None = None**

Controls how *App.__call__()* and *App.run_async()* handle command return values. By default (`"print_non_int_sys_exit"`), the app will call `sys.exit()` with an appropriate exit code. This default was chosen for consistent functionality between standalone scripts, and console entrypoints.

Can be a predefined literal string, a custom callable that takes the result and returns a processed value, or a **sequence of actions** to be applied left-to-right in a pipeline.

Each predefined mode's exact behavior is shown below:

**"print_non_int_sys_exit"** (default)

The default CLI mode. Prints non-int values to stdout, then calls `sys.exit()` with the appropriate exit code.

```
if isinstance(result, bool):
    sys.exit(0 if result else 1)  # i.e. True is success
elif isinstance(result, int):
    sys.exit(result)
elif result is not None:
    print(result)
    sys.exit(0)
else:
    sys.exit(0)
```

**"return_value"**

Returns the command's value unchanged. Use for embedding Cyclopts in other Python code or testing.

```
return result
```

**"call_if_callable"**

Calls the result if it's callable (with no arguments), otherwise returns it unchanged. Useful for the dataclass command pattern where commands return class instances with `__call__` methods. Intended to be used in composition with other result actions (e.g., `["call_if_callable", "print_non_int_sys_exit"]`).

```
return result() if callable(result) else result
```

See *Dataclass Commands* for usage examples.

**"sys_exit"**

Never prints output. Calls `sys.exit()` with the appropriate exit code. Useful for CLI apps that handle their own output and just need exit code handling.

```
if isinstance(result, bool):
    sys.exit(0 if result else 1)  # i.e. True is success
elif isinstance(result, int):
    sys.exit(result)
else:
    sys.exit(0)
```

**"print_non_int_return_int_as_exit_code"**

Prints non-int values, returns int/bool as exit codes. Useful for testing and embedding.

```
if isinstance(result, bool):
    return 0 if result else 1  # i.e. True is success
elif isinstance(result, int):
    return result
elif result is not None:
    print(result)
    return 0
else:
    return 0
```

**"print_str_return_int_as_exit_code"**

Only prints string return values. Returns int/bool as exit codes, silently returns 0 for other types.

```
if isinstance(result, str):
    print(result)
    return 0
elif isinstance(result, bool):
    return 0 if result else 1  # i.e. True is success
elif isinstance(result, int):
    return result
else:
    return 0
```

**"print_str_return_zero"**

Only prints string return values, always returns 0. Useful for simple output-only CLIs.

```
if isinstance(result, str):
    print(result)
return 0
```

**"print_non_none_return_int_as_exit_code"**

Prints all non-None values (including ints), returns int/bool as exit codes.

```
if result is not None:
    print(result)
if isinstance(result, bool):
    return 0 if result else 1  # i.e. True is success
elif isinstance(result, int):
    return result
return 0
```

**"print_non_none_return_zero"**

Prints all non-None values (including ints), always returns 0.

```
if result is not None:
    print(result)
return 0
```

**"return_int_as_exit_code_else_zero"**

Never prints output. Returns int/bool as exit codes, 0 for all other types. Useful for silent CLIs.

```
if isinstance(result, bool):
    return 0 if result else 1  # i.e. True is success
elif isinstance(result, int):
    return result
else:
    return 0
```

**"return_none"**

Always returns None, regardless of the command's return value.

```
return None
```

**"return_zero"**

Always returns 0, regardless of the command's return value.

```
return 0
```

**"print_return_zero"**

Always prints the result (even None), then always returns 0.

```
print(result)
return 0
```

**"sys_exit_zero"**

Always calls `sys.exit(0)`, regardless of the command's return value.

```
sys.exit(0)
```

**"print_sys_exit_zero"**

Always prints the result (even None), then calls `sys.exit(0)`.

```
print(result)
sys.exit(0)
```

**Custom Callable**

Provide a function for fully custom result handling. Receives the command's return value and returns a processed value.

```
def custom_handler(result):
    if result is None:
        return 0
    elif isinstance(result, str):
        print(f"[OUTPUT] {result}")
        return 0
    return result

app = App(result_action=custom_handler)
```

**Sequence of Actions**

Provide a sequence (list or tuple) of actions to create a result-processing pipeline. Actions are applied left-to-right, with each action receiving the result of the previous action.

```
def uppercase(result):
    return result.upper() if isinstance(result, str) else result

def add_prefix(result):
    return f"[OUTPUT] {result}" if isinstance(result, str) else result

# Pipeline: result → uppercase → add_prefix → return
app = App(result_action=[uppercase, add_prefix, "return_value"])
```

(continues on next page)

(continued from previous page)

```
@app.command
def greet(name: str) -> str:
    return f"hello {name}"

result = app(["greet", "world"])
# result == "[OUTPUT] HELLO WORLD"
```

Actions in a sequence can be any combination of predefined literal strings and custom callables. Empty sequences raise a `ValueError` at app initialization.

Example:

```
from cyclopts import App

# For CLI applications with console_scripts entry points
app = App(result_action="print_non_int_return_int_as_exit_code")

@app.command
def greet(name: str) -> str:
    return f"Hello {name}!"

app()
```

See *Result Action* for detailed examples and usage patterns.

**version_print**(*console: Annotated[Console | None, Parameter(parse=False)] = None*) → None

Print the application version.

**Parameters**

**console** (*Console*) -- Console to print version string to. If not provided, follows the resolution order defined in *App.console*.

**__getitem__**(*key: str*) → *App*

Get the subapp from a command string.

All commands get registered to Cyclopts as subapps. The actual function handler is at `app[key].default_command`.

If the command was registered via lazy loading (import path string), it will be imported and resolved on first access.

Example usage:

```
from cyclopts import App

app = App()
app.command(App(name="foo"))


@app["foo"].command
def bar():
    print("Running bar.")


app()
```

**__iter__**() → Iterator[str]

Iterate over command & meta command names.

Example usage:

```
from cyclopts import App

app = App()


@app.command
def foo():
    pass


@app.command
def bar():
    pass


# help and version flags are treated as commands.
assert list(app) == ["--help", "-h", "--version", "foo", "bar"]
```

**parse_commands**(*tokens: None | str | Iterable[str] = None*, *, *include_parent_meta=True*) → tuple[tuple[str, ...], tuple[*App*, ...], list[str]]

Extract out the command tokens from a command.

You are probably actually looking for *parse_args()*.

**Parameters**

- **tokens** (*None | str | Iterable[str]*) -- Either a string, or a list of strings to launch a command. Defaults to `sys.argv[1:]`
- **include_parent_meta** (*bool*) -- Controls whether parent meta apps are included in the execution path.

  When True (default): - Parent meta apps (i.e. the "normal" app ) are added to the apps list. - Meta app options are consumed while parsing commands. - Used for getting the inheritance hierarchy.

  When False: - Meta app options are treated as regular arguments. - Used for getting the execution hierarchy.

  This parameter is primarily for internal use.

**Returns**

- *tuple[str, ...]* -- Strings that are interpreted as a valid command chain.
- *tuple[App, ...]* -- The execution path - apps that will be invoked in order.
- *list[str]* -- The remaining non-command tokens.

**command**(*obj: T*, *name: None | str | Iterable[str] = None*, *, *alias: None | str | Iterable[str] = None*, ***kwargs: object*) → T

**command**(*obj: None = None*, *name: None | str | Iterable[str] = None*, *, *alias: None | str | Iterable[str] = None*, ***kwargs: object*) → Callable[[T], T]

**command**(*obj: str*, *name: None | str | Iterable[str] = None*, ***, *alias: None | str | Iterable[str] = None*, ***kwargs: object*) → None

Decorator to register a function as a CLI command.

Example usage:

```
from cyclopts import App

app = App()

@app.command
def foo():
    print("foo!")

@app.command(name="buzz")
def bar():
    print("bar!")

# Lazy loading via import path
app.command("myapp.commands:create_user", name="create")

app()
```

```
$ my-script foo
foo!

$ my-script buzz
bar!

$ my-script create
# Imports and runs myapp.commands:create_user
```

**Parameters**

- **obj** (*Callable | App | str | None*) -- Function, *App*, or import path string to be registered as a command. For lazy loading, provide a string in format "module.path:function_or_app_name".
- **name** (*None | str | Iterable[str]*) -- Name(s) to register the command to. If not provided, defaults to:
  - If registering an *App*, then the app's name.
  - If registering a **function**, then the function's name after applying *name_transform*.
  - If registering via **import path**, then the attribute name after applying *name_transform*.

  Special value "*" flattens all sub-App commands into this app (App instances only). See *Flattening SubCommands* for details.
- ****kwargs** -- Any argument that *App* can take.

**default**(*obj: T*, ***, *validator: Callable[[...], Any] | None = None*) → T

**default**(*obj: None = None*, ***, *validator: Callable[[...], Any] | None = None*) → Callable[[T], T]

Decorator to register a function as the default action handler.

Example usage:

```python
from cyclopts import App

app = App()


@app.default
def main():
    print("Hello world!")


app()
```

```
$ my-script
Hello world!
```

**assemble_argument_collection**(*, *default_parameter:* Parameter | *None* = *None*, *parse_docstring:* *bool* = *False*) → *ArgumentCollection*

Assemble the argument collection for this app.

**Parameters**

- **default_parameter** (Parameter | *None*) -- Default parameter with highest priority.
- **parse_docstring** (*bool*) -- Parse the docstring of `default_command`. Set to `True` if we need help strings, otherwise set to `False` for performance reasons.

**Returns**

All arguments for this app.

**Return type**

*ArgumentCollection*

**parse_known_args**(*tokens:* *None* | *str* | *Iterable[str]* = *None*, *, *console:* *Console* | *None* = *None*, *error_console:* *Console* | *None* = *None*, *end_of_options_delimiter:* *str* | *None* = *None*) → tuple[Callable[[...], Any], BoundArguments, list[str], dict[str, Any]]

Interpret arguments into a registered function, `BoundArguments`, and any remaining unknown tokens.

**Parameters**

- **tokens** (*None | str | Iterable[str]*) -- Either a string, or a list of strings to launch a command. Defaults to `sys.argv[1:]`
- **console** (*Console*) -- Console to print help and runtime Cyclopts errors. If not provided, follows the resolution order defined in *App.console*.
- **error_console** (*Console*) -- Console to print error messages. If not provided, follows the resolution order defined in *App.error_console*.
- **end_of_options_delimiter** (*str | None*) -- All tokens after this delimiter will be force-interpreted as positional arguments. If None, inherits from *App.end_of_options_delimiter*, eventually defaulting to POSIX-standard `"--"`. Set to an empty string to disable.

**Returns**

- **command** (*Callable*) -- Bare function to execute.
- **bound** (*inspect.BoundArguments*) -- Bound arguments for `command`.
- **unused_tokens** (*list[str]*) -- Any remaining CLI tokens that didn't get parsed for `command`.

- **ignored** (*dict[str, Any]*) -- A mapping of python-variable-name to annotated type of any parameter with annotation `parse=False`. `Annotated` will be resolved. Intended to simplify *meta apps*.

**parse_args**(*tokens: None | str | Iterable[str] = None*, ***, *console: Console | None = None*, *error_console: Console | None = None*, *print_error: bool | None = None*, *exit_on_error: bool | None = None*, *help_on_error: bool | None = None*, *verbose: bool | None = None*, *end_of_options_delimiter: str | None = None*) → tuple[Callable, BoundArguments, dict[str, Any]]

Interpret arguments into a function and `BoundArguments`.

**Raises**

***UnusedCliTokensError*** -- If any tokens remain after parsing.

**Parameters**

- **tokens** (*None | str | Iterable[str]*) -- Either a string, or a list of strings to launch a command. Defaults to `sys.argv[1:]`.
- **console** (*Console*) -- Console to print help and runtime Cyclopts errors. If not provided, follows the resolution order defined in *App.console*.
- **error_console** (*Console*) -- Console to print error messages. If not provided, follows the resolution order defined in *App.error_console*.
- **print_error** (*bool | None*) -- Print a rich-formatted error on error. If `None`, inherits from *App.print_error*, eventually defaulting to `True`.
- **exit_on_error** (*bool | None*) -- If there is an error parsing the CLI tokens invoke `sys.exit(1)`. Otherwise, continue to raise the exception. If `None`, inherits from *App.exit_on_error*, eventually defaulting to `True`.
- **help_on_error** (*bool | None*) -- Prints the help-page before printing an error. If `None`, inherits from *App.help_on_error*, eventually defaulting to `False`.
- **verbose** (*bool | None*) -- Populate exception strings with more information intended for developers. If `None`, inherits from *App.verbose*, eventually defaulting to `False`.
- **end_of_options_delimiter** (*str | None*) -- All tokens after this delimiter will be force-interpreted as positional arguments. If `None`, inherits from *App.end_of_options_delimiter*, eventually defaulting to POSIX-standard `"--"`. Set to an empty string to disable.

**Returns**

- **command** (*Callable*) -- Function associated with command action.
- **bound** (*inspect.BoundArguments*) -- Parsed and converted `args` and `kwargs` to be used when calling `command`.
- **ignored** (*dict[str, Any]*) -- A mapping of python-variable-name to type-hint of any parameter with annotation `parse=False`. `Annotated` will be resolved. Intended to simplify *meta apps*.

__call__(*tokens: None | str | Iterable[str] = None*, ***, *console: Console | None = None*, *error_console: Console | None = None*, *print_error: bool | None = None*, *exit_on_error: bool | None = None*, *help_on_error: bool | None = None*, *verbose: bool | None = None*, *end_of_options_delimiter: str | None = None*, *backend: Literal['asyncio', 'trio'] | None = None*, *result_action: Literal['return_value', 'call_if_callable', 'print_non_int_return_int_as_exit_code', 'print_str_return_int_as_exit_code', 'print_str_return_zero', 'print_non_none_return_int_as_exit_code', 'print_non_none_return_zero', 'return_int_as_exit_code_else_zero', 'print_non_int_sys_exit', 'sys_exit', 'return_none', 'return_zero', 'print_return_zero', 'sys_exit_zero', 'print_sys_exit_zero'] | Callable[[Any], Any] | Iterable[Literal['return_value', 'call_if_callable', 'print_non_int_return_int_as_exit_code', 'print_str_return_int_as_exit_code', 'print_str_return_zero', 'print_non_none_return_int_as_exit_code', 'print_non_none_return_zero', 'return_int_as_exit_code_else_zero', 'print_non_int_sys_exit', 'sys_exit', 'return_none', 'return_zero', 'print_return_zero', 'sys_exit_zero', 'print_sys_exit_zero'] | Callable[[Any], Any]] | None = None*) → Any

Interprets and executes a command.

**Parameters**

- **tokens** (*None | str | Iterable[str]*) -- Either a string, or a list of strings to launch a command. Defaults to `sys.argv[1:]`.
- **console** (*Console*) -- Console to print help and runtime Cyclopts errors. If not provided, follows the resolution order defined in *App.console*.
- **error_console** (*Console*) -- Console to print error messages. If not provided, follows the resolution order defined in *App.error_console*.
- **print_error** (*bool | None*) -- Print a rich-formatted error on error. If None, inherits from *App.print_error*, eventually defaulting to True.
- **exit_on_error** (*bool | None*) -- If there is an error parsing the CLI tokens invoke `sys.exit(1)`. Otherwise, continue to raise the exception. If None, inherits from *App.exit_on_error*, eventually defaulting to True.
- **help_on_error** (*bool | None*) -- Prints the help-page before printing an error. If None, inherits from *App.help_on_error*, eventually defaulting to False.
- **verbose** (*bool | None*) -- Populate exception strings with more information intended for developers. If None, inherits from *App.verbose*, eventually defaulting to False.
- **end_of_options_delimiter** (*str | None*) -- All tokens after this delimiter will be force-interpreted as positional arguments. If None, inherits from *App.end_of_options_delimiter*, eventually defaulting to POSIX-standard "--". Set to an empty string to disable.
- **backend** (*Literal["asyncio", "trio"] | None*) -- Override the async backend to use (if an async command is invoked). If None, inherits from *App.backend*, eventually defaulting to "asyncio". If passing backend="trio", ensure trio is installed via the extra: *cyclopts[trio]*.
- **result_action** (*ResultAction | None*) -- Controls how command return values are handled. Can be a predefined literal string or a custom callable that takes the result and returns a processed value. If None, inherits from *App.result_action*, eventually defaulting to "print_non_int_return_int_as_exit_code". See *App.result_action* for available modes.

**Returns**

**return_value** -- The value the command function returns.

**Return type**
Any

async **run_async**(*tokens: None | str | Iterable[str] = None*, ***, *console: Console | None = None*, *error_console: Console | None = None*, *print_error: bool | None = None*, *exit_on_error: bool | None = None*, *help_on_error: bool | None = None*, *verbose: bool | None = None*, *end_of_options_delimiter: str | None = None*, *backend: Literal['asyncio', 'trio'] | None = None*, *result_action: Literal['return_value', 'call_if_callable', 'print_non_int_return_int_as_exit_code', 'print_str_return_int_as_exit_code', 'print_str_return_zero', 'print_non_none_return_int_as_exit_code', 'print_non_none_return_zero', 'return_int_as_exit_code_else_zero', 'print_non_int_sys_exit', 'sys_exit', 'return_none', 'return_zero', 'print_return_zero', 'sys_exit_zero', 'print_sys_exit_zero'] | Callable[[Any], Any] | Iterable[Literal['return_value', 'call_if_callable', 'print_non_int_return_int_as_exit_code', 'print_str_return_int_as_exit_code', 'print_str_return_zero', 'print_non_none_return_int_as_exit_code', 'print_non_none_return_zero', 'return_int_as_exit_code_else_zero', 'print_non_int_sys_exit', 'sys_exit', 'return_none', 'return_zero', 'print_return_zero', 'sys_exit_zero', 'print_sys_exit_zero'] | Callable[[Any], Any]] | None = None*) → Any

Async equivalent of `__call__()` for use within existing event loops.

This method should be used when you're already in an async context (e.g., Jupyter notebooks, existing async applications) and need to execute a Cyclopts command without creating a new event loop.

**Parameters**

- **tokens** (*None | str | Iterable[str]*) -- Either a string, or a list of strings to launch a command. Defaults to `sys.argv[1:]`.
- **console** (*Console*) -- Console to print help and runtime Cyclopts errors. If not provided, follows the resolution order defined in *App.console*.
- **error_console** (*Console*) -- Console to print error messages. If not provided, follows the resolution order defined in *App.error_console*.
- **print_error** (*bool | None*) -- Print a rich-formatted error on error. If `None`, inherits from *App.print_error*, eventually defaulting to `True`.
- **exit_on_error** (*bool | None*) -- If there is an error parsing the CLI tokens invoke `sys.exit(1)`. Otherwise, continue to raise the exception. If `None`, inherits from *App.exit_on_error*, eventually defaulting to `True`.
- **help_on_error** (*bool | None*) -- Prints the help-page before printing an error. If `None`, inherits from *App.help_on_error*, eventually defaulting to `False`.
- **verbose** (*bool | None*) -- Populate exception strings with more information intended for developers. If `None`, inherits from *App.verbose*, eventually defaulting to `False`.
- **end_of_options_delimiter** (*str | None*) -- All tokens after this delimiter will be force-interpreted as positional arguments. If `None`, inherits from *App.end_of_options_delimiter*, eventually defaulting to POSIX-standard `"--"`. Set to an empty string to disable.
- **backend** (*Literal["asyncio", "trio"] | None*) -- Override the async backend to use (if an async command is invoked). If `None`, inherits from *App.backend*, eventually defaulting to "asyncio". If passing backend="trio", ensure trio is installed via the extra: *cyclopts[trio]*.
- **result_action** (*ResultAction | None*) -- Controls how command return values are handled. Can be a predefined literal string or a custom callable that takes the result and re-

turns a processed value. If None, inherits from *App.result_action*, eventually defaulting to "print_non_int_return_int_as_exit_code". See *App.result_action* for available modes.

**Returns**
**return_value** -- The value the command function returns.

**Return type**
Any

### Examples

```
import asyncio
from cyclopts import App

app = App()

@app.command
async def my_async_command():
    await asyncio.sleep(1)
    return "Done!"


# In an async context (e.g., Jupyter notebook or existing async app):
async def main():
    result = await app.run_async(["my-async-command"])
    print(result)  # Prints: Done!


asyncio.run(main())
```

**help_print**(*tokens: Annotated[None | str | Iterable[str], Parameter(show=False)] = None*, ***, *console: Annotated[Console | None, Parameter(parse=False)] = None*) → None

Print the help page.

**Parameters**

- **tokens** (*None | str | Iterable[str]*) -- Tokens to interpret for traversing the application command structure. If not provided, defaults to `sys.argv`.
- **console** (*Console*) -- Console to print help and runtime Cyclopts errors. If not provided, follows the resolution order defined in *App.console*.

**generate_docs**(*output_format: DocFormat = 'markdown'*, *recursive: bool = True*, *include_hidden: bool = False*, *heading_level: int = 1*, *max_heading_level: int = 6*, *flatten_commands: bool = False*) → str

Generate documentation for this CLI application.

**Parameters**

- **output_format** (*DocFormat*) -- Output format for the documentation. Accepts "markdown"/"md", "html"/"htm", or "rst"/"rest"/"restructuredtext". Default is "markdown".
- **recursive** (*bool*) -- If True, generate documentation for all subcommands recursively. Default is True.
- **include_hidden** (*bool*) -- If True, include hidden commands/parameters in documentation. Default is False.

- **heading_level** (*int*) -- Starting heading level for the main application title. Default is 1 (single # for markdown, = for RST).
- **max_heading_level** (*int*) -- Maximum heading level to use. Headings deeper than this will be capped at this level. Standard Markdown and HTML support levels 1-6. Default is 6.
- **flatten_commands** (*bool*) -- If True, generate all commands at the same heading level instead of nested. Default is False.

**Returns**
The generated documentation.

**Return type**
str

**Raises**
**ValueError** -- If an unsupported output format is specified.

### Examples

```
>>> app = App(name="myapp", help="My CLI Application")
>>> docs = app.generate_docs()  # Generate markdown as string
>>> html_docs = app.generate_docs(output_format="html")  # Generate HTML
>>> rst_docs = app.generate_docs(output_format="rst")  # Generate RST
>>> # To write to file, caller can do:
>>> # Path("docs/cli.md").write_text(docs)
```

**generate_completion**(*, *prog_name: str | None = None*, *shell: Literal['zsh', 'bash', 'fish'] | None = None*) → str

Generate shell completion script for this application.

**Parameters**

- **prog_name** (*str | None*) -- Program name for completion. If None, uses first name from app.name.
- **shell** (*Literal["zsh", "bash", "fish"] | None*) -- Shell type. If None, automatically detects current shell. Supported shells: "zsh", "bash", "fish".

**Returns**
Complete shell completion script.

**Return type**
str

### Examples

Auto-detect shell and generate completion:

```
>>> app = App(name="myapp")
>>> script = app.generate_completion()
>>> Path("_myapp").write_text(script)
```

Explicitly specify shell type:

```
>>> script = app.generate_completion(shell="zsh")
```

**Raises**

- **ValueError** -- If app has no name or shell type is unsupported.
- **ShellDetectionError** -- If shell is None and auto-detection fails.

**install_completion**(*, *shell: Literal['zsh', 'bash', 'fish'] | None = None*, *output: Path | None = None*, *add_to_startup: bool = True*) → Path

Install shell completion script to appropriate location.

Generates and writes the completion script to a shell-specific location.

**Parameters**

- **shell** (*Literal["zsh", "bash", "fish"] | None*) -- Shell type for completion. If not specified, attempts to auto-detect current shell.
- **output** (*Path | None*) -- Output path for the completion script. If not specified, uses shell-specific default: - zsh: ~/.zsh/completions/_<prog_name> - bash: ~/.local/share/bash-completion/completions/<prog_name> - fish: ~/.config/fish/completions/<prog_name>.fish
- **add_to_startup** (*bool*) -- If True (default), adds source line to shell RC file to ensure completion is loaded. Set to False if completions are already configured to auto-load.

**Returns**

Path where the completion script was installed.

**Return type**

*Path*

### Examples

Auto-detect shell and install:

```
>>> app = App(name="myapp")
>>> path = app.install_completion()
```

Install for specific shell:

```
>>> path = app.install_completion(shell="zsh")
```

Install to custom path:

```
>>> path = app.install_completion(output=Path("/custom/path"))
```

Install without modifying RC files:

```
>>> path = app.install_completion(shell="bash", add_to_startup=False)
```

**Raises**

- **ShellDetectionError** -- If shell is None and auto-detection fails.
- **ValueError** -- If shell type is unsupported.

**register_install_completion_command**(*name: str | Iterable[str] = '--install-completion'*, *add_to_startup: bool = True*, ***kwargs*) → None

Register a command for installing shell completion.

This is a convenience method that creates a command which calls *install_completion()*. For more control over the command implementation, users can manually define their own command.

**Parameters**

- **name** (*str | Iterable[str]*) -- Command name(s) for the install completion command. Defaults to "--install-completion".
- **add_to_startup** (*bool*) -- If True (default), adds source line to shell RC file to ensure completion is loaded. Set to False if completions are already configured to auto-load.
- ****kwargs** -- Additional keyword arguments to pass to *command()*. Can be used to customize the command registration (e.g., *help*, *group*, *help_flags*, *version_flags*).

### Examples

Register install-completion command:

```
>>> app = App(name="myapp")
>>> app.register_install_completion_command()
>>> app()  # Now responds to: myapp --install-completion
```

Use a custom command name:

```
>>> app.register_install_completion_command(name="--setup-completion")
```

Customize help text:

```
>>> app.register_install_completion_command(help="Install shell completion for␣
↪myapp.")
```

Customize command registration:

```
>>> app.register_install_completion_command(group="Setup", help_flags=[])
```

Install without modifying RC files:

```
>>> app.register_install_completion_command(add_to_startup=False)
```

**See also**

***install_completion***
The underlying method that performs the installation.

**interactive_shell**(*prompt: str = '$ ', quit: None | str | Iterable[str] = None, dispatcher: Dispatcher | None = None, console: Console | None = None, exit_on_error: bool = False, result_action: Literal['return_value', 'call_if_callable', 'print_non_int_return_int_as_exit_code', 'print_str_return_int_as_exit_code', 'print_str_return_zero', 'print_non_none_return_int_as_exit_code', 'print_non_none_return_zero', 'return_int_as_exit_code_else_zero', 'print_non_int_sys_exit', 'sys_exit', 'return_none', 'return_zero', 'print_return_zero', 'sys_exit_zero', 'print_sys_exit_zero'] | Callable[[Any], Any] | Iterable[Literal['return_value', 'call_if_callable', 'print_non_int_return_int_as_exit_code', 'print_str_return_int_as_exit_code', 'print_str_return_zero', 'print_non_none_return_int_as_exit_code', 'print_non_none_return_zero', 'return_int_as_exit_code_else_zero', 'print_non_int_sys_exit', 'sys_exit', 'return_none', 'return_zero', 'print_return_zero', 'sys_exit_zero', 'print_sys_exit_zero'] | Callable[[Any], Any]] | None = None, **kwargs*) → None

Create a blocking, interactive shell.

All registered commands can be executed in the shell.

**Parameters**

- **prompt** (*str*) -- Shell prompt. Defaults to `"$ "`.
- **quit** (*str | Iterable[str]*) -- String or list of strings that will cause the shell to exit and this method to return. Defaults to `["q", "quit"]`.
- **dispatcher** (*Dispatcher | None*) -- Optional function that subsequently invokes the command. The `dispatcher` function must have signature:

```
def dispatcher(command: Callable, bound: inspect.BoundArguments,␣
↪ignored: dict[str, Any]) -> Any:
    return command(*bound.args, **bound.kwargs)
```

  The above is the default dispatcher implementation.

- **console** (*Console | None*) -- Rich Console to use for output. If `None`, uses *App.console*.
- **exit_on_error** (*bool*) -- Whether to call `sys.exit` on parsing errors. Defaults to `False`.
- **result_action** (*ResultAction | None*) -- How to handle command return values in the interactive shell. Defaults to `"print_non_int_return_int_as_exit_code"` which prints non-int results and returns int/bool as exit codes without calling sys.exit. If `None`, inherits from *App.result_action*.
- ****kwargs** -- Get passed along to *parse_args()*.

**update**(*app:* App)

Copy over all commands from another *App*.

Commands from the meta app will **not** be copied over.

**Parameters**

**app** (`cyclopts.App`) -- All commands from this application will be copied over.

**class** cyclopts.**Parameter**(*name=None, *, converter: ~collections.abc.Callable[[...], ~typing.Any] | str | None = None, validator=(), alias=None, negative: None | ~typing.Any | ~collections.abc.Iterable[~typing.Any] = None, group: None | ~typing.Any | ~collections.abc.Iterable[~typing.Any] = None, parse=None, show: bool | None = None, show_default: None | bool | ~collections.abc.Callable[[~typing.Any], ~typing.Any] = None, show_choices=None, help: str | None = None, show_env_var=None, env_var=None, env_var_split: ~collections.abc.Callable = <function env_var_split>, negative_bool=None, negative_iterable=None, negative_none=None, required: bool | None = None, allow_leading_hyphen: bool = False, name_transform: ~collections.abc.Callable[[str], str] | None = None, accepts_keys: bool | None = None, consume_multiple=None, json_dict: bool | None = None, json_list: bool | None = None, count=None, n_tokens: int | None = None*)

Cyclopts configuration for individual function parameters with `Annotated`.

Example usage:

```
from cyclopts import app, Parameter
from typing import Annotated
```

(continues on next page)

(continued from previous page)

```
app = App()


@app.default
def main(foo: Annotated[int, Parameter(name="bar")]):
    print(foo)


app()
```

```
$ my-script 100
100

$ my-script --bar 100
100
```

**name:** `None | str | Iterable[str] = None`

Name(s) to expose to the CLI. If not specified, cyclopts will apply *name_transform* to the python parameter name.

```
from cyclopts import App, Parameter
from typing import Annotated

app = App()

@app.default
def main(foo: Annotated[int, Parameter(name=("bar", "-b"))]):
   print(f"{foo=}")

app()
```

```
$ my-script --help
Usage: main COMMAND [ARGS] [OPTIONS]

╭─ Commands ─────────────────────────────────────────────────────╮
│ --help -h  Display this message and exit.                      │
│ --version  Display application version.                        │
╰────────────────────────────────────────────────────────────────╯
╭─ Parameters ───────────────────────────────────────────────────╮
│ *  BAR --bar  -b  [required]                                   │
╰────────────────────────────────────────────────────────────────╯

$ my-script --bar 100
foo=100

$ my-script -b 100
foo=100
```

If specifying name in a nested data structure (e.g. a dataclass), beginning the name with a hyphen - will override any hierarchical dot-notation.

```python
from cyclopts import App, Parameter
from dataclasses import dataclass
from typing import Annotated

app = App()

@dataclass
class User:
   id: int  # default behavior
   email: Annotated[str, Parameter(name="--email")]  # overrides
   pwd: Annotated[str, Parameter(name="password")]  # dot-notation with parent

@app.command
def create(user: User):
   print(f"Creating {user=}")

app()
```

```
$ my-script create --help
Usage: scratch.py create [ARGS] [OPTIONS]

╭─ Parameters ─────────────────────────────────────────────────────────────────╮
│ *  USER.ID --user.id  [required]                                             │
│ *  EMAIL --email      [required]                                             │
│ *  USER.PASSWORD      [required]                                             │
│      --user.password                                                         │
╰──────────────────────────────────────────────────────────────────────────────╯
```

**alias: None | str | Iterable[str] = None**

Additional name(s) to expose to the CLI. Unlike *name*, this does not override Cyclopts-derived names.

The following two examples are functionally equivalent:

```python
@app.default
def main(foo: Annotated[int, Parameter(name=["--foo", "-f"])]):
    pass
```

```python
@app.default
def main(foo: Annotated[int, Parameter(alias="-f")]):
    pass
```

**converter: Callable | None = None**

A function that converts tokens into an object. The converter should have signature:

```python
def converter(type_, tokens) -> Any:
    pass
```

Where `type_` is the parameter's type hint, and `tokens` is either:

- A `list[cyclopts.Token]` of CLI tokens (most commonly).

```python
from cyclopts import App, Parameter
from typing import Annotated
```

(continues on next page)

(continued from previous page)

```
app = App()

def converter(type_, tokens):
    assert type_ == tuple[int, int]
    return tuple(2 * int(x.value) for x in tokens)

@app.default
def main(coordinates: Annotated[tuple[int, int],␣
↪Parameter(converter=converter)]):
    print(f"{coordinates=}")

app()
```

```
$ python my-script.py 7 12
coordinates=(14, 24)
```

- A `dict` of *Token* if keys are specified in the CLI. E.g.

```
$ python my-script.py --foo.key1=val1
```

would be parsed into:

```
tokens = {
    "key1": ["val1"],
}
```

If not provided, defaults to Cyclopts's internal coercion engine. If a pydantic type-hint is provided, Cyclopts will disable its internal coercion engine (including this *converter* argument) and leave the coercion to pydantic.

The number of tokens passed to the converter is inferred from the type hint by default, but can be explicitly controlled with *n_tokens*. This is useful when the type signature doesn't match the desired CLI token consumption. When loading complex objects with multiple fields, it may also be useful to combine with *accepts_keys*.

**Decorating Converters:** Converter functions can be decorated with *Parameter* to define reusable conversion behavior:

```
@Parameter(n_tokens=1, accepts_keys=False)
def load_from_id(type_, tokens):
    """Load object from database by ID."""
    return fetch_from_db(tokens[0].value)

@app.default
def main(obj: Annotated[MyType, Parameter(converter=load_from_id)]):
    # Automatically inherits n_tokens=1 and accepts_keys=False
    pass
```

**Classmethod Support:** Converters can be classmethods. Use string references for class decoration or direct references in annotations. Classmethod signature should be `(cls, tokens)` instead of `(type_, tokens)`:

```python
@Parameter(converter="from_env")
class Config:
    @Parameter(n_tokens=1, accepts_keys=False)
    @classmethod
    def from_env(cls, tokens):
        env = tokens[0].value
        configs = {"dev": ("localhost", 8080), "prod": ("api.example.com", 443)}
        return cls(*configs[env])
```

**validator:** `None | Callable | Iterable[Callable] = None`

A function (or list of functions) that validates data returned by the *converter*.

```python
def validator(type_, value: Any) -> None:
    pass  # Raise a TypeError, ValueError, or AssertionError here if data is␣
↪invalid.
```

**group:** `None | str | Group | Iterable[str | Group] = None`

The group(s) that this parameter belongs to. This can be used to better organize the help-page, and/or to add additional conversion/validation logic (such as ensuring mutually-exclusive arguments).

If `None`, defaults to one of the following groups:

1. Parenting *App.group_arguments* if the parameter is `POSITIONAL_ONLY`. By default, this is `Group("Arguments")`.
2. Parenting *App.group_parameters* otherwise. By default, this is `Group("Parameters")`.

See *Groups* for examples.

**negative:** `None | str | Iterable[str] = None`

Name(s) for empty iterables or false boolean flags.

- For booleans, defaults to `no-{name}` (see *negative_bool*).
- For iterables, defaults to `empty-{name}` (see *negative_iterable*).

Set to an empty list or string to disable the creation of negative flags.

Example usage:

```python
from cyclopts import App, Parameter
from typing import Annotated

app = App()

@app.default
def main(*, verbose: Annotated[bool, Parameter(negative="--quiet")] = False):
   print(f"{verbose=}")

app()
```

```
$ my-script --help
Usage: main COMMAND [ARGS] [OPTIONS]

─ Commands ──────────────────────────────────────────────────────────────────
│ --help -h  Display this message and exit.                                 │
```

(continues on next page)

(continued from previous page)

```
│ --version  Display application version.                                    │
╰────────────────────────────────────────────────────────────────────────────╯
╭─ Parameters ───────────────────────────────────────────────────────────────╮
│ --verbose --quiet  [default: False]                                        │
╰────────────────────────────────────────────────────────────────────────────╯
```

**negative_bool:** `str | None = None`

Prefix for negative boolean flags. Defaults to `"no-"`.

**negative_iterable:** `str | None = None`

Prefix for empty iterables (like lists and sets) flags. Defaults to `"empty-"`.

**negative_none:** `str | None = None`

Prefix for setting optional parameters to `None`. Not enabled by default (no prefixes set).

Example:

```python
from pathlib import Path
from typing import Annotated

from cyclopts import App, Parameter

app = App(
   default_parameter=Parameter(negative_none="none-")
)

@app.default
def default(path: Path | None = Path("data.bin")):
    print(f"{path=}")

app()
```

```
$ my-script
path=PosixPath('data.bin')

$ my-script --path=cat.jpeg
path=PosixPath('cat.jpeg')

$ my-script --none-path
path=None
```

**allow_leading_hyphen:** `bool = False`

Allow parsing non-numeric values that begin with a hyphen `-`. This is disabled (`False`) by default, allowing for more helpful error messages for unknown CLI options.

**parse:** `None | bool | str | re.Pattern = None`

Attempt to use this parameter while parsing. Annotated parameter **must** be keyword-only or have a default value. This is intended to be used with *meta apps* for injecting values.

- `True` - Parse this parameter from CLI tokens.
- `False` - Do not parse; parameter will appear in the `ignored` dict from *`App.parse_args()`*.
- `None` - Default behavior (parse).

- `str` - A regex pattern; parse **if the pattern matches the parameter name**, otherwise skip. String patterns are automatically compiled to `re.Pattern` for performance.
- `re.Pattern` - A pre-compiled regex pattern; same behavior as string patterns.

Regex patterns are primarily intended for use with *App.default_parameter* to define app-wide skip patterns. For example, if we wanted to skip all fields that begin with an underscore _:

```
import re
from cyclopts import App, Parameter

# Skip parsing underscore-prefixed KEYWORD_ONLY parameters (i.e. private␣
↪parameters)
# Both string and pre-compiled patterns are supported:
app = App(default_parameter=Parameter(parse="^(?!_)"))
# or: app = App(default_parameter=Parameter(parse=re.compile("^(?!_)")))

@app.default
def main(visible: str, *, _injected: str = "default"):
    # _injected is NOT parsed from CLI; uses default value
    pass
```

**required:** `bool | None = None`

Indicates that the parameter must be supplied. Defaults to inferring from the function signature; i.e. `False` if the parameter has a default, `True` otherwise.

**show:** `bool | None = None`

Show this parameter on the help screen. Defaults to whether the parameter is *parsed* (usually `True`).

**show_default:** `None | bool | Callable[[Any], Any] = None`

If a variable has a default, display the default on the help page. Defaults to `None`, similar to `True`, but will **not** display the default if it is `None`.

If set to a function with signature:

```
def formatter(value: Any) -> Any:
    ...
```

Then the function will be called with the default value, and the returned value will be used as the displayed default value.

Example formatting function:

```
def hex_formatter(value: int) -> str
   """Will result in something like "[default: 0xFF]" instead of "[default: 255]
↪"."""
   return f"0x{value:X}"
```

**show_choices:** `bool | None = True`

If a variable has a set of choices, display the choices on the help page.

**help:** `str | None = None`

Help string to be displayed on the help page. If not specified, defaults to the docstring.

**show_env_var:** `bool | None = True`

If a variable has *env_var* set, display the variable name on the help page.

**env_var: None | str | Iterable[str] = None**

Fallback to environment variable(s) if CLI value not provided. If multiple environment variables are given, the left-most environment variable **with a set value** will be used. If no environment variable is set, Cyclopts will fallback to the function-signature default.

**env_var_split: Callable = cyclopts.env_var_split**

Function that splits up the read-in *env_var* value. The function must have signature:

```
def env_var_split(type_: type, val: str) -> list[str]:
    ...
```

where `type_` is the associated parameter type-hint, and `val` is the environment value.

**name_transform: Callable[[str], str] | None = None**

A function that converts python parameter names to their CLI command counterparts.

The function must have signature:

```
def name_transform(s: str) -> str:
    ...
```

If `None` (default value), uses *cyclopts.default_name_transform()*.

**accepts_keys: bool | None = None**

If `False`, treat the user-defined class annotation similar to a tuple. Individual class sub-parameters will not be addressable by CLI keywords. The class will consume enough tokens to populate all required positional parameters.

Default behavior (`accepts_keys=True`):

```
from cyclopts import App, Parameter
from typing import Annotated

app = App()

class Image:
   def __init__(self, path, label):
      self.path = path
      self.label = label

   def __repr__(self):
      return f"Image(path={self.path!r}, label={self.label!r})"

@app.default
def main(image: Image):
   print(f"{image=}")

app()
```

```
$ my-program --help
Usage: main COMMAND [ARGS] [OPTIONS]

╭─ Commands ─────────────────────────────────────────────────────────────────╮
│ --help -h  Display this message and exit.                                  │
│ --version  Display application version.                                    │
```

(continues on next page)

(continued from previous page)

```
╭──────────────────────────────────────────────────────────────────────────────╮
│ Parameters ──────────────────────────────────────────────────────────────────│
│ *  IMAGE.PATH --image.path    [required]                                     │
│ *  IMAGE.LABEL --image.label  [required]                                     │
╰──────────────────────────────────────────────────────────────────────────────╯

$ my-program foo.jpg nature
image=Image(path='foo.jpg', label='nature')

$ my-program --image.path foo.jpg --image.label nature
image=Image(path='foo.jpg', label='nature')
```

Behavior when `accepts_keys=False`:

```python
# Modify the default command function's signature.
@app.default
def main(image: Annotated[Image, Parameter(accepts_keys=False)]):
    print(f"{image=}")
```

```
$ my-program --help
Usage: main COMMAND [ARGS] [OPTIONS]

╭─ Commands ───────────────────────────────────────────────────────────────────╮
│ --help -h  Display this message and exit.                                    │
│ --version  Display application version.                                      │
╰──────────────────────────────────────────────────────────────────────────────╯
╭─ Parameters ─────────────────────────────────────────────────────────────────╮
│ *  IMAGE --image  [required]                                                 │
╰──────────────────────────────────────────────────────────────────────────────╯

$ my-program foo.jpg nature
image=Image(path='foo.jpg', label='nature')

$ my-program --image foo.jpg nature
image=Image(path='foo.jpg', label='nature')
```

The `accepts_keys=False` option is commonly used with *converter* and *n_tokens*.

**consume_multiple: bool | None = None**

Lists use different parsing rules depending on whether the values are provided positionally or by keyword. If the parameter is specified **positionally**, *Parameter.consume_multiple* is ignored.

If the parameter is specified **by keyword** and `consume_multiple=True`, all remaining CLI tokens will be consumed until the stream is exhausted or an option-like token (typically a keyword) is reached (unless *Parameter.allow_leading_hyphen* is `True`, in which case it will also be consumed).

```python
from cyclopts import App, Parameter
from typing import Annotated

app = App()

@app.command
def name_ext(
```

(continues on next page)

(continued from previous page)

```
    name: str,
    ext: Annotated[list[str], Parameter(consume_multiple=True)],
):
    for extension in ext:
        print(f"{name}.{extension}")

app()
```

```
$ my-program --name "my_file" --ext "txt" "pdf"  # stream is exhausted
my_file.txt
my_file.pdf

$ my-program --ext "txt" "pdf" --name "my_file"  # a keyword is reached
my_file.txt
my_file.pdf
```

When `consume_multiple=True`, providing the keyword flag without any values will create an empty container, equivalent to using the *negative_iterable* prefix (e.g., `--empty-ext`):

```
$ my-program --name "my_file" --ext
# No output - ext is an empty list []

$ my-program --name "my_file" --empty-ext
# No output - ext is an empty list []
```

If the parameter is specified **by keyword** and `consume_multiple=False` (the default), only a single element worth of CLI tokens will be consumed.

```
from cyclopts import App
from pathlib import Path

app = App()

@app.default
def name_ext(name: str, ext: list[str]): # same as `ext: Annotated[list[str],␣
↪Parameter(consume_multiple=False)]``
    for extension in ext:
        print(f"{name}.{extension}")

app()
```

```
$ my-program --name "my_file" --ext "txt" "pdf"
╭─ Error ──────────────────────────────────────────╮
│ Unused Tokens: ['pdf'].                          │
╰──────────────────────────────────────────────────╯
```

**json_dict: bool | None = None**

Allow for the parsing of json-dict-strings as data. If `None` (default behavior), acts like `True`, **unless** the annotated type is union'd with `str`. When `True`, data will be parsed as json if the following conditions are met:

1. The parameter is specified as a keyword option; e.g. `--movie`.
2. The referenced parameter is dataclass-like.

3. The first character of the token is a `{`.

**json_list: bool | None = None**

Allow for the parsing of json-list-strings as data. If `None` (default behavior), acts like `True`, **unless** the annotated type has each element type `str`. When `True`, data will be parsed as json if the following conditions are met:

1. The referenced parameter is iterable (not including `str`).
2. The first character of the token is a `[`.

**count: bool = False**

If `True`, count the number of times the flag appears instead of parsing a value. Each occurrence increments the count by 1 (e.g., `-vvv` results in `3`).

Requirements and behavior:

- The parameter **must** have an `int` type hint (or `Optional[int]`).
- Short flags can be concatenated: `-vvv` is equivalent to `-v -v -v`.
- Long flags can be repeated: `--verbose --verbose` results in `2`.
- Negative flag variants (e.g., `--no-verbose`) are **not** generated.

Common use case: verbosity levels.

```
from cyclopts import App, Parameter
from typing import Annotated

app = App()

@app.default
def main(verbose: Annotated[int, Parameter(name="-v", count=True)] = 0):
    print(f"Verbosity level: {verbose}")

app()
```

```
$ my-script -vvv
Verbosity level: 3
```

See *Coercion Rules* for more details.

**n_tokens: int | None = None**

Explicitly override the number of CLI tokens this parameter consumes.

By default, Cyclopts infers the token count from the parameter's type hint (e.g., `int` consumes 1 token, `tuple[int, int]` consumes 2, `list` consumes all remaining). This attribute allows you to override that inference, which is particularly useful when:

- Using custom converters that need a different token count than the type suggests.
- Loading complex types from a single token (e.g., loading from a file path).
- Implementing selection/lookup patterns where one token identifies an object.

Values:

- `None` (default): Infer token count from the type hint.
- non-negative integer: Consume exactly that many tokens.
- `-1`: Consume all remaining tokens (similar to iterables).

For *args parameters, n_tokens specifies tokens **per element**. For example, n_tokens=2 with 6 tokens creates 3 elements.

```python
from cyclopts import App, Parameter
from typing import Annotated

class Config:
    def __init__(self, host: str, port: int):
        self.host = host
        self.port = port

def load_config(type_, tokens):
    # Load config from a file path (single token)
    filepath = tokens[0].value
    # ... load from file ...
    return Config("example.com", 8080)

app = App()

@app.default
def main(
    config: Annotated[
        Config,
        Parameter(n_tokens=1, converter=load_config, accepts_keys=False)
    ]
):
    print(f"Connecting to {config.host}:{config.port}")

app()
```

```
$ my-script --config prod.conf
Connecting to example.com:8080
```

**classmethod combine**(**parameters:* Parameter | *None*) → *Parameter*

Returns a new Parameter with combined values of all provided parameters.

**Parameters**

***parameters** (Parameter | *None*) -- Parameters who's attributes override self attributes. Ordered from least-to-highest attribute priority.

**classmethod default**() → Self

Create a Parameter with all Cyclopts-default values.

This is different than just *Parameter* because the default values will be recorded and override all upstream parameter values.

**__call__**(*obj: T*) → T

Decorator interface for annotating a function/class with a *Parameter*.

Most commonly used for directly configuring a class:

```python
@Parameter(...)
class Foo: ...
```

**class** cyclopts.**Group**(*name: str = ''*, *help: str = ''*, ***, *show: bool | None = None*, *sort_key: Any = None*, *validator=None*, *default_parameter:* Parameter | *None = None*, *help_formatter: None | Literal['default', 'plain'] | Any = None*)

A group of parameters and/or commands in a CLI application.

**name:** `str = ""`

Group name used for the help-page and for group-referenced-by-string. This is a title, so the first character should be capitalized. If a name is not specified, it will not be shown on the help-page.

**help:** `str = ""`

Additional documentation shown on the help-page. This will be displayed inside the group's panel, above the parameters/commands.

**show:** `bool | None = None`

Show this group on the help-page. Defaults to `None`, which will only show the group if a `name` is provided.

**help_formatter:** `None | Literal['default', 'plain'] | HelpFormatter = None`

Help formatter to use for rendering this group's help panel.

- If `None` (default), inherits from the *App*'s *help_formatter*.
- If `"default"`, uses *DefaultFormatter*.
- If `"plain"`, uses *PlainFormatter* for no-frills plain text output.
- If a callable (see *HelpFormatter* protocol), uses the provided formatter.

This allows per-group customization of help appearance:

```
from cyclopts import App, Group, Parameter
from cyclopts.help import DefaultFormatter, PanelSpec
from typing import Annotated

app = App()

# Using string literal
simple_group = Group(
    "Simple Options",
    help_formatter="plain"
)

# Using custom formatter instance
custom_group = Group(
    "Custom Options",
    help_formatter=DefaultFormatter(
        panel_spec=PanelSpec(border_style="red")
    )
)

@app.default
def main(
    opt1: Annotated[str, Parameter(group=simple_group)],
    opt2: Annotated[str, Parameter(group=custom_group)]
):
    pass
```

See *Help Customization* for detailed examples.

**sort_key:** `Any = None`

Modifies group-panel display order on the help-page.

1. If *sort_key* is a generator, it will be consumed immediately with `next()` to get the actual value.
2. If *sort_key*, or any of it's contents, are `Callable`, then invoke it `sort_key(group)` and apply the rules below.
3. The *App* default groups (`App.group_command`, *App.group_arguments*, *App.group_parameters*) will be displayed first. If you want to further customize the ordering of these default groups, you can define custom values and they will be treated like any other group:

```python
from cyclopts import App, Group

app = App(
    group_parameters=Group("Parameters", sort_key=1),
    group_arguments=Group("Arguments", sort_key=2),
    group_commands=Group("Commands", sort_key=3),
)


@app.default
def main(foo, /, bar):
    pass


if __name__ == "__main__":
    app()
```

```
$ python main.py --help
Usage: main [ARGS] [OPTIONS]

╭─ Parameters ─────────────────────────────────────────────────────────────────╮
│ *  BAR --bar  [required]                                                     │
╰──────────────────────────────────────────────────────────────────────────────╯
╭─ Arguments ──────────────────────────────────────────────────────────────────╮
│ *  FOO  [required]                                                           │
╰──────────────────────────────────────────────────────────────────────────────╯
╭─ Commands ───────────────────────────────────────────────────────────────────╮
│ --help -h  Display this message and exit.                                    │
│ --version  Display application version.                                      │
╰──────────────────────────────────────────────────────────────────────────────╯
```

4. For all groups with `sort_key!=None`, sort them by `(sort_key, group.name)`. That is, sort them by their `sort_key`, and then break ties alphabetically. It is the user's responsibility that `sort_key` are comparable.
5. For all groups with `sort_key==None` (default value), sort them alphabetically after (4), *App.group_commands*, *App.group_arguments*, and *App.group_parameters*.

Example usage:

```python
from cyclopts import App, Group

app = App()

@app.command(group=Group("4", sort_key=5))
```

(continues on next page)

(continued from previous page)

```python
def cmd1():
    pass


@app.command(group=Group("3", sort_key=lambda x: 10))
def cmd2():
    pass


@app.command(group=Group("2", sort_key=lambda x: None))
def cmd3():
    pass


@app.command(group=Group("1"))
def cmd4():
    pass

app()
```

Resulting help-page:

```
Usage: app COMMAND

╭─ 4 ──────────────────────────────────────────────────────────────────────────╮
│ cmd1                                                                         │
╰──────────────────────────────────────────────────────────────────────────────╯
╭─ 3 ──────────────────────────────────────────────────────────────────────────╮
│ cmd2                                                                         │
╰──────────────────────────────────────────────────────────────────────────────╯
╭─ 1 ──────────────────────────────────────────────────────────────────────────╮
│ cmd4                                                                         │
╰──────────────────────────────────────────────────────────────────────────────╯
╭─ 2 ──────────────────────────────────────────────────────────────────────────╮
│ cmd3                                                                         │
╰──────────────────────────────────────────────────────────────────────────────╯
╭─ Commands ───────────────────────────────────────────────────────────────────╮
│ --help,-h  Display this message and exit.                                    │
│ --version  Display application version.                                      │
╰──────────────────────────────────────────────────────────────────────────────╯
```

**default_parameter: *Parameter* | None = None**

Default *Parameter* in the parameter-resolution-stack that goes between *App.default_parameter* and the function signature's `Annotated` *Parameter*. The provided *Parameter* is not allowed to have a *group* value.

**validator: Callable | None = None**

A function (or list of functions) that validates an *ArgumentCollection*.

Example usage:

```python
def validator(argument_collection: ArgumentCollection):
    "Raise an exception if something is invalid."
```

The *ArgumentCollection* will contain all arguments that belong to that group. The validator(s) will **always be invoked**, regardless if any argument within the collection has token(s).

Validators are **not** invoked for command groups.

**classmethod create_ordered**(*name=''*, *help=''*, ***, *show=None*, *sort_key=None*, *validator=None*, *default_parameter=None*, *help_formatter=None*) → Self

Create a group with a globally incrementing *sort_key*.

Used to create a group that will be displayed **after** a previously instantiated *Group.create_ordered()* group on the help-page.

**Parameters**

- **name** (*str*) -- Group name used for the help-page and for group-referenced-by-string. This is a title, so the first character should be capitalized. If a name is not specified, it will not be shown on the help-page.
- **help** (*str*) -- Additional documentation shown on the help-page. This will be displayed inside the group's panel, above the parameters/commands.
- **show** (*bool | None*) -- Show this group on the help-page. Defaults to None, which will only show the group if a name is provided.
- **sort_key** (*Any*) -- If provided, **prepended** to the globally incremented counter value (i.e. has priority during sorting).
- **validator** (*None | Callable[[ArgumentCollection], Any] | Iterable[Callable[[ArgumentCollection], Any]]*) -- Group validator to collectively apply.
- **default_parameter** (cyclopts.Parameter *| None*) -- Default parameter for elements within the group.
- **help_formatter** (cyclopts.help.protocols.HelpFormatter *| None*) -- Custom help formatter for this group's help display.

**class** cyclopts.**Token**(***, *keyword: str | None = None*, *value: str = ''*, *source: str = ''*, *index: int = 0*, *keys: tuple[str, ...] = ()*, *implicit_value: ~typing.Any = <UNSET>*)

Tracks how a user supplied a value to the application.

**keyword:** str | None = None

**Unadulterated** user-supplied keyword like `--foo` or `--foo.bar.baz`; None when token was pared positionally. Could also be something like `tool.project.foo` if from non-cli sources.

**value:** str = ""

The parsed token value (unadulterated).

**source:** str = ""

Where the token came from; used for error message purposes. Cyclopts uses the string `cli` for cli-parsed tokens.

**index:** int = 0

The relative positional index in which the value was provided.

**keys:** tuple[str, ...] = ()

The additional parsed **python** variable keys from *keyword*.

Only used for Arguments that take arbitrary keys.

**implicit_value: Any = cyclopts.UNSET**

Final value that should be used instead of converting from *value*.

Commonly used for boolean flags.

Ignored if *UNSET*.

class cyclopts.field_info.**FieldInfo**(*names: tuple[str, ...] = ()*, *kind: _ParameterKind = _ParameterKind.POSITIONAL_OR_KEYWORD*, ***, *required: bool = False*, *default: Any*, *annotation: Any*, *help: str | None = None*)

Extension of `inspect.Parameter`.

class cyclopts.**Argument**(***, *tokens: list[~cyclopts.token.Token] = NOTHING*, *field_info: ~cyclopts.field_info.FieldInfo = NOTHING*, *parameter: ~cyclopts.parameter.Parameter = NOTHING*, *hint: ~typing.Any = <class 'str'>*, *index: int | None = None*, *keys: tuple[str, ...] = ()*, *value: ~typing.Any = <UNSET>*)

Encapsulates functionality and additional contextual information for parsing a parameter.

An argument is defined as anything that would have its own entry in the help page.

**tokens: list[*Token*]**

List of *Token* parsed from various sources. Do not directly mutate; see *append()*.

**field_info: *FieldInfo***

Additional information about the parameter from surrounding python syntax.

**parameter: *Parameter***

Fully resolved user-provided *Parameter*.

**hint: Any**

The type hint for this argument; may be different from `FieldInfo.annotation`.

**index: int | None**

Associated python positional index for argument. If `None`, then cannot be assigned positionally.

**keys: tuple[str, ...]**

**Python** keys that lead to this leaf.

`self.parameter.name` and `self.keys` can naively disagree! For example, a `self.parameter.name="--foo.bar.baz"` could be aliased to "--fizz". The resulting `self.keys` would be `("bar", "baz")`.

This is populated based on type-hints and class-structure, not `Parameter.name`.

```python
from cyclopts import App, Parameter
from dataclasses import dataclass
from typing import Annotated

app = App()


@dataclass
class User:
    id: int
    name: Annotated[str, Parameter(name="--fullname")]


@app.default
```

(continues on next page)

(continued from previous page)

```
def main(user: User):
    pass


for argument in app.assemble_argument_collection():
    print(f"name: {argument.name:16} hint: {str(argument.hint):16} keys:
↪{str(argument.keys)}")
```

```
$ my-script
name: --user.id        hint: <class 'int'>    keys: ('id',)
name: --fullname       hint: <class 'str'>    keys: ('name',)
```

**children:** *ArgumentCollection*

Collection of other *Argument* that eventually culminate into the python variable represented by *field_info*.

**property value**

Converted value from last *convert()* call.

This value may be stale if fields have changed since last *convert()* call. *UNSET* if *convert()* has not yet been called with tokens.

**property show_default:** bool | Callable[[Any], str]

Show the default value on the help page.

**match**(*term: str | int*, *, *transform: Callable[[str], str] | None = None*, *delimiter: str = '.'*) → tuple[tuple[str, ...], Any]

Match a name search-term, or a positional integer index.

**Raises**

**ValueError** -- If no match is found.

**Returns**

- *tuple[str, ...]* -- Leftover keys after matching to this argument. Used if this argument accepts_arbitrary_keywords.
- *Any* -- Implicit value. *UNSET* if no implicit value is applicable.

**append**(*token:* Token)

Safely add a *Token*.

**property has_tokens:** bool

This argument, or a child argument, has at least 1 parsed token.

**property children_recursive:** *ArgumentCollection*

**convert**(*converter: Callable | None = None*)

Converts *tokens* into *value*.

**Parameters**

**converter** (*Callable | None*) -- Converter function to use. Overrides `self.parameter.converter`

**Returns**

The converted data. Same as *value*.

**Return type**
Any

`validate`(*value*)

Validates provided value.

**Parameters**
**value** -- Value to validate.

**Returns**
The converted data. Same as *value*.

**Return type**
Any

`convert_and_validate`(*converter: Callable | None = None*)

Converts and validates *tokens* into *value*.

**Parameters**
**converter** (*Callable | None*) -- Converter function to use. Overrides `self.parameter.converter`

**Returns**
The converted data. Same as *value*.

**Return type**
Any

`token_count`(*keys: tuple[str, ...] = ()*)

The number of string tokens this argument consumes.

**Parameters**
**keys** (*tuple[str, ...]*) -- The **python** keys into this argument. If provided, returns the number of string tokens that specific data type within the argument consumes.

**Returns**

- *int* -- Number of string tokens to create 1 element.
- **consume_all** (*bool*) -- `True` if this data type is iterable.

`property negatives`

Negative flags from `Parameter.get_negatives()`.

`property name: str`

The **first** provided name this argument goes by.

`property names: tuple[str, ...]`

Names the argument goes by (both positive and negative).

`env_var_split`(*value: str, delimiter: str | None = None*) → list[str]

Split a given value with `Parameter.env_var_split()`.

`property show: bool`

Show this argument on the help page.

If an argument has child arguments, don't show it on the help-page. Returns False for arguments that won't be parsed (including underscore-prefixed params).

**property parse:** bool

Whether this argument should be parsed from CLI tokens.

If `Parameter.parse` is a regex pattern, parse if the pattern matches the field name; otherwise don't parse.

**property required:** bool

Whether or not this argument requires a user-provided value.

**is_positional_only**() → bool

**is_var_positional**() → bool

**is_flag**() → bool

Check if this argument is a flag (consumes no CLI tokens).

Flags are arguments that don't consume command-line tokens after the option name. They typically have implicit values (e.g., *--verbose* for bool, *--no-items* for list).

**Returns**

True if the argument consumes zero tokens from the command line.

**Return type**

bool

### Examples

```
>>> from cyclopts import Parameter
>>> bool_arg = Argument(hint=bool, parameter=Parameter(name="--verbose"))
>>> bool_arg.is_flag()
True
>>> str_arg = Argument(hint=str, parameter=Parameter(name="--name"))
>>> str_arg.is_flag()
False
```

**get_choices**(*force: bool = False*) → tuple[str, ...] | None

Extract completion choices from type hint.

Extracts choices from Literal types, Enum types, and Union types containing them. Respects the Parameter.show_choices setting unless force=True.

**Parameters**

**force** (*bool*) -- If True, return choices even when show_choices=False. Used by shell completion to always provide choices.

**Returns**

Tuple of choice strings if choices exist and should be shown, None otherwise.

**Return type**

tuple[str, ...] | None

### Examples

```
>>> argument = Argument(hint=Literal["dev", "staging", "prod"],␣
↪parameter=Parameter(show_choices=True))
>>> argument.get_choices()
('dev', 'staging', 'prod')
>>> argument = Argument(hint=Literal["dev", "staging", "prod"],␣
↪parameter=Parameter(show_choices=False))
```

(continues on next page)

(continued from previous page)

```
>>> argument.get_choices()  # Returns None for help text
>>> argument.get_choices(force=True)  # Returns choices for completion
('dev', 'staging', 'prod')
```

**class** cyclopts.**ArgumentCollection**(**args*)

A list-like container for *Argument*.

**copy**() → *ArgumentCollection*

Returns a shallow copy of the *ArgumentCollection*.

**__contains__**(*item: object*, /) → bool

Check if an argument or argument name exists in the collection.

**Parameters**

**item** (Argument | str) -- Either an Argument object or a string name/alias to search for.

**Returns**

True if the item is in the collection.

**Return type**

bool

### Examples

```
>>> argument_collection = ArgumentCollection(
...     [
...         Argument(parameter=Parameter(name="--foo")),
...         Argument(parameter=Parameter(name=("--bar", "-b"))),
...     ]
... )
>>> "--foo" in argument_collection
True
>>> "-b" in argument_collection  # Alias matching
True
>>> "--baz" in argument_collection
False
```

**get**(*term: str | int, default: type[<UNSET>] = <UNSET>, *, transform: ~collections.abc.Callable[[str], str] | None = None, delimiter: str = '.'*) → *Argument*

**get**(*term: str | int, default: T, *, transform: Callable[[str], str] | None = None, delimiter: str = '.'*) → *Argument* | T

Get an *Argument* by name or index.

This is a convenience wrapper around *match()* that returns just the *Argument* object instead of a tuple.

**Parameters**

- **term** (*str | int*) -- Either a string keyword or an integer positional index.
- **default** (*Any*) -- Default value to return if term not found. If *UNSET* (default), will raise KeyError/IndexError.
- **transform** (*Callable[[str], str] | None*) -- Optional function to transform string terms before matching.
- **delimiter** (*str*) -- Delimiter for nested field access.

**Returns**
The matched *Argument*, or `default` if provided and not found.

**Return type**
*Argument* | None

**Raises**

- **KeyError** -- If `term` is a string and not found (when `default` is *UNSET*).
- **IndexError** -- If `term` is an int and is out-of-range (when `default` is *UNSET*).

**See also**

*match()*
Returns a tuple of (*Argument*, keys, value) with more detailed information.

**match**(*term: str | int*, *, *transform: Callable[[str], str] | None = None*, *delimiter: str = '.'*) → tuple[*Argument*, tuple[str, ...], Any]

Matches CLI keyword or index to their *Argument*.

**Parameters**
**term** (*str | int*) -- One of:

- `str` keyword like `"--foo"` or `"-f"` or `"--foo.bar.baz"`.
- `int` global positional index.

**Raises**
**ValueError** -- If the provided `term` doesn't match.

**Returns**

- *Argument* -- Matched *Argument*.
- *tuple[str, ...]* -- Python keys into *Argument*. Non-empty iff *Argument* accepts keys.
- *Any* -- Implicit value (if a flag). *UNSET* otherwise.

**property groups**

**filter_by**(*, *group: Group | None = None*, *has_tokens: bool | None = None*, *has_tree_tokens: bool | None = None*, *keys_prefix: tuple[str, ...] | None = None*, *kind: _ParameterKind | None = None*, *parse: bool | None = None*, *show: bool | None = None*, *value_set: bool | None = None*) → *ArgumentCollection*

Filter the *ArgumentCollection*.

All non-None filters will be applied.

**Parameters**

- **group** (*Group | None*) -- The *Group* the arguments should be in.
- **has_tokens** (*bool | None*) -- Immediately has tokens (not including children).
- **has_tree_tokens** (*bool | None*) -- *Argument* and/or it's children have parsed tokens.
- **kind** (*inspect._ParameterKind | None*) -- The `kind` of the argument.
- **parse** (*bool | None*) -- If the argument is intended to be parsed or not.
- **show** (*bool | None*) -- The *Argument* is intended to be show on the help page.

- **value_set** (*bool | None*) -- The converted value is set.

**class** cyclopts.**UNSET**

Special sentinel value indicating that no data was provided. **Do not instantiate**.

cyclopts.**default_name_transform**(*s: str*) → str

Converts a python identifier into a CLI token.

Performs the following operations (in order):

1. Convert PascalCase to snake_case.
2. Convert the string to all lowercase.
3. Replace _ with -.
4. Strip any leading/trailing - (also stripping _, due to point 3).

Intended to be used with *App.name_transform* and *Parameter.name_transform*.

**Parameters**
**s** (*str*) -- Input python identifier string.

**Returns**
Transformed name.

**Return type**
str

cyclopts.**env_var_split**(*type_: Any*, *val: str*, ***, *delimiter: str | None = None*) → list[str]

Type-dependent environment variable value splitting.

Converts a single string into a list of strings. Splits when:

- The `type_` is some variant of `Iterable[pathlib.Path]` objects. If Windows, split on `;`, otherwise split on `:`.
- Otherwise, if the `type_` is an `Iterable`, split on whitespace. Leading/trailing whitespace of each output element will be stripped.

This function is the default value for `cyclopts.App.env_var_split`.

**Parameters**

- **type** (*type*) -- Type hint that we will eventually coerce into.
- **val** (*str*) -- String to split.
- **delimiter** (*str | None*) -- Delimiter to split `val` on. If None, defaults to whitespace.

**Returns**
List of individual string tokens.

**Return type**
list[str]

cyclopts.**edit**(*initial_text: str = ''*, ***, *fallback_editors: Sequence[str] = ('nano', 'vim', 'notepad', 'gedit')*, *editor_args: Sequence[str] = ()*, *path: str | Path = ''*, *encoding: str = 'utf-8'*, *save: bool = True*, *required: bool = True*) → str

Get text input from a user by launching their default text editor.

**Parameters**

- **initial_text** (*str*) -- Initial text to populate the text file with.

- **fallback_editors** (*Sequence[str]*) -- If the text editor cannot be determined from the environment variable `EDITOR`, attempt to use these text editors in the order provided.
- **editor_args** (*Sequence[str]*) -- Additional CLI arguments that are passed along to the editor-launch command.
- **path** (*Union[str, Path]*) -- If specified, the path to the file that should be opened. Text editors typically display this, so a custom path may result in a better user-interface. Defaults to a temporary text file.
- **encoding** (*str*) -- File encoding to use.
- **save** (*bool*) -- **Require** the user to save before exiting the editor. Otherwise raises *EditorDidNotSaveError*.
- **required** (*bool*) -- **Require** for the saved text to be different from `initial_text`. Otherwise raises *EditorDidNotChangeError*.

**Raises**

- ***EditorError*** -- Base editor error exception. Explicitly raised if editor subcommand returned a non-zero exit code.
- ***EditorNotFoundError*** -- A suitable text editor could not be found.
- ***EditorDidNotSaveError*** -- The user exited the text-editor without saving and `save=True`.
- ***EditorDidNotChangeError*** -- The user did not change the file contents and `required=True`.

**Returns**

The resulting text that was saved by the text editor.

**Return type**

str

cyclopts.**run**(*callable: Callable[[...], Coroutine[None, None, V]]*, */*, ***, *result_action: Literal['return_value']*) → V

cyclopts.**run**(*callable: Callable[[...], V]*, */*, ***, *result_action: Literal['return_value']*) → V

cyclopts.**run**(*callable: Callable[[...], Coroutine[None, None, Any]]*, */*, ***, *result_action: Literal['return_value', 'call_if_callable', 'print_non_int_return_int_as_exit_code', 'print_str_return_int_as_exit_code', 'print_str_return_zero', 'print_non_none_return_int_as_exit_code', 'print_non_none_return_zero', 'return_int_as_exit_code_else_zero', 'print_non_int_sys_exit', 'sys_exit', 'return_none', 'return_zero', 'print_return_zero', 'sys_exit_zero', 'print_sys_exit_zero'] | Callable[[Any], Any] | Iterable[Literal['return_value', 'call_if_callable', 'print_non_int_return_int_as_exit_code', 'print_str_return_int_as_exit_code', 'print_str_return_zero', 'print_non_none_return_int_as_exit_code', 'print_non_none_return_zero', 'return_int_as_exit_code_else_zero', 'print_non_int_sys_exit', 'sys_exit', 'return_none', 'return_zero', 'print_return_zero', 'sys_exit_zero', 'print_sys_exit_zero'] | Callable[[Any], Any]] | None = None*) → Any

cyclopts.**run**(*callable: Callable[[...], Any]*, */*, ***, *result_action: Literal['return_value', 'call_if_callable', 'print_non_int_return_int_as_exit_code', 'print_str_return_int_as_exit_code', 'print_str_return_zero', 'print_non_none_return_int_as_exit_code', 'print_non_none_return_zero', 'return_int_as_exit_code_else_zero', 'print_non_int_sys_exit', 'sys_exit', 'return_none', 'return_zero', 'print_return_zero', 'sys_exit_zero', 'print_sys_exit_zero'] | Callable[[Any], Any] | Iterable[Literal['return_value', 'call_if_callable', 'print_non_int_return_int_as_exit_code', 'print_str_return_int_as_exit_code', 'print_str_return_zero', 'print_non_none_return_int_as_exit_code', 'print_non_none_return_zero', 'return_int_as_exit_code_else_zero', 'print_non_int_sys_exit', 'sys_exit', 'return_none', 'return_zero', 'print_return_zero', 'sys_exit_zero', 'print_sys_exit_zero'] | Callable[[Any], Any]] | None = None*) → Any

Run the given callable as a CLI command.

The callable may also be a coroutine function. This function is syntax sugar for very simple use cases, and is roughly equivalent to:

```
from cyclopts import App

app = App()
app.default(callable)
app()
```

**Parameters**

- **callable** -- The function to execute as a CLI command.
- **result_action** -- How to handle the command's return value. If not specified, uses the default `"print_non_int_sys_exit"` which calls `sys.exit()` with the appropriate code. Can be set to `"return_value"` to return the result directly for testing/embedding.
- **usage** (*Example*)
- **code-block:** (`..`) -- python: import cyclopts

  **def main(name: str, age: int):**
  print(f"Hello {name}, you are {age} years old.")

  cyclopts.run(main)

class cyclopts.**CycloptsPanel**(*message: Any*, *title: str = 'Error'*, *style: str = 'red'*)

Create a `Panel` with a consistent style.

The resulting panel can be displayed using a `Console`.

```
╭─ Title ──────────────────────────────────╮
│ Message content here.                    │
╰──────────────────────────────────────────╯
```

**Parameters**

- **message** (*Any*) -- The body of the panel will be filled with the stringified version of the message.
- **title** (*str*) -- Title of the panel that appears in the top-left corner.
- **style** (*str*) -- Rich style for the panel border.

**Returns**

Formatted panel object.

**Return type**
*Panel*

## 18.1 Validators

Cyclopts has several builtin validators for common CLI inputs.

**class** cyclopts.validators.**LimitedChoice**(*min: int = 0*, *max: int | None = None*, *allow_none: bool = False*)

Group validator that limits the number of selections per group.

Commonly used for enforcing mutually-exclusive parameters (default behavior).

**Parameters**

- **min** (*int*) -- The minimum (inclusive) number of CLI parameters allowed. If negative, then **all** parameters in the group must have CLI values provided.
- **max** (*int | None*) -- The maximum (inclusive) number of CLI parameters allowed. Defaults to 1 if `min==0`, `min` otherwise.
- **allow_none** (*bool*) -- If `True`, also allow 0 CLI parameters (even if `min` is greater than 0). Defaults to `False`.

**class** cyclopts.validators.**MutuallyExclusive**

Alias for *LimitedChoice* to make intentions more obvious.

Only 1 argument in the group can be supplied a value.

cyclopts.validators.**mutually_exclusive = <cyclopts.validators._group.MutuallyExclusive object>**

Instantiated version of *MutuallyExclusive*. Can be used directly in group validators:

```
import cyclopts
from cyclopts import Group

mutually_exclusive_group = Group(validator=cyclopts.validators.mutually_exclusive)
```

cyclopts.validators.**all_or_none = <cyclopts.validators._group.LimitedChoice object>**

Group validator that enforces that either all parameters in the group must be supplied an argument, or none of them.

**class** cyclopts.validators.**Number**(**, lt: int | float | None = None, lte: int | float | None = None, gt: int | float | None = None, gte: int | float | None = None, modulo: int | float | None = None*)

Limit input number to a value range.

Example Usage:

```
from cyclopts import App, Parameter, validators
from typing import Annotated

app = App()


@app.default
def main(age: Annotated[int, Parameter(validator=validators.Number(gte=0,␣
```

(continues on next page)

(continued from previous page)

```
↪lte=150))]):
    print(f"You are {age} years old.")


app()
```

```
$ my-script 100
You are 100 years old.

$ my-script -1
╭─ Error ──────────────────────────────────────────────────────────────────────╮
│ Invalid value "-1" for "AGE". Must be >= 0.                                  │
╰──────────────────────────────────────────────────────────────────────────────╯

$ my-script 200
╭─ Error ──────────────────────────────────────────────────────────────────────╮
│ Invalid value "200" for "AGE". Must be <= 150.                               │
╰──────────────────────────────────────────────────────────────────────────────╯
```

**lt:** `int | float | None`

Input value must be **less than** this value.

**lte:** `int | float | None`

Input value must be **less than or equal** this value.

**gt:** `int | float | None`

Input value must be **greater than** this value.

**gte:** `int | float | None`

Input value must be **greater than or equal** this value.

**modulo:** `int | float | None`

Input value must be a multiple of this value.

**class** cyclopts.validators.**Path**(*, *exists: bool = False*, *file_okay: bool = True*, *dir_okay: bool = True*, *ext: None | Any | Iterable[Any] = None*)

Assertions on properties of `pathlib.Path`.

Example Usage:

```
from cyclopts import App, Parameter, validators
from pathlib import Path
from typing import Annotated

app = App()


@app.default
def main(
    # ``src`` must be a file that exists.
    src: Annotated[Path, Parameter(validator=validators.Path(exists=True, dir_
↪okay=False))],
    # ``dst`` must be a path that does **not** exist.
```

(continues on next page)

(continued from previous page)

```
    dst: Annotated[Path, Parameter(validator=validators.Path(dir_okay=False, file_
↪okay=False))],
):
    "Copies src->dst."
    dst.write_bytes(src.read_bytes())


app()
```

```
$ my-script foo.bin bar.bin  # if foo.bin does not exist
╭─ Error ──────────────────────────────────────────────────────────────────────╮
│ Invalid value "foo.bin" for "SRC". "foo.bin" does not exist.                 │
╰──────────────────────────────────────────────────────────────────────────────╯

$ my-script foo.bin bar.bin  # if bar.bin exists
╭─ Error ──────────────────────────────────────────────────────────────────────╮
│ Invalid value "bar.bin" for "DST". "bar.bin" already exists.                 │
╰──────────────────────────────────────────────────────────────────────────────╯
```

**exists:** `bool`

If `True`, specified path **must** exist. Defaults to `False`.

**file_okay:** `bool`

If path exists, check it's type:

- If `True`, specified path may be an **existing** file.
- If `False`, then **existing** files are not allowed.

Defaults to `True`.

**dir_okay:** `bool`

If path exists, check it's type:

- If `True`, specified path may be an **existing** directory.
- If `False`, then **existing** directories are not allowed.

Defaults to `True`.

**ext:** `str | Sequence[str]`

Supplied path must have this extension (case insensitive). May or may not include the ".".

## 18.2 Types

Cyclopts has builtin pre-defined annotated-types for common conversion and validation configurations. Most definitions in this section are simply predefined annotations for convenience:

```
Annotated[..., Parameter(...)]
```

Custom classes that provide additional functionality beyond simple annotations will be noted.

Due to Cyclopts's advanced *Parameter* resolution engine, these annotations can themselves be annotated to further configure behavior. E.g:

```
Annotated[PositiveInt, Parameter(...)]
```

### 18.2.1 Path

Path annotated types for checking existence, type, and performing path-resolution. All of these types will also work on sequence of paths (e.g. `tuple[Path, Path]` or `list[Path]`).

**class** cyclopts.types.**StdioPath**

> **Note**
>
> This is a custom class, not a simple `Annotated` type alias.
>
> Requires **Python 3.12+** due to `Path` subclassing support.

A `Path` subclass that treats `-` as stdin (for reading) or stdout (for writing). This follows common Unix convention.

`StdioPath` is pre-configured with `allow_leading_hyphen=True`, so `-` can be passed as an argument without being interpreted as an option.

**STDIO_STRING: str = "-"**

Class attribute defining the string that triggers stdio behavior. Override in subclasses to use a different string.

**is_stdio: bool**

Returns `True` if this path represents stdin/stdout (i.e., `str(self) == STDIO_STRING`). Override this property in subclasses for custom matching logic (e.g., matching multiple strings).

Basic usage:

```python
from cyclopts import App
from cyclopts.types import StdioPath

app = App()

@app.default
def main(input_file: StdioPath):
    data = input_file.read_text()
    print(data.upper())

app()
```

```
$ echo "hello" | python my_script.py -
HELLO

$ python my_script.py data.txt
<contents of data.txt uppercased>
```

To default to stdin/stdout when no argument is provided:

```python
@app.default
def main(input_file: StdioPath = StdioPath("-")):
    data = input_file.read_text()
    print(data.upper())
```

See *Reading/Writing From File or Stdin/Stdout* for more examples.

**Subclassing**

To use a different trigger string or custom matching logic, subclass `StdioPath`:

```
from cyclopts.types import StdioPath

# Simple: different trigger string
class StdinPath(StdioPath):
    STDIO_STRING = "STDIN"

class StdoutPath(StdioPath):
    STDIO_STRING = "STDOUT"

# Advanced: match multiple strings
class MultiStdioPath(StdioPath):
    @property
    def is_stdio(self) -> bool:
        return str(self) in ("-", "STDIN", "STDOUT")
```

`cyclopts.types.`**`ExistingPath`**

A `Path` file or directory that **must** exist.

alias of `Annotated`[`Path`, Parameter(validator=(Path(exists=True, file_okay=True, dir_okay=True, ext=()),))]

`cyclopts.types.`**`NonExistentPath`**

A `Path` file or directory that **must not** exist.

alias of `Annotated`[`Path`, Parameter(validator=(Path(exists=False, file_okay=False, dir_okay=False, ext=()),))]

`cyclopts.types.`**`ResolvedPath`**

A `Path` file or directory. `resolve()` is invoked prior to returning the path.

alias of `Annotated`[`Path`, Parameter(converter=<function _path_resolve_converter at 0x74d0ff96cee0>)]

`cyclopts.types.`**`ResolvedExistingPath`**

A `Path` file or directory that **must** exist. `resolve()` is invoked prior to returning the path.

alias of `Annotated`[`Path`, Parameter(validator=(Path(exists=True, file_okay=True, dir_okay=True, ext=()),)), Parameter(converter=<function _path_resolve_converter at 0x74d0ff96cee0>)]

`cyclopts.types.`**`Directory`**

A `Path` that **must** be a directory (or not exist).

alias of `Annotated`[`Path`, Parameter(validator=(Path(exists=False, file_okay=False, dir_okay=True, ext=()),))]

`cyclopts.types.`**`ExistingDirectory`**

A `Path` directory that **must** exist.

alias of `Annotated`[`Path`, Parameter(validator=(Path(exists=True, file_okay=False, dir_okay=True, ext=()),))]

`cyclopts.types.`**`NonExistentDirectory`**

A `Path` directory that **must not** exist.

alias of `Annotated`[`Path`, Parameter(validator=(Path(exists=False, file_okay=False, dir_okay=False, ext=()),))]

`cyclopts.types.`**`ResolvedDirectory`**

A `Path` directory. `resolve()` is invoked prior to returning the path.

alias of Annotated[Path, Parameter(validator=(Path(exists=False, file_okay=False, dir_okay=True, ext=()),)), Parameter(converter=<function _path_resolve_converter at 0x74d0ff96cee0>)]

cyclopts.types.**ResolvedExistingDirectory**

A Path directory that **must** exist. resolve() is invoked prior to returning the path.

alias of Annotated[Path, Parameter(validator=(Path(exists=True, file_okay=False, dir_okay=True, ext=()),)), Parameter(converter=<function _path_resolve_converter at 0x74d0ff96cee0>)]

cyclopts.types.**File**

A File that **must** be a file (or not exist).

alias of Annotated[Path, Parameter(validator=(Path(exists=False, file_okay=True, dir_okay=False, ext=()),))]

cyclopts.types.**ExistingFile**

A Path file that **must** exist.

alias of Annotated[Path, Parameter(validator=(Path(exists=True, file_okay=True, dir_okay=False, ext=()),))]

cyclopts.types.**NonExistentFile**

A Path file that **must not** exist.

alias of Annotated[Path, Parameter(validator=(Path(exists=False, file_okay=False, dir_okay=False, ext=()),))]

cyclopts.types.**ResolvedFile**

A Path file. resolve() is invoked prior to returning the path.

alias of Annotated[Path, Parameter(validator=(Path(exists=False, file_okay=True, dir_okay=False, ext=()),)), Parameter(converter=<function _path_resolve_converter at 0x74d0ff96cee0>)]

cyclopts.types.**ResolvedExistingFile**

A Path file that **must** exist. resolve() is invoked prior to returning the path.

alias of Annotated[Path, Parameter(validator=(Path(exists=True, file_okay=True, dir_okay=False, ext=()),)), Parameter(converter=<function _path_resolve_converter at 0x74d0ff96cee0>)]

cyclopts.types.**BinPath**

A Path that **must** have extension bin.

alias of Annotated[Path, Parameter(validator=(Path(exists=False, file_okay=True, dir_okay=False, ext=('bin',)),))]

cyclopts.types.**ExistingBinPath**

A Path that **must** exist and have extension bin.

alias of Annotated[Path, Parameter(validator=(Path(exists=True, file_okay=True, dir_okay=False, ext=('bin',)),))]

cyclopts.types.**NonExistentBinPath**

A Path that **must not** exist and have extension bin.

alias of Annotated[Path, Parameter(validator=(Path(exists=False, file_okay=False, dir_okay=False, ext=('bin',)),))]

cyclopts.types.**CsvPath**

A Path that **must** have extension csv.

alias of Annotated[Path, Parameter(validator=(Path(exists=False, file_okay=True, dir_okay=False, ext=('csv',)),))]

cyclopts.types.**ExistingCsvPath**

A Path that **must** exist and have extension csv.

alias of Annotated[Path, Parameter(validator=(Path(exists=True, file_okay=True, dir_okay=False, ext=('csv',)),))]

cyclopts.types.**NonExistentCsvPath**

A Path that **must not** exist and have extension csv.

alias of Annotated[Path, Parameter(validator=(Path(exists=False, file_okay=False, dir_okay=False, ext=('csv',)),))]

cyclopts.types.**TxtPath**

A Path that **must** have extension txt.

alias of Annotated[Path, Parameter(validator=(Path(exists=False, file_okay=True, dir_okay=False, ext=('txt',)),))]

cyclopts.types.**ExistingTxtPath**

A Path that **must** exist and have extension txt.

alias of Annotated[Path, Parameter(validator=(Path(exists=True, file_okay=True, dir_okay=False, ext=('txt',)),))]

cyclopts.types.**NonExistentTxtPath**

A Path that **must not** exist and have extension txt.

alias of Annotated[Path, Parameter(validator=(Path(exists=False, file_okay=False, dir_okay=False, ext=('txt',)),))]

cyclopts.types.**ImagePath**

A Path that **must** have extension in {png, jpg, jpeg}.

alias of Annotated[Path, Parameter(validator=(Path(exists=False, file_okay=True, dir_okay=False, ext=('png', 'jpg', 'jpeg')),))]

cyclopts.types.**ExistingImagePath**

A Path that **must** exist and have extension in {png, jpg, jpeg}.

alias of Annotated[Path, Parameter(validator=(Path(exists=True, file_okay=True, dir_okay=False, ext=('png', 'jpg', 'jpeg')),))]

cyclopts.types.**NonExistentImagePath**

A Path that **must not** exist and have extension in {png, jpg, jpeg}.

alias of Annotated[Path, Parameter(validator=(Path(exists=False, file_okay=False, dir_okay=False, ext=('png', 'jpg', 'jpeg')),))]

cyclopts.types.**Mp4Path**

A Path that **must** have extension mp4.

alias of Annotated[Path, Parameter(validator=(Path(exists=False, file_okay=True, dir_okay=False, ext=('mp4',)),))]

cyclopts.types.**ExistingMp4Path**

A Path that **must** exist and have extension mp4.

alias of Annotated[Path, Parameter(validator=(Path(exists=True, file_okay=True, dir_okay=False, ext=('mp4',)),))]

cyclopts.types.**NonExistentMp4Path**

A Path that **must not** exist and have extension mp4.

alias of Annotated[Path, Parameter(validator=(Path(exists=False, file_okay=False, dir_okay=False, ext=('mp4',)),))]

cyclopts.types.**JsonPath**

A Path that **must** have extension json.

alias of Annotated[Path, Parameter(validator=(Path(exists=False, file_okay=True, dir_okay=False, ext=('json',)),))]

cyclopts.types.**ExistingJsonPath**

A Path that **must** exist and have extension json.

alias of Annotated[Path, Parameter(validator=(Path(exists=True, file_okay=True, dir_okay=False, ext=('json',)),))]

cyclopts.types.**NonExistentJsonPath**

A Path that **must not** exist and have extension json.

alias of Annotated[Path, Parameter(validator=(Path(exists=False, file_okay=False, dir_okay=False, ext=('json',)),))]

cyclopts.types.**TomlPath**

A Path that **must** have extension toml.

alias of Annotated[Path, Parameter(validator=(Path(exists=False, file_okay=True, dir_okay=False, ext=('toml',)),))]

cyclopts.types.**ExistingTomlPath**

A Path that **must** exist and have extension toml.

alias of Annotated[Path, Parameter(validator=(Path(exists=True, file_okay=True, dir_okay=False, ext=('toml',)),))]

cyclopts.types.**NonExistentTomlPath**

A Path that **must not** exist and have extension toml.

alias of Annotated[Path, Parameter(validator=(Path(exists=False, file_okay=False, dir_okay=False, ext=('toml',)),))]

cyclopts.types.**YamlPath**

A Path that **must** have extension yaml.

alias of Annotated[Path, Parameter(validator=(Path(exists=False, file_okay=True, dir_okay=False, ext=('yaml',)),))]

cyclopts.types.**ExistingYamlPath**

A Path that **must** exist and have extension yaml.

alias of Annotated[Path, Parameter(validator=(Path(exists=True, file_okay=True, dir_okay=False, ext=('yaml',)),))]

cyclopts.types.**NonExistentYamlPath**

A Path that **must not** exist and have extension yaml.

alias of Annotated[Path, Parameter(validator=(Path(exists=False, file_okay=False, dir_okay=False, ext=('yaml',)),))]

### 18.2.2 Number

Annotated types for checking common int/float value constraints. All of these types will also work on sequence of numbers (e.g. `tuple[int, int]` or `list[float]`).

`cyclopts.types.`**PositiveFloat**

A float that **must** be >0.

alias of `Annotated`[`float`, Parameter(validator=(Number(lt=None, lte=None, gt=0, gte=None, modulo=None),))]

`cyclopts.types.`**NonNegativeFloat**

A float that **must** be >=0.

alias of `Annotated`[`float`, Parameter(validator=(Number(lt=None, lte=None, gt=None, gte=0, modulo=None),))]

`cyclopts.types.`**NegativeFloat**

A float that **must** be <0.

alias of `Annotated`[`float`, Parameter(validator=(Number(lt=0, lte=None, gt=None, gte=None, modulo=None),))]

`cyclopts.types.`**NonPositiveFloat**

A float that **must** be <=0.

alias of `Annotated`[`float`, Parameter(validator=(Number(lt=None, lte=0, gt=None, gte=None, modulo=None),))]

`cyclopts.types.`**PositiveInt**

An int that **must** be >0.

alias of `Annotated`[`int`, Parameter(validator=(Number(lt=None, lte=None, gt=0, gte=None, modulo=None),))]

`cyclopts.types.`**NonNegativeInt**

An int that **must** be >=0.

alias of `Annotated`[`int`, Parameter(validator=(Number(lt=None, lte=None, gt=None, gte=0, modulo=None),))]

`cyclopts.types.`**NegativeInt**

An int that **must** be <0.

alias of `Annotated`[`int`, Parameter(validator=(Number(lt=0, lte=None, gt=None, gte=None, modulo=None),))]

`cyclopts.types.`**NonPositiveInt**

An int that **must** be <=0.

alias of `Annotated`[`int`, Parameter(validator=(Number(lt=None, lte=0, gt=None, gte=None, modulo=None),))]

`cyclopts.types.`**UInt8**

An unsigned 8-bit integer.

alias of `Annotated`[`int`, Parameter(validator=(Number(lt=None, lte=255, gt=None, gte=0, modulo=None),))]

`cyclopts.types.`**HexUInt8**

An unsigned 8-bit integer who's default value will be displayed as hexadecimal in the help-page.

alias of `Annotated`[`int`, Parameter(validator=(Number(lt=None, lte=255, gt=None, gte=0, modulo=None),)), Parameter(show_default=functools.partial(<function _hex_formatter at 0x74d0ff96fbe0>, digits=2))]

cyclopts.types.**Int8**

A signed 8-bit integer.

alias of Annotated[int, Parameter(validator=(Number(lt=None, lte=127, gt=None, gte=-128, modulo=None),))]

cyclopts.types.**UInt16**

An unsigned 16-bit integer.

alias of Annotated[int, Parameter(validator=(Number(lt=None, lte=65535, gt=None, gte=0, modulo=None),))]

cyclopts.types.**HexUInt16**

An unsigned 16-bit integer who's default value will be displayed as hexadecimal in the help-page.

alias of Annotated[int, Parameter(validator=(Number(lt=None, lte=65535, gt=None, gte=0, modulo=None),)), Parameter(show_default=functools.partial(<function _hex_formatter at 0x74d0ff96fbe0>, digits=4))]

cyclopts.types.**Int16**

A signed 16-bit integer.

alias of Annotated[int, Parameter(validator=(Number(lt=None, lte=32767, gt=None, gte=-32768, modulo=None),))]

cyclopts.types.**UInt32**

An unsigned 32-bit integer.

alias of Annotated[int, Parameter(validator=(Number(lt=4294967296, lte=None, gt=None, gte=0, modulo=None),))]

cyclopts.types.**HexUInt32**

An unsigned 32-bit integer who's default value will be displayed as hexadecimal in the help-page.

alias of Annotated[int, Parameter(validator=(Number(lt=4294967296, lte=None, gt=None, gte=0, modulo=None),)), Parameter(show_default=functools.partial(<function _hex_formatter at 0x74d0ff96fbe0>, digits=8))]

cyclopts.types.**Int32**

A signed 32-bit integer.

alias of Annotated[int, Parameter(validator=(Number(lt=2147483648, lte=None, gt=None, gte=-2147483648, modulo=None),))]

cyclopts.types.**UInt64**

An unsigned 64-bit integer.

alias of Annotated[int, Parameter(validator=(Number(lt=18446744073709551616, lte=None, gt=None, gte=0, modulo=None),))]

cyclopts.types.**HexUInt64**

An unsigned 64-bit integer who's default value will be displayed as hexadecimal in the help-page.

alias of Annotated[int, Parameter(validator=(Number(lt=18446744073709551616, lte=None, gt=None, gte=0, modulo=None),)), Parameter(show_default=functools.partial(<function _hex_formatter at 0x74d0ff96fbe0>, digits=16))]

cyclopts.types.**Int64**

A signed 64-bit integer.

alias of Annotated[int, Parameter(validator=(Number(lt=9223372036854775808, lte=None, gt=None, gte=-9223372036854775808, modulo=None),))]

### 18.2.3 Json

Annotated types for parsing a json-string from the CLI.

cyclopts.types.**Json**

Parse a json-string from the CLI.

Note: Since Cyclopts v3.6.0, all dataclass-like classes now natively attempt to parse json-strings, so practical use-case of this annotation is limited.

Usage example:

```
from cyclopts import App, types

app = App()

@app.default
def main(json: types.Json):
    print(json)

app()
```

```
$ my-script '{"foo": 1, "bar": 2}'
{'foo': 1, 'bar': 2}
```

alias of Annotated[Any, Parameter(converter=<function _json_converter at 0x74d0ff96e9e0>)]

### 18.2.4 Web

Annotated types for common web-related values.

cyclopts.types.**Email**

An email address string with simple validation.

alias of Annotated[str, Parameter(validator=(<function _email_validator at 0x74d0ff96e3b0>,))]

cyclopts.types.**Port**

An int limited to range [0, 65535].

alias of Annotated[int, Parameter(validator=(Number(lt=None, lte=65535, gt=None, gte=0, modulo=None),))]

cyclopts.types.**URL**

A str URL string with some simple validation.

alias of Annotated[str, Parameter(validator=(<function _url_validator at 0x74d0ff96e440>,))]

.. autodata:: cyclopts.types.URL

## 18.3 Help Formatting

Cyclopts provides a flexible help formatting system for customizing the help-page's appearance.

**class** cyclopts.help.protocols.**HelpFormatter**(**args*, ***kwargs*)

Protocol for help **formatter** functions.

It's the Formatter's job to transform a *HelpPanel* into rendered text on the display.

Implementations may optionally provide the following methods for custom rendering of "usage" and "description". If these methods are not provided, default rendering will be used.

```
def render_usage(self, console: Console, options: ConsoleOptions, usage: Any) ->␣
↪None:
    """Render the usage line."""
    ...

def render_description(self, console: Console, options: ConsoleOptions,␣
↪description: Any) -> None:
    """Render the description."""
    ...
```

**__call__**(*console: Console*, *options: ConsoleOptions*, *panel:* HelpPanel) → None

Format and render a single help panel.

**Parameters**

- **console** (*Console*) -- Console to render to.
- **options** (*ConsoleOptions*) -- Console rendering options.
- **panel** (HelpPanel) -- Help panel to render (commands, parameters, etc).

**class** cyclopts.help.**DefaultFormatter**(*, *panel_spec:* PanelSpec | *None* = *None*, *table_spec:* TableSpec | *None* = *None*, *column_specs:* *tuple*[ColumnSpec, ...] | ColumnSpecBuilder | *None* = *None*)

Default help formatter using Rich library with customizable specs.

**Parameters**

- **panel_spec** (*Optional*[PanelSpec]) -- Panel specification for the outer box/panel styling.
- **table_spec** (*Optional*[TableSpec]) -- Table specification for table styling (borders, padding, etc).
- **column_specs** (*Optional*[*Union*[*tuple*[ColumnSpec, ...], ColumnSpecBuilder]]) -- Column specifications or builder function for table columns.

### Notes

The relationship between these specs can be visualized as:

```
─ Commands ──────────────────────────────────────────────────────────  ← panel_spec
│ serve     Start the development server                            │     (border,␣
↪title)
│ --help    Display this message and exit.                          │
──────────────────────────────────────────────────────────────────────
 ↑         ↑
 col[0]    col[1]
 (name)    (description)

─ Parameters ────────────────────────────────────────────────────────  ← panel_spec
│ *  PORT --port        Server port number [required]                 │
│    VERBOSE --verbose  Enable verbose output [default: False]        │
──────────────────────────────────────────────────────────────────────
 ↑  ↑                  ↑
 │  col[1]             col[2]
 │  (name/flags)       (description)
```

(continues on next page)

(continued from previous page)

```
|
col[0]
(required marker)
```

Where:

- `panel_spec` controls the outer panel appearance (border, title, etc.)
- `table_spec` controls the inner table styling (no visible borders by default)
- `column_specs` defines individual columns (width, style, alignment, etc.)

**panel_spec:** *PanelSpec* | None

Panel specification for the outer box/panel styling (border, title, padding, etc).

**table_spec:** *TableSpec* | None

Table specification for table styling (borders, padding, column separation, etc).

**column_specs:** tuple[*ColumnSpec*, ...] | *ColumnSpecBuilder* | None

Column specifications or builder function for table columns (width, style, alignment, etc).

**classmethod with_newline_metadata**(***kwargs*)

Create formatter with metadata on separate lines.

Returns a DefaultFormatter configured to display parameter metadata (choices, env vars, defaults) on separate indented lines rather than inline with descriptions.

**Parameters**

****kwargs** -- Additional keyword arguments to pass to DefaultFormatter constructor.

**Returns**

Configured formatter instance with newline metadata display.

**Return type**

*DefaultFormatter*

### Examples

```
>>> from cyclopts import App
>>> from cyclopts.help import DefaultFormatter
>>> app = App(help_formatter=DefaultFormatter.with_newline_metadata())
```

**__call__**(*console: Console*, *options: ConsoleOptions*, *panel:* HelpPanel) → None

Format and render a single help panel using Rich.

**Parameters**

- **console** (*Console*) -- Console to render to.
- **options** (*ConsoleOptions*) -- Console rendering options.
- **panel** (HelpPanel) -- Help panel to render.

**render_usage**(*console: Console*, *options: ConsoleOptions*, *usage: Any*) → None

Render the usage line.

**Parameters**

- **console** (*Console*) -- Console to render to.
- **options** (*ConsoleOptions*) -- Console rendering options.

- **usage** (*Any*) -- The usage line (Text or str).

**render_description**(*console: Console*, *options: ConsoleOptions*, *description: Any*) → None

Render the description.

**Parameters**

- **console** (*Console*) -- Console to render to.
- **options** (*ConsoleOptions*) -- Console rendering options.
- **description** (*Any*) -- The description (can be various Rich renderables).

**class** cyclopts.help.**PlainFormatter**(*indent_width: int = 2*, *max_width: int | None = None*)

Plain text formatter for improved accessibility.

**Parameters**

- **indent_width** (*int*) -- Number of spaces to indent entries (default: 2).
- **max_width** (*Optional[int]*) -- Maximum line width for wrapping text.

**__call__**(*console: Console*, *options: ConsoleOptions*, *panel: HelpPanel*) → None

Format and render a single help panel as plain text.

**Parameters**

- **console** (*Console*) -- Console to render to.
- **options** (*ConsoleOptions*) -- Console rendering options.
- **panel** (*HelpPanel*) -- Help panel to render.

**render_usage**(*console: Console*, *options: ConsoleOptions*, *usage: Any*) → None

Render the usage line.

**Parameters**

- **console** (*Console*) -- Console to render to.
- **options** (*ConsoleOptions*) -- Console rendering options.
- **usage** (*Any*) -- The usage line (Text or str).

**render_description**(*console: Console*, *options: ConsoleOptions*, *description: Any*) → None

Render the description.

**Parameters**

- **console** (*Console*) -- Console to render to.
- **options** (*ConsoleOptions*) -- Console rendering options.
- **description** (*Any*) -- The description (can be various Rich renderables).

**class** cyclopts.help.protocols.**ColumnSpecBuilder**(**args*, ***kwargs*)

Protocol for ColumnSpecBuilders.

**__call__**(*console: Console*, *options: ConsoleOptions*, *entries: list[*HelpEntry*]*) → tuple[*ColumnSpec*, ...]

Build column specifications based on console settings and entries.

**Parameters**

- **console** (*Console*) -- The Rich console instance.
- **options** (*ConsoleOptions*) -- Console rendering options.

- **entries** (*list*[HelpEntry]) -- List of help entries to be displayed.

**Returns**
Tuple of column specifications for table rendering.

**Return type**
tuple[*ColumnSpec*, ...]

class cyclopts.help.**PanelSpec**(*title: RenderableType | None = None*, *subtitle: RenderableType | None = None*, *title_align: Literal['left', 'center', 'right'] = 'left'*, *subtitle_align: Literal['left', 'center', 'right'] = 'center'*, *style: StyleType | None = 'none'*, *border_style: StyleType | None = 'none'*, *box: Box | None = None*, *padding: PaddingDimensions = (0, 1)*, *expand: bool = True*, *width: int | None = None*, *height: int | None = None*, *safe_box: bool | None = None*, *highlight: bool = False*)

Specification for panel (outer box) styling.

Used by *DefaultFormatter* to control the appearance of the outer panel that wraps help sections. This spec defines the panel's border, title, subtitle, and overall styling.

**See also**

*DefaultFormatter*
The formatter that uses these specs.

*TableSpec*
Specification for the inner table.

*ColumnSpec*
Specification for individual columns.

**title: RenderableType | None**
Title text displayed at the top of the panel.

Corresponds to the `title` parameter of `Panel`.

**subtitle: RenderableType | None**
Subtitle text displayed at the bottom of the panel.

Corresponds to the `subtitle` parameter of `Panel`.

**title_align: Literal['left', 'center', 'right']**
Alignment of the title text within the panel.

Corresponds to the `title_align` parameter of `Panel`.

**subtitle_align: Literal['left', 'center', 'right']**
Alignment of the subtitle text within the panel.

Corresponds to the `subtitle_align` parameter of `Panel`.

**style: StyleType | None**
Style applied to the panel background.

Corresponds to the `style` parameter of `Panel`.

**border_style:** `StyleType | None`

Style applied to the panel border.

Corresponds to the `border_style` parameter of `Panel`.

**box:** `Box | None`

Box drawing style for the panel border.

Corresponds to the `box` parameter of `Panel`. See `rich.box` for available styles. Defaults to `rich.box.ROUNDED`.

**padding:** `PaddingDimensions`

Padding inside the panel (top/bottom, left/right) or (top, right, bottom, left).

Corresponds to the `padding` parameter of `Panel`.

**expand:** `bool`

Whether the panel should expand to fill available width.

Corresponds to the `expand` parameter of `Panel`.

**width:** `int | None`

Fixed width for the panel in characters.

Corresponds to the `width` parameter of `Panel`.

**height:** `int | None`

Fixed height for the panel in lines.

Corresponds to the `height` parameter of `Panel`.

**safe_box:** `bool | None`

Whether to use ASCII-safe box characters for compatibility.

Corresponds to the `safe_box` parameter of `Panel`.

**highlight:** `bool`

Enable automatic highlighting of panel contents.

Corresponds to the `highlight` parameter of `Panel`.

**build**(*renderable: RenderableType*, ***overrides*) → Panel

Create a Panel around *renderable*. Use kwargs to override spec per render.

**copy**(***kwargs*)

class cyclopts.help.**TableSpec**(*title: str | None = None*, *caption: str | None = None*, *style: StyleType | None = None*, *border_style: StyleType | None = None*, *header_style: StyleType | None = None*, *footer_style: StyleType | None = None*, *box: Box | None = None*, *show_header: bool = False*, *show_footer: bool = False*, *show_lines: bool = False*, *show_edge: bool = True*, *expand: bool = False*, *pad_edge: bool = False*, *padding: PaddingDimensions = (0, 2, 0, 0)*, *collapse_padding: bool = False*, *width: int | None = None*, *min_width: int | None = None*, *safe_box: bool | None = None*)

Specification for table layout and styling.

Used by *DefaultFormatter* to control the appearance of tables that display commands and parameters. This spec defines table-wide properties like borders, headers, and padding.

**See also**

*DefaultFormatter*
: The formatter that uses these specs.

*ColumnSpec*
: Specification for individual columns.

*PanelSpec*
: Specification for the outer panel.

**title:** `str | None`
: Title text displayed above the table.

Corresponds to the `title` parameter of `Table`.

**caption:** `str | None`
: Caption text displayed below the table.

Corresponds to the `caption` parameter of `Table`.

**style:** `StyleType | None`
: Default style applied to the entire table.

Corresponds to the `style` parameter of `Table`.

**border_style:** `StyleType | None`
: Style applied to table borders.

Corresponds to the `border_style` parameter of `Table`.

**header_style:** `StyleType | None`
: Default style for all table headers (can be overridden per column).

Corresponds to the `header_style` parameter of `Table`.

**footer_style:** `StyleType | None`
: Default style for all table footers (can be overridden per column).

Corresponds to the `footer_style` parameter of `Table`.

**box:** `Box | None`
: Box drawing style for the table borders.

Corresponds to the `box` parameter of `Table`. See `rich.box` for available styles.

**show_header:** `bool`
: Whether to display column headers.

Corresponds to the `show_header` parameter of `Table`.

**show_footer:** `bool`
: Whether to display column footers.

Corresponds to the `show_footer` parameter of `Table`.

**show_lines:** `bool`
: Whether to show horizontal lines between rows.

Corresponds to the `show_lines` parameter of `Table`.

**show_edge:** bool

Whether to draw a box around the outside of the table.

Corresponds to the show_edge parameter of Table.

**expand:** bool

Whether the table should expand to fill available width.

Corresponds to the expand parameter of Table.

**pad_edge:** bool

Whether to add padding to the table edges.

Corresponds to the pad_edge parameter of Table.

**padding:** PaddingDimensions

Padding around cell content (top, right, bottom, left).

Corresponds to the padding parameter of Table.

**collapse_padding:** bool

Whether to collapse padding when adjacent cells are empty.

Corresponds to the collapse_padding parameter of Table.

**width:** int | None

Fixed width for the table in characters.

Corresponds to the width parameter of Table.

**min_width:** int | None

Minimum width for the table in characters.

Corresponds to the min_width parameter of Table.

**safe_box:** bool | None

Whether to use ASCII-safe box characters for compatibility.

Corresponds to the safe_box parameter of Table.

**build**(*columns: tuple[*ColumnSpec*, ...]*, *entries: Iterable[*HelpEntry*]*, ***overrides*) → Table

Construct and populate a rich.Table.

**Parameters**

- **columns** (*tuple[ColumnSpec, ...]*) -- Column specifications defining the table structure.
- **entries** (*Iterable[HelpEntry]*) -- Table entries to populate the table with.
- ****overrides** -- Per-render overrides for table settings.

**Returns**

A populated Rich Table.

**Return type**

Table

**copy**(***kwargs*)

class cyclopts.help.**ColumnSpec**(*renderer: str | Renderer*, *header: str = ''*, *footer: str = ''*, *header_style: StyleType | None = None*, *footer_style: StyleType | None = None*, *style: StyleType | None = None*, *justify: Literal['default', 'left', 'center', 'right', 'full'] = 'left'*, *vertical: Literal['top', 'middle', 'bottom'] = 'top'*, *overflow: Literal['fold', 'crop', 'ellipsis', 'ignore'] = 'ellipsis'*, *width: int | None = None*, *min_width: int | None = None*, *max_width: int | None = None*, *ratio: int | None = None*, *no_wrap: bool = False*, *highlight: bool | None = None*)

Specification for a single column in a help table.

Used by *DefaultFormatter* to define how individual columns are rendered in help tables. Each column can have its own renderer, styling, and layout properties.

**See also**

*DefaultFormatter*
: The formatter that uses these specs.

*TableSpec*
: Specification for the entire table.

*PanelSpec*
: Specification for the outer panel.

**renderer:** `str | Renderer`

Specifies how to extract and render cell content from a *HelpEntry*.

Can be either:

- A string: The attribute name to retrieve from *HelpEntry* (e.g., 'names', 'description', 'required', 'type'). The string is displayed as-is.
- A callable: A function matching the `Renderer` protocol. The function receives a *HelpEntry* and should return a `RenderableType` (str, `Text`, or other Rich renderable).

Examples:

```
# String renderer - get attribute directly
ColumnSpec(renderer="description")

# Callable renderer - custom formatting
def format_names(entry: HelpEntry) -> str:
    return ", ".join(entry.names) if entry.names else ""
ColumnSpec(renderer=format_names)
```

**header:** `str`

Column header text displayed at the top of the column.

Example:

```
header="Options" renders:
┌─────────┬─────────────┐
│ Options │ Description │
├─────────┼─────────────┤
│ --help  │ Show help   │
└─────────┴─────────────┘
```

**footer:** `str`

Column footer text displayed at the bottom of the column.

Example:

```
footer="Required" renders:
┌──────────────────────┐
│ --help   │ Show help │
├──────────┼───────────┤
│ Required │           │
└──────────┴───────────┘
```

**header_style:** `StyleType | None`

Style applied to the column header text.

Corresponds to the `header_style` parameter of `rich.table.Table.add_column()`.

**footer_style:** `StyleType | None`

Style applied to the column footer text.

Corresponds to the `footer_style` parameter of `rich.table.Table.add_column()`.

**style:** `StyleType | None`

Default style applied to all cells in this column.

Corresponds to the `style` parameter of `rich.table.Table.add_column()`.

**justify:** `Literal['default', 'left', 'center', 'right', 'full']`

Text justification within the column.

Corresponds to the `justify` parameter of `rich.table.Table.add_column()`.

**vertical:** `Literal['top', 'middle', 'bottom']`

Vertical alignment of text within cells.

Corresponds to the `vertical` parameter of `rich.table.Table.add_column()`.

**overflow:** `Literal['fold', 'crop', 'ellipsis', 'ignore']`

How to handle text that exceeds column width.

Corresponds to the `overflow` parameter of `rich.table.Table.add_column()`.

**width:** `int | None`

Fixed width for the column in characters.

Corresponds to the `width` parameter of `rich.table.Table.add_column()`.

**min_width:** `int | None`

Minimum width for the column in characters.

Corresponds to the `min_width` parameter of `rich.table.Table.add_column()`.

**max_width:** `int | None`

Maximum width for the column in characters.

Corresponds to the `max_width` parameter of `rich.table.Table.add_column()`.

**ratio:** `int | None`

Relative width ratio compared to other columns.

Corresponds to the `ratio` parameter of `rich.table.Table.add_column()`.

**no_wrap:** bool

Prevent text wrapping in the column.

Corresponds to the no_wrap parameter of rich.table.Table.add_column().

**highlight:** bool | None

Enable automatic highlighting of text in the column.

Corresponds to the highlight parameter of rich.table.Table.add_column().

**copy**(**kwargs)

class cyclopts.help.**NameRenderer**(*max_width: int | None = None*)

Renderer for parameter/command names with optional text wrapping.

**Parameters**
**max_width** (*int | None*) -- Maximum width for wrapping. If None, no wrapping is applied.

Initialize the renderer with formatting options.

**Parameters**
**max_width** (*int | None*) -- Maximum width for wrapping. If None, no wrapping is applied.

**__call__**(*entry:* HelpEntry) → RenderableType

Render the names column with optional text wrapping.

**Parameters**
**entry** (HelpEntry) -- The table entry to render.

**Returns**
Combined names and shorts, optionally wrapped. Order: positive_names, positive_shorts, negative_names, negative_shorts

**Return type**
*RenderableType*

class cyclopts.help.**DescriptionRenderer**(*newline_metadata: bool = False*)

Renderer for descriptions with configurable metadata formatting.

**Parameters**
**newline_metadata** (*bool*) -- If True, display metadata (choices, env vars, defaults) on separate lines. If False (default), display metadata inline with the description.

Initialize the renderer with formatting options.

**Parameters**
**newline_metadata** (*bool*) -- If True, display metadata on separate lines instead of inline.

**__call__**(*entry:* HelpEntry) → RenderableType

Render parameter description with metadata annotations.

Enriches the base description with choices, environment variables, default values, and required status.

**Parameters**
**entry** (HelpEntry) -- The table entry to render.

**Returns**
Description with appended metadata.

**Return type**
*RenderableType*

class cyclopts.help.**AsteriskRenderer**

Renderer for required parameter asterisk indicator.

A simple renderer that displays an asterisk (*) for required parameters.

**__call__**(*entry:* HelpEntry) → RenderableType

Render an asterisk for required parameters.

**Parameters**
**entry** (HelpEntry) -- The table entry to render.

**Returns**
An asterisk if the entry is required, empty string otherwise.

**Return type**
*RenderableType*

class cyclopts.help.**HelpPanel**(*format: Literal['command', 'parameter']*, *title: RenderableType*, *description=None*, *entries: list[*HelpEntry*] = NOTHING*)

Data container for help panel information.

**format:** Literal['command', 'parameter']

Panel format type.

**title:** RenderableType

The title text displayed at the top of the help panel.

**description:** Any

Optional description text displayed below the title.

Typically a str or a RenderableType

**entries:** list[*HelpEntry*]

List of help entries to display (in order) in the panel.

**copy**(***kwargs*)

class cyclopts.help.**HelpEntry**(***, *positive_names: tuple[str, ...] = ()*, *positive_shorts: tuple[str, ...] = ()*, *negative_names: tuple[str, ...] = ()*, *negative_shorts: tuple[str, ...] = ()*, *description: Any | None = None*, *required: bool = False*, *sort_key: Any | None = None*, *type: Any | None = None*, *choices: tuple[str, ...] | None = None*, *env_var: tuple[str, ...] | None = None*, *default: str | None = None*)

Container for help table entry data.

**positive_names:** tuple[str, ...]

Positive long option names (e.g., "--verbose", "--dry-run").

**positive_shorts:** tuple[str, ...]

Positive short option names (e.g., "-v", "-n").

**negative_names:** tuple[str, ...]

Negative long option names (e.g., "--no-verbose", "--no-dry-run").

**negative_shorts:** tuple[str, ...]

Negative short option names (e.g., "-N"). Rarely used.

**property names:** tuple[str, ...]

All long option names (positive + negative). For backward compatibility.

**property shorts:** `tuple[str, ...]`

All short option names (positive + negative). For backward compatibility.

**property all_options:** `tuple[str, ...]`

positive longs, positive shorts, negative longs, negative shorts.

**Type**
All options in display order

**description:** `Any`

Help text description for this entry.

Typically a `str` or a `RenderableType`

**required:** `bool`

Whether this parameter/command is required.

**sort_key:** `Any`

Custom sorting key for ordering entries.

**type:** `Any | None`

Type annotation of the parameter.

**choices:** `tuple[str, ...] | None`

Available choices for this parameter.

**env_var:** `tuple[str, ...] | None`

Environment variable names that can set this parameter.

**default:** `str | None`

Default value for this parameter to display. None means no default to show.

**copy**(***kwargs*)

## 18.4 Config

Cyclopts has builtin configuration classes to be used with *App.config* for loading user-defined defaults in many common scenarios. All Cyclopts builtins index into the configuration file with the following rules:

1. Apply `root_keys` (if provided) to enter the project's configuration namespace.
2. Apply the command name(s) to enter the current command's configuration namespace.
3. Apply each key/value pair if CLI arguments have **not** been provided for that parameter.

**class** cyclopts.config.**Toml**(*path*, ***, *root_keys: None | Any | Iterable[Any] = ()*, *allow_unknown: bool = False*, *use_commands_as_keys: bool = True*, *source: str | None = None*, *must_exist: bool = False*, *search_parents: bool = False*)

Automatically read configuration from Toml file.

**path:** `str | pathlib.Path`

Path to TOML configuration file.

**source:** `str | None = None`

Identifier for the configuration source, used in error messages. If not provided, defaults to the absolute path of the configuration file.

**root_keys: Iterable[str] = None**

The key or sequence of keys that lead to the root configuration structure for this app. For example, if referencing a `pyproject.toml`, it is common to store all of your projects configuration under:

```
[tool.myproject]
```

So, your Cyclopts *App* should be configured as:

```
app = cyclopts.App(config=cyclopts.config.Toml("pyproject.toml", root_keys=(
↪"tool", "myproject")))
```

**must_exist: bool = False**

The configuration file MUST exist. Raises `FileNotFoundError` if it does not exist.

**search_parents: bool = False**

If `path` doesn't exist, iteratively search parenting directories for a same-named configuration file. Raises `FileNotFoundError` if no configuration file is found.

**allow_unknown: bool = False**

Allow for unknown keys. Otherwise, if an unknown key is provided, raises `UnknownOptionError`.

**use_commands_as_keys: bool = True**

Use the sequence of commands as keys into the configuration.

For example, the following CLI invocation:

```
$ python my-script.py my-command
```

Would search into `["my-command"]` for values.

**class** cyclopts.config.**Yaml**(*path*, *, *root_keys: None | Any | Iterable[Any] = ()*, *allow_unknown: bool = False*, *use_commands_as_keys: bool = True*, *source: str | None = None*, *must_exist: bool = False*, *search_parents: bool = False*)

Automatically read configuration from YAML file.

**path: str | pathlib.Path**

Path to YAML configuration file.

**source: str | None = None**

Identifier for the configuration source, used in error messages. If not provided, defaults to the absolute path of the configuration file.

**root_keys: Iterable[str] = None**

The key or sequence of keys that lead to the root configuration structure for this app. For example, if referencing a common `config.yaml` that is shared with other applications, it is common to store your projects configuration under a key like `myproject:`.

Your Cyclopts *App* would be configured as:

```
app = cyclopts.App(config=cyclopts.config.Yaml("config.yaml", root_keys=
↪"myproject"))
```

**must_exist: bool = False**

The configuration file MUST exist. Raises `FileNotFoundError` if it does not exist.

**search_parents:** `bool = False`

If `path` doesn't exist, iteratively search parenting directories for a same-named configuration file. Raises `FileNotFoundError` if no configuration file is found.

**allow_unknown:** `bool = False`

Allow for unknown keys. Otherwise, if an unknown key is provided, raises `UnknownOptionError`.

**use_commands_as_keys:** `bool = True`

Use the sequence of commands as keys into the configuration.

For example, the following CLI invocation:

```
$ python my-script.py my-command
```

Would search into `["my-command"]` for values.

class cyclopts.config.**Json**(*path*, *, *root_keys: None | Any | Iterable[Any] = ()*, *allow_unknown: bool = False*, *use_commands_as_keys: bool = True*, *source: str | None = None*, *must_exist: bool = False*, *search_parents: bool = False*)

Automatically read configuration from Json file.

**path:** `str | pathlib.Path`

Path to JSON configuration file.

**source:** `str | None = None`

Identifier for the configuration source, used in error messages. If not provided, defaults to the absolute path of the configuration file. Can be customized to provide more descriptive error context.

Example:

```
app = cyclopts.App(config=cyclopts.config.Json("config.json", source=
↪"production-config"))
```

**root_keys:** `Iterable[str] = None`

The key or sequence of keys that lead to the root configuration structure for this app. For example, if referencing a common `config.json` that is shared with other applications, it is common to store your projects configuration under a key like `"myproject":`.

Your Cyclopts *App* would be configured as:

```
app = cyclopts.App(config=cyclopts.config.Json("config.json", root_keys=
↪"myproject"))
```

**must_exist:** `bool = False`

The configuration file MUST exist. Raises `FileNotFoundError` if it does not exist.

**search_parents:** `bool = False`

If `path` doesn't exist, iteratively search parenting directories for a same-named configuration file. Raises `FileNotFoundError` if no configuration file is found.

**allow_unknown:** `bool = False`

Allow for unknown keys. Otherwise, if an unknown key is provided, raises `UnknownOptionError`.

**use_commands_as_keys:** `bool = True`

Use the sequence of commands as keys into the configuration.

For example, the following CLI invocation:

```
$ python my-script.py my-command
```

Would search into `["my-command"]` for values.

**class** cyclopts.config.**Dict**(*data: dict[str, Any]*, ***, *root_keys: None | Any | Iterable[Any] = ()*, *allow_unknown: bool = False*, *use_commands_as_keys: bool = True*, *source: str | None = None*)

Configuration source from an in-memory dictionary.

Useful for programmatically generated configurations.

Use an in-memory Python dictionary as configuration source.

**data:** `dict[str, Any]`

The configuration dictionary.

**source:** `str = "dict"`

Identifier for the configuration source, used in error messages.

**root_keys:** `Iterable[str] = ()`

The key or sequence of keys that lead to the root configuration structure for this app.

**allow_unknown:** `bool = False`

Allow for unknown keys. Otherwise, if an unknown key is provided, raises `UnknownOptionError`.

**use_commands_as_keys:** `bool = True`

Use the sequence of commands as keys into the configuration.

**class** cyclopts.config.**Env**(*prefix: str = ''*, ***, *source: str = 'env'*, *command: bool = True*, *show: bool = True*)

Automatically derive environment variable names to read configurations from.

For example, consider the following app:

```
import cyclopts

app = cyclopts.App(config=cyclopts.config.Env("MY_SCRIPT_"))

@app.command
def my_command(foo, bar):
    print(f"{foo=} {bar=}")

app()
```

If values for `foo` and `bar` are not supplied by the command line, the app will check the environment variables `MY_SCRIPT_MY_COMMAND_FOO` and `MY_SCRIPT_MY_COMMAND_BAR`, respectively:

```
$ python my_script.py my-command 1 2
foo=1 bar=2

$ export MY_SCRIPT_MY_COMMAND_FOO=100
$ python my_script.py my-command --bar=2
foo=100 bar=2
$ python my_script.py my-command 1 2
foo=1 bar=2
```

**prefix:** str = ""

String to prepend to all autogenerated environment variable names. Typically ends in _, and is something like MY_APP_.

**source:** str = "env"

Identifier for the configuration source, used in error messages and token tracking. Defaults to "env".

**command:** bool = True

If True, add the command's name (uppercase) after *prefix*.

**show:** bool = True

If True, then show the environment variables on the help-page.

## 18.5 Exceptions

**exception** cyclopts.**CycloptsError**

Bases: Exception

Root exception for runtime errors.

As CycloptsErrors bubble up the Cyclopts call-stack, more information is added to it.

**msg:** str | None

If set, override automatic message generation.

**verbose:** bool

More verbose error messages; aimed towards developers debugging their Cyclopts app. Defaults to False.

**root_input_tokens:** list[str] | None

The parsed CLI tokens that were initially fed into the *App*.

**unused_tokens:** list[str] | None

Leftover tokens after parsing is complete.

**target:** Callable | None

The python function associated with the command being parsed.

**argument:** *Argument* | None

*Argument* that was matched.

**command_chain:** Sequence[str] | None

List of command that lead to target.

**app:** *App* | None

The Cyclopts application itself.

**console:** Console | None

Console to display runtime errors.

**exception** cyclopts.**ValidationError**

Bases: *CycloptsError*

Validator function raised an exception.

**exception_message:** str

Parenting Assertion/Value/Type Error message.

**group:** *Group* | None

If a group validator caused the exception.

**value:** Any

Converted value that failed validation.

**exception** cyclopts.**UnknownOptionError**

Bases: *CycloptsError*

Unknown/unregistered option provided by the cli.

A nearest-neighbor parameter suggestion may be printed.

**token:** *Token*

Token without a matching parameter.

**argument_collection:** *ArgumentCollection*

Argument collection of plausible options.

**exception** cyclopts.**CoercionError**

Bases: *CycloptsError*

There was an error performing automatic type coercion.

**token:** *Token* | None

Input token that couldn't be coerced.

**target_type:** type | None

Intended type to coerce into.

**exception** cyclopts.**UnknownCommandError**

Bases: *CycloptsError*

CLI token combination did not yield a valid command.

**msg:** str | None

If set, override automatic message generation.

**verbose:** bool

More verbose error messages; aimed towards developers debugging their Cyclopts app. Defaults to `False`.

**root_input_tokens:** list[str] | None

The parsed CLI tokens that were initially fed into the *App*.

**unused_tokens:** list[str] | None

Leftover tokens after parsing is complete.

**target:** Callable | None

The python function associated with the command being parsed.

**argument:** 'Argument' | None

*Argument* that was matched.

**command_chain:** Sequence[str] | None

List of command that lead to `target`.

**app:** 'App' | None

The Cyclopts application itself.

**console:** 'Console' | None

Console to display runtime errors.

**exception** cyclopts.**UnusedCliTokensError**

Bases: *CycloptsError*

Not all CLI tokens were used as expected.

**exception** cyclopts.**MissingArgumentError**

Bases: *CycloptsError*

A required argument was not provided.

**tokens_so_far:** list[str]

If the matched parameter requires multiple tokens, these are the ones we have parsed so far.

**keyword:** str | None

The keyword that was used when the error was raised (e.g., '-o' instead of '--option').

**exception** cyclopts.**RepeatArgumentError**

Bases: *CycloptsError*

The same parameter has erroneously been specified multiple times.

**token:** *Token*

The repeated token.

**exception** cyclopts.**MixedArgumentError**

Bases: *CycloptsError*

Cannot supply keywords and non-keywords to the same argument.

**exception** cyclopts.**CommandCollisionError**

Bases: Exception

A command with the same name has already been registered to the app.

**exception** cyclopts.**CombinedShortOptionError**

Bases: *CycloptsError*

Cannot combine short, token-consuming options with short flags.

**exception** cyclopts.**EditorError**

Bases: Exception

Root editor-related error.

Root exception raised by all exceptions in *edit()*.

**exception** cyclopts.**EditorNotFoundError**

Bases: *EditorError*

Could not find a valid text editor for :func`.edit`.

**exception** cyclopts.**EditorDidNotSaveError**

Bases: *EditorError*

User did not save upon exiting *edit()*.

**exception** cyclopts.**EditorDidNotChangeError**

Bases: *EditorError*

User did not edit file contents in *edit()*.

CHAPTER

# NINETEEN

# CLI REFERENCE

The `cyclopts` package includes a command-line interface for various development tasks.

## 19.1 cyclopts --install-completion

```
cyclopts --install-completion [OPTIONS]
```

Register shell-completion for the cyclopts CLI itself.

**Parameters:**

**`--shell`**
Shell type for completion. If not specified, attempts to auto-detect current shell. [Choices: `zsh`, `bash`, `fish`]

**`--output, -o`**
Output path for the completion script. If not specified, uses shell-specific default.

## 19.2 cyclopts generate-docs

```
cyclopts generate-docs [OPTIONS] SCRIPT [ARGS]
```

Generate documentation for a Cyclopts application.

**Parameters:**

**`SCRIPT, --script`**
Python script path, optionally with `':app_object'` notation to specify the App object. If not specified, will search for App objects in the script's global namespace. [**Required**]

**`OUTPUT, --output, -o`**
Output file path. If not specified, prints to stdout.

**`--format, -f`**
Output format for documentation. If not specified, inferred from output file extension. [Choices: `markdown`, `md`, `html`, `htm`, `rst`, `rest`, `restructuredtext`]

**`--include-hidden`**
Include hidden commands in documentation. [Default: `False`]

**`--heading-level`**
Starting heading level for markdown format. [Default: `1`]

## 19.3 cyclopts run

```
cyclopts run SCRIPT [ARGS...]
```

Run a Cyclopts application from a Python script with dynamic shell completion.

All arguments after the script path are passed to the loaded application.

Shell completion is available. Run once to install (persistent): `cyclopts --install-completion`

**Arguments:**

**SCRIPT**
: Python script path with optional ':app_object' notation. [**Required**]

**ARGS**
: Arguments to pass to the loaded application.

CHAPTER

TWENTY

# KNOWN ISSUES

This document intends to record any known long-standing issues/limitations with Cyclopts. While this document should always be up to date, please also visit the github-issues page for more information & discussion.

## 20.1 `from __future__ import annotations`

Due to quirks in the python-typing system, Cyclopts can only support some scenarios surrounding PEP-0563, the strinigization of type hints via `from __future__ import annotations`. Notably, this can also sometimes break `dataclass` definitions when inheritance from multiple python modules is involved. Attempts have been made to improve Cyclopts support, but there are the following blockers:

1. CPython has some bugs around `typing.get_type_hints()`. It is outside the scope of Cyclopts to compensate for the complex task of type-hint scoping and resolution.
2. Particularly with dataclasses, it looks like they will be fixing these bugs, but it would only be backported to 3.13 and 3.14. This limitation to very modern python versions makes a lot of PEP-0563 moot.
3. PEP-0649 and PEP-0749 deprecate the usage of `from __future__ import annotations`. This suggests that it is not worth the long-term maintenance of supporting the complications of this feature.

Original discussion on GitHub.

CHAPTER

# TWENTYONE

# LAZY LOADING

Lazy loading allows you to register commands **using import path strings instead of direct function references**. This defers importing command modules until they are actually executed, which could significantly improve CLI startup time for large applications that have expensive per-command imports.

## 21.1 Basic Usage

Instead of importing and registering a function directly:

```
from cyclopts import App
from myapp.commands.users import create, delete, list_users  # Imported immediately

app = App()
user_app = App(name="user")

user_app.command(create)
user_app.command(delete)
user_app.command(list_users, name="list")

app.command(user_app)
```

Use an import path string:

```
from cyclopts import App

app = App()
user_app = App(name="user")

# No imports! Modules loaded only when commands execute
user_app.command("myapp.commands.users:create")
user_app.command("myapp.commands.users:delete")
user_app.command("myapp.commands.users:list_users", name="list")

app.command(user_app)
```

The import path format is `"module.path:function_name"`, similar to setuptools entry points.

Lazy commands are resolved/imported in these situations:

- **Command Execution** - When the user runs that specific command
- **Help Generation** - When displaying help that includes the command
- **Direct Access** - When accessing via `app["command_name"]`

In order to benefit from lazy loading, you have to make sure that the files are not imported by other means when your CLI starts up.

## 21.2 Import Path Format

The import path string has two parts separated by a colon (`:`):

**Module Path (before the `:`)**
: The Python **module** to import, using dot notation (e.g., `myapp.commands.users`).

**Attribute Name (after the `:`)**
: The function or App to get from the module using `getattr()`.

Examples:

```
# Simple function in a module
app.command("myapp.commands:create_user")

# Nested module path
app.command("myapp.admin.database.operations:migrate")

# Import an App instance, exposed to the CLI as "admin"
app.command("myapp.admin:admin_app", name="admin")
```

> **Note**
>
> The attribute name (after `:`) is the **actual Python name**, not the CLI command name. Use the `name` parameter to specify the CLI command name.

## 21.3 Name vs Function Name

The `name` parameter specifies **how the command appears in the CLI**, while the import path specifies **what code to execute**. They can be completely different:

```
from cyclopts import App

user_app = App(name="user")

# Function name: "list_users"
# CLI command name: "list"
user_app.command("myapp.commands.users:list_users", name="list")

# Function name: "delete"
# CLI command name: "remove"
user_app.command("myapp.commands.users:delete", name="remove")
```

```
$ myapp user list --limit 10
# Imports and runs myapp.commands.users:list_users

$ myapp user remove --username alice
# Imports and runs myapp.commands.users:delete
```

If `name` is not specified, Cyclopts derives it from the function name with *App.name_transform* applied (typically converting underscores to hyphens).

## 21.4 Error Handling

If an import path/configuration is invalid, the error occurs **when the command is executed**, not when it's registered:

```
from cyclopts import App

app = App()

# This won't error immediately - registration succeeds
app.command("nonexistent.module:func")

app()
```

```
$ myapp func
# Now the error occurs:
ImportError: Cannot import module 'nonexistent.module'
```

To catch import errors early, you can access the command during testing:

```
import pytest
from cyclopts import App

def test_lazy_commands_are_importable():
    app = App()
    app.command("myapp.commands:create")

    # This will trigger the import and fail if path is wrong
    resolved = app["create"]
    assert resolved is not None
```

## 21.5 Groups and Lazy Loading

> **Tip**
>
> **TL;DR:** Define *Group* objects used by commands in your main CLI module, NOT in lazy-loaded modules.

*Group* objects defined in **unresolved lazy modules** won't be available until those modules are **explicitly imported**. To avoid this, define *Group* objects in non-lazy modules.

```
# myapp/cli.py (always imported)
from cyclopts import App, Group

# Define Group objects here
admin_group = Group("Admin Commands", validator=require_admin_role)
db_group = Group("Database", default_parameter=Parameter(envvar_prefix="DB_"))

app = App()
```

(continues on next page)

(continued from previous page)

```
# Lazy commands can reference the Group objects
app.command("myapp.admin:create_user", group=admin_group)
app.command("myapp.admin:delete_user", group=admin_group)
app.command("myapp.db:migrate", group=db_group)
```

**What to avoid:** Defining `Group` objects inside lazy-loaded modules:

```
# myapp/admin.py (lazy-loaded)
from cyclopts import App, Group

# BAD: This Group won't be available to other commands until this module is imported
admin_group = Group("Admin Commands", validator=require_admin_role)

def create_user():
    ...
```

If you reference a group by string (e.g., `group="Admin Commands"`) and the *Group* object with that name is only defined in an unresolved lazy module, the group won't be available until that lazy module is imported. This means that:

- Validators defined on the lazy-loaded *Group* won't be applied to commands in other modules.
- *Group.default_parameter* and other settings won't be inherited by commands **referencing the group by string**.

Once the lazy module is imported (e.g., by executing one of its commands), the *Group* object becomes available and subsequent operations will use it correctly.

CHAPTER

# TWENTYTWO

# HELP CUSTOMIZATION

Cyclopts provides extensive customization options for help screen appearance and formatting through the `help_formatter` parameter available on both *App* and *Group*. These parameters accept formatters that follow the *HelpFormatter* protocol.

## 22.1 Setting Help Formatters

### 22.1.1 App-Level Formatting

The *App* class accepts a `help_formatter` parameter that controls the default formatting for all help output:

```python
from cyclopts import App
from cyclopts.help import DefaultFormatter, PlainFormatter

# Use a built-in formatter by name: {"default", "plain"}
app = App(help_formatter="plain")

# Or pass a formatter instance with custom configuration
app = App(
    help_formatter=DefaultFormatter(
        # Custom configuration options
    )
)

# Or use a completely custom formatter; see HelpFormatter protocol.
app = App(help_formatter=MyCustomFormatter())
```

### 22.1.2 Group-Level Formatting

Individual *Group* instances can have their own `help_formatter` that overrides the app-level default:

```python
from cyclopts import App, Group
from cyclopts.help import DefaultFormatter, PanelSpec
from rich.box import DOUBLE

# Create a group with custom formatting
advanced_group = Group(
    "Advanced Options",
    help_formatter=DefaultFormatter(
        panel_spec=PanelSpec(
            border_style="red",
```

(continues on next page)

(continued from previous page)

```
            box=DOUBLE,
        )
    )
)

# The app can have a different default formatter
app = App(help_formatter="plain")

# Parameters in advanced_group will use the group's formatter,
# while other parameters use the app's formatter
```

## 22.2 Built-in Formatters

### 22.2.1 DefaultFormatter

The *DefaultFormatter* is the default help formatter that uses Rich for beautiful terminal output with colors, borders, and structured layouts.

```
from cyclopts import App

# Explicitly use the default formatter (same as not specifying)
app = App(help_formatter="default")

@app.default
def main(name: str, count: int = 1):
    """A simple greeting application.

    Parameters
    ----------
    name : str
        Person to greet.
    count : int
        Number of times to greet.
    """
    for _ in range(count):
        print(f"Hello, {name}!")

if __name__ == "__main__":
    app()
```

Output:

### 22.2.2 PlainFormatter

The *PlainFormatter* provides accessibility-focused plain text output without colors or special characters, ideal for screen readers and simpler terminals.

```
from cyclopts import App

# Use plain text formatter for accessibility
app = App(help_formatter="plain")
```

(continues on next page)

(continued from previous page)

```
@app.default
def main(name: str, count: int = 1):
    """A simple greeting application.

    Parameters
    ----------
    name : str
        Person to greet.
    count : int
        Number of times to greet.
    """
    for _ in range(count):
        print(f"Hello, {name}!")

if __name__ == "__main__":
    app()
```

Output:

```
Usage: demo.py [ARGS] [OPTIONS]

A simple greeting application.

Commands:
--help, -h: Display this message and exit.
--version: Display application version.

Parameters:
NAME, --name: Person to greet.
COUNT, --count: Number of times to greet.
```

## 22.3 Basic Customization

The *DefaultFormatter* accepts several customization options through its initialization parameters.

### 22.3.1 Panel Customization

The *PanelSpec* controls the outer panel appearance:

```
from cyclopts import App
from cyclopts.help import DefaultFormatter, PanelSpec
from rich.box import DOUBLE

app = App(
    help_formatter=DefaultFormatter(
        panel_spec=PanelSpec(
            box=DOUBLE,              # Use double-line borders
            border_style="blue",     # Blue border color
            padding=(1, 2),          # (vertical, horizontal) padding
            expand=True,             # Expand to full terminal width
        )
```

(continues on next page)

(continued from previous page)

```
    )
)

@app.default
def main(path: str, verbose: bool = False):
    """Process a file with custom panel styling."""
    print(f"Processing {path}")

if __name__ == "__main__":
    app()
```

Output:

### 22.3.2 Table Customization

The *TableSpec* controls the table styling within panels:

```
from cyclopts import App
from cyclopts.help import DefaultFormatter, TableSpec

app = App(
    help_formatter=DefaultFormatter(
        table_spec=TableSpec(
            show_header=True,  # Show column headers
            show_lines=True,  # Show lines between rows
            show_edge=False,  # Remove outer table border
            border_style="green",  # Green table elements
            padding=(0, 2, 0, 0),  # Extra right padding
            box=SQUARE,  # otherwise we won't see the lines
        )
    )
)

@app.default
def main(path: str, verbose: bool = False):
    """Process a file with custom table styling."""
    print(f"Processing {path}")

if __name__ == "__main__":
    app()
```

Output:

### 22.3.3 Combining Customizations

You can combine both panel and table specifications:

```
from cyclopts import App
from cyclopts.help import DefaultFormatter, PanelSpec, TableSpec
from rich.box import ROUNDED

app = App(
    help_formatter=DefaultFormatter(
```

(continues on next page)

(continued from previous page)

```
        panel_spec=PanelSpec(
            box=ROUNDED,
            border_style="cyan",
            padding=(0, 1),
        ),
        table_spec=TableSpec(
            show_header=False,
            show_lines=False,
            padding=(0, 1),
        )
    )
)

@app.default
def main(path: str, verbose: bool = False):
    """Process a file with combined customizations."""
    print(f"Processing {path}")

if __name__ == "__main__":
    app()
```

Output:

## 22.4 Group-Level Formatting

Different parameter groups can have different formatting styles, allowing you to visually distinguish between different types of options:

```
from cyclopts import App, Group, Parameter
from cyclopts.help import DefaultFormatter, PanelSpec
from rich.box import DOUBLE, MINIMAL
from typing import Annotated

# Create groups with different styles
required_group = Group(
    "Required Options",
    help_formatter=DefaultFormatter(
        panel_spec=PanelSpec(
            box=DOUBLE,
            border_style="red bold",
        )
    )
)

optional_group = Group(
    "Optional Settings",
    help_formatter=DefaultFormatter(
        panel_spec=PanelSpec(
            box=MINIMAL,
            border_style="green",
        )
    )
```

(continues on next page)

(continued from previous page)

```
)

app = App()

@app.default
def main(
    # Required parameters with red double border
    input_file: Annotated[str, Parameter(group=required_group)],
    output_dir: Annotated[str, Parameter(group=required_group)],

    # Optional parameters with green minimal border
    verbose: Annotated[bool, Parameter(group=optional_group)] = False,
    threads: Annotated[int, Parameter(group=optional_group)] = 4,
):
    """Process files with styled help groups."""
    print(f"Processing {input_file} -> {output_dir}")
    if verbose:
        print(f"Using {threads} threads")

if __name__ == "__main__":
    app()
```

Output:

## 22.5 Custom Column Layout

For complete control over the help table layout, you can define custom columns using *ColumnSpec*:

```
from cyclopts import App, Group, Parameter
from cyclopts.help import DefaultFormatter, ColumnSpec, TableSpec
from typing import Annotated

# Define custom column renderers
def names_renderer(entry):
    """Combine parameter names and shorts."""
    names = " ".join(entry.names) if entry.names else ""
    shorts = " ".join(entry.shorts) if entry.shorts else ""
    return f"{names} {shorts}".strip()

def type_renderer(entry):
    """Show the parameter type."""
    from cyclopts.annotations import get_hint_name
    return get_hint_name(entry.type) if entry.type else ""

# Create custom columns
custom_group = Group(
    "Custom Layout",
    help_formatter=DefaultFormatter(
        table_spec=TableSpec(show_header=True),
        column_specs=(
            ColumnSpec(
                renderer=lambda e: "" if e.required else " ",
```

(continues on next page)

(continued from previous page)

```
                header="",
                width=2,
                style="yellow bold",
            ),
            ColumnSpec(
                renderer=names_renderer,
                header="Option",
                style="cyan",
                max_width=30,
            ),
            ColumnSpec(
                renderer=type_renderer,
                header="Type",
                style="magenta",
                justify="center",
            ),
            ColumnSpec(
                renderer="description",  # Use attribute name
                header="Description",
                overflow="fold",
            ),
        )
    )
)

app = App()

@app.default
def main(
    input_path: Annotated[str, Parameter(group=custom_group, help="Input file path")],
    output_path: Annotated[str, Parameter(group=custom_group, help="Output file path")],
    count: Annotated[int, Parameter(group=custom_group, help="Number of iterations")] =␣
↪1,
):
    """Demo custom column layout."""
    print(f"Processing {input_path} -> {output_path} ({count} times)")

if __name__ == "__main__":
    app()
```

Output:

### 22.5.1 Dynamic Column Builders

For even more flexibility, you can create columns dynamically based on runtime conditions:

```
from cyclopts import App, Parameter
from cyclopts.help import DefaultFormatter, ColumnSpec
from typing import Annotated

def dynamic_columns(console, options, entries):
    """Build columns based on console width and entries."""
```

(continues on next page)

(continued from previous page)

```
    columns = []

    # Only show required indicator if there are required params
    if any(e.required for e in entries):
        columns.append(ColumnSpec(
            renderer=lambda e: "*" if e.required else "",
            width=2,
            style="red",
        ))

    # Adjust name column width based on console size
    max_width = min(40, int(console.width * 0.3))
    columns.append(ColumnSpec(
        renderer=lambda e: " ".join(e.names + e.shorts),
        header="Option",
        max_width=max_width,
        style="cyan",
    ))

    # Always include description
    columns.append(ColumnSpec(
        renderer="description",
        header="Description",
        overflow="fold",
    ))

    return tuple(columns)

app = App(
    help_formatter=DefaultFormatter(
        column_specs=dynamic_columns
    )
)

@app.default
def main(
    input_file: str,
    output_file: str,
    verbose: bool = False,
):
    """Process files with dynamic columns."""
    print(f"Processing {input_file} -> {output_file}")

if __name__ == "__main__":
    app()
```

Output (adjusts based on terminal width):

## 22.6 Creating Custom Formatters

For complete control, you can implement your own formatter by following the *HelpFormatter* protocol. The formatter methods receive the console and options first, followed by the content to render:

```
from cyclopts import App
from cyclopts.help import HelpPanel
from rich.console import Console, ConsoleOptions
from rich.table import Table
from rich.panel import Panel

class MyCustomFormatter:
    """A custom formatter with unique styling."""

    def __call__(self, console: Console, options: ConsoleOptions, panel: HelpPanel) ->␣
↪None:
        """Render a help panel with custom styling."""
        if not panel.entries:
            return

        # Create a custom table
        table = Table(show_header=True, header_style="bold magenta")
        table.add_column("Option", style="cyan", no_wrap=True)
        table.add_column("Description", style="white")

        for entry in panel.entries:
            name = " ".join(entry.names + entry.shorts)
            # Extract plain text from description (handles InlineText, etc)
            desc = ""
            if entry.description:
                if hasattr(entry.description, 'plain'):
                    desc = entry.description.plain
                elif hasattr(entry.description, '__rich_console__'):
                    # Render to plain text without styles
                    with console.capture() as capture:
                        console.print(entry.description, end="")
                    desc = capture.get()
                else:
                    desc = str(entry.description)
            table.add_row(name, desc)

        # Wrap in a custom panel
        panel_title = panel.title or "Options"
        styled_panel = Panel(
            table,
            title=f"[bold blue]{panel_title}[/bold blue]",
            border_style="blue",
        )

        console.print(styled_panel)

    def render_usage(self, console: Console, options: ConsoleOptions, usage) -> None:
        """Render the usage line."""
```

(continues on next page)

(continued from previous page)

```
        if usage:
            console.print(f"[bold green]Usage:[/bold green] {usage}")

    def render_description(self, console: Console, options: ConsoleOptions, description)␣
↪-> None:
        """Render the description."""
        if description:
            console.print(f"\n[italic]{description}[/italic]\n")

# Use the custom formatter
app = App(help_formatter=MyCustomFormatter())

@app.default
def main(input_file: str, output_file: str, verbose: bool = False):
    """Process files with custom formatter."""
    print(f"Processing {input_file} -> {output_file}")

if __name__ == "__main__":
    app()
```

Output:

## 22.7 Reference

For complete API documentation of help formatting components, see:

- *cyclopts.help.DefaultFormatter* - Rich-based formatter with full customization
- *cyclopts.help.PlainFormatter* - Plain text formatter for accessibility
- *cyclopts.help.PanelSpec* - Panel appearance specification
- *cyclopts.help.TableSpec* - Table styling specification
- *cyclopts.help.ColumnSpec* - Column definition and rendering
- *cyclopts.help.protocols.HelpFormatter* - Protocol for custom formatters

See also:

- Help - General help system documentation
- *Groups* - Organizing parameters into groups

CHAPTER

# TWENTYTHREE

# USER CLASSES

Cyclopts supports classically defined user classes, as well as classes defined by the following dataclass-like libraries:

- attrs
- dataclass
- pydantic
- NamedTuple
- TypedDict

## 23.1 Basic Example

As an example, let's consider using the builtin `dataclass` to make a CLI that manages a movie collection.

```python
from cyclopts import App
from dataclasses import dataclass

app = App(name="movie-maintainer")

@dataclass
class Movie:
   title: str
   year: int

@app.command
def add(movie: Movie):
   print(f"Adding movie: {movie}")

app()
```

```
$ movie-maintainer add --help
Usage: movie-maintainer add [ARGS] [OPTIONS]

╭─ Parameters ─────────────────────────────────────────────────╮
│ *  MOVIE.TITLE              [required]                       │
│      --movie.title                                           │
│ *  MOVIE.YEAR --movie.year  [required]                       │
╰──────────────────────────────────────────────────────────────╯

$ movie-maintainer add 'Mad Max: Fury Road' 2015
```

(continues on next page)

(continued from previous page)

```
Adding movie: Movie(title='Mad Max: Fury Road', year=2015)

$ movie-maintainer add --movie.title 'Furiosa: A Mad Max Saga' --movie.year 2024
Adding movie: Movie(title='Furiosa: A Mad Max Saga', year=2024)
```

In most circumstances, Cyclopts will also parse a json-string for a dataclass-like parameter:

```
$ movie-maintainer add --movie='{"title": "Mad Max: Fury Road", "year": 2024}'
Adding movie: Movie(title='Mad Max: Fury Road', year=2024)
```

## 23.2 JSON Dict Parsing

JSON dict parsing will be performed when:

1. The parameter is specified as a keyword option; e.g. `--movie`.
2. The referenced parameter type has various sub-arguments (is dataclass-like).
3. The referenced parameter is **not** union'd with a `str`.
4. The first character is a `{`.

This behavior can be configured via *Parameter.json_dict*.

```python
from cyclopts import App
from dataclasses import dataclass

app = App(name="movie-manager")

@dataclass
class Movie:
   title: str
   year: int
   rating: float = 8.0

@app.command
def add(movie: Movie):
   print(f"Adding: {movie}")

app()
```

```
$ movie-manager add --movie '{"title": "Mad Max: Fury Road", "year": 2015, "rating": 8.1}
↪'
Adding: Movie(title='Mad Max: Fury Road', year=2015, rating=8.1)

$ movie-manager add --movie '{"title": "Furiosa", "year": 2024}'
Adding: Movie(title='Furiosa', year=2024, rating=8.0)
```

Note that JSON parsing only works when using the keyword option format (`--movie`). The traditional positional argument format still works with individual fields:

```
$ movie-manager add --movie.title "Dune" --movie.year 2021 --movie.rating 8.5
Adding: Movie(title='Dune', year=2021, rating=8.5)
```

## 23.3 JSON List Parsing

Cyclopts also supports JSON parsing for lists of dataclasses. This allows you to pass multiple structured objects via JSON:

```
from cyclopts import App
from dataclasses import dataclass

app = App(name="movie-collection")

@dataclass
class Movie:
   title: str
   year: int

@app.command
def add_batch(movies: list[Movie]):
   for movie in movies:
       print(f"Adding: {movie}")

app()
```

You can provide the list in several ways:

1. JSON Array - Multiple objects in a single argument:

```
$ movie-collection add-batch --movies '[{"title": "Mad Max", "year": 2015}, {"title
↪": "Furiosa", "year": 2024}]'
Adding: Movie(title='Mad Max', year=2015)
Adding: Movie(title='Furiosa', year=2024)
```

2. Individual JSON - Each object as a separate argument:

```
$ movie-collection add-batch --movies '{"title": "Mad Max", "year": 2015}' --movies
↪'{"title": "Furiosa", "year": 2024}'
Adding: Movie(title='Mad Max', year=2015)
Adding: Movie(title='Furiosa', year=2024)
```

3. Mixed - Combining arrays and individual objects:

```
$ movie-collection add-batch --movies '{"title": "Mad Max", "year": 2015}' --movies
↪'[{"title": "Furiosa", "year": 2024}, {"title": "Dune", "year": 2021}]'
Adding: Movie(title='Mad Max', year=2015)
Adding: Movie(title='Furiosa', year=2024)
Adding: Movie(title='Dune', year=2021)
```

JSON list parsing is automatically enabled for `list` types containing dataclasses. The same rules apply as for dict parsing:

- The element type cannot be union'd with `str`
- JSON objects must start with `{` or be arrays starting with `[`

This behavior can be configured via *Parameter.json_list*.

## 23.4 Namespace Flattening

It is likely that the actual movie class/object is not important to the CLI user, and the parameter names like `--movie.title` are unnecessarily verbose. We can remove `movie` from the name by giving the `Movie` type annotation the special name `"*"`.

```python
from cyclopts import App, Parameter
from dataclasses import dataclass
from typing import Annotated

app = App(name="movie-maintainer")

@dataclass
class Movie:
   title: str
   year: int

@app.command
def add(movie: Annotated[Movie, Parameter(name="*")]):
   print(f"Adding movie: {movie}")

app()
```

```
$ movie-maintainer add --help
Usage: movie-maintainer add [ARGS] [OPTIONS]

╭─ Parameters ─────────────────────────────────────────────╮
│ *  TITLE --title  [required]                             │
│ *  YEAR --year    [required]                             │
╰──────────────────────────────────────────────────────────╯
```

An alternative way of supplying the *Parameter* configuration is via a decorator. This way can be cleaner and terser in many scenarios. The *Parameter* configuration will also be inherited by subclasses.

```python
from cyclopts import App, Parameter
from dataclasses import dataclass

app = App(name="movie-maintainer")

@Parameter(name="*")
@dataclass
class Movie:
   title: str
   year: int

@app.command
def add(movie: Movie):
   print(f"Adding movie: {movie}")

app()
```

## 23.5 Sharing Parameters

A flattened dataclass provides a natural way of easily sharing a set of parameters between commands.

```python
from cyclopts import App, Parameter
from dataclasses import dataclass

app = App(name="movie-maintainer")

@Parameter(name="*")
@dataclass
class Config:
    user: str
    server: str = "media.sqlite"

@dataclass
class Movie:
    title: str
    year: int

@app.command
def add(movie: Movie, *, config: Config):
    print(f"Config: {config}")
    print(f"Adding movie: {movie}")

@app.command
def remove(movie: Movie, *, config: Config):
    print(f"Config: {config}")
    print(f"Removing movie: {movie}")

app()
```

```
$ movie-maintainer remove --help
Usage: movie-maintainer remove [ARGS] [OPTIONS]

╭─ Parameters ─────────────────────────────────────────────────╮
│ *  MOVIE.TITLE              [required]                       │
│      --movie.title                                           │
│ *  MOVIE.YEAR --movie.year  [required]                       │
│ *  --user                   [required]                       │
│    --server                 [default: media.sqlite]          │
╰──────────────────────────────────────────────────────────────╯

$ movie-maintainer remove 'Mad Max: Fury Road' 2015 --user Guido
Config: Config(user='Guido', server='media.sqlite')
Removing movie: Movie(title='Mad Max: Fury Road', year=2015)
```

## 23.6 Config File

Having the user specify `--user` every single call is a bit cumbersome, especially if they're always going to provide the same value. We can have Cyclopts fallback to a *toml configuration file*.

Consider the following toml data saved to `config.toml`:

```
# config.toml
user = "Guido"
```

We can update our app to fill in missing CLI parameters from this file:

```
from cyclopts import App, Parameter, config
from dataclasses import dataclass
from typing import Annotated

app = App(
   name="movie-maintainer",
   config=config.Toml("config.toml", use_commands_as_keys=False),
)

@Parameter(name="*")
@dataclass
class Config:
   user: str
   server: str = "media.sqlite"

@dataclass
class Movie:
   title: str
   year: int

@app.command
def add(movie: Movie, *, config: Config):
   print(f"Config: {config}")
   print(f"Adding movie: {movie}")

app()
```

```
$ movie-maintainer add 'Mad Max: Fury Road' 2015
Config: Config(user='Guido', server='media.sqlite')
Adding movie: Movie(title='Mad Max: Fury Road', year=2015)
```

CHAPTER

# TWENTYFOUR

# ARGS & KWARGS

In python, a function can consume a variable number of positional and keyword arguments:

```
def foo(normal_required_variable, *args, **kwargs):
    pass
```

There is **nothing special** about the names `args` and `kwargs`; the functionality is derived from the leading * and **. `args` and `kwargs` are the defacto standard names for these variables. In this document, we'll usually just refer to them as `*args` and `**kwargs`.

Cyclopts commands may consume a variable number of positional and keyword arguments. The priority ruleset is as follows:

1. `--keyword` CLI arguments first get matched to normal variable parameters.
2. Unmatched keywords get consumed by `**kwargs`, if specified.
3. All remaining tokens get consumed by `*args`, if specified. A prevalant use-case is in a typical *Meta App*.

## 24.1 Args (Variable Positional)

A variable number of positional arguments consume all remaining positional arguments from the command-line. Individual elements are converted to the annotated type.

```
from cyclopts import App

app = App()

@app.command
def foo(name: str, *favorite_numbers: int):
    print(f"{name}'s favorite numbers are: {favorite_numbers}")

app()
```

```
$ my-script foo Brian
Brian's favorite numbers are: ()

$ my-script foo Brian 777
Brian's favorite numbers are: (777,)

$ my-script foo Brian 777 2
Brian's favorite numbers are: (777, 2)
```

## 24.2 Kwargs (Variable Keywords)

A variable number of keyword arguments consume all remaining CLI tokens starting with `--`. Individual values are converted to the annotated type.

```python
from cyclopts import App

app = App()

@app.command
def add(**country_to_capitols):
    for country, capitol in country_to_capitols.items():
        print(f"Adding {country} with capitol {capitol}.")

app()
```

```
$ my-script add --united-states="Washington, D.C." --canada=Ottawa
Adding united-states with capitol Washington, D.C..
Adding canada with capitol Ottawa.
```

CHAPTER

# TWENTYFIVE

# CONFIG FILES

For more complicated CLI applications, it is common to have an external user configuration file. For example, the popular python tools `poetry`, `ruff`, and `pytest` are all configurable from a `pyproject.toml` file. The *App.config* attribute accepts a callable (or list of callables) that add (or remove) values to the parsed CLI tokens. The provided callable must have signature:

```
def config(app: "App", commands: Tuple[str, ...], arguments: ArgumentCollection):
    """Modifies the argument collection inplace with some injected values.

    Parameters
    ----------
    app: App
       The current command app being executed.
    commands: Tuple[str, ...]
       The CLI strings that led to the current command function.
    arguments: ArgumentCollection
       Complete ArgumentCollection for the app.
       Modify this collection inplace to influence values provided to the function.
    """
    ...
```

The provided `config` does not have to be a function; all the Cyclopts builtin configs are classes that implement the `__call__` method. The Cyclopts builtins offer good standard functionality for common configuration files like yaml or toml.

## 25.1 TOML Example

In this example, we create a small CLI tool that counts the number of times a given character occurs in a file.

```
# character-counter.py
import cyclopts
from cyclopts import App
from pathlib import Path

app = App(
    name="character-counter",
    config=cyclopts.config.Toml(
        "pyproject.toml",  # Name of the TOML File
        root_keys=["tool", "character-counter"],  # The project's namespace in the TOML.
        # If "pyproject.toml" is not found in the current directory,
        # then iteratively search parenting directories until found.
```

(continues on next page)

(continued from previous page)

```
        search_parents=True,
    ),
)

@app.command
def count(filename: Path, *, character="-"):
    print(filename.read_text().count(character))

if __name__ == "__main__":
    app()
```

Running this code without a `pyproject.toml` present:

```
$ python character-counter.py count README.md
70
$ python character-counter.py count README.md --character=t
380
```

We can have the new default character be `t` by adding the following to `pyproject.toml`:

```
[tool.character-counter.count]
character = "t"
```

Rerunning the app without a specified `--character` will result in using the toml-provided value:

```
$ python character-counter.py count README.md
380
```

## 25.2 User-Specified Config File

Extending the above *TOML Example*, what if we want to allow the user to specify the toml configuration file? This can be accomplished via a *Meta App*.

```
# character-counter.py
from pathlib import Path
from typing import Annotated

import cyclopts
from cyclopts import App, Parameter

app = App(name="character-counter")

@app.command
def count(filename: Path, *, character="-"):
    print(filename.read_text().count(character))

@app.meta.default
def meta(
    *tokens: Annotated[str, Parameter(show=False, allow_leading_hyphen=True)],
    config: Path = Path("pyproject.toml"),
):
    app.config = cyclopts.config.Toml(
```

(continues on next page)

(continued from previous page)

```
        config,
        root_keys=["tool", "character-counter"],
        search_parents=True,
    )

    app(tokens)

if __name__ == "__main__":
    app.meta()
```

## 25.3 Environment Variable Example

To automatically derive and read appropriate environment variables, use the *cyclopts.config.Env* class. Continuing the above TOML example:

```
# character-counter.py
import cyclopts
from pathlib import Path

app = cyclopts.App(
    name="character-counter",
    config=cyclopts.config.Env(
        "CHAR_COUNTER_",  # Every environment variable will begin with this.
    ),
)

@app.command
def count(filename: Path, *, character="-"):
    print(filename.read_text().count(character))

app()
```

*Env* assembles the environment variable name by joining the following components (in-order):

1. The provided `prefix`. In this case, it is `"CHAR_COUNTER_"`.
2. The command and subcommand(s) that lead up to the function being executed.
3. The parameter's CLI name, with the leading `--` stripped, and hyphens `-` replaced with underscores `_`.

Running this code without a specified `--character` results in counting the default `-` character.

```
$ python character-counter.py count README.md
70
```

By exporting a value to `CHAR_COUNTER_COUNT_CHARACTER`, that value will now be used as the default:

```
$ export CHAR_COUNTER_COUNT_CHARACTER=t
$ python character-counter.py count README.md
380
$ python character-counter.py count README.md --character=q
3
```

## 25.4 In-Memory Dict

For configurations that come from sources other than files, use *cyclopts.config.Dict*.

```
# character-counter.py
import json
import cyclopts
from pathlib import Path

def fetch_config():
    """Simulate fetching configuration from an API."""
    return {"count": {"character": "e"}}

config_data = fetch_config_from_api()

app = cyclopts.App(
    name="character-counter",
    config=cyclopts.config.Dict(
        fetch_config,
        # Optional: provide custom source identifier for better error messages
        source="api",
    ),
)

@app.command
def count(filename: Path, *, character="-"):
    print(filename.read_text().count(character))

if __name__ == "__main__":
    app()
```

## 25.5 Combining Multiple Config Sources

You can combine multiple config sources in a single application by passing a sequence to *App.config*. Each configuration is applied sequentially.

In the following example, we combine a TOML file and environment variables, allowing environment variables to override TOML settings:

```
# character-counter.py
import cyclopts
from pathlib import Path

app = cyclopts.App(
    name="character-counter",
    config=[
        # Since Env comes before Toml, it has priority.
        cyclopts.config.Env("CHAR_COUNTER_"),
        cyclopts.config.Toml(
            "pyproject.toml",
            root_keys=["tool", "character-counter"],
            search_parents=True,
        ),
```

(continues on next page)

(continued from previous page)

```
    ],
)

@app.command
def count(filename: Path, *, character="-"):
    print(filename.read_text().count(character))

if __name__ == "__main__":
    app()
```

With this setup, the configuration is resolved in the following order:

1. **CLI arguments** (if provided) override everything else
2. **Environment variables** (prefixed with `CHAR_COUNTER_`) can override TOML values
3. **TOML file** (pyproject.toml) provides the base configuration
4. **Python default** the default value `-` in the python code.

For example, with `pyproject.toml` containing:

```
[tool.character-counter.count]
character = "t"
```

You can override it via environment variable:

```
$ CHAR_COUNTER_COUNT_CHARACTER=a python character-counter.py count README.md
```

Or via CLI argument:

```
$ python character-counter.py count README.md --character=x
```

CHAPTER

# TWENTYSIX

# SPHINX INTEGRATION

Cyclopts provides builtin Sphinx support.

**Table of Contents**


- *Quick Start*
- *Directive Usage*
  - *Basic Syntax*
  - *Module Path Formats*
- *Directive Options*
  - `:heading-level:` - *Heading Level*
  - `:max-heading-level:` - *Maximum Heading Level*
  - `:no-recursive:` - *Exclude Subcommands*
  - `:include-hidden:` - *Show Hidden Commands*
  - `:flatten-commands:` - *Generate Flat Command Hierarchy*
  - `:code-block-title:` - *Render Titles as Inline Code*
  - `:commands:` - *Filter Specific Commands*
  - `:exclude-commands:` - *Exclude Specific Commands*
  - `:skip-preamble:` - *Skip Description and Usage*
  - *Automatic Reference Labels*
- *Complete Example*
  - *CLI Application (`myapp/cli.py`):*
  - *Sphinx Configuration (`docs/conf.py`):*
  - *Documentation File (`docs/cli.rst`):*
- *Advanced Usage*
  - *Using Distinct Command Headings*
  - *Selective Command Documentation*
- *Output Formats*

- *See Also*

## 26.1 Quick Start

1. Add the extension to your Sphinx configuration (`docs/conf.py`):

```
extensions = [
    'cyclopts.sphinx_ext',  # Add this line
    # ... your other extensions
]
```

2. Use the directive in your RST files:

```
.. cyclopts:: mypackage.cli:app
```

## 26.2 Directive Usage

### 26.2.1 Basic Syntax

The `cyclopts` directive accepts a module path to your Cyclopts `App` object:

```
.. cyclopts:: mypackage.cli:app
```

### 26.2.2 Module Path Formats

The directive accepts two module path formats:

1. **Explicit format** (`module.path:app_name`):

```
.. cyclopts:: mypackage.cli:app
.. cyclopts:: myapp.commands:main_app
.. cyclopts:: src.cli:cli
```

   This explicitly specifies which `App` object to document.

2. **Automatic discovery** (`module.path`):

```
.. cyclopts:: mypackage.cli
.. cyclopts:: myapp.main
```

   The extension will search the module for an `App` instance, looking for common names like `app`, `cli`, or `main`.

## 26.3 Directive Options

The directive supports several options to customize the generated documentation:

### 26.3.1 `:heading-level:` - Heading Level

Set the starting heading level for the generated documentation (1-6, default: 2):

```
.. cyclopts:: mypackage.cli:app
   :heading-level: 3
```

This is useful when you need to adjust the heading hierarchy. The default of 2 works well for most cases where the directive is placed under a page title.

### 26.3.2 `:max-heading-level:` - Maximum Heading Level

Set the maximum heading level to use (1-6, default: 6):

```
.. cyclopts:: mypackage.cli:app
   :max-heading-level: 4
```

Headings deeper than this level will be capped at this value. This is useful for deeply nested command hierarchies where you want to prevent headings from becoming too small.

### 26.3.3 `:no-recursive:` - Exclude Subcommands

Disable recursive documentation of subcommands (by default, subcommands are included):

```
.. cyclopts:: mypackage.cli:app
   :no-recursive:
```

When this flag is present, only the top-level commands are documented.

### 26.3.4 `:include-hidden:` - Show Hidden Commands

Include commands marked with `show=False` in the documentation:

```
.. cyclopts:: mypackage.cli:app
   :include-hidden: true
```

By default, hidden commands are not included in the generated documentation.

### 26.3.5 `:flatten-commands:` - Generate Flat Command Hierarchy

Generate all commands at the same heading level instead of nested hierarchy:

```
.. cyclopts:: mypackage.cli:app
   :flatten-commands:
```

This creates distinct, equally-weighted headings for each command and subcommand, making them easier to reference and navigate in the documentation. Without this option, subcommands are nested with incrementing heading levels.

### 26.3.6 `:code-block-title:` - Render Titles as Inline Code

Render command titles with inline code formatting instead of plain text:

```
.. cyclopts:: mypackage.cli:app
   :code-block-title:
```

When this flag is present, command titles are rendered with monospace formatting, which can be useful for certain documentation themes or to make command names stand out visually.

### 26.3.7 :commands: - Filter Specific Commands

Document only specific commands from your CLI application:

```
.. cyclopts:: mypackage.cli:app
   :commands: init, build, deploy
```

This will only document the specified commands. You can also use nested command paths with dot notation:

```
.. cyclopts:: mypackage.cli:app
   :commands: db.migrate, db.backup, api
```

- `db.migrate` - Documents only the `migrate` subcommand under `db`
- `db.backup` - Documents only the `backup` subcommand under `db`
- `api` - Documents the `api` command and all its subcommands

You can use either underscore or dash notation in command names - they will be normalized automatically.

### 26.3.8 :exclude-commands: - Exclude Specific Commands

Exclude specific commands from the documentation:

```
.. cyclopts:: mypackage.cli:app
   :exclude-commands: debug, internal-test
```

This is useful for hiding internal or debug commands from user-facing documentation. Like `:commands:`, this also supports nested command paths with dot notation.

### 26.3.9 :skip-preamble: - Skip Description and Usage

Skip the description and usage sections for the target command when filtering to a single command:

```
.. cyclopts:: mypackage.cli:app
   :commands: deploy
   :skip-preamble:
```

When you filter to a single command using `:commands:` and provide your own section heading in the RST, you may not want the directive to generate the command's description and usage block. Adding `:skip-preamble:` suppresses these sections while still generating the command's parameters and subcommands.

This is useful when you want to write your own introduction for a command section:

```
Deployment
==========

Deploy your application to production with these commands.

.. cyclopts:: mypackage.cli:app
   :commands: deploy
   :skip-preamble:
```

Without `:skip-preamble:`, the output would include both your introduction and the command's docstring description, which can be redundant.

### 26.3.10 Automatic Reference Labels

The Sphinx directive automatically generates RST reference labels for all commands, enabling cross-referencing throughout your documentation. The anchor format is `cyclopts-{app-name}-{command-path}`, which prevents naming conflicts when documenting multiple CLIs.

For example: - Root application: `cyclopts-myapp` - Subcommand: `cyclopts-myapp-deploy` - Nested subcommand: `cyclopts-myapp-deploy-production`

You can reference these commands elsewhere in your documentation using :ref:`cyclopts-myapp-deploy`.

## 26.4 Complete Example

Here's a complete example showing a CLI application and its Sphinx documentation:

### 26.4.1 CLI Application (myapp/cli.py):

```python
from pathlib import Path
from typing import Optional
from cyclopts import App

app = App(
    name="myapp",
    help="My awesome CLI application",
    version="1.0.0"
)

@app.command
def init(path: Path = Path("."), template: str = "default"):
    """Initialize a new project.

    Parameters
    ----------
    path : Path
        Directory where the project will be created
    template : str
        Project template to use
    """
    print(f"Initializing project at {path}")

@app.command
def build(source: Path, output: Optional[Path] = None, *, minify: bool = False):
    """Build the project.

    Parameters
    ----------
    source : Path
        Source directory
    output : Path, optional
        Output directory (defaults to source/dist)
    minify : bool
        Minify the output files
    """
    output = output or source / "dist"
```

(continues on next page)

(continued from previous page)

```
    print(f"Building from {source} to {output}")

if __name__ == "__main__":
    app()
```

### 26.4.2 Sphinx Configuration (docs/conf.py):

```
import sys
from pathlib import Path

# Add your package to the path
sys.path.insert(0, str(Path(__file__).parent.parent))

# Extensions
extensions = [
    'cyclopts.sphinx_ext',
    'sphinx.ext.autodoc',  # For API docs
    'sphinx.ext.napoleon',  # For NumPy-style docstrings
]

# Project info
project = 'MyApp'
author = 'Your Name'
version = '1.0.0'

# HTML theme
html_theme = 'sphinx_rtd_theme'
```

### 26.4.3 Documentation File (docs/cli.rst):

```
CLI Reference
=============

This section documents all available CLI commands.

.. cyclopts:: myapp.cli:app

The above directive will automatically generate documentation for all
commands, including their parameters, types, defaults, and help text.
```

## 26.5 Advanced Usage

### 26.5.1 Using Distinct Command Headings

When you want each command to have its own distinct heading for better navigation and referencing:

```
CLI Command Reference
=====================

.. cyclopts:: myapp.cli:app
```

(continues on next page)

(continued from previous page)

```
   :flatten-commands:
   :code-block-title:

This generates:

- All commands at the same heading level (not nested)
- Command titles with monospace formatting
- Automatic reference labels for cross-linking

You can then reference specific commands:

See :ref:`cyclopts-myapp-deploy` for deployment options.
The :ref:`cyclopts-myapp-init` command sets up your project.
```

### 26.5.2 Selective Command Documentation

Split your CLI documentation across multiple sections or pages:

```
Database Commands
=================

The following commands manage database operations:

.. cyclopts:: myapp.cli:app
   :commands: db

API Management
==============

Commands for controlling the API server:

.. cyclopts:: myapp.cli:app
   :commands: api

Development Tools
=================

Utilities for development (excluding internal debug commands):

.. cyclopts:: myapp.cli:app
   :commands: dev
   :exclude-commands: dev.debug, dev.internal
```

This approach allows you to:

- Organize large CLI applications into logical sections
- Document different command groups on separate pages
- Exclude internal or debug commands from user documentation
- Create targeted documentation for different audiences

## 26.6 Output Formats

While the Sphinx directive uses RST internally, you can also generate documentation programmatically in multiple formats:

```
from myapp.cli import app

# Generate reStructuredText
rst_docs = app.generate_docs(output_format="rst")

# Generate Markdown
md_docs = app.generate_docs(output_format="markdown")

# Generate HTML
html_docs = app.generate_docs(output_format="html")
```

This is useful for generating documentation outside of Sphinx, such as for GitHub README files or other documentation systems.

## 26.7 See Also

- *Help* - Customizing help output
- *Commands* - Creating commands and subcommands
- *Parameters* - Parameter types and validation
- Sphinx Documentation - Official Sphinx documentation

CHAPTER

# TWENTYSEVEN

# MKDOCS INTEGRATION

Cyclopts provides builtin MkDocs support.

**Warning**

The MkDocs plugin is **experimental** and may have breaking changes in future releases. If you encounter any issues or have feedback, please report them on GitHub.

**Table of Contents**


- *Quick Start*
- *Directive Usage*
  - *Basic Syntax*
  - *Module Path Formats*
- *Directive Options*
  - `module` - *Module Path (Required)*
  - `heading_level` - *Heading Level*
  - `max_heading_level` - *Maximum Heading Level*
  - `recursive` - *Include Subcommands*
  - `include_hidden` - *Show Hidden Commands*
  - `flatten_commands` - *Generate Flat Command Hierarchy*
  - `generate_toc` - *Generate Table of Contents*
  - `code_block_title` - *Render Titles as Inline Code*
  - `commands` - *Filter Specific Commands*
  - `exclude_commands` - *Exclude Specific Commands*
  - `skip_preamble` - *Skip Description and Usage*
- *Complete Example*
  - *CLI Application (`myapp/cli.py`):*
  - *MkDocs Configuration (`mkdocs.yml`):*

- *Documentation File (`docs/cli-reference.md`):*
- *Advanced Usage*
  - *Using Flat Command Structure*
  - *Selective Command Documentation*
- *See Also*


## 27.1 Quick Start

1. Install Cyclopts with MkDocs support:

```
pip install cyclopts[mkdocs]
```

2. Add the plugin to your MkDocs configuration (`mkdocs.yml`):

```
plugins:
  - cyclopts
```

3. Use the directive in your Markdown files:

```
::: cyclopts
    module: mypackage.cli:app
```

## 27.2 Directive Usage

### 27.2.1 Basic Syntax

The `::: cyclopts` directive uses YAML format and accepts a module path to your Cyclopts `App` object:

```
::: cyclopts
    module: mypackage.cli:app
```

### 27.2.2 Module Path Formats

The directive accepts two module path formats:

1. **Explicit format** (`module.path:app_name`):

```
::: cyclopts
    module: mypackage.cli:app

::: cyclopts
    module: myapp.commands:main_app

::: cyclopts
    module: src.cli:cli
```

   This explicitly specifies which `App` object to document.

2. **Automatic discovery** (`module.path`):

```
::: cyclopts
    module: mypackage.cli

::: cyclopts
    module: myapp.main
```

The plugin will search the module for an `App` instance, looking for common names like `app`, `cli`, or `main`.

## 27.3 Directive Options

The directive supports several options to customize the generated documentation. All options use standard YAML syntax:

### 27.3.1 `module` - Module Path (Required)

The module path to your Cyclopts App instance:

```
::: cyclopts
    module: mypackage.cli:app
```

This is the only required option.

### 27.3.2 `heading_level` - Heading Level

Set the starting heading level for the generated documentation (1-6, default: 2):

```
::: cyclopts
    module: mypackage.cli:app
    heading_level: 3
```

This is useful when you need to adjust the heading hierarchy. The default of 2 works well for most cases where the directive is placed under a page title.

### 27.3.3 `max_heading_level` - Maximum Heading Level

Set the maximum heading level to use (1-6, default: 6):

```
::: cyclopts
    module: mypackage.cli:app
    max_heading_level: 4
```

Headings deeper than this level will be capped at this value. This is useful for deeply nested command hierarchies where you want to prevent headings from becoming too small.

### 27.3.4 `recursive` - Include Subcommands

Control whether to document subcommands recursively (default: true):

```
::: cyclopts
    module: mypackage.cli:app
    recursive: false
```

Set to `false` to only document the top-level commands.

### 27.3.5 `include_hidden` - Show Hidden Commands

Include commands marked with `show=False` in the documentation:

```
::: cyclopts
    module: mypackage.cli:app
    include_hidden: true
```

By default, hidden commands are not included in the generated documentation.

### 27.3.6 `flatten_commands` - Generate Flat Command Hierarchy

Generate all commands at the same heading level instead of nested hierarchy:

```
::: cyclopts
    module: mypackage.cli:app
    flatten_commands: true
```

This creates distinct, equally-weighted headings for each command and subcommand, making them easier to reference and navigate in the documentation. Without this option, subcommands are nested with incrementing heading levels.

### 27.3.7 `generate_toc` - Generate Table of Contents

Control whether to generate a table of contents for multi-command apps (default: true):

```
::: cyclopts
    module: mypackage.cli:app
    generate_toc: false
```

This is useful when you want to suppress the automatic table of contents, especially when using multiple directives on the same page or when you have your own navigation structure.

### 27.3.8 `code_block_title` - Render Titles as Inline Code

Render command titles with inline code formatting (backticks) instead of plain text:

```
::: cyclopts
    module: mypackage.cli:app
    code_block_title: true
```

When enabled, command titles are rendered as `#### `command-name`` instead of `#### command-name`. This makes command names appear with monospace formatting, which can be useful for certain documentation themes or to make command names stand out visually.

### 27.3.9 `commands` - Filter Specific Commands

Document only specific commands from your CLI application:

```
::: cyclopts
    module: mypackage.cli:app
    commands:
      - init
      - build
      - deploy
```

This will only document the specified commands. You can also use nested command paths with dot notation:

```
::: cyclopts
    module: mypackage.cli:app
    commands:
      - db.migrate
      - db.backup
      - api
```

Or use inline YAML list syntax:

```
::: cyclopts
    module: mypackage.cli:app
    commands: [db.migrate, db.backup, api]
```

- `db.migrate` - Documents only the `migrate` subcommand under `db`
- `db.backup` - Documents only the `backup` subcommand under `db`
- `api` - Documents the `api` command and all its subcommands

You can use either underscore or dash notation in command names - they will be normalized automatically.

### 27.3.10 `exclude_commands` - Exclude Specific Commands

Exclude specific commands from the documentation:

```
::: cyclopts
    module: mypackage.cli:app
    exclude_commands:
      - debug
      - internal-test
```

This is useful for hiding internal or debug commands from user-facing documentation. Like `commands`, this also supports nested command paths with dot notation and inline YAML list syntax.

### 27.3.11 `skip_preamble` - Skip Description and Usage

Skip the description and usage sections for the target command when filtering to a single command:

```
::: cyclopts
    module: mypackage.cli:app
    commands: [deploy]
    skip_preamble: true
```

When you filter to a single command using `commands` and provide your own section heading in the Markdown, you may not want the plugin to generate the command's description and usage block. Setting `skip_preamble: true` suppresses these sections while still generating the command's parameters and subcommands.

This is useful when you want to write your own introduction for a command section:

```
## Deployment

Deploy your application to production with these commands.

::: cyclopts
    module: mypackage.cli:app
```

(continues on next page)

(continued from previous page)

```
    commands: [deploy]
    skip_preamble: true
```

Without `skip_preamble`, the output would include both your introduction and the command's docstring description, which can be redundant.

## 27.4 Complete Example

Here's a complete example showing a CLI application and its MkDocs documentation:

### 27.4.1 CLI Application (`myapp/cli.py`):

```python
from pathlib import Path
from typing import Optional
from cyclopts import App

app = App(
    name="myapp",
    help="My awesome CLI application",
    version="1.0.0"
)

@app.command
def init(path: Path = Path("."), template: str = "default"):
    """Initialize a new project.

    Parameters
    ----------
    path : Path
        Directory where the project will be created
    template : str
        Project template to use
    """
    print(f"Initializing project at {path}")

@app.command
def build(source: Path, output: Optional[Path] = None, *, minify: bool = False):
    """Build the project.

    Parameters
    ----------
    source : Path
        Source directory
    output : Path, optional
        Output directory (defaults to source/dist)
    minify : bool
        Minify the output files
    """
    output = output or source / "dist"
    print(f"Building from {source} to {output}")
```

(continues on next page)

(continued from previous page)

```
if __name__ == "__main__":
    app()
```

### 27.4.2 MkDocs Configuration (`mkdocs.yml`):

```
site_name: MyApp Documentation
site_description: Documentation for MyApp CLI

theme:
  name: readthedocs

plugins:
  - search
  - cyclopts

nav:
  - Home: index.md
  - CLI Reference: cli-reference.md
  - User Guide: guide.md

markdown_extensions:
  - admonition
  - codehilite
  - toc:
      permalink: true
```

### 27.4.3 Documentation File (`docs/cli-reference.md`):

```
# CLI Reference

This section documents all available CLI commands.

::: cyclopts
    module: myapp.cli:app
    heading_level: 2
    recursive: true

The above directive will automatically generate documentation for all
commands, including their parameters, types, defaults, and help text.
```

## 27.5 Advanced Usage

### 27.5.1 Using Flat Command Structure

When you want each command to have its own distinct heading for better navigation:

```
# CLI Command Reference

::: cyclopts
    module: myapp.cli:app
```

(continues on next page)

(continued from previous page)

```
    flatten_commands: true

This generates all commands at the same heading level (not nested),
making it easier to navigate and reference specific commands.
```

### 27.5.2 Selective Command Documentation

Split your CLI documentation across multiple sections or pages:

```
## Database Commands

The following commands manage database operations:

::: cyclopts
    module: myapp.cli:app
    commands: [db]
    recursive: true

## API Management

Commands for controlling the API server:

::: cyclopts
    module: myapp.cli:app
    commands: [api]
    recursive: true

## Development Tools

Utilities for development (excluding internal debug commands):

::: cyclopts
    module: myapp.cli:app
    commands: [dev]
    exclude_commands: [dev.debug, dev.internal]
    recursive: true
```

This approach allows you to:

- Organize large CLI applications into logical sections
- Document different command groups on separate pages
- Exclude internal or debug commands from user documentation
- Create targeted documentation for different audiences

## 27.6 See Also

- *Sphinx Integration* - Sphinx integration (similar functionality)
- *Help* - Customizing help output
- *Commands* - Creating commands and subcommands

- *Parameters* - Parameter types and validation
- MkDocs Documentation - Official MkDocs documentation

CHAPTER

TWENTYEIGHT

# PACKAGING

Packaging is bundling up your python library so that it can be easily `pip install` by others.

Typically this involves:

1. Bundling the code into a Built Distribution (wheel) and/or Source Distribution (sdist).
2. Uploading (publishing) the distribution(s) to python package repository, like PyPI.

This section is a brief bootcamp on package **configuration** for a CLI application. This is **not** intended to be a complete tutorial on python packaging and publishing. In this tutorial, replace all instances of `mypackage` with your own project name.

## 28.1 __main__.py

In python, if you have a module `mypackage/__main__.py`, it will be executed with the bash command `python -m mypackage`.

A pretty bare-bones Cyclopts `mypackage/__main__.py` will look like:

```
# mypackage/__main__.py

import cyclopts

app = cyclopts.App()

@app.command
def foo(name: str):
    print(f"Hello {name}!")

if __name__ == "__main__":
    app()
```

```
$ python -m mypackage World
Hello World!
```

## 28.2 Entrypoints

If you want your application to be callable like a standard bash executable (i.e. `my-package` instead of `python -m mypackage`), we'll need to add an entrypoint.

Modern Python projects typically use `pyproject.toml` for configuration. The standard way to define console scripts is:

```toml
# pyproject.toml
[project.scripts]
my-package = "mypackage.__main__:main"
```

This creates an executable named `my-package` that executes function `main` (from the right of the colon) from the python module `mypackage.__main__`. Note that this configuration is independent of any special naming, like `__main__` or `main`.

### 28.2.1 Legacy Configurations

For older projects, you may encounter these alternative formats:

**setup.py:**

```python
# setup.py
from setuptools import setup

setup(
    # There should be a lot more fields populated here.
    entry_points={
        "console_scripts": [
            "my-package = mypackage.__main__:main",
        ]
    },
)
```

**setup.cfg:**

```
# setup.cfg
[options.entry_points]
console_scripts =
    my-package = mypackage.__main__:main
```

**Poetry:**

```toml
# pyproject.toml
[tool.poetry.scripts]
my-package = "mypackage.__main__:main"
```

The setuptools entrypoint documentation goes into further detail.

## 28.3 Result Action

When using Cyclopts as a CLI application, command return values are automatically handled appropriately. By default, *App* uses `"print_non_int_sys_exit"` mode, which calls `sys.exit()` with the appropriate exit code:

- String returns are printed to stdout, then `sys.exit(0)` is called
- Integer returns are passed to `sys.exit(int)` as the exit code
- Boolean returns are converted: `True` → `sys.exit(0)`, `False` → `sys.exit(1)`
- `None` returns call `sys.exit(0)`

This default behavior makes Cyclopts applications work consistently whether run directly as scripts or installed via console_scripts entry points. The *result_action* can be customized if different behavior is needed:

CHAPTER

# TWENTYNINE

# APP CALLING & RETURN VALUES

In this section, we'll take a closer look at the *App.__call__()* method.

## 29.1 Input Command

Typically, a Cyclopts app looks something like:

```
from cyclopts import App

app = App()

@app.command
def foo(a: int, b: int, c: int):
    print(a + b + c)

app()
```

```
$ my-script 1 2 3
6
```

*App.__call__()* takes in an optional input that it parses into an action. If not specified, Cyclopts defaults to `sys.argv[1:]`, i.e. the list of command line arguments. An explicit string or list of strings can instead be passed in.

```
app("foo 1 2 3")
# 6
app(["foo", "1", "2", "3"])
# 6
```

If a string is passed in, it will be internally converted into a list using shlex.split.

## 29.2 Return Value

The app invocation processes the command's return value based on *App.result_action*. By default, Cyclopts calls `sys.exit()` with an appropriate exit code:

```
from cyclopts import App

app = App()  # Default result_action="print_non_int_sys_exit"

@app.command
```

(continues on next page)

(continued from previous page)

```python
def success():
    return 0  # Exit code for success

@app.command
def greet(name: str) -> str:
    return f"Hello {name}!"  # Prints and exits with 0

if __name__ == "__main__":
    app()  # Will call sys.exit with the returned 0 error code (success).
```

Installed scripts call `sys.exit()` with the returned value of the entry point. So the default Cyclopts *App.result_action* will have consistent behavior for standalone scripts and installed apps.

For embedding Cyclopts in other Python code or testing, use `result_action="return_value"` to get the raw command return value without calling `sys.exit()`:

```python
from cyclopts import App

app = App(result_action="return_value")

@app.command
def foo(a: int, b: int, c: int):
    return a + b + c

return_value = app("foo 1 2 3")  # no longer exits!
print(f"The return value was: {return_value}.")
# The return value was: 6.
```

See *Result Action* for all available modes and detailed behavior.

## 29.3 Exception Handling and Exiting

For the most part, Cyclopts is **hands-off** when it comes to handling exceptions and exiting the application. However, by default, if there is a **Cyclopts runtime error**, like *CoercionError* or a *ValidationError*, then Cyclopts will perform a `sys.exit(1)`. This is to avoid displaying the unformatted, uncaught exception to the CLI user.

These behaviors can be controlled via *App* attributes or method parameters:

- *App.exit_on_error* - Calls `sys.exit(1)` on errors (defaults to `True`)
- *App.print_error* - Formatted errors are printed (defaults to `True`)
- *App.help_on_error* - The help-page is printed before errors (defaults to `False`)
- *App.verbose* - Include verbose error information that might be useful for **developers** using Cyclopts (defaults to `False`)

These attributes are inherited by child apps and can be overridden by providing parameters to method calls.

**Note**

Cyclopts separates normal output from error messages using two different consoles:

- *App.console* - Used for normal output like help messages and version information (defaults to stdout)
- *App.error_console* - Used for error messages like parsing errors and exceptions (defaults to stderr)

**Setting at App Level:**

```
# Configure error handling at the app level
app = App(
    exit_on_error=False,  # Don't exit on errors
    print_error=False,    # Don't print formatted errors
)

# Child apps inherit these settings
child_app = App(name="child")
app.command(child_app)
```

**Method-Level Override:**

```
app("this-is-not-a-registered-command")
print("this will not be printed since cyclopts exited above.")
# ╭─ Error ──────────────────────────────────────────────────────────────────────╮
# │ Unknown command "this-is-not-a-registered-command".                          │
# ╰──────────────────────────────────────────────────────────────────────────────╯

app("this-is-not-a-registered-command", exit_on_error=False, print_error=False)
# Traceback (most recent call last):
#   File "/cyclopts/scratch.py", line 9, in <module>
#     app("this-is-not-a-registered-command", exit_on_error=False, print_error=False)
#   File "/cyclopts/cyclopts/core.py", line 1102, in __call__
#     command, bound, _ = self.parse_args(
#   File "/cyclopts/cyclopts/core.py", line 1037, in parse_args
#     command, bound, unused_tokens, ignored, argument_collection = self._parse_known_
↪args(
#   File "/cyclopts/cyclopts/core.py", line 966, in _parse_known_args
#     raise UnknownCommandError(unused_tokens=unused_tokens)
# cyclopts.exceptions.UnknownCommandError: Unknown command "this-is-not-a-registered-
↪command".

try:
    app("this-is-not-a-registered-command", exit_on_error=False, print_error=False)
except CycloptsError:
    pass
print("Execution continues since we caught the exception.")
```

With `exit_on_error=False`, the `UnknownCommandError` is raised the same as a normal python exception.

CHAPTER

# THIRTY

# META APP

What if you want more control over the application launch process? Cyclopts provides the option of launching an app from an app; a meta app!

## 30.1 Meta Sub App

Typically, a Cyclopts application is launched by calling the *App* object:

```python
from cyclopts import App

app = App()
# Register some commands here (not shown)
app()  # Run the app
```

To change how the primary app is run, you can use the meta-app feature of Cyclopts. The meta app is a special *App* that inherits configuration from its parent and has its help-page merged with the parent app's help.

```python
from cyclopts import App, Group, Parameter
from typing import Annotated

app = App()
# Rename the meta's "Parameter" -> "Session Parameters".
# Set sort_key so it will be drawn higher up the help-page.
app.meta.group_parameters = Group("Session Parameters", sort_key=0)

@app.command
def foo(loops: int):
    for i in range(loops):
        print(f"Looping! {i}")

@app.meta.default
def my_app_launcher(*tokens: Annotated[str, Parameter(show=False, allow_leading_
↪hyphen=True)], user: str):
    print(f"Hello {user}")
    app(tokens)

app.meta()
```

```
$ my-script --user=Bob foo 3
Hello Bob
Looping! 0
```

(continues on next page)

(continued from previous page)

```
Looping! 1
Looping! 2
```

The variable positional `*tokens` will aggregate all remaining tokens, including those starting with a hyphen (typically options). We can then pass them along to the primary `app`.

The `meta` app inherits many configuration values from its parent app and is additionally scanned when generating help screens. `*tokens` is annotated with `show=False` since we do not want this variable to show up in the help screen.

```
$ my-script --help
Usage: my-script COMMAND

╭─ Session Parameters ─────────────────────────────────────────────────────────╮
│ *  --user  [required]                                                        │
╰──────────────────────────────────────────────────────────────────────────────╯
╭─ Commands ───────────────────────────────────────────────────────────────────╮
│ foo                                                                          │
│ --help,-h  Display this message and exit.                                    │
│ --version  Display application version.                                      │
╰──────────────────────────────────────────────────────────────────────────────╯
```

## 30.2 Meta Commands

If you want a command to circumvent `my_app_launcher`, add it as you would any other command to the meta app.

```
@app.meta.command
def info():
    print("CLI didn't have to provide --user to call this.")
```

```
$ my-script info
CLI didn't have to provide --user to call this.

$ my-script --help
Usage: my-script COMMAND

╭─ Session Parameters ─────────────────────────────────────────────────────────╮
│ *  --user  [required]                                                        │
╰──────────────────────────────────────────────────────────────────────────────╯
╭─ Commands ───────────────────────────────────────────────────────────────────╮
│ foo                                                                          │
│ info                                                                         │
│ --help,-h  Display this message and exit.                                    │
│ --version  Display application version.                                      │
╰──────────────────────────────────────────────────────────────────────────────╯
```

Just like a standard application, the parsed `command` executes instead of `default`.

## 30.3 Custom Command Invocation

The core logic of *App.__call__()* method is the following:

```
def __call__(self, tokens=None, **kwargs):
    command, bound, ignored = self.parse_args(tokens, **kwargs)
    return command(*bound.args, **bound.kwargs)
```

Knowing this, we can easily customize how we actually invoke actions with Cyclopts. Let's imagine that we want to instantiate an object, `User` in our meta app, and pass it to subsequent commands that need it. This might be useful to share an expensive-to-create object amongst commands in a single session; see *Command Chaining*.

```
from cyclopts import App, Parameter
from typing import Annotated

app = App()

class User:
    def __init__(self, name):
        self.name = name

@app.command
def create(
    age: int,
    *,
    user_obj: Annotated[User, Parameter(parse=False)],
):
    print(f"Creating user {user_obj.name} with age {age}.")

@app.meta.default
def launcher(*tokens: Annotated[str, Parameter(show=False, allow_leading_hyphen=True)],␣
↪user: str):
    additional_kwargs = {}
    command, bound, ignored = app.parse_args(tokens)
    # "ignored" is a dict mapping python-variable-name to it's type annotation for␣
↪parameters with "parse=False".
    if "user_obj" in ignored:
        # 'ignored["user_obj"]' is the class "User"
        additional_kwargs["user_obj"] = ignored["user_obj"](user)
    return command(*bound.args, **bound.kwargs, **additional_kwargs)

if __name__ == "__main__":
    app.meta()
```

```
$ my-script create --user Alice 30
Creating user Alice with age 30.
```

The `parse=False` configuration tells Cyclopts to not try and bind arguments to this parameter. Cyclopts will pass it along to `ignored` to make custom meta-app logic easier. The annotated parameter **must** be a keyword-only parameter.

> **Tip**
>
> For app-wide control over which parameters are parsed, *parse* can also accept a **regex pattern**. This can be useful

for automatically skipping all "private" parameters (e.g., those prefixed with _) with the regex pattern `"^(?!_)"`. See *Skipping Private Parameters* for details.

CHAPTER

# THIRTYONE

## COMMAND CHAINING

Cyclopts does not natively support command chaining. This is because Cyclopts opted for more flexible and robust CLI parsing, rather than a compromised, inconsistent parsing experience. With that said, Cyclopts gives you the tools to create your own command chaining experience. In this example, we will use a special delimiter token (e.g. `"AND"`) to separate commands.

```python
import itertools
from cyclopts import App, Parameter
from typing import Annotated

app = App()

@app.command
def foo(val: int):
    print(f"FOO {val=}")

@app.command
def bar(flag: bool):
    print(f"BAR {flag=}")

@app.meta.default
def main(*tokens: Annotated[str, Parameter(show=False, allow_leading_hyphen=True)]):
    # tokens is ``["foo", "123", "AND", "foo", "456", "AND", "bar", "--flag"]``
    delimiter = "AND"

    groups = [list(group) for key, group in itertools.groupby(tokens, lambda x: x ==␣
↪delimiter) if not key] or [[]]
    # groups is ``[['foo', '123'], ['foo', '456'], ['bar', '--flag']]``

    for group in groups:
        # Execute each group
        app(group)

if __name__ == "__main__":
    app.meta(["foo", "123", "AND", "foo", "456", "AND", "bar", "--flag"])
    # FOO val=123
    # FOO val=456
    # BAR flag=True
```

CHAPTER

# THIRTYTWO

# AUTOREGISTRY

AutoRegistry is a python library that automatically creates string-to-functionality mappings, making it trivial to instantiate classes or invoke functions from CLI parameters.

Lets consider the following program that can download a file from either a GCP, AWS, or Azure bucket (without worrying about the implementation):

```
import cyclopts
from pathlib import Path
from typing import Literal

def _download_gcp(bucket: str, key: str, dst: Path):
    print("Downloading data from Google.")

def _download_s3(bucket: str, key: str, dst: Path):
    print("Downloading data from Amazon.")

def _download_azure(bucket: str, key: str, dst: Path):
    print("Downloading data from Azure.")

_downloaders = {
    "gcp": _download_gcp,
    "s3": _download_s3,
    "azure": _download_azure,
}

app = cyclopts.App()

@app.command
def download(bucket: str, key: str, dst: Path, provider: Literal[tuple(_downloaders)] =
↪"gcp"):
    downloader = _downloaders[provider]
    downloader(bucket, key, dst)

app()
```

```
$ my-script download --help
╭─ Parameters ─────────────────────────────────────────────────────────────────╮
│ *  BUCKET,--bucket      [required]                                           │
│ *  KEY,--key            [required]                                           │
│ *  DST,--dst            [required]                                           │
│    PROVIDER,--provider  [choices: gcp,s3,azure] [default: gcp]               │
```

(continues on next page)

(continued from previous page)

```
──────────────────────────────────────────────────────────────────────────────

$ my-script my-bucket my-key local.bin --provider=s3
Downloading data from Amazon.
```

Not bad, but let's see how this would look with AutoRegistry.

```
import cyclopts
from autoregistry import Registry
from pathlib import Path
from typing import Literal

_downloaders = Registry(prefix="_download_")

@_downloaders
def _download_gcp(bucket: str, key: str, dst: Path):
    print("Downloading data from Google.")

@_downloaders
def _download_s3(bucket: str, key: str, dst: Path):
    print("Downloading data from Amazon.")

@_downloaders
def _download_azure(bucket: str, key: str, dst: Path):
    print("Downloading data from Azure.")

app = cyclopts.App()

@app.command
def download(bucket: str, key: str, dst: Path, provider: Literal[tuple(_downloaders)] =
↪"gcp"):
    downloader = _downloaders[provider]
    downloader(bucket, key, dst)

app()
```

```
$ my-script download --help
─ Parameters ─────────────────────────────────────────────────────────────────
│ *  BUCKET,--bucket      [required]                                          │
│ *  KEY,--key            [required]                                          │
│ *  DST,--dst            [required]                                          │
│    PROVIDER,--provider  [choices: gcp,s3,azure] [default: gcp]              │
──────────────────────────────────────────────────────────────────────────────

$ my-script my-bucket my-key local.bin --provider=s3
Downloading data from Amazon.
```

Exactly the same functionality, but more terse and organized. With Autoregistry, the download providers are much more self-contained, do not require changes in other code locations, and reduce duplication.

CHAPTER

# THIRTYTHREE

# APP UPGRADE

It's best practice for users to install python-based CLIs via pipx, where each application gets it's own python virtual environment. Whether done via `pipx` or standard `pip`, updating your application can be done via the `upgrade` command. i.e.:

```
$ pipx upgrade mypackage
```

If you would like your CLI application to be able to upgrade itself, you can add the following command to your application:

```
import mypackage
import subprocess
import sys
from cyclopts import App

app = App()

@app.command
def upgrade():
    """Update mypackage to latest stable version."""
    old_version = mypackage.__version__
    subprocess.check_output([sys.executable, "-m", "pip", "install", "--upgrade", "pip"])
    subprocess.check_output([sys.executable, "-m", "pip", "install", "--upgrade",
↪"mypackage"])
    res = subprocess.run([sys.executable, "-m", "mypackage", "--version"],␣
↪stdout=subprocess.PIPE, check=True)
    new_version = res.stdout.decode().strip()
    if old_version == new_version:
        print(f"mypackage up-to-date (v{new_version}).")
    else:
        print(f"mypackage updated from v{old_version} to v{new_version}.")

app()
```

`sys.executable` points to the currently used python interpreter's path; if your package was installed via pipx, then it points to the python interpreter in it's respective virtual environment.

CHAPTER

# THIRTYFOUR

# DATACLASS COMMANDS

An alternative command syntax is to use dataclasses with a `__call__` method. To support this pattern, Cyclopts provides the `"call_if_callable"` result action, which can be composed with other result actions.

## 34.1 Basic Example

Here's a simple example using the dataclass command pattern:

```python
# greeter.py
from dataclasses import dataclass, KW_ONLY
from cyclopts import App

app = App(result_action=["call_if_callable", "print_non_int_sys_exit"])

@app.command
@dataclass
class Greet:
    """Greet someone with a message."""

    name: str = "World"
    _: KW_ONLY
    formal: bool = False

    def __call__(self):
        greeting = "Hello" if self.formal else "Hey"
        return f"{greeting} {self.name}."

if __name__ == "__main__":
    app()
```

Running this application:

```
$ python greeter.py greet
Hey World.

$ python greeter.py greet Alice
Hey Alice.

$ python greeter.py greet Bob --formal
Hello Bob.
```

## 34.2 How It Works

The `result_action=["call_if_callable", "print_non_int_sys_exit"]` creates a pipeline:

1. **call_if_callable**: After parsing, Cyclopts creates an instance of the `Greet` dataclass. This action checks if the result is callable (it is, because of `__call__`), and calls it with no arguments.
2. **print_non_int_sys_exit**: Takes the string returned by `__call__` and prints it, then exits.

Without `"call_if_callable"`, the app would try to print the dataclass instance itself instead of calling it and printing the result.

CHAPTER

# THIRTYFIVE

# INTERACTIVE SHELL & HELP

Cyclopts has a builtin interactive shell-like feature:

```
from cyclopts import App

app = App()

@app.command
def foo(p1):
    """Foo Docstring.

    Parameters
    ----------
    p1: str
        Foo's first parameter.
    """
    print(f"foo {p1}")

@app.command
def bar(p1):
    """Bar Docstring.

    Parameters
    ----------
    p1: str
        Bar's first parameter.
    """
    print(f"bar {p1}")

# A blocking call, launching an interactive shell.
app.interactive_shell(prompt="cyclopts> ")
```

To make the application still work as-expected from the CLI, it is more appropriate to set a command (or `@app.default`) to launch the shell:

```
@app.command
def shell():
    app.interactive_shell()

if __name__ == "__main__":
    app()  # Don't call ``app.interactive_shell()`` here.
```

Special flags like `--help` and `--version` work in the shell, but could be a bit awkward for the root-help:

```
$ python interactive-shell-demo.py
Interactive shell. Press Ctrl-D to exit.
cyclopts> --help
Usage: interactive-shell-demo.py COMMAND

╭─ Parameters ─────────────────────────────────────────────────╮
│ --version      Display application version.                  │
│ --help     -h  Display this message and exit.                │
╰──────────────────────────────────────────────────────────────╯
╭─ Commands ───────────────────────────────────────────────────╮
│ bar  Bar Docstring.                                          │
│ foo  Foo Docstring.                                          │
╰──────────────────────────────────────────────────────────────╯
cyclopts> foo --help
Usage: interactive-shell-demo.py foo [ARGS] [OPTIONS]

Foo Docstring

╭─ Parameters ─────────────────────────────────────────────────╮
│ *  P1,--p1  Foo's first parameter. [required]                │
╰──────────────────────────────────────────────────────────────╯
cyclopts>
```

To resolve this, we can explicitly add a `help` command:

```python
@app.command
def help():
    """Display the help screen."""
    app.help_print()
```

```
$ python interactive-shell-demo.py
Interactive shell. Press Ctrl-D to exit.
cyclopts> help
Usage: interactive-shell-demo.py COMMAND

╭─ Parameters ─────────────────────────────────────────────────╮
│ --version      Display application version.                  │
│ --help     -h  Display this message and exit.                │
╰──────────────────────────────────────────────────────────────╯
╭─ Commands ───────────────────────────────────────────────────╮
│ bar   Bar Docstring.                                         │
│ foo   Foo Docstring.                                         │
│ help  Display the help screen.                               │
╰──────────────────────────────────────────────────────────────╯
```

CHAPTER

# THIRTYSIX

# RICH FORMATTED EXCEPTIONS

Tracebacks of uncaught exceptions provide valuable feedback for debugging. This guide demonstrates how to enhance your error messages using rich formatting.

## 36.1 Standard Python Traceback

Consider the following example:

```
from cyclopts import App

app = App()

@app.default
def main(name: str):
    print(name + 3)

if __name__ == "__main__":
    app()
```

Running this script will produce a standard Python traceback:

```
$ python my-script.py foo
Traceback (most recent call last):
  File "/cyclopts/my-script.py", line 12, in <module>
    app()
  File "/cyclopts/cyclopts/core.py", line 903, in __call__
    return command(*bound.args, **bound.kwargs)
           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "/cyclopts/my-script.py", line 8, in main
    print(name + 3)
          ~~~~~^~~
TypeError: can only concatenate str (not "int") to str
```

## 36.2 Rich Formatted Traceback

To create a more visually appealing and informative traceback, you can use the Rich library's traceback handler. Here's how to modify your script:

```
import sys
from cyclopts import App
```

(continues on next page)

(continued from previous page)

```python
from rich.console import Console
from rich.traceback import install as install_rich_traceback

error_console = Console(stderr=True)
app = App(console=console, error_console=error_console)

# Install rich traceback handler using the error console
install_rich_traceback(console=error_console)

@app.default
def main(name: str):
    print(name + 3)

if __name__ == "__main__":
    app()
```

Now, running the updated script will display a rich-formatted traceback:

```
$ python my-script.py foo
──────────────── Traceback (most recent call last) ────────────────
 /cyclopts/my-script.py:16 in <module>                             
                                                                   
   13                                                              
   14 if __name__ == "__main__":                                   
   15 │   try:                                                     
  16 │   │   app()                                                 
   17 │   except Exception:                                        
   18 │   │   console.print_exception(width=70)                    
   19                                                              
                                                                   
 /cyclopts/cyclopts/core.py:903 in __call__                        
                                                                   
    900 │   │   │   │                                              
    901 │   │   │   │   return asyncio.run(command(*bound.args, **b
    902 │   │   │   else:                                          
   903 │   │   │   │   return command(*bound.kwargs, **bound.kwargs)
    904 │   │   except Exception as e:                             
    905 │   │   │   try:                                           
    906 │   │   │   │   from pydantic import ValidationError as Pyd
                                                                   
 /cyclopts/my-script.py:11 in main                                 
                                                                   
    8                                                              
    9 @app.default                                                 
   10 def main(name: str):                                        
  11 │   print(name + 3)                                          
   12                                                              
   13                                                              
   14 if __name__ == "__main__":                                  
──────────────────────────────────────────────────────────────────
```

This rich-formatted traceback provides a more readable and visually appealing representation of the error, but may make copy/pasting for sharing a bit more cumbersome.

CHAPTER

# THIRTYSEVEN

# SHARING PARAMETERS

Many subcommands within a CLI may take the same parameters. For example, all commands for a CLI that deals with a remote server might need a `url` and `port` number. Furthermore, there might be common setup required, such as connecting to the remote server. If you are familiar with Click, this would be accomplished with contexts. In Cyclopts, there are 2 ways to accomplish this:

1. With a *meta app*. While powerful, it's admittantly a bit heavy-handed and clunky.
2. Via a common dataclass that is passed to each command. While less powerful than using a meta-app, it still accomplishes many of the same goals with simpler, terser code.

In this section, we'll be investigating option (2) by constructing an example application that has 2 commands:

1. `create` - Connect to a server and send a POST command to it.
2. `info` - Connect to a server and GET information about a user.

```python
# demo.py
from cyclopts import App, Parameter
from cyclopts.types import UInt16
from dataclasses import dataclass
from functools import cached_property
from httpx import Client

@Parameter(name="*")  # Flatten the namespace; i.e. option will be "--url" instead of "--
↪common.url"
@dataclass
class Common:
    url: str = "http://cyclopts.readthedocs.io"
    "URL of remote server."

    port: UInt16  = 8080  # an "int" that is limited to range [0, 65535]
    "Port of remote server."

    verbose: bool = False
    "Increased printing verbosity."

    def __post_init__(self):
       # dataclasses call this method after calling the auto-generated __init__.
       if self.verbose:
          print(f"Server: {self.base_url}")

    @property
    def base_url(self) -> str:
```

(continues on next page)

(continued from previous page)

```
        return f"{self.url}:{self.port}"

    @cached_property
    def client(self) -> Client:
        return Client(base_url=self.base_url)

app = App()

@app.command
def create(name: str, age: int, *, common: Common | None = None):
    """Create a user on remote server.

    Parameters
    ----------
    name: str
        Name of the user to create.
    age: int
        Age of the user in years.
    """
    if common is None:
        common = Common()
    json = {"name": name, "age": age}
    if common.verbose:
        print(f"Creating user: {json}")
    common.client.post("/users", json=json)
    # TODO: in a real application, we should error-check the response here.

@app.command
def info(name: str, *, common: Common | None = None):
    """List a user on remote server.

    Parameters
    ----------
    name: str
        Name of the user to get info about.
    """
    if common is None:
        common = Common()
    response = common.client.get("/users", params={"name": name})
    user = response.json()
    print(f"User: {user}")

if __name__ == "__main__":
    app()
```

From the root help-page, we can see our two commands:

```
$ python demo.py --help
Usage: demo.py COMMAND

- Commands ------------------------------------------------------------------
| create     Create a user on remote server.                                 |
```

(continues on next page)

(continued from previous page)

```
│ info       List a user on remote server.                                    │
│ --help -h  Display this message and exit.                                   │
│ --version  Display application version.                                     │
──────────────────────────────────────────────────────────────────────────────
```

From the `create` help-page, we can see all of our parameters:

```
$ python demo.py create --help
Usage: demo.py create [ARGS] [OPTIONS]

Create a user on remote server.

─ Parameters ─────────────────────────────────────────────────────────────────
│ *  NAME --name             Name of the user to create. [required]           │
│ *  AGE --age               Age of the user in years. [required]             │
│    --url                   URL of remote server. [default:                  │
│                            http://cyclopts.readthedocs.io]                  │
│    --port                  Port of remote server. [default: 8080]           │
│    --verbose --no-verbose  Increased printing verbosity. [default: False]   │
──────────────────────────────────────────────────────────────────────────────
```

Some example command-line invocations:

```
$ python demo.py create Alice 42
# No response from the CLI.

$ python demo.py create Alice 42 --verbose
Creating user: {'name': 'Alice', 'age': 42}
```

By organizing the code this way, we can centralize shared parameters and logic between many commands.

CHAPTER

# THIRTYEIGHT

# UNIT TESTING

It is important to have unit-tests to verify that your CLI is behaving correctly. For unit-testing, we will be using the defacto-standard python unit-testing library, pytest. This section demonstrates some common scenarios you may encounter when unit-testing your CLI app.

Lets make a small application that checks PyPI if a library name is available:

```
# pypi_checker.py
import sys
import urllib.error
import urllib.request
import cyclopts

def _check_pypi_name_available(name):
    try:
        urllib.request.urlopen(f"https://pypi.org/pypi/{name}/json")
    except urllib.error.HTTPError as e:
        if e.code == 404:
            return True  # Package does not exist (name is available)
    return False  # Package exists (name is not available)

app = cyclopts.App(
      config=[
         cyclopts.config.Env("PYPI_CHECKER_"),
         cyclopts.config.Json("config.json"),
      ],
)

@app.default
def pypi_checker(name: str, *, silent: bool = False) -> bool:
    """Check if a package name is available on PyPI.

    Returns True if available; False otherwise.

    Parameters
    ----------
    name: str
        Name of the package to check.
    silent: bool
        Do not print anything to stdout.
    """
    is_available = _check_pypi_name_available(name)
```

(continues on next page)

(continued from previous page)

```
    if not silent:
        if is_available:
            print(f"{name} is available.")
        else:
            print(f"{name} is not available.")
    return is_available

if __name__ == "__main__":
    app()
```

Running the app from the console:

```
$ python pypi_checker.py --help
Usage: pypi_checker COMMAND [ARGS] [OPTIONS]

Check if a package name is available on PyPI.

Returns True if available; False otherwise.

─ Commands␣
↪──────────────────────────────────────────────────────────────────────────────
│ --help -h  Display this message and exit.                                  ␣
↪           │
│ --version  Display application version.                                    ␣
↪           │
──────────────────────────────────────────────────────────────────────────────
─ Parameters␣
↪──────────────────────────────────────────────────────────────────────────────
│ *  NAME --name           Name of the package to check. [env var: PYPI_CHECKER_NAME]␣
↪[required]    │
│    --silent --no-silent  Do not print anything to stdout. [env var: PYPI_CHECKER_
↪SILENT]          │
│                          [default: False]                                  ␣
↪           │
──────────────────────────────────────────────────────────────────────────────

$ python pypi_checker.py cyclopts
cyclopts is not available.

$ python pypi_checker.py cyclopts --silent
$ echo $?  # Check the exit code of the previous command.
1

$ python pypi_checker.py the-next-big-project
the-next-big-project is available.
$ echo $?  # Check the exit code of the previous command.
0
```

We will slowly introduce unit-testing concepts and build up a fairly comprehensive set of unit-tests for this application.

## 38.1 Mocking

First off, it's good code-hygiene to separate "business logic" from "user interface." In this example, that means putting all the actual logic of determining whether or not a package name is available into the `_check_pypi_name_available` function, and putting all of the CLI logic (like printing to `stdout` and exit-codes) in the Cyclopts-decorated function `pypi_checker`. This makes it easier to unit-test the app because it allows us to mock out portions of our app, allowing us to isolate our CLI unit-tests to just the CLI components.

We can use pytest-mock to simplify mocking `_check_pypi_name_available`. Let's define a fixture that declares this mock.

```
# test.py
import pytest
from pypi_checker import app

@pytest.fixture
def mock_check_pypi_name_available(mocker):
    return mocker.patch("pypi_checker._check_pypi_name_available")
```

Unit tests that use this fixture can define it's return value, as well as check the arguments it was called with. This will be demonstrated in the next section.

## 38.2 Exit Codes

Our command function returns a boolean. By default, Cyclopts uses *result_action* of "print_non_int_sys_exit", which calls `sys.exit()` with the appropriate code: `True` → `0` (success), `False` → 1 (failure).

```
import pytest

def test_unavailable_name_cli_behavior(mock_check_pypi_name_available):
    # Set the mock return_value to False (i.e. the name is NOT available).
    mock_check_pypi_name_available.return_value = False
    with pytest.raises(SystemExit) as exc_info:
        app("foo")  # Default result_action calls sys.exit
    mock_check_pypi_name_available.assert_called_once_with("foo")
    assert exc_info.value.code == 1  # Package unavailable exits with code 1
```

We can then run pytest on this file:

```
$ pytest test.py
============================= test session starts ==============================
platform darwin -- Python 3.13.0, pytest-8.3.4, pluggy-1.5.0
rootdir: /cyclopts-demo
configfile: pyproject.toml
plugins: cov-6.0.0, anyio-4.8.0, mock-3.14.0
collected 1 item

test.py .                                                                [100%]

============================== 1 passed in 0.05s ===============================
```

## 38.3 Checking stdout

We also want to make sure that our message is displayed to the user. The built-in capsys fixture gives us access to our application's `stdout`. We can use this to confirm our app prints the correct statement.

By passing `result_action="return_value"` to the app call, we can get the return value directly without `sys.exit()` being called:

```
# test.py - continued from "Exit Codes"
def test_unavailable_name_with_output(capsys, mock_check_pypi_name_available):
    mock_check_pypi_name_available.return_value = False
    is_available = app("foo", result_action="return_value")
    mock_check_pypi_name_available.assert_called_once_with("foo")
    assert is_available is False
    assert capsys.readouterr().out == "foo is not available.\n"
```

**Note**

Normal output goes to *console* (stdout), while errors go to *error_console* (stderr). Use `capsys.readouterr().err` to check error messages, or provide a custom `error_console` to capture both streams together.

## 38.4 Environment Variables

Because we configured our *App* with *cyclopts.config.Env*, we can pass arguments into our application via environment variables. The pytest monkeypatch fixture allows us to modify environment variables within the context of a unit-test.

In this test, we only want to test if our environment variable is being passed in correctly. We will use *App.parse_args()*, which performs all the parsing, but doesn't actually invoke the command.

```
# test.py
def test_name_env_var(monkeypatch):
    from pypi_checker import pypi_checker
    monkeypatch.setenv("PYPI_CHECKER_NAME", "foo")
    command, bound, _ = app.parse_args([])  # An empty list - no CLI arguments passed in.
    assert command == pypi_checker
    assert bound.arguments['name'] == "foo"
```

**Warning**

A common mistake is accidentally calling `app()` or `app.parse_args()` with the **intent of providing no arguments**. Calling these methods with no arguments will read from `sys.argv`, the same as in a typical application. This is rarely the intention in a unit-test, and Cyclopts **will produce a warning.** For example, this code in a unit test:

```
app()  # Wrong: will produce a warning
```

Will generate this warning:

```
============================== warnings summary ===============================
test.py::test_no_args
  /my_project/test.py:64: UserWarning: Cyclopts application invoked without tokens
  under unit-test framework "pytest". Did you mean "app([])"?
    app()
```

The proper way to specify no CLI arguments is to provide an empty string or list:

```
app([])
```

## 38.5 File Config

To explicitly test that configurations from the *Cyclopts configuration system* are loading properly, we can create a configuration file in a temporary directory and change our current-working-directory (cwd) to that temporary directory. The pytest built-in `tmp_path` fixture gives us a temporary directory, and the `monkeypatch` fixture allows us to change the cwd. We have to change the cwd because typically configuration files are discovered relative to the directory where the CLI was invoked. If your CLI searches other locations (such as the home directory), you will need to modify this example appropriately.

```
# test.py
import json
from pypi_checker import pypi_checker

@pytest.fixture(autouse=True)
def chdir_to_tmp_path(tmp_path, monkeypatch):
    "Automatically change current directory to tmp_path"
    monkeypatch.chdir(tmp_path)

@pytest.fixture
def config_path(tmp_path):
    "Path to JSON configuration file in tmp_path"
    return tmp_path / "config.json"  # same name that was provided to cyclopts.config.
↪Json

def test_config(config_path):
    with config_path.open("w") as f:
        json.dump({"name": "bar"}, f)
    command, bound, _ = app.parse_args([])  # An empty list - no CLI arguments passed in.
    assert command == pypi_checker
    assert bound.arguments['name'] == "bar"
```

## 38.6 Help Page

Cyclopts uses Rich to pretty-print messages to the console. Rich interprets the console environment, and can change how it displays text depending on the terminal's capabilities. For unit testing, we will explicitly set a lot of these parameters in a pytest fixture to make it easier to compare against known good values:

```
@pytest.fixture
def console():
    from rich.console import Console
    return Console(width=70, force_terminal=True, highlight=False, color_system=None,␣
↪legacy_windows=False)
```

Since the help-page is just printed to `stdout`, we will be using the capsys fixture again.

```
import pytest
from textwrap import dedent
```

(continues on next page)

(continued from previous page)

```
def test_help_page(capsys, console):
    with pytest.raises(SystemExit):
        app("--help", console=console)
    actual = capsys.readouterr().out
    assert actual == dedent(
        """\
        Usage: pypi_checker COMMAND [ARGS] [OPTIONS]

        Check if a package name is available on PyPI.

        Returns True if available; False otherwise.

        ╭─ Commands ─────────────────────────────────────────────────────────────────╮
        │ --help -h  Display this message and exit.                                  │
        │ --version  Display application version.                                    │
        ╰────────────────────────────────────────────────────────────────────────────╯
        ╭─ Parameters ───────────────────────────────────────────────────────────────╮
        │ *  NAME --name           Name of the package to check. [required]          │
        │    --silent --no-silent  Do not print anything to stdout.                  │
        │                          [default: False]                                  │
        ╰────────────────────────────────────────────────────────────────────────────╯
        """
    )
```

The `textwrap.dedent()` function allows us to have our expected-help-string nicely indented within our code. Alternatively, we could have used the `rich.console.Console.capture()` context manager to directly capture the `rich.console.Console` output.

**Note**

Unit-testing the help-page is probably overkill for most projects (and may get in the way more often than it helps!).

CHAPTER

# THIRTYNINE

# READING/WRITING FROM FILE OR STDIN/STDOUT

In many CLI applications, it's common to be able to read from a file or stdin, and write to a file or stdout. This allows for the chaining of many CLI applications via pipes `|`.

## 39.1 StdioPath

> **Note**
>
> *`StdioPath`* requires **Python 3.12+**. For older Python versions, see *Alternative Approach (Python < 3.12)* below.

The recommended approach is to use *`StdioPath`*, a `Path` subclass that treats `-` as stdin (for reading) or stdout (for writing). This follows common Unix convention used by many command-line tools.

```
from cyclopts import App
from cyclopts.types import StdioPath

app = App()

@app.default
def scream(input_: StdioPath, output: StdioPath):
    """Uppercase all input data.

    Parameters
    ----------
    input_:
        Input file path, or "-" for stdin.
    output:
        Output file path, or "-" for stdout.
    """
    data = input_.read_text()
    output.write_text(data.upper())

if __name__ == "__main__":
    app()
```

```
$ echo "hello cyclopts users." > demo.txt

$ python scream.py demo.txt -
HELLO CYCLOPTS USERS.
```

(continues on next page)

(continued from previous page)

```
$ python scream.py demo.txt output.txt && cat output.txt
HELLO CYCLOPTS USERS.

$ echo "foo" | python scream.py - -
FOO
```

*StdioPath* is pre-configured with `allow_leading_hyphen=True`, so `-` can be passed as an argument without being interpreted as an option.

### 39.1.1 Defaulting to Stdin/Stdout

To make stdin/stdout the default when no argument is provided, use `StdioPath("-")` as the default value:

```python
from cyclopts import App
from cyclopts.types import StdioPath

app = App()

@app.default
def scream(input_: StdioPath = StdioPath("-"), output: StdioPath = StdioPath("-")):
    """Uppercase all input data.

    Parameters
    ----------
    input_:
        Input file path. Defaults to stdin if not provided.
    output:
        Output file path. Defaults to stdout if not provided.
    """
    data = input_.read_text()
    output.write_text(data.upper())

if __name__ == "__main__":
    app()
```

```
$ echo "hello cyclopts users." > demo.txt

$ python scream.py demo.txt
HELLO CYCLOPTS USERS.

$ python scream.py demo.txt output.txt && cat output.txt
HELLO CYCLOPTS USERS.

$ echo "foo" | python scream.py
FOO
```

### 39.1.2 Binary Data

`StdioPath` also supports binary reading and writing:

```
@app.default
def process_binary(input_: StdioPath = StdioPath("-"), output: StdioPath = StdioPath("-
↪")):
    data = input_.read_bytes()
    output.write_bytes(data)
```

Or using the context manager interface:

```
@app.default
def process_binary(input_: StdioPath = StdioPath("-"), output: StdioPath = StdioPath("-
↪")):
    with input_.open("rb") as f_in, output.open("wb") as f_out:
        f_out.write(f_in.read())
```

## 39.2 Alternative Approach (Python < 3.12)

For Python versions before 3.12, or when you prefer an `Optional[Path]` pattern where `None` indicates stdin/stdout, you can use helper functions:

```
import sys
from cyclopts import App
from pathlib import Path
from typing import Optional

def read_str(input_: Optional[Path]) -> str:
    return sys.stdin.read() if input_ is None else input_.read_text()

def write_str(output: Optional[Path], data: str):
    sys.stdout.write(data) if output is None else output.write_text(data)

def read_bytes(input_: Optional[Path]) -> bytes:
    return sys.stdin.buffer.read() if input_ is None else input_.read_bytes()

def write_bytes(output: Optional[Path], data: bytes):
    sys.stdout.buffer.write(data) if output is None else output.write_bytes(data)

app = App()

@app.default
def scream(input_: Optional[Path] = None, output_: Optional[Path] = None):
    """Uppercase all input data.

    Parameters
    ----------
    input_ : Optional[Path]
        If provided, read data from file. If not provided, read from stdin.
    output_ : Optional[Path]
        If provided, write data to file. If not provided, write to stdout.
    """
    data = read_str(input_)
    processed = data.upper()
    write_str(output_, processed)
```

(continues on next page)

(continued from previous page)

```
if __name__ == "__main__":
    app()
```

```
$ echo "hello cyclopts users." > demo.txt
$ python scream.py demo.txt
HELLO CYCLOPTS USERS.
$ python scream.py demo.txt output.txt
$ cat output.txt
HELLO CYCLOPTS USERS.
$ echo "foo" | python scream.py
FOO
```

CHAPTER

FORTY

# MIGRATING FROM TYPER

Much of Cyclopts's syntax is Typer-inspired. Migrating from Typer should be pretty straightforward; it is recommended to first read the *Getting Started* and *Commands* sections. The below table offers a jumping off point for translating the various portions of the APIs. The *Typer Comparison* page also provides many examples comparing the APIs.

Table 1: Typer-to-Cyclopts API Reference

| Typer | Cyclopts | Notes |
|---|---|---|
| `typer.Typer()` | *`cyclopts.App()`* | **Same/similar fields:**<br>• *`App.name`* - Optional name of application or sub-command.<br>**Cyclopts has more user-friendly default features:**<br>• Equivalent `no_args_is_help=True`.<br>• Equivalent `pretty_exceptions_enable=False`. |
| `@app.command()` | *`@app.command()`* | In Cyclopts, `@app.command` *always results in a command.* To define an action when no command is provided, see *`@app.default`*. |
| `app.add_typer(...)()` | `app.command(...)()` | Sub applications and commands are registered the same way in Cyclopts. |
| `@app.callback()` | *`@app.default()`*<br>*`@app.meta.default()`* | Typer's callback always executes before executing an app. If used to provide functionality when no command was specified from the CLI, then use *`@app.default()`*. Otherwise, checkout Cyclopt's *Meta App*. |
| `Annotated[..., typer.Argument(...)]`<br>`Annotated[..., typer.Option(...)]` | *`Annotated[..., cyclopts.Parameter(...)]`* | In Cyclopts, Positional/Keyword arguments *are determined from the function signature.* Some of Typer's validation fields, like `exists` for `Path` types are handled in Cyclopts *by explicit validators.* |

Cyclopts and Typer mostly handle type-hints the same way, but there are a few notable exceptions:

Table 2: Typer-to-Cyclopts Type-Hints

| Type Annotation | Notes |
|---|---|
| `Enum` | Compared to Typer, Cyclopts handles `Enum` lookups *in the reverse direction.* Frequently, `Literal` *offers a more terse, intuitive choice option.* |
| `Union` | Typer does **not** support type unions. *Cyclopts does.* |

## 40.1 General Steps

1. Add the following import: `from cyclopts import App, Parameter`.
2. Change `app = Typer(...)` to just `app = App()`. Revisit more advanced configuration later.
3. Remove all `@app.callback` stuff. Cyclopts already provides a good `--version` handler for you.
4. Replace all `Annotated[..., Argument/Option]` type-hints with *`Annotated[..., Parameter()]`*. If only supplying a *`help`* string, *it's better to supply it via docstring.*
5. Cyclopts has similar boolean-flag handling as Typer, *but has different configuration parameters.*

```
#########
# Typer #
#########
# Overriding the name results in no "False" flag generation.
my_flag: Annotated[bool, Option("--my-custom-flag")]
# However, it can be custom specified:
my_flag: Annotated[bool, Option("--my-custom-flag/--disable-my-custom-flag")]

#############
# Cyclopts #
#############
# Overriding the name still results in "False" flag generation:
#    --my-custom-flag --no-my-custom-flag
my_flag: Annotated[bool, Parameter("--my-custom-flag")]
# Negative flag generation can be disabled:
#    --my-custom-flag
my_flag: Annotated[bool, Parameter("--my-custom-flag", negative="")]
# Or the prefix can be changed:
#    --my-custom-flag --disable-my-custom-flag
my_flag: Annotated[bool, Parameter("--my-custom-flag", negative_bool="--disable-")]
```

After the basic migration is done, it is recommended to read through the rest of Cyclopts's documentation to learn about some of the better functionality it has, which could result in cleaner, terser code.

CHAPTER

# FORTYONE

# TYPER COMPARISON

Much of Cyclopts was inspired by the excellent Typer library. Despite its popularity, Typer has some traits that I (and others) find less than ideal. Part of this stems from Typer's age, with its first release in late 2019, soon after Python 3.8's release. Because of this, most of its API was initially designed around assigning proxy default values to function parameters. This made the decorated command functions difficult to use outside of Typer. With the introduction of `Annotated` in python3.9, type-hints were able to be directly annotated, allowing for the removal of these proxy defaults.

Additionally, Typer is built on top of Click. This makes it difficult for newcomers to figure out which elements are Typer-related and which elements are click-related. It's also hard to tell whether the following criticisms stem from Typer, or the underlying Click. For better-or-worse, Cyclopts uses its own internal parsing strategy, gaining complete control over the process.

This section was originally written about Typer `v0.9.0` (May 2023). Some criticisms have been addressed in later Typer versions; updates are noted in the respective sections below.

## 41.1 Argument vs Option

In Typer, there are two primary classes for providing CLI parameter configuration:

- `Argument` results in a positional CLI argument.
- `Option` results in a keyword CLI argument, preceded with a `--`.

With more modern python type annotations, this distinction is unnecessary, because parameters (positional or keyword) can be determined directly from the function signature.

Consider the following function signatures:

```python
def pos_or_keyword(a, b):
    pass

def pos_only(a, b, /):
    pass

def keyword_only(*, a, b=2):
    pass

def mixture(a, /, b, *, c=3):
    pass
```

If you aren't familiar with these declarations, refer to the official PEP570, or a more user-friendly tutorial.

From these function signatures, we can deduce:

1. Which parameters are position-only, keyword-only, or both.

2. Which parameters are required, by their lack of defaults.

Because of these builtin python mechanisms, Cyclopts has a single *Parameter* class used for providing additional parameter metadata.

I believe that Typer's separate `Argument` and `Option` classes are a relic from when they must be supplied as a parameter's proxy default value.

```
app = typer.Typer()

@app.command()
def foo(a=Argument(), b=Option(default=2)):
    pass
```

When used as such, we lose the ability to define the function signature with position-only or keyword-only markers. We also lose the ability to directly inspect which parameters are optional by having "real" defaults and which ones are required.

## 41.2 Positional or Keyword Arguments

A limitation of Typer is that a parameter cannot be both positional and keyword.

For example, lets say we want to implement a `mv`-like program that takes in a source path, and a destination path:

```
typer_app = typer.Typer()

@typer_app.command()
def mv(src, dst):
    print(f"Moving {src} -> {dst}")

typer_app(["foo", "bar"], standalone_mode=False)
# Moving foo -> bar
```

The code works when supplying the inputs as positional arguments, but fails when trying to specify them as keywords.

```
print("Typer keyword:")
typer_app(["--src", "foo", "--dst", "bar"], standalone_mode=False)
# No such option: --src
```

Cyclopts handles both situations:

```
cyclopts_app = cyclopts.App()

@cyclopts_app.default()
def mv(src, dst):
    print(f"Moving {src} -> {dst}")

cyclopts_app(["foo", "bar"])
# Moving foo -> bar
cyclopts_app(["--src", "foo", "--dst", "bar"])
# Moving foo -> bar
```

## 41.3 Choices

### 41.3.1 Enum

Frequently, a CLI will want to limit values provided to a parameter to a specific set of choices. With Typer, this is accomplished via declaring an `Enum`.

```
import typer
from enum import Enum

class Environment(str, Enum):
    # Values end in "_value" to avoid confusion in this example.
    DEV = "dev_value"
    STAGING = "staging_value"
    PROD = "prod_value"

typer_app = typer.Typer()

@typer_app.command
def foo(env: Environment = Environment.DEV):
    print(f"Using: {env.name}")

print("Typer (Enum):")
typer_app(["--env", "staging_value"])
# Using: STAGING
```

Typer looks for the CLI-provided *value*, and supplies the function with the enum member. IMHO, this is backwards; typically the enum name (e.g. `DEV`) is intended to be more human-friendly, while the value (e.g. `dev_value`) more frequently has a programmatic-meaning. **When using enums, Cyclopts will do the opposite of Typer**, performing a **case-insensitive** lookup by **name**.

```
import cyclopts

cyclopts_app = cyclopts.App()

@cyclopts_app.default
def foo(env: Environment = Environment.DEV):
    print(f"Using: {env.name}")

print("Cyclopts (Enum):")
cyclopts_app(["--env", "staging"])
# Using: STAGING
```

### 41.3.2 Literal

Enums don't work well with everyone's workflow. Many people prefer to directly use strings for their functions' options. The much more intuitive, convenient method of doing this is with the `Literal` type annotation.

> **Note**
>
> Typer added support for `Literal` in version 0.19.0 (September 2025), resolving a feature request from early 2020.

Cyclopts has builtin support for `Literal`, see *Coercion Rules - Literal*.

```python
import cyclopts
from typing import Literal

cyclopts_app = cyclopts.App()

@cyclopts_app.default
def foo(env: Literal["dev", "staging", "prod"] = "staging"):
    print(f"Using: {env}")


print("Cyclopts (Literal):")
cmd = ["--env", "staging"]
print(cmd)
cyclopts_app(cmd)
# Using: staging
```

## 41.4 Default Command

Typer has an annoying design quirk where if you register a single command, it **won't** expect you to provide the command name in the CLI. For example:

```python
import typer

typer_app = typer.Typer()

@typer_app.command()
def foo():
    print("FOO")

typer_app([], standalone_mode=False)
# FOO
typer_app(["foo"], standalone_mode=False)
# raises exception: Got unexpected extra argument (foo)
```

Once you add a second command, then the CLI expects the command to be provided:

```python
typer_app(["foo"], standalone_mode=False)
# FOO
typer_app(["bar"], standalone_mode=False)
# BAR
```

This behavior catches many people off guard. If you want a single command, you have to unintuitively declare a `callback`. Github user ajlive's callback solution is copied below.

```python
@app.callback()
def dummy_to_force_subcommand() -> None:
    """
    This function exists because Typer won't let you force a single subcommand.
    Since we know we will add other subcommands in the future and don't want to
    break the interface, we have to use this workaround.

    Delete this when a second subcommand is added.
```

(continues on next page)

(continued from previous page)

```
    """
    pass
```

To avoid this confusion, Cyclopts has two ways of registering a function:

1. *@app.command* - Register a function as a command.
2. *@app.default* - Invoked if no registered command can be parsed from the CLI.

```
import cyclopts

cyclopts_app = cyclopts.App()

@cyclopts_app.command
def foo():
    print("FOO")

cyclopts_app(["foo"])
# FOO
```

## 41.5 Docstring Parsing

Typer performs no docstring parsing. Frequently, Typer's Argument/Option is only used to provide a `help` string. However, this `help` string commonly mirrors the function's docstring.

Consider the following Typer program:

```
import typer

typer_app = typer.Typer()

@typer_app.callback()
def dummy():
    # So that ``foo`` is considered a command.
    pass

@typer_app.command()
def foo(bar):
    """Foo Docstring.

    Parameters
    ----------
    bar: str
        Bar parameter docstring.
    """

typer_app()
```

```
$ my-script --help
─ Commands ──────────────────────────────────────────────────────────────────
│ foo                Foo Docstring.                                         │
─────────────────────────────────────────────────────────────────────────────
```

(continues on next page)

(continued from previous page)

```
$ my-script foo --help
Foo Docstring.
Parameters ---------- bar: str     Bar parameter docstring.

╭─ Arguments ──────────────────────────────────────────────────────────────────╮
│ *    bar      TEXT  [default: None] [required]                               │
╰──────────────────────────────────────────────────────────────────────────────╯
```

The `foo` command's short description was properly parsed from the docstring. However, it mangles the Numpy-style docstring (or any docstring format for that matter) and doesn't correctly display `bar`'s help. Typer just displays the entire docstring.

To achieve the desired result with Typer, we have to explicitly annotate the parameter `bar`:

```
@typer_app.command()
def foo(bar: Annotated[str, Argument(help="Bar parameter docstring.")]):
    ...
```

For any serious application, this means that every function parameter must be annotated this way, significantly bloating the function signature.

Compare this to Cyclopts:

```
import cyclopts

cyclopts_app = cyclopts.App()

@cyclopts_app.command()
def foo(bar):
    """Foo Docstring.

    Parameters
    ----------
    bar: str
        Bar parameter docstring.
    """

cyclopts_app()
```

```
$ my-script --help
╭─ Commands ───────────────────────────────────────────────────────────────────╮
│ foo  Foo Docstring.                                                          │
╰──────────────────────────────────────────────────────────────────────────────╯

$ my-script foo --help

Foo Docstring.

╭─ Parameters ─────────────────────────────────────────────────────────────────╮
│ *  BAR,--bar  Bar parameter docstring. [required]                            │
╰──────────────────────────────────────────────────────────────────────────────╯
```

Cyclopts did not mangle the docstring into the long description, and it correctly parsed `bar`'s help. This ends up

significantly simplifying function signatures in the common situation where just a help string needs to be added. The common case in Cyclopts does not require the lengthy `Annotated[str, Parameter(help="Bar parameter docstring")]`.

Internally, Cyclopts uses the excellent docstring_parser library for parsing docstrings. Check their project out!

## 41.6 Decorator Parentheses

A minor nitpick, but all of Typer's decorators require parentheses.

```
import typer

typer_app = typer.Typer()

# This doesn't work! Missing ()
@typer_app.command
def foo():
    pass
```

Cyclopts works with and without parentheses.

```
import cyclopts

cyclopts_app = cyclopts.App()

# This works! Missing ()
@cyclopts_app.command
def foo():
    pass

# This also works.
@cyclopts_app.command()
def bar():
    pass
```

## 41.7 Optional Lists

**Note**

This issue has been addressed in Typer v0.10.0.

Typer does not handle optional lists particularly well. In Typer, if a list argument is not provided via the CLI, an empty list is passed to the command by default. While this might be acceptable in some scenarios, it can be unexpected and differs semantically from the default value. Because lists are mutable, and setting mutable defaults is strongly discouraged, setting list parameters' default to `None` is common practice. This approach can also help differentiate between the intention of using a default list and explicitly requesting an empty list.

Consider the following Typer example:

```
import typer

typer_app = typer.Typer()
```

(continues on next page)

(continued from previous page)

```
@typer_app.command()
def foo(favorite_numbers: Optional[list[int]] = None):
    if favorite_numbers is None:
        favorite_numbers = [1, 2, 3]
    print(f"My favorite numbers are: {favorite_numbers}")

typer_app(["--favorite-numbers", "100", "--favorite-numbers", "200"], standalone_
↪mode=False)
# My favorite numbers are: [100, 200]
typer_app([], standalone_mode=False)
# My favorite numbers are: []
```

In this example, we expect the default list `[1, 2, 3]` to be used when no input is provided. However, Typer supplies an empty list instead of `None`.

Cyclopts has a more intuitive solution. If no CLI option is specified, no argument is bound, so the parameter's default value `None` is used. If we wish to pass an empty iterable (e.g. `set` or `list`), Cyclopts provides an `--empty-*` flag for each iterable parameter. This feature is configurable via *`Parameter.negative_iterable`*.

```
import cyclopts

cyclopts_app = cyclopts.App()

@cyclopts_app.default()
def foo(favorite_numbers: Optional[list[int]] = None):
    if favorite_numbers is None:
        favorite_numbers = [1, 2, 3]
    print(f"My favorite numbers are: {favorite_numbers}")

cyclopts_app(["--favorite-numbers", "100", "--favorite-numbers", "200"])
# My favorite numbers are: [100, 200]
cyclopts_app([])
# My favorite numbers are: [1, 2, 3]
cyclopts_app(["--empty-favorite-numbers"])
# My favorite numbers are: []
```

## 41.8 Keyword Multiple Values

In some applications, it is desirable to supply multiple values to a keyword argument. For example, lets consider an application where we want to specify multiple input files. We want our application to look like the following:

```
$ my-program output.bin --input input1.bin input2.bin input3.bin
```

Interpreted as:

```
output=PosixPath('output.bin')
input=[PosixPath('input1.bin'), PosixPath('input2.bin'), PosixPath('input3.bin')]
```

In Typer, it is impossible to accomplish this. With Typer, the keyword must be specified before each value:

```
$ my-program output.bin --input input1.bin --input input2.bin --input input3.bin
```

By default, Cyclopts behavior mimics Typer, where a single element worth of CLI tokens are consumed. However, by setting *Parameter.consume_multiple* to True, multiple elements worth of CLI tokens will be consumed. Consider the following example program with a single output path, and multiple input paths.

```
from cyclopts import App, Parameter
from pathlib import Path
from typing import Annotated

app = App()

@app.default
def main(output: Path, input: Annotated[list[Path], Parameter(consume_multiple=True)]):
    print(f"{input=} {output=}")

if __name__ == "__main__":
    app()
```

All of the following invocations are equivalent:

```
$ my-program output.bin input1.bin input2.bin input3.bin                    #␣
↪Supplying arguments positionally.
$ my-program output.bin --input input1.bin --input input2.bin --input input3.bin #␣
↪Supplying input arguments via multiple keywords.
$ my-program output.bin --input input1.bin input2.bin input3.bin            #␣
↪Supplying input arguments via a single keyword.
$ my-program --input input1.bin input2.bin input3.bin --output output.bin   #␣
↪Supplying all arguments via keywords.
$ my-program --input input1.bin input2.bin input3.bin -- output.bin         # Using␣
↪the POSIX convention to indicate the end of keywords
```

To set this configuration for your entire application, supply it to your root *App.default_parameter*:

```
from cyclopts import App, Parameter

app = App(default_parameter=Parameter(consume_multiple=True))
```

## 41.9 Flag Negation

For boolean parameters, Typer adds a `--no-MY-FLAG-NAME` to specify a `False` argument.

```
import typer

typer_app = typer.Typer()

@typer_app.command()
def foo(my_flag: bool = False):
    print(f"{my_flag=}")

typer_app(["--my-flag"], standalone_mode=False)
# my_flag=True
typer_app(["--no-my-flag"], standalone_mode=False)
# my_flag=False
```

Overriding the option's name will disable Typer's negative-flag generation logic:

```python
import typer
from typing import Annotated

typer_app = typer.Typer()

@typer_app.command()
def foo(my_flag: Annotated[bool, Option("--my-flag")] = False):
    print(f"{my_flag=}")

typer_app(["--my-flag"], standalone_mode=False)
# my_flag=True
typer_app(["--no-my-flag"], standalone_mode=False)
# NoSuchOption: No such option: --no-my-flag
```

This is not the worst, but there is a tiny bit of duplication. To use a different negative flag, you can supply the name after a slash in your option-name-string.

```python
import typer

typer_app = typer.Typer()

@typer_app.command()
def foo(my_flag: Annotated[bool, Option("--my-flag/--your-flag")] = False):
    print(f"{my_flag=}")

typer_app(["--my-flag"], standalone_mode=False)
# my_flag=True
typer_app(["--your-flag"], standalone_mode=False)
# my_flag=False
```

Cyclopts's *Parameter* takes in an optional *negative* flag. To suppress the negative-flag generation, set this argument to either an empty string or list.

```python
import cyclopts
from typing import Annotated

cyclopts_app = cyclopts.App()

@cyclopts_app.default
def foo(my_flag: Annotated[bool, cyclopts.Parameter(negative="")] = False):
    print(f"{my_flag=}")

print("Cyclopts:")
cyclopts_app(["--my-flag"])
# my_flag=True
cyclopts_app(["--your-flag"], exit_on_error=False)
# ╭─ Error ──────────────────────────────────────────────────────────────────────╮
# │ Error converting value "--your-flag" to <class 'bool'> for "--my-flag".      │
# ╰──────────────────────────────────────────────────────────────────────────────╯
# CoercionError: Error converting value "--your-flag" to <class 'bool'> for "--my-flag".
```

To define your own custom negative flag, just provide it as a string or list of strings.

```
@cyclopts_app.default
def foo(my_flag: Annotated[bool, cyclopts.Parameter(negative="--your-flag")] = False):
    print(f"{my_flag=}")

print("Cyclopts:")
cyclopts_app(["--my-flag"])
# my_flag=True
cyclopts_app(["--your-flag"])
# my_flag=False
```

The default `--no-` negation prefix can also be customized with *negative_bool*.

```
@cyclopts_app.default
def foo(my_flag: Annotated[bool, cyclopts.Parameter(negative_bool="--disable-")] =␣
↪False):
    print(f"{my_flag=}")

print("Cyclopts:")
cyclopts_app(["--my-flag"])
# my_flag=True
cyclopts_app(["--disable-my-flag"])
# my_flag=False
```

## 41.10 Help Defaults

In Typer's `--help` display, default values are unhelpfully shown for required arguments.

**Note**

This was fixed in Typer 0.17.2 (August 2025) when using Rich for help display.

```
import typer

typer_app = typer.Typer()

@typer_app.command()
def compress(
    src: Annotated[Path, typer.Argument(help="File to compress.")],
    dst: Annotated[Path, typer.Argument(help="Path to save compressed data to.")] = Path(
↪"out.zip"),
):
    print(f"Compressing data from {src} to {dst}")

print("Typer positional:")
typer_app(["--help"], standalone_mode=False)
# ─ Arguments ──────────────────────────────────────────────────────────────────
# │ *    src      PATH   File to compress. [default: None] [required]          │
# │      dst      [DST]  Path to save compressed data to. [default: out.zip]   │
# ──────────────────────────────────────────────────────────────────────────────
```

It doesn't make any sense to show a default for a parameter that is required and has no default. Cyclopts fixes this:

```python
import cyclopts

cyclopts_app = cyclopts.App()

@cyclopts_app.default()
def compress(
    src: Annotated[Path, cyclopts.Parameter(help="File to compress.")],
    dst: Annotated[Path, cyclopts.Parameter(help="Path to save compressed data to.")] =␣
↪Path("out.zip"),
):
    print(f"Compressing data from {src} to {dst}")

cyclopts_app(["--help"])
# ─ Parameters ──────────────────────────────────────────────────────────────
# │ *  SRC,--src  File to compress. [required]                              │
# │    DST,--dst  Path to save compressed data to. [default: out.zip]       │
# ───────────────────────────────────────────────────────────────────────────
```

Additionally, if the default value is `None`, cyclopts's default configuration will **not** display `[default: None]`. Doing so doesn't convey much meaning to the end-user. Typically `None` is a sentinel value who's true value gets set inside the function.

Additionally, the cleaner, docstring-centric way of writing this program with Cyclopts would be:

```python
import cyclopts
from pathlib import Path

cyclopts_app = cyclopts.App()

@cyclopts_app.default()
def compress(src: Path, dst: Path = Path("out.zip")):
    """Compress a file.

    Parameters
    ----------
    src: Path
       File to compress.
    dst: Path
       Path to save compressed data to.
    """
    print(f"Compressing data from {src} to {dst}")

cyclopts_app(["--help"])
# ─ Parameters ──────────────────────────────────────────────────────────────
# │ *  SRC,--src  File to compress. [required]                              │
# │    DST,--dst  Path to save compressed data to. [default: out.zip]       │
# ───────────────────────────────────────────────────────────────────────────
```

## 41.11 Validation

Typer has builtin argument validation for certain type annotations.

```python
import typer

typer_app = typer.Typer()

@typer_app.command()
def foo(age: Annotated[int, typer.Argument(min=0)]):
    pass
```

This works for a select few builtins, but the Typer solution doesn't abstract out validation properly. Why does the generic `typer.Argument` have fields that only have meaning if the annotated type is a number? The `typer.Argument` signature has a ridiculous number of fields that only apply for certain types.

```python
def Argument(
    # Parameter
    default: Optional[Any] = ...,
    *,
    callback: Optional[Callable[..., Any]] = None,
    metavar: Optional[str] = None,
    expose_value: bool = True,
    is_eager: bool = False,
    envvar: Optional[Union[str, List[str]]] = None,
    shell_complete: Optional[
        Callable[
            [click.Context, click.Parameter, str],
            Union[List["click.shell_completion.CompletionItem"], List[str]],
        ]
    ] = None,
    autocompletion: Optional[Callable[..., Any]] = None,
    # Custom type
    parser: Optional[Callable[[str], Any]] = None,
    # TyperArgument
    show_default: Union[bool, str] = True,
    show_choices: bool = True,
    show_envvar: bool = True,
    help: Optional[str] = None,
    hidden: bool = False,
    # Choice
    case_sensitive: bool = True,
    # Numbers
    min: Optional[Union[int, float]] = None,
    max: Optional[Union[int, float]] = None,
    clamp: bool = False,
    # DateTime
    formats: Optional[List[str]] = None,
    # File
    mode: Optional[str] = None,
    encoding: Optional[str] = None,
    errors: Optional[str] = "strict",
    lazy: Optional[bool] = None,
    atomic: bool = False,
    # Path
    exists: bool = False,
    file_okay: bool = True,
```

(continues on next page)

(continued from previous page)

```
    dir_okay: bool = True,
    writable: bool = False,
    readable: bool = True,
    resolve_path: bool = False,
    allow_dash: bool = False,
    path_type: Union[None, Type[str], Type[bytes]] = None,
    # Rich settings
    rich_help_panel: Union[str, None] = None,
) -> Any:
    ...
```

Cyclopts has an explicit *validator* field that accepts a function:

```
from cyclopts import App, parameter
from typing import Annotated

cyclopts_app = App()

def age_validator(type_, value: int):
    if value < 0:
        raise ValueError

@cyclopts_app.command()
def foo(age: Annotated[int, Parameter(validator=age_validator)]):
    pass

 cyclopts_app()
```

This solution is similar to how other libraries, like Attrs or Pydantic, perform validation.

Cyclopts has builtin validators for common use-cases.

```
# Typer
typer.Argument(file_okay=True, exists=True)

# Cyclopts
cyclopts.Parameter(validator=cyclopts.validators.Path(file_okay=True, exists=True))
```

## 41.12 Union/Optional Support

Currently, Typer does not support Union type annotations.

```
import typer

typer_app = typer.Typer()

@typer_app.command()
def foo(value: Union[int, str] = "default_str"):
    print(f"{type(value)=} {value=}")

typer_app(["123"])
# AssertionError: Typer Currently doesn't support Union types
```

Cyclopts fully supports `Union` annotations. Cyclopt's *Coercion Rules* iterate left-to-right over the unioned types until a coercion can be performed without error.

```
import cyclopts

cyclopts_app = cyclopts.App()

@cyclopts_app.default
def foo(value: Union[int, str] = "default_str"):
    print(f"{type(value)=} {value=}")

print("Cyclopts:")
cyclopts_app(["123"])
# type(value)=<class 'int'> value=123
cyclopts_app(["bar"])
# type(value)=<class 'str'> value='bar'
```

Naturally, Cyclopts also supports `Optional` types, since `Optional` is syntactic sugar for `Union[..., None]`.

## 41.13 Adding a Version Flag

It's common to check a CLI app's version via a `--version` flag.

Concretely, we want the following behavior:

```
$ mypackage --version
1.2.3
```

To achieve this in Typer, we need the following bulky implementation:

```
import typer
from typing import Annotated

typer_app = typer.Typer()

def version_callback(value: bool):
    if value:
        print("1.2.3")
        raise typer.Exit()

@typer_app.callback()
def common(
    version: Annotated[
        bool,
        typer.Option(
            "--version",
            callback=version_callback,
            help="Print version.",
        ),
    ] = False,
):
    print("Callback body executed.")

print("Typer:")
```

(continues on next page)

(continued from previous page)

```
typer_app(["--version"])
# 1.2.3
```

Not only is this a lot of boilerplate, but it also has some nasty side-effects, such as impacting whether or not you need to specify the command in a single-command program. On top of that, it's not very intuitive. Would you expect `"Callback body executed."` to be printed? When does `version_callback` get called? What is `value`?

With Cyclopts, the version is automatically detected by checking the version of the package instantiating *App*. If you prefer explicitness, *version* can also be explicitly supplied to *App*.

```
import cyclopts

cyclopts_app = cyclopts.App(version="1.2.3")
cyclopts_app(["--version"])
# 1.2.3
```

## 41.14 Documentation

Documentation is a major component of any library.

Typer's documentation contains many good tutorials and demonstrations on how to use the library, **but has very little information on the API itself**.

Frequently the only way to discover options and behavior is to dive into the source code. This becomes further confusing as the lines of where Typer ends and Click begins is quite blurred.

Cyclopts has a full *API* page, containing all the configurable options and defined behaviors in a single place.

CHAPTER

# FORTYTWO

## FIRE COMPARISON

Fire is a CLI parsing library by Google that attempts to generate a CLI from any Python object. To that end, I think Fire definitely achieves its goal. However, I think Fire has too much magic, and not enough control.

From the Fire documentation:

> The types of the arguments are determined by their values, rather than by the function signature where they're used. You can pass any Python literal from the command line: numbers, strings, tuples, lists, dictionaries, (sets are only supported in some versions of Python). You can also nest the collections arbitrarily as long as they only contain literals.

Essentially, Fire ignores type hints and parses CLI parameters as if they were python expressions.

```
import fire


def hello(name: str = "World"):
    print(f"{name=} {type(name)=}")


if __name__ == "__main__":
    fire.Fire(hello)
```

```
$ my-script foo
name='foo' type(name)=<class 'str'>

$ my-script 100
name=100 type(name)=<class 'int'>

$ my-script true
name='true' type(name)=<class 'str'>

$ my-script True
name=True type(name)=<class 'bool'>
```

The equivalent in Cyclopts:

```
import cyclopts


app = cyclopts.App()
```

(continues on next page)

(continued from previous page)

```python
@app.default
def hello(name: str = "World"):
    print(f"{name=} {type(name)=}")


if __name__ == "__main__":
    app()
```

```
$ my-script foo
name='foo' type(name)=<class 'str'>

$ my-script 100
name='100' type(name)=<class 'str'>

$ my-script true
name='true' type(name)=<class 'str'>

$ my-script True
name='True' type(name)=<class 'str'>
```

Fire is fine for some quick prototyping, but is not suitable for a serious CLI. Therefore, I wouldn't say Fire is a direct competitor to Cyclopts.

CHAPTER

FORTYTHREE

# ARGUABLY COMPARISON

Arguably is another Typer-inspired type-annotation-based CLI library. Arguably was created in response to the overly intrusive nature of Typer, with the goal of minimizing clutter and maintaining code simplicity. Like Cyclopts, Arguably mostly skirts using `Annotated` by interpreting as much data as possible from the function docstring. Unlike the *Typer comparison*, many of the topics in this section are simply comparing/contrasting with Arguably, rather than claiming to be strictly better.

## 43.1 Global State

Unlike Cyclopts or Typer, with `arguably` you directly jump into decorating functions:

```
import arguably


@arguably.command
def some_function(required, not_required=2, *others: int, option: float = 3.14):
    """
    this function is on the command line!

    Args:
        required: a required argument
        not_required: this one isn't required, since it has a default value
        *others: all the other positional arguments go here
        option: [-x] keyword-only args are options, short name is in brackets
    """
    print(f"{required=}, {not_required=}, {others=}, {option=}")


if __name__ == "__main__":
    arguably.run()
```

With Arguably, no application object is created. This immediately becomes an issue if you use a library that uses arguably on import.

Lets consider the following file:

```
# library_using_arguably.py
import arguably


@arguably.command
```

(continues on next page)

(continued from previous page)

```
def some_library_function(name):
    print(f"{name=}")


if __name__ == "__main__":
    arguably.run()
```

```
$ python library_using_arguably.py foo
name='foo'
```

So this by itself works fine, but lets create another script that imports this library:

```
import arguably
import library_using_arguably


@arguably.command
def my_function(name):
    print(f"{name=}")


if __name__ == "__main__":
    arguably.run()
```

Now, lets check the help screen:

```
$ python my-script.py --help
usage: my-script.py [-h] command ...

positional arguments:
  command
    some-library-function
    my-function

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

The two CLI applications got combined into one, making Arguably dangerous for CLIs that are also libraries.

## 43.2 Subcommands

Arguably parses the command tree based on __ delimited function names.

```
import arguably


@arguably.command
def ec2__start_instances(*instances):
    """Start instances.

    Args:
        *instances: {instance}s to start
```

(continues on next page)

(continued from previous page)

```
    """
    for inst in instances:
        print(f"Starting {inst}")


@arguably.command
def ec2__stop_instances(*instances):
    """Stop instances.

    Args:
        *instances: {instance}s to stop
    """
    for inst in instances:
        print(f"Stopping {inst}")


if __name__ == "__main__":
    arguably.run()
```

```
$ python main.py ec2 --help
positional arguments:
   command
     start-instances  start instances.
     stop-instances   stop instances.
```

Cyclopts handles the command tree by creating and registering recursive *App* objects:

```
from cyclopts import App

app = App()
ec2 = app.command(App(name="ec2"))


@ec2.command
def start_instances(*instances):
    """Start instances.

    Args:
        *instances: {instance}s to start
    """
    for inst in instances:
        print(f"Starting {inst}")


@ec2.command
def stop_instances(*instances):
    """Stop instances.

    Args:
        *instances: {instance}s to stop
    """
    for inst in instances:
```

(continues on next page)

(continued from previous page)

```
        print(f"Stopping {inst}")


if __name__ == "__main__":
    app()
```

```
$ python main.py ec2 --help
─ Commands ──────────────────────────────────────────────────────────────────
│ start-instances  start instances.                                        │
│ stop-instances   stop instances.                                         │
─────────────────────────────────────────────────────────────────────────────
```

# INDEX

## Symbols


__call__() (*cyclopts.App method*), 102
__call__() (*cyclopts.Parameter method*), 120
__call__() (*cyclopts.help.AsteriskRenderer method*), 155
__call__() (*cyclopts.help.DefaultFormatter method*), 146
__call__() (*cyclopts.help.DescriptionRenderer method*), 154
__call__() (*cyclopts.help.NameRenderer method*), 154
__call__() (*cyclopts.help.PlainFormatter method*), 147
__call__() (*cyclopts.help.protocols.ColumnSpecBuilder method*), 147
__call__() (*cyclopts.help.protocols.HelpFormatter method*), 145
__contains__() (*cyclopts.ArgumentCollection method*), 129
__getitem__() (*cyclopts.App method*), 98
__iter__() (*cyclopts.App method*), 98


## A


accepts_keys (*cyclopts.Parameter attribute*), 116
alias (*cyclopts.App attribute*), 88
alias (*cyclopts.Parameter attribute*), 111
all_options (*cyclopts.help.HelpEntry property*), 156
all_or_none (*in module cyclopts.validators*), 134
allow_leading_hyphen (*cyclopts.Parameter attribute*), 114
allow_unknown (*cyclopts.config.Dict attribute*), 159
allow_unknown (*cyclopts.config.Json attribute*), 158
allow_unknown (*cyclopts.config.Toml attribute*), 157
allow_unknown (*cyclopts.config.Yaml attribute*), 158
App (*class in cyclopts*), 87
app (*cyclopts.CycloptsError attribute*), 160
app (*cyclopts.UnknownCommandError attribute*), 161
append() (*cyclopts.Argument method*), 126
Argument (*class in cyclopts*), 125
argument (*cyclopts.CycloptsError attribute*), 160
argument (*cyclopts.UnknownCommandError attribute*), 161
argument_collection (*cyclopts.UnknownOptionError attribute*), 161
ArgumentCollection (*class in cyclopts*), 129
assemble_argument_collection() (*cyclopts.App method*), 101
AsteriskRenderer (*class in cyclopts.help*), 154


## B


backend (*cyclopts.App attribute*), 94
BinPath (*in module cyclopts.types*), 139
border_style (*cyclopts.help.PanelSpec attribute*), 148
border_style (*cyclopts.help.TableSpec attribute*), 150
box (*cyclopts.help.PanelSpec attribute*), 149
box (*cyclopts.help.TableSpec attribute*), 150
build() (*cyclopts.help.PanelSpec method*), 149
build() (*cyclopts.help.TableSpec method*), 151


## C


caption (*cyclopts.help.TableSpec attribute*), 150
children (*cyclopts.Argument attribute*), 126
children_recursive (*cyclopts.Argument property*), 126
choices (*cyclopts.help.HelpEntry attribute*), 156
CoercionError, 161
collapse_padding (*cyclopts.help.TableSpec attribute*), 151
column_specs (*cyclopts.help.DefaultFormatter attribute*), 146
ColumnSpec (*class in cyclopts.help*), 151
ColumnSpecBuilder (*class in cyclopts.help.protocols*), 147
combine() (*cyclopts.Parameter class method*), 120
CombinedShortOptionError, 162
command (*cyclopts.config.Env attribute*), 160
command() (*cyclopts.App method*), 99
command_chain (*cyclopts.CycloptsError attribute*), 160
command_chain (*cyclopts.UnknownCommandError attribute*), 161
CommandCollisionError, 162
config (*cyclopts.App attribute*), 93
console (*cyclopts.App attribute*), 92
console (*cyclopts.CycloptsError attribute*), 160
console (*cyclopts.UnknownCommandError attribute*), 161

consume_multiple (*cyclopts.Parameter attribute*), 117
convert() (*cyclopts.Argument method*), 126
convert_and_validate() (*cyclopts.Argument method*), 127
converter (*cyclopts.Parameter attribute*), 111
copy() (*cyclopts.ArgumentCollection method*), 129
copy() (*cyclopts.help.ColumnSpec method*), 154
copy() (*cyclopts.help.HelpEntry method*), 156
copy() (*cyclopts.help.HelpPanel method*), 155
copy() (*cyclopts.help.PanelSpec method*), 149
copy() (*cyclopts.help.TableSpec method*), 151
count (*cyclopts.Parameter attribute*), 119
create_ordered() (*cyclopts.Group class method*), 124
CsvPath (*in module cyclopts.types*), 139
cyclopts.types.StdioPath (*built-in class*), 137
CycloptsError, 160
CycloptsPanel (*class in cyclopts*), 133

## D

data (*cyclopts.config.Dict attribute*), 159
default (*cyclopts.help.HelpEntry attribute*), 156
default() (*cyclopts.App method*), 100
default() (*cyclopts.Parameter class method*), 120
default_name_transform() (*in module cyclopts*), 131
default_parameter (*cyclopts.App attribute*), 92
default_parameter (*cyclopts.Group attribute*), 123
DefaultFormatter (*class in cyclopts.help*), 145
description (*cyclopts.help.HelpEntry attribute*), 156
description (*cyclopts.help.HelpPanel attribute*), 155
DescriptionRenderer (*class in cyclopts.help*), 154
Dict (*class in cyclopts.config*), 159
dir_okay (*cyclopts.validators.Path attribute*), 136
Directory (*in module cyclopts.types*), 138

## E

edit() (*in module cyclopts*), 131
EditorDidNotChangeError, 162
EditorDidNotSaveError, 162
EditorError, 162
EditorNotFoundError, 162
Email (*in module cyclopts.types*), 144
end_of_options_delimiter (*cyclopts.App attribute*), 94
entries (*cyclopts.help.HelpPanel attribute*), 155
Env (*class in cyclopts.config*), 159
env_var (*cyclopts.help.HelpEntry attribute*), 156
env_var (*cyclopts.Parameter attribute*), 115
env_var_split (*cyclopts.Parameter attribute*), 116
env_var_split() (*cyclopts.Argument method*), 127
env_var_split() (*in module cyclopts*), 131
error_console (*cyclopts.App attribute*), 92
exception_message (*cyclopts.ValidationError attribute*), 160
ExistingBinPath (*in module cyclopts.types*), 139
ExistingCsvPath (*in module cyclopts.types*), 139
ExistingDirectory (*in module cyclopts.types*), 138
ExistingFile (*in module cyclopts.types*), 139
ExistingImagePath (*in module cyclopts.types*), 140
ExistingJsonPath (*in module cyclopts.types*), 141
ExistingMp4Path (*in module cyclopts.types*), 140
ExistingPath (*in module cyclopts.types*), 138
ExistingTomlPath (*in module cyclopts.types*), 141
ExistingTxtPath (*in module cyclopts.types*), 140
ExistingYamlPath (*in module cyclopts.types*), 141
exists (*cyclopts.validators.Path attribute*), 136
exit_on_error (*cyclopts.App attribute*), 90
expand (*cyclopts.help.PanelSpec attribute*), 149
expand (*cyclopts.help.TableSpec attribute*), 151
ext (*cyclopts.validators.Path attribute*), 136

## F

field_info (*cyclopts.Argument attribute*), 125
FieldInfo (*class in cyclopts.field_info*), 125
File (*in module cyclopts.types*), 139
file_okay (*cyclopts.validators.Path attribute*), 136
filter_by() (*cyclopts.ArgumentCollection method*), 130
footer (*cyclopts.help.ColumnSpec attribute*), 152
footer_style (*cyclopts.help.ColumnSpec attribute*), 153
footer_style (*cyclopts.help.TableSpec attribute*), 150
format (*cyclopts.help.HelpPanel attribute*), 155

## G

generate_completion() (*cyclopts.App method*), 106
generate_docs() (*cyclopts.App method*), 105
get() (*cyclopts.ArgumentCollection method*), 129
get_choices() (*cyclopts.Argument method*), 128
Group (*class in cyclopts*), 120
group (*cyclopts.App attribute*), 93
group (*cyclopts.Parameter attribute*), 113
group (*cyclopts.ValidationError attribute*), 160
group_arguments (*cyclopts.App attribute*), 93
group_commands (*cyclopts.App attribute*), 93
group_parameters (*cyclopts.App attribute*), 93
groups (*cyclopts.ArgumentCollection property*), 130
gt (*cyclopts.validators.Number attribute*), 135
gte (*cyclopts.validators.Number attribute*), 135

## H

has_tokens (*cyclopts.Argument property*), 126
header (*cyclopts.help.ColumnSpec attribute*), 152
header_style (*cyclopts.help.ColumnSpec attribute*), 153
header_style (*cyclopts.help.TableSpec attribute*), 150
height (*cyclopts.help.PanelSpec attribute*), 149
help (*cyclopts.App attribute*), 88

help (*cyclopts.Group attribute*), 121
help (*cyclopts.Parameter attribute*), 115
help_epilogue (*cyclopts.App attribute*), 89
help_flags (*cyclopts.App attribute*), 88
help_format (*cyclopts.App attribute*), 88
help_formatter (*cyclopts.App attribute*), 89
help_formatter (*cyclopts.Group attribute*), 121
help_on_error (*cyclopts.App attribute*), 90
help_print() (*cyclopts.App method*), 105
HelpEntry (*class in cyclopts.help*), 155
HelpFormatter (*class in cyclopts.help.protocols*), 144
HelpPanel (*class in cyclopts.help*), 155
HexUInt16 (*in module cyclopts.types*), 143
HexUInt32 (*in module cyclopts.types*), 143
HexUInt64 (*in module cyclopts.types*), 143
HexUInt8 (*in module cyclopts.types*), 142
highlight (*cyclopts.help.ColumnSpec attribute*), 154
highlight (*cyclopts.help.PanelSpec attribute*), 149
hint (*cyclopts.Argument attribute*), 125

## I

ImagePath (*in module cyclopts.types*), 140
implicit_value (*cyclopts.Token attribute*), 124
index (*cyclopts.Argument attribute*), 125
index (*cyclopts.Token attribute*), 124
install_completion() (*cyclopts.App method*), 107
Int16 (*in module cyclopts.types*), 143
Int32 (*in module cyclopts.types*), 143
Int64 (*in module cyclopts.types*), 143
Int8 (*in module cyclopts.types*), 142
interactive_shell() (*cyclopts.App method*), 108
is_flag() (*cyclopts.Argument method*), 128
is_positional_only() (*cyclopts.Argument method*), 128
is_stdio (*cyclopts.types.StdioPath attribute*), 137
is_var_positional() (*cyclopts.Argument method*), 128

## J

Json (*class in cyclopts.config*), 158
Json (*in module cyclopts.types*), 144
json_dict (*cyclopts.Parameter attribute*), 118
json_list (*cyclopts.Parameter attribute*), 119
JsonPath (*in module cyclopts.types*), 141
justify (*cyclopts.help.ColumnSpec attribute*), 153

## K

keys (*cyclopts.Argument attribute*), 125
keys (*cyclopts.Token attribute*), 124
keyword (*cyclopts.MissingArgumentError attribute*), 162
keyword (*cyclopts.Token attribute*), 124

## L

LimitedChoice (*class in cyclopts.validators*), 134
lt (*cyclopts.validators.Number attribute*), 135
lte (*cyclopts.validators.Number attribute*), 135

## M

match() (*cyclopts.Argument method*), 126
match() (*cyclopts.ArgumentCollection method*), 130
max_width (*cyclopts.help.ColumnSpec attribute*), 153
min_width (*cyclopts.help.ColumnSpec attribute*), 153
min_width (*cyclopts.help.TableSpec attribute*), 151
MissingArgumentError, 162
MixedArgumentError, 162
modulo (*cyclopts.validators.Number attribute*), 135
Mp4Path (*in module cyclopts.types*), 140
msg (*cyclopts.CycloptsError attribute*), 160
msg (*cyclopts.UnknownCommandError attribute*), 161
must_exist (*cyclopts.config.Json attribute*), 158
must_exist (*cyclopts.config.Toml attribute*), 157
must_exist (*cyclopts.config.Yaml attribute*), 157
mutually_exclusive (*in module cyclopts.validators*), 134
MutuallyExclusive (*class in cyclopts.validators*), 134

## N

n_tokens (*cyclopts.Parameter attribute*), 119
name (*cyclopts.App attribute*), 87
name (*cyclopts.Argument property*), 127
name (*cyclopts.Group attribute*), 121
name (*cyclopts.Parameter attribute*), 110
name_transform (*cyclopts.App attribute*), 93
name_transform (*cyclopts.Parameter attribute*), 116
NameRenderer (*class in cyclopts.help*), 154
names (*cyclopts.Argument property*), 127
names (*cyclopts.help.HelpEntry property*), 155
negative (*cyclopts.Parameter attribute*), 113
negative_bool (*cyclopts.Parameter attribute*), 114
negative_iterable (*cyclopts.Parameter attribute*), 114
negative_names (*cyclopts.help.HelpEntry attribute*), 155
negative_none (*cyclopts.Parameter attribute*), 114
negative_shorts (*cyclopts.help.HelpEntry attribute*), 155
NegativeFloat (*in module cyclopts.types*), 142
NegativeInt (*in module cyclopts.types*), 142
negatives (*cyclopts.Argument property*), 127
no_wrap (*cyclopts.help.ColumnSpec attribute*), 153
NonExistentBinPath (*in module cyclopts.types*), 139
NonExistentCsvPath (*in module cyclopts.types*), 140
NonExistentDirectory (*in module cyclopts.types*), 138
NonExistentFile (*in module cyclopts.types*), 139
NonExistentImagePath (*in module cyclopts.types*), 140
NonExistentJsonPath (*in module cyclopts.types*), 141
NonExistentMp4Path (*in module cyclopts.types*), 140
NonExistentPath (*in module cyclopts.types*), 138
NonExistentTomlPath (*in module cyclopts.types*), 141

NonExistentTxtPath (*in module cyclopts.types*), 140
NonExistentYamlPath (*in module cyclopts.types*), 141
NonNegativeFloat (*in module cyclopts.types*), 142
NonNegativeInt (*in module cyclopts.types*), 142
NonPositiveFloat (*in module cyclopts.types*), 142
NonPositiveInt (*in module cyclopts.types*), 142
Number (*class in cyclopts.validators*), 134

## O

overflow (*cyclopts.help.ColumnSpec attribute*), 153

## P

pad_edge (*cyclopts.help.TableSpec attribute*), 151
padding (*cyclopts.help.PanelSpec attribute*), 149
padding (*cyclopts.help.TableSpec attribute*), 151
panel_spec (*cyclopts.help.DefaultFormatter attribute*), 146
PanelSpec (*class in cyclopts.help*), 148
Parameter (*class in cyclopts*), 109
parameter (*cyclopts.Argument attribute*), 125
parse (*cyclopts.Argument property*), 127
parse (*cyclopts.Parameter attribute*), 114
parse_args() (*cyclopts.App method*), 102
parse_commands() (*cyclopts.App method*), 99
parse_known_args() (*cyclopts.App method*), 101
Path (*class in cyclopts.validators*), 135
path (*cyclopts.config.Json attribute*), 158
path (*cyclopts.config.Toml attribute*), 156
path (*cyclopts.config.Yaml attribute*), 157
PlainFormatter (*class in cyclopts.help*), 147
Port (*in module cyclopts.types*), 144
positive_names (*cyclopts.help.HelpEntry attribute*), 155
positive_shorts (*cyclopts.help.HelpEntry attribute*), 155
PositiveFloat (*in module cyclopts.types*), 142
PositiveInt (*in module cyclopts.types*), 142
prefix (*cyclopts.config.Env attribute*), 159
print_error (*cyclopts.App attribute*), 90

## R

ratio (*cyclopts.help.ColumnSpec attribute*), 153
register_install_completion_command() (*cyclopts.App method*), 107
render_description() (*cyclopts.help.DefaultFormatter method*), 147
render_description() (*cyclopts.help.PlainFormatter method*), 147
render_usage() (*cyclopts.help.DefaultFormatter method*), 146
render_usage() (*cyclopts.help.PlainFormatter method*), 147
renderer (*cyclopts.help.ColumnSpec attribute*), 152
RepeatArgumentError, 162
required (*cyclopts.Argument property*), 128
required (*cyclopts.help.HelpEntry attribute*), 156
required (*cyclopts.Parameter attribute*), 115
ResolvedDirectory (*in module cyclopts.types*), 138
ResolvedExistingDirectory (*in module cyclopts.types*), 139
ResolvedExistingFile (*in module cyclopts.types*), 139
ResolvedExistingPath (*in module cyclopts.types*), 138
ResolvedFile (*in module cyclopts.types*), 139
ResolvedPath (*in module cyclopts.types*), 138
result_action (*cyclopts.App attribute*), 94
root_input_tokens (*cyclopts.CycloptsError attribute*), 160
root_input_tokens (*cyclopts.UnknownCommandError attribute*), 161
root_keys (*cyclopts.config.Dict attribute*), 159
root_keys (*cyclopts.config.Json attribute*), 158
root_keys (*cyclopts.config.Toml attribute*), 156
root_keys (*cyclopts.config.Yaml attribute*), 157
run() (*in module cyclopts*), 132
run_async() (*cyclopts.App method*), 104

## S

safe_box (*cyclopts.help.PanelSpec attribute*), 149
safe_box (*cyclopts.help.TableSpec attribute*), 151
search_parents (*cyclopts.config.Json attribute*), 158
search_parents (*cyclopts.config.Toml attribute*), 157
search_parents (*cyclopts.config.Yaml attribute*), 157
shorts (*cyclopts.help.HelpEntry property*), 155
show (*cyclopts.App attribute*), 90
show (*cyclopts.Argument property*), 127
show (*cyclopts.config.Env attribute*), 160
show (*cyclopts.Group attribute*), 121
show (*cyclopts.Parameter attribute*), 115
show_choices (*cyclopts.Parameter attribute*), 115
show_default (*cyclopts.Argument property*), 126
show_default (*cyclopts.Parameter attribute*), 115
show_edge (*cyclopts.help.TableSpec attribute*), 150
show_env_var (*cyclopts.Parameter attribute*), 115
show_footer (*cyclopts.help.TableSpec attribute*), 150
show_header (*cyclopts.help.TableSpec attribute*), 150
show_lines (*cyclopts.help.TableSpec attribute*), 150
sort_key (*cyclopts.App attribute*), 91
sort_key (*cyclopts.Group attribute*), 121
sort_key (*cyclopts.help.HelpEntry attribute*), 156
source (*cyclopts.config.Dict attribute*), 159
source (*cyclopts.config.Env attribute*), 160
source (*cyclopts.config.Json attribute*), 158
source (*cyclopts.config.Toml attribute*), 156
source (*cyclopts.config.Yaml attribute*), 157
source (*cyclopts.Token attribute*), 124
STDIO_STRING (*cyclopts.types.StdioPath attribute*), 137

style (*cyclopts.help.ColumnSpec attribute*), 153
style (*cyclopts.help.PanelSpec attribute*), 148
style (*cyclopts.help.TableSpec attribute*), 150
subtitle (*cyclopts.help.PanelSpec attribute*), 148
subtitle_align (*cyclopts.help.PanelSpec attribute*), 148
suppress_keyboard_interrupt (*cyclopts.App attribute*), 94

## T

table_spec (*cyclopts.help.DefaultFormatter attribute*), 146
TableSpec (*class in cyclopts.help*), 149
target (*cyclopts.CycloptsError attribute*), 160
target (*cyclopts.UnknownCommandError attribute*), 161
target_type (*cyclopts.CoercionError attribute*), 161
title (*cyclopts.help.HelpPanel attribute*), 155
title (*cyclopts.help.PanelSpec attribute*), 148
title (*cyclopts.help.TableSpec attribute*), 150
title_align (*cyclopts.help.PanelSpec attribute*), 148
Token (*class in cyclopts*), 124
token (*cyclopts.CoercionError attribute*), 161
token (*cyclopts.RepeatArgumentError attribute*), 162
token (*cyclopts.UnknownOptionError attribute*), 161
token_count() (*cyclopts.Argument method*), 127
tokens (*cyclopts.Argument attribute*), 125
tokens_so_far (*cyclopts.MissingArgumentError attribute*), 162
Toml (*class in cyclopts.config*), 156
TomlPath (*in module cyclopts.types*), 141
TxtPath (*in module cyclopts.types*), 140
type (*cyclopts.help.HelpEntry attribute*), 156

## U

UInt16 (*in module cyclopts.types*), 143
UInt32 (*in module cyclopts.types*), 143
UInt64 (*in module cyclopts.types*), 143
UInt8 (*in module cyclopts.types*), 142
UnknownCommandError, 161
UnknownOptionError, 161
UNSET (*class in cyclopts*), 131
unused_tokens (*cyclopts.CycloptsError attribute*), 160
unused_tokens (*cyclopts.UnknownCommandError attribute*), 161
UnusedCliTokensError, 162
update() (*cyclopts.App method*), 109
URL (*in module cyclopts.types*), 144
usage (*cyclopts.App attribute*), 90
use_commands_as_keys (*cyclopts.config.Dict attribute*), 159
use_commands_as_keys (*cyclopts.config.Json attribute*), 158
use_commands_as_keys (*cyclopts.config.Toml attribute*), 157
use_commands_as_keys (*cyclopts.config.Yaml attribute*), 158

## V

validate() (*cyclopts.Argument method*), 127
ValidationError, 160
validator (*cyclopts.App attribute*), 93
validator (*cyclopts.Group attribute*), 123
validator (*cyclopts.Parameter attribute*), 113
value (*cyclopts.Argument property*), 126
value (*cyclopts.Token attribute*), 124
value (*cyclopts.ValidationError attribute*), 161
verbose (*cyclopts.App attribute*), 90
verbose (*cyclopts.CycloptsError attribute*), 160
verbose (*cyclopts.UnknownCommandError attribute*), 161
version (*cyclopts.App attribute*), 92
version_flags (*cyclopts.App attribute*), 92
version_format (*cyclopts.App attribute*), 90
version_print() (*cyclopts.App method*), 98
vertical (*cyclopts.help.ColumnSpec attribute*), 153

## W

width (*cyclopts.help.ColumnSpec attribute*), 153
width (*cyclopts.help.PanelSpec attribute*), 149
width (*cyclopts.help.TableSpec attribute*), 151
with_newline_metadata() (*cyclopts.help.DefaultFormatter class method*), 146

## Y

Yaml (*class in cyclopts.config*), 157
YamlPath (*in module cyclopts.types*), 141