cyclopts

Release 0.0.0

Brian Pugh

May 16, 2024

USAGE

Documentation: https://cyclopts.readthedocs.io

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

Cyclopts is a modern, easy-to-use command-line interface (CLI) framework. It offers a streamlined approach for building CLI applications with an emphasis on simplicity, extensibility, and robustness. Cyclopts aims to provide an intuitive and efficient developer experience, making python CLI development more accessible and enjoyable.

CHAPTER

ONE

WHY CYCLOPTS?

- **Intuitive API**: Cyclopts features a straightforward and intuitive API, making it easy for developers to create complex CLI applications with minimal code.
- **Advanced Type Hinting**: Cyclopts offers advanced type hinting features, allowing for more accurate and informative command-line interfaces.
- **Rich Help Generation**: Automatically generates beautiful, user-friendly help messages, ensuring that users can easily understand and utilize your CLI application.
- **Extensible and Customizable**: Designed with extensibility in mind, Cyclopts allows developers to easily add custom behaviors and integrate with other systems.

CHAPTER

TWO

INSTALLATION

Cyclopts requires Python >=3.8; to install Cyclopts, run:

```
pip install cyclopts
```

CHAPTER

THREE

QUICK START

- 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 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 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".               │
╰──────────────────────────────────────────────────────────────────────────────────────────────────────────────────╯
```

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

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

app = 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)


@app.callback()
def dummy_callback():
```

(continues on next page)

(continued from previous page)

```python
    pass


@app.command(help="Deploy code to an environment.")
def deploy(
    env: Annotated[Environment, Argument(help="Environment to deploy to.")],
    replicas: Annotated[
        int,
        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 [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␣
↪──────────────────────────────────────────────────────────────────────────────────────────────────────────────
│ --install-completion        [bash|zsh|fish|powershell|pwsh]  Install completion for␣
↪the specified shell. [default: None]                │
│ --show-completion           [bash|zsh|fish|powershell|pwsh]  Show completion for the␣
↪specified shell, to copy it or customize the     │
│                                                              installation.           ␣
↪                                                  │
│                                                              [default: None]         ␣
↪                                                  │
│ --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 deploy [OPTIONS] ENV:{dev|staging|prod} [REPLICAS]
Try 'my-script deploy --help' for help.
╭─ Error ──────────────────────────────────────────────────────────────────────╮
│ Invalid value for 'ENV:{dev|staging|prod}': 'nonexistent-env' is not one of │
│ 'dev', 'staging', 'prod'.                                                    │
╰──────────────────────────────────────────────────────────────────────────────╯
```

The Typer implementation is 43 lines long, while the Cyclopts implementation is just 30, all while including a proper docstring. Since Typer doesn't support Unions, the choices for `replica` could not be displayed on the help page. We also had to include a dummy callback since our application currently only has a single command. Cyclopts is much more terse, much more readable, and much more intuitive to use.

CHAPTER

FIVE

API

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

5.1 Installation

cyclopts requires Python >=3.8 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 development, its recommended to use Poetry:

```
git clone https://github.com/BrianPugh/cyclopts.git
cd cyclopts
poetry install
```

5.2 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.

5.2.1 A Basic Cyclopts Application

The most basic Cyclopts application is as follows:

```
import cyclopts

app = cyclopts.App()


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

(continues on next page)

(continued from previous page)

```
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 investigate in future guides. The `app` object has a decorator method, *default*, which registers a function as the default action. An application can have only a single default action. In this example, the `main` function is our default, and is executed when no CLI command is provided.

5.2.2 Function Arguments

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

```
import cyclopts

app = cyclopts.App()


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


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

Execute it with an argument:

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

Here's what's happening:

1. The function `main` was registered to `app` as the default action.
2. Calling `app()` triggers Cyclopts to parse CLI inputs.
3. Cyclopts identifies `"Alice"` as a positional argument, matching it to the parameter `name`. In the absence of an explicit type hint, Cyclopts defaults to `str`.
4. Cyclopts calls the registered default `main("Alice")`, and the greeting is printed.

5.2.3 Multiple Arguments

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

```python
import cyclopts

app = cyclopts.App()


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


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

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

The command line input `"3"` is converted to an integer because of `count`'s type hint `int`. Cyclopts natively handles all python builtin types, see *Coercion Rules* for more details. Cyclopts adheres to Python's argument binding rules, allowing both positional and keyword arguments. Therefore, all these commands are equivalent:

```
$ python main.py Alice 3
$ python main.py --name Alice --count 3
$ python main.py --name=Alice --count=3
$ python main.py --count 3 --name=Alice
$ python main.py Alice --count 3
$ python main.py --count 3 Alice
$ python main.py --name=Alice 3
$ python main.py 3 --name=Alice
```

Cyclopts parses keyword arguments first, then fills in the gaps with positional arguments.

5.2.4 Adding Help

We can add application-level help documentation when creating our `app`:

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

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

Help string for this demo application.

─ Commands ─────────────────────────────────────────────────────────────
│ --help,-h  Display this message and exit.                            │
```

(continues on next page)

(continued from previous page)

```
| --version  Display application version.                                    |
```

By default, Cyclopts adds `--help` and `--version` commands to your CLI. If *App.help* is not set, Cyclopts will fallback to the first line (short description) of the registered `@app.default` function's docstring.

5.3 Commands

There are 2 function-registering decorators:

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*. The default *app.default* handler runs *app.help_print*.

2. *@app.command* - Registers a function or *App* as a command. Commands require explicit CLI invocation.

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

5.3.1 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()
```

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

$ my-script buzz 4
BUZZ: 4
```

5.3.2 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:  app.command(sub_app := 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 bar 4
BAZ: 4
```

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

5.3.3 Changing Name

By default, a command is registered to the function name with underscores replaced with hyphens. Any leading or trailing underscore/hyphens will also be stripped. For example, the function `_foo_bar()` will become the command `foo-bar`. This automatic command 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
app = App(name_transform=lambda s: s)
```

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

```python
@app.command(name="bar")
def foo():
    print("Hello World!")


app(["bar"])
# Hello World!
```

5.3.4 Adding Help

There are a few ways to adding 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.
2. If the registered command is a sub app, the sub app's *help* field will be used.

```
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.

```
app = cyclopts.App()


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


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

```
$ 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.                                  │
╰────────────────────────────────────────────────────────────────────────────╯
```

5.3.5 Async

Cyclopts works with async functions too, it will run async function with `asyncio.run`

```
app = cyclopts.App()


@app.command
async def foo():
    await asyncio.sleep(10)


app()
```

5.3.6 Decorated Function Details

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

5.4 Parameters

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

```python
import cyclopts

app = cyclopts.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()
```

```
$ 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 terse and clean as possible. However, if more control is required, we can use *Parameter* along with the python builtin `Annotated`. Prior to Python 3.9, `Annotated` has to be imported from `typing_extensions`; in more recent python versions it can be directly imported from the `typing` module.

```python
from cyclopts import Parameter
from typing_extensions import Annotated


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

Parameter gives complete control on how Cyclopts processes the annotated parameter. See the API page for all configurable options.

5.4.1 Naming

Like *command names*, commandline parameters are derived from their python function argument counterparts. This automatic command name transform can be configured by *Parameter.name_transform*. Note that the resulting string is **before** the standard `--` is prepended.

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

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

Manually set names via *Parameter.name* are not subject to *Parameter.name_transform*.

5.4.2 Help

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

```python
@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]                   │
╰──────────────────────────────────────────────────────────────╯
```

5.4.3 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.

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
from typing_extensions import Annotated
from pathlib import Path

app = App()

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


def byte_units(type_, *values):
    value = values[0].lower()
    try:
        return int(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 all the string tokens parsed for this argument. The returned value gets used by the function.

5.4.4 Validating Input

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

```
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: int):
    if age < 21:
        print("Under 21: prohibited.")
    else:
        print("Good to go!")


app()
```

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

$ my-script 10
Under 21: prohibited.

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

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

5.4.5 Parameter Resolution

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:

```python
def must_be_multiple_of_4096(type_, value):
    assert value % 4096 == 0


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

See *Parameter Resolution Order* for more details.

5.5 Default Parameter

The default values of `Parameter` can be configured via *App.default_parameter*.

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

```python
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 provided:

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

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

Explicitly setting *negative* in the function signature overrides this configuration and works as expected:

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

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

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

5.5.1 Resolution Order

When resolving what the `Parameter` values for an individual function parameter should be, explicitly set attributes of higher priority Parameters override lower priority Parameters. 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. All App/Group/Parameter `default_parameter` values default to `None`.

5.6 Groups

Groups offer a way of organizing parameters and commands on the help-page. They also provide an additional abstraction layer that converters and validators can operate on.

Groups can be created in 2 ways:

1. Creating an instance of the *`Group`* object.
2. Implicitly with string title name. This is short for `Group(my_group_name)`. If there exists a *`Group`* object with the same name within the command/parameter context, it will join that group.

Every command and parameter belongs to one (or more) groups.

Group(s) can be provided to the `group` keyword argument of *`@app.command`* and *`Parameter`*. The *`Group`* class itself only marks objects with metadata and doesn't contain a set of it's members. This means that groups can be re-used across commands.

5.6.1 Command Groups

An example of using groups with commands:

```
from cyclopts import App, Group, Parameter

app = App()

# Change the group of "--help" and "--version" to the implicit "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
```

(continues on next page)

(continued from previous page)

```python
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_command`, which defaults to a group named `"Commands"`.

5.6.2 Parameter Groups

An example of using groups with parameters:

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

app = App()


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


@app.command
def create(
    *,
    # Using an explicitly created group object.
    car: Annotated[bool, Parameter(group=vehicle_type_group)] = False,
```

(continues on next page)

(continued from previous page)

```
    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]

─ 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]                                            │
─────────────────────────────────────────────────────────────────────────────────

$ python my-script.py create --car --truck
─ Error ─────────────────────────────────────────────────────────────────────────
│ Mutually exclusive arguments: {--car, --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".

5.6.3 Converters

Converters offer a way of having parameters within a group interact during processing. Groups with an empty name, or with `show=False`, are a way of using converters without impacting the help-page. See *Group.converter* for details.

5.6.4 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 validators without impacting the help-page.

```
mutually_exclusive = Group(validator=LimitedChoice(), default_parameter=Parameter(show_
↪default=False, negative=""))


@app.command
def foo(
    car: Annotated[bool, Parameter(group=(app.group_parameters, mutually_exclusive))],
    truck: Annotated[bool, Parameter(group=(app.group_parameters, mutually_exclusive))],
):
    pass
```

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

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

See *Group.validator* for details.

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

5.6.5 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.

```
from cyclopts import App, Group

app = App()

g_plants = Group.create_ordered("Plants")
g_animals = Group.create_ordered("Animals")
```

(continues on next page)

(continued from previous page)

```
g_mushrooms = Group.create_ordered("Mushrooms")


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


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


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


app()
```

```
╭─ Plants ─────────────────────────────────────────────────────────────────────╮
│ daisy                                                                        │
╰──────────────────────────────────────────────────────────────────────────────╯
╭─ Animals ────────────────────────────────────────────────────────────────────╮
│ zebra                                                                        │
╰──────────────────────────────────────────────────────────────────────────────╯
╭─ Mushrooms ──────────────────────────────────────────────────────────────────╮
│ portobello                                                                   │
╰──────────────────────────────────────────────────────────────────────────────╯
╭─ Commands ───────────────────────────────────────────────────────────────────╮
│ --help,-h  Display this message and exit.                                    │
│ --version  Display application version.                                      │
╰──────────────────────────────────────────────────────────────────────────────╯
```

A *sort_key* can still be supplied; the global counter will only be used to break sorting ties.

5.7 Parameter Validators

In a CLI application, 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. 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*. 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.

5.7.1 Path

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

```python
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 ──────────────────────────────────────────────────────────────────────╮
│ Invalid value for --path. this_file_does_not_exist.txt does not exist.       │
╰──────────────────────────────────────────────────────────────────────────────╯
```

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

5.7.2 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
```

(continues on next page)

(continued from previous page)

```
Your number in hex is f.

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

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

5.8 Group Validators

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

5.8.1 LimitedChoice

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

```python
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]

╭─ Vehicle (choose one) ───────────────────────────────────────────────────────╮
│ --car    [default: False]                                                    │
```

(continues on next page)

(continued from previous page)

```
│ --truck  [default: False]                                                    │
──────────────────────────────────────────────────────────────────────────────
─ Commands ────────────────────────────────────────────────────────────────────
│ --help,-h  Display this message and exit.                                    │
│ --version  Display application version.                                      │
──────────────────────────────────────────────────────────────────────────────

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

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

See the *LimitedChoice* docs for more info.

5.9 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.

   ```
   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*.

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

(continues on next page)

(continued from previous page)

```
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.

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


@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."""
```

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

```
app = cyclopts.App()


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

5.9.1 Markup Format

The standard markup language for docstrings in python is reStructuredText (see PEP-0287). By default, Cyclopts parses docstring descriptions as restructuredtext and renders it appropriately. To change the markup format, set the *`App.help_format`* field accordingly.

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.

PlainText

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

```python
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.
    """
```

```
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.
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.

Rich

Displays text as Rich Markup.

Note: Newlines are interpreted literally.

```
app = App(help_format="rich")
```

ReStructuredText

ReStructuredText is the default parsing behavior of Cyclopts, so *help_format* won't need to be explicitly set.

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


@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.
    """
```

Resulting help:

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

Markdown

Markdown is another popular markup language that Cyclopts can render.

```
app = App(help_format="markdown")  # or "md"


@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:

5.9.2 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=[])
```

5.10 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.

The resolution order for determining the version string is as follows:

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

   ```
   app = cyclopts.App(version="7.5.8")
   ```

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

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


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

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.

   ```
   # 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=[])
```

5.11 Coercion Rules

This page intends to serve as a terse set of type coercion rules that Cyclopts follows. If a specific type (including custom types) is not specified, the coercion defaults to `type(token: str)`. For example, Cyclopts does not have an explicit rule for `pathlib.Path`, so if the value `"foo.bin"` is provided, Cyclopts will default to coercing it as `pathlib.Path("foo.bin")`.

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`).

5.11.1 No Hint

If no explicit type hint is provided:

- If the parameter is optional and has a non-None default value, interpret the type `type(default_value)`.

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

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

- Otherwise, interpret the type as string. See *Str*.

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

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

5.11.2 Any

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

5.11.3 Str

No operation is performed, CLI tokens are natively strings.

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

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

5.11.4 List

- The inner annotation type will be applied independently to each element.
- If `Parameter.allow_leading_hyphen=False` (default behavior), all tokens will be consumed until a hyphenated-option is reached.
- If `Parameter.allow_leading_hyphen=True`, all remaining tokens will be unconditionally consumed.

```
@app.default
def main(*, favorite_numbers: List[int]):
    pass
```

```
$ my-program --favorite-numbers 1 2 3
# favorite_numbers argument is a list containing 3 integers: ``[1, 2, 3]``.
```

- To get an empty list pass in the flag `--empty-MY-LIST-NAME`. Continuing the previous example:

  ```
  $ my-program --empty-favorite-numbers
  # favorite_numbers argument is an empty list: ``[]``.
  ```

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

5.11.5 Iterable

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

5.11.6 Set

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

5.11.7 Tuple

- Parses the same number of tokens as the size of the annotated tuple.
- The inner annotation type will be applied independently to each element.
- 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*.

```
@app.default
def default(coordinates: Tuple[float, float, str]):
    pass
```

And invoke our script:

```
$ my-program --coordinates 3.14 2.718 my-coord-name
# coordinates argument is a tuple containing two floats and a string: ``(3.14, 2.718,
↪"my-coord-name")``
```

5.11.8 Union

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

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

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

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

5.11.9 Optional

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

5.11.10 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 hexadecimal values (strings starting with *0x*).
- Accepts binary values (strings starting with *0b*)

5.11.11 Float

Not explicitly handled by Cyclopts, token gets cast as `float(token)`. For example, `float("3.14")`.

5.11.12 Complex

Not explicitly handled by Cyclopts, token gets cast as `complex(token)`. For example, `complex("3+5j")`

5.11.13 Bool

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

   Example:

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

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

```
$ my-program foo 1
True

$ my-program foo 0
False
```

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). Only the positive flag can be specified this way, attempting to assign a value to the negative value will result in a `ValidationError`.

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

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

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

$ my-program foo --no-my-flag=true
╭─ Error ──────────────────────────────────────────────────────────────────────
```

(continues on next page)

(continued from previous page)

```
| Cannot assign value to negative flag "--no-my-flag".                    |
```

5.11.14 Literal

The `Literal` type is a good option for limiting the user input to a set of choices. 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
@app.default
def default(value: Literal["foo", "bar", 3]):
    print(f"{value=} {type(value)=}")
```

```
$ 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 ──────────────────────────────────────────────────────────────────────╮
│ Error converting value "fizz" to typing.Literal['foo', 'bar', 3] for "--value". │
╰──────────────────────────────────────────────────────────────────────────────╯
```

5.11.15 Enum

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

`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**.

```python
class Language(str, Enum):
    ENGLISH = "en"
    SPANISH = "es"
    GERMAN = "de"


@app.default
def default(language: Language = Language.ENGLISH):
    print(f"Using: {language}")
```

```
$ my-program english
Using: Language.ENGLISH
```

(continues on next page)

(continued from previous page)

```
$ my-program german
Using: Language.GERMAN

$ my-program french
╭─ Error ──────────────────────────────────────────────────────────────────────╮
│ Error converting value "french" to <enum 'Language'> for "--language".       │
╰──────────────────────────────────────────────────────────────────────────────╯
```

5.12 API

class cyclopts.**App**(*name=None*, *help=None*, *usage=None*, ***, *default_command=None*, *default_parameter=None*, *version=_Nothing.NOTHING*, *version_flags=['--version']*, *show=True*, *console=None*, *help_flags=['--help', '-h']*, *help_format=None*, *group=None*, *group_arguments=None*, *group_parameters=None*, *group_commands=None*, *converter=None*, *validator=None*, *name_transform=None*)

Cyclopts Application.

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

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

1. User specified `name`.
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]`.

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

help: `str | None = None`

Text to display on help screen.

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

Tokens that trigger *help_print()*. Set to an empty list to disable help feature. Defaults to `["--help", "-h"]`. Cannot be changed after instantiating the app.

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

The markup language of docstring function descriptions. If `None`, fallback to parenting *help_format*. If no *help_format* is defined, falls back to `"restructuredtext"`.

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.

version: `None | str | Callable = None`

Version to be displayed when a token of `version_flags` is parsed. Defaults to attempting to using 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"]`. Cannot be changed after instantiating the app.

console: rich.console.Console = None

Default `rich.console.Console` to use when displaying runtime errors. 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.

default_parameter: *Parameter* = None

Default *Parameter* configuration.

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.

converter: Callable | None = None

A function 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.

```
def converter(**kwargs) -> Dict[str, Any]:
    "Return an updated dictionary."
```

The returned dictionary will be used passed along to the command invocation. This converter runs **after** *Parameter* and *Group* converters.

validator: None | Callable | List[Callable] = []

A function 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.

The raised error message will be presented to the user with python-variables prepended with "--" remapped to their CLI counterparts.

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:
    ...
```

If None (default value), uses *cyclopts.default_name_transform()*. If a subapp, inherits from first non-None parent.

version_print()

Print the application version.

Return type
None

__getitem__(*key*)

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`.

Return type
App

__iter__()

Iterate over command & meta command names.

Return type
Iterator[str]

parse_commands(*tokens=None*)

Extract out the command tokens from a command.

Parameters
tokens (*Union[None, str, Iterable[str]]*) -- Either a string, or a list of strings to launch a command. Defaults to `sys.argv[1:]`

Returns

- *List[str]* -- Tokens that are interpreted as a valid command chain.
- *List[App]* -- The associated *App* object with each of those tokens.
- *List[str]* -- The remaining non-command tokens.

command(*obj=None*, *name=None*, ***kwargs*)

Decorator to register a function as a CLI command.

Return type
TypeVar(T, bound= Callable)

Parameters

- **obj** (*Optional[Callable]*) -- Function or *App* to be registered as a command.

- **name** (*Union[None, str, Iterable[str]]*) -- Name(s) to register the `obj` to. If not provided, defaults to:
  - If registering an *App*, then the app's name.
  - If registering a function, then the function's name.
- ****kwargs** -- Any argument that *App* can take.

default(*obj=None*, ***, *converter=None*, *validator=None*)

Decorator to register a function as the default action handler.

Return type
TypeVar(T, bound= Callable)

parse_known_args(*tokens=None*, ***, *console=None*)

Interpret arguments into a function, `BoundArguments`, and any remaining unknown tokens.

Return type
Tuple[Callable, BoundArguments, List[str]]

Parameters
tokens (*Union[None, str, Iterable[str]]*) -- Either a string, or a list of strings to launch a command. Defaults to `sys.argv[1:]`

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`.

parse_args(*tokens=None*, ***, *console=None*, *print_error=True*, *exit_on_error=True*, *verbose=False*)

Interpret arguments into a function and `BoundArguments`.

Return type
Tuple[Callable, BoundArguments]

Raises
UnusedCliTokensError -- If any tokens remain after parsing.

Parameters

- **tokens** (*Union[None, str, Iterable[str]]*) -- Either a string, or a list of strings to launch a command. Defaults to `sys.argv[1:]`.
- **console** (*rich.console.Console*) -- Console to print help and runtime Cyclopts errors. If not provided, follows the resolution order defined in *App.console*.
- **print_error** (*bool*) -- Print a rich-formatted error on error. Defaults to `True`.
- **exit_on_error** (*bool*) -- If there is an error parsing the CLI tokens invoke `sys.exit(1)`. Otherwise, continue to raise the exception. Defaults to `True`.
- **verbose** (*bool*) -- Populate exception strings with more information intended for developers. Defaults to `False`.

Returns

- **command** (*Callable*) -- Function associated with command action.
- **bound** (*inspect.BoundArguments*) -- Parsed and converted `args` and `kwargs` to be used when calling `command`.

__call__(*tokens=None*, ***, *console=None*, *print_error=True*, *exit_on_error=True*, *verbose=False*)

Interprets and executes a command.

Parameters

- **tokens** (*Union[None, str, Iterable[str]]*) -- Either a string, or a list of strings to launch a command. Defaults to `sys.argv[1:]`.
- **console** (*rich.console.Console*) -- Console to print help and runtime Cyclopts errors. If not provided, follows the resolution order defined in *App.console*.
- **print_error** (*bool*) -- Print a rich-formatted error on error. Defaults to `True`.
- **exit_on_error** (*bool*) -- If there is an error parsing the CLI tokens invoke `sys.exit(1)`. Otherwise, continue to raise the exception. Defaults to `True`.
- **verbose** (*bool*) -- Populate exception strings with more information intended for developers. Defaults to `False`.

Returns

return_value -- The value the parsed command handler returns.

Return type

Any

help_print(*tokens=None*, ***, *console=None*)

Print the help page.

Return type

None

Parameters

- **tokens** (*Union[None, str, Iterable[str]]*) -- Tokens to interpret for traversing the application command structure. If not provided, defaults to `sys.argv`.
- **console** (*rich.console.Console*) -- Console to print help and runtime Cyclopts errors. If not provided, follows the resolution order defined in *App.console*.

interactive_shell(*prompt='$ '*, *quit=None*, *dispatcher=None*, ***kwargs*)

Create a blocking, interactive shell.

All registered commands can be executed in the shell.

Return type

None

Parameters

- **prompt** (*str*) -- Shell prompt. Defaults to `"$ "`.
- **quit** (*Union[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** (*Optional[Dispatcher]*) -- Optional function that subsequently invokes the command. The `dispatcher` function must have signature:

```
def dispatcher(command: Callable, bound: inspect.BoundArguments) ->␣
↪Any:
    return command(*bound.args, **bound.kwargs)
```

  The above is the default dispatcher implementation.

- ****kwargs** -- Get passed along to *parse_args()*.

class cyclopts.**Parameter**(*name=None*, *converter=None*, *validator=()*, *negative=None*, *group=None*, *parse=None*, *show=None*, *show_default=None*, *show_choices=None*, *help=None*, *show_env_var=None*, *env_var=None*, *negative_bool=None*, *negative_iterable=None*, *required=None*, *allow_leading_hyphen=False*, ***, *name_transform=None*)

Cyclopts configuration for individual function parameters.

Cyclopts configuration for individual function parameters.

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

Name(s) to expose to the CLI. Defaults to the python parameter's name, prepended with `--`. Single-character options should start with `-`. Full-name options should start with `--`.

converter: `Callable | None = None`

A function that converts string token(s) into an object. The converter must have signature:

```
def converter(type_, *args) -> Any:
    pass
```

If not provided, defaults to Cyclopts's internal coercion engine.

validator: `None | Callable | Iterable[Callable] = None`

A function (or list of functions) that validates data returned by the `converter`.

```
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")`.

negative: `None | str | Iterable[str] = None`

Name(s) for empty iterables or false boolean flags. For booleans, defaults to `--no-{name}`. For iterables, defaults to `--empty-{name}`. Set to an empty list to disable this feature.

negative_bool: `str | None = None`

Prefix for negative boolean flags. Must start with "`--`". Defaults to "`--no-`".

negative_iterable: `str | None = None`

Prefix for empty iterables (like lists and sets) flags. Must start with "`--`". Defaults to "`--empty-`".

allow_leading_hyphen: `bool = False`

Allow parsing non-numeric values that begin with a hyphen `-`. This is disabled by default, allowing for more helpful error messages for unknown CLI options.

parse: `bool | None = True`

Attempt to use this parameter while parsing. Annotated parameter **must** be keyword-only.

required: `bool | None = None`

Indicates that the parameter must be supplied on the help-page. Does **not** directly enforce whether or not a parameter must be supplied; only influences the help-page. 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. If `False`, state of all other `show_*` flags are ignored. Defaults to `parse` value (`True`).

show_default: `bool | None = 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's `None`.

show_choices: `bool | None = True`

If a variable has a set of choices, display the choices on the help page. Defaults to `True`.

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. Defaults to `True`.

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.

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()*.

classmethod combine(**parameters*)

Returns a new Parameter with values of `parameters`.

Return type
Parameter

Parameters
***parameters** (*Optional[Parameter]*) -- Parameters who's attributes override `self` attributes. Ordered from least-to-highest attribute priority.

classmethod default()

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.

Return type
Parameter

class cyclopts.**Group**(*name=''*, *help=''*, *sort_key=None*, ***, *show=None*, *converter=None*, *validator=None*, *default_parameter=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.

sort_key: `Any = None`

Modifies group-panel display order on the help-page.

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

Example usage:

```
@app.command(group=Group("4", sort_key=5))
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
```

Resulting help-page:

```
Usage: app COMMAND

App Help String Line 1.

╭─ 4 ──────────────────────────────────────────────────────────────────────────╮
│ cmd1                                                                         │
╰──────────────────────────────────────────────────────────────────────────────╯
╭─ 3 ──────────────────────────────────────────────────────────────────────────╮
│ cmd2                                                                         │
╰──────────────────────────────────────────────────────────────────────────────╯
```

(continues on next page)

(continued from previous page)

```
─ 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.

converter: Callable | None

A function where the CLI-provided group 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.

```
def converter(**kwargs) -> Dict[str, Any]:
    """Return an updated dictionary."""
```

The **python variable names will be used**, which may differ from their CLI names. If a variable isn't populated from the CLI or environment variable, it will not be provided to the converter. I.e. defaults from the function signature are **not** applied prior.

The returned dictionary will be used for subsequent execution. Removing variables from the returned dictionary will unbind them. When used with *@app.command*, all function arguments are provided.

validator: Callable | None = None

A function (or list of functions) where the CLI-provided group 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."
```

Validators are **not** invoked on command groups. The group-validator runs **after** the group-converter.

The raised error message will be presented to the user with python-variables prepended with "--" remapped to their CLI counterparts.

In the following example, the python variable name "--bar" in the error message is remapped to "--buzz".

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

app = App()
```

(continues on next page)

(continued from previous page)

```python
def upper_case_only(**kwargs):
    for k, v in kwargs.items():
        if not v.isupper():
            raise ValueError(f'--{k} value "{v}" needs to be uppercase.')


group = Group("", validator=upper_case_only)


@app.default
def foo(
    bar: Annotated[str, Parameter(name="--fizz", group=group)],
    baz: Annotated[str, Parameter(name="--buzz", group=group)],
):
    pass


app()
```

```
$ python meow.py ALICE bob
╭─ Error ──────────────────────────────────────────────╮
│ --buzz value "bob" needs to be uppercase.            │
╰──────────────────────────────────────────────────────╯
```

classmethod create_ordered(**args*, *sort_key=None*, ***kwargs*)

Create a group with a globally incremented *sort_key*.

Used to create a group that will be displayed **after** a previously declared *Group.create_ordered()* group on the help-page.

If a *sort_key* is provided, it is **prepended** to the globally incremented counter value (i.e. has priority during sorting).

cyclopts.**convert**(*type_*, **args*, *converter=None*, *name_transform=None*)

Coerce variables into a specified type.

Internally used to coercing string CLI tokens into python builtin types. Externally, may be useful in a custom converter. See Cyclopt's automatic coercion rules *Coercion Rules*.

If `type_` **is not** iterable, then each element of `*args` will be converted independently. If there is more than one element, then the return type will be a `Tuple[type_, ...]`. If there is a single element, then the return type will be `type_`.

If `type_` **is** iterable, then all elements of `*args` will be collated.

Parameters

- **type** (*Type*) -- A type hint/annotation to coerce `*args` into.
- ***args** (*str*) -- String tokens to coerce.
- **converter** (*Optional[Callable[[Type, str], Any]]*) -- An optional function to convert tokens to the inner-most types. The converter should have signature:

```python
def converter(type_: type, value: str) -> Any:
    ...
```

This allows to use the *convert()* function to handle the the difficult task of traversing lists/tuples/unions/etc, while leaving the final conversion logic to the caller.

- **name_transform** (*Optional[Callable[[str], str]]*) -- Currently only used for `Enum` type hints. A function that transforms enum names and CLI values into a normalized format.

  The function should have signature:

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

  where the returned value is the name to be used on the CLI.

  If `None`, defaults to `cyclopts.default_name_transform`.

Returns
Coerced version of input `*args`.

Return type
Any

cyclopts.**default_name_transform**(*s*)

Converts a python identifier into a CLI token.

Performs the following operations (in order): :rtype: `str`

1. Convert the string to all lowercase.
2. Replace _ with -.
3. Strip any leading/trailing - (also stripping _, due to point 2).

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

5.12.1 Validators

Cyclopts has several builtin validators for common CLI inputs.

class cyclopts.validators.**LimitedChoice**(*min=0*, *max=None*)

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.
- **max** (*Optional[int]*) -- The maximum (inclusive) number of CLI parameters allowed. Defaults to `1` if `min==0`, `min` otherwise.

class cyclopts.validators.**Number**(*, *lt=None*, *lte=None*, *gt=None*, *gte=None*)

Limit input number to a value range.

lt: Union[int, float, None]

Input value must be **less than** this value.

lte: Union[int, float, None]

Input value must be **less than or equal** this value.

gt: Union[int, float, None]

Input value must be **greater than** this value.

gte: Union[int, float, None]

Input value must be **greater than or equal** this value.

class cyclopts.validators.**Path**(*, *exists=False*, *file_okay=True*, *dir_okay=True*)

Assertions on properties of pathlib.Path.

exists: bool

If True, specified path **must** exist. Defaults to False.

file_okay: bool

If True, specified path may be a file. If False, then files are not allowed. Defaults to True.

dir_okay: bool

If True, specified path may be a directory. If False, then directories are not allowed. Defaults to True.

5.12.2 Types

Cyclopts has builtin pre-defined annotated-types for common validation configurations. All definitions in this section are just predefined annotations for convenience:

```
Annotated[..., Parameter(...)]
```

Due to Cyclopts's advanced *Parameter* resolution engine, these annotations can themselves be annotated. E.g:

```
Annotated[PositiveInt, Parameter(...)]
```

Path

Path annotated types for checking existence, type, and performing path-resolution.

cyclopts.types.**ExistingPath**

A Path file or directory that **must** exist.

alias of Path[Path]

cyclopts.types.**ResolvedPath**

A Path file or directory. resolve() is invoked prior to returning the path.

alias of Path[Path]

cyclopts.types.**ResolvedExistingPath**

A Path file or directory that **must** exist. resolve() is invoked prior to returning the path.

alias of Path[Path]

cyclopts.types.**Directory**

A `Path` that **must** be a directory (or not exist).

alias of `Path`[`Path`]

cyclopts.types.**ExistingDirectory**

A `Path` directory that **must** exist.

alias of `Path`[`Path`]

cyclopts.types.**ResolvedDirectory**

A `Path` directory. `resolve()` is invoked prior to returning the path.

alias of `Path`[`Path`]

cyclopts.types.**ResolvedExistingDirectory**

A `Path` directory that **must** exist. `resolve()` is invoked prior to returning the path.

alias of `Path`[`Path`]

cyclopts.types.**File**

A `File` that **must** be a file (or not exist).

alias of `Path`[`Path`]

cyclopts.types.**ExistingFile**

A `Path` file that **must** exist.

alias of `Path`[`Path`]

cyclopts.types.**ResolvedFile**

A `Path` file. `resolve()` is invoked prior to returning the path.

alias of `Path`[`Path`]

cyclopts.types.**ResolvedExistingFile**

A `Path` file that **must** exist. `resolve()` is invoked prior to returning the path.

alias of `Path`[`Path`]

Number

Annotated types for checking common int/float value constraints.

cyclopts.types.**PositiveFloat**

A float that **must** be >`0`.

alias of `Annotated`[`float`]

cyclopts.types.**NonNegativeFloat**

A float that **must** be >=`0`.

alias of `Annotated`[`float`]

cyclopts.types.**NegativeFloat**

A float that **must** be <`0`.

alias of `Annotated`[`float`]

cyclopts.types.**NonPositiveFloat**

A float that **must** be <=0.

alias of Annotated[float]

cyclopts.types.**PositiveInt**

An int that **must** be >0.

alias of Annotated[int]

cyclopts.types.**NonNegativeInt**

An int that **must** be >=0.

alias of Annotated[int]

cyclopts.types.**NegativeInt**

An int that **must** be <0.

alias of Annotated[int]

cyclopts.types.**NonPositiveInt**

An int that **must** be <=0.

alias of Annotated[int]

5.12.3 Exceptions

exception cyclopts.**CycloptsError**(*, *msg=None*, *verbose=True*, *root_input_tokens=None*, *unused_tokens=None*, *target=None*, *cli2parameter=None*, *parameter2cli=None*, *command_chain=None*, *app=None*, *console=None*)

Bases: Exception

Root exception for runtime errors.

As CycloptsErrors bubble up the Cyclopts stack, more information is added to it. Finally, cyclopts.exceptions.format_cyclopts_error() formats the message nicely for the user.

msg: Optional[str]

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: Optional[List[str]]

The parsed CLI tokens that were initially fed into the *App*.

unused_tokens: Optional[List[str]]

Leftover tokens after parsing is complete.

target: Optional[Callable]

The python function associated with the command being parsed.

cli2parameter: Optional[Dict[str, Tuple[Parameter, Any]]]

Dictionary mapping CLI strings to python parameters.

parameter2cli: Optional[ParameterDict]

Dictionary mapping function parameters to possible CLI tokens.

command_chain: Optional[List[str]]

List of command that lead to `target`.

app: Optional[App]

The Cyclopts application itself.

console: Optional[Console]

Rich console to display runtime errors.

exception cyclopts.**ValidationError**(*, *msg=None*, *verbose=True*, *root_input_tokens=None*, *unused_tokens=None*, *target=None*, *cli2parameter=None*, *parameter2cli=None*, *command_chain=None*, *app=None*, *console=None*, *value*, *parameter=None*, *group=None*)

Bases: *CycloptsError*

Validator function raised an exception.

value: str

Parenting Assertion/Value/Type Error message.

parameter: Optional[Parameter]

Parameter who's `validator` function failed.

group: Optional[*Group*]

Group who's `validator` function failed.

exception cyclopts.**UnknownOptionError**(*, *msg=None*, *verbose=True*, *root_input_tokens=None*, *unused_tokens=None*, *target=None*, *cli2parameter=None*, *parameter2cli=None*, *command_chain=None*, *app=None*, *console=None*, *token*)

Bases: *CycloptsError*

Unknown/unregistered option provided by the cli.

token: str

exception cyclopts.**CoercionError**(*, *msg=None*, *verbose=True*, *root_input_tokens=None*, *unused_tokens=None*, *target=None*, *cli2parameter=None*, *parameter2cli=None*, *command_chain=None*, *app=None*, *console=None*, *input_value=''*, *target_type=None*, *parameter=None*)

Bases: *CycloptsError*

There was an error performing automatic type coercion.

input_value: str

String input token that couldn't be coerced.

target_type: Optional[Type]

Intended type to coerce into.

parameter: Optional[Parameter]

exception cyclopts.**InvalidCommandError**(*, *msg=None*, *verbose=True*, *root_input_tokens=None*, *unused_tokens=None*, *target=None*, *cli2parameter=None*, *parameter2cli=None*, *command_chain=None*, *app=None*, *console=None*)

Bases: *CycloptsError*

CLI token combination did not yield a valid command.

msg: Optional[str]

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: Optional[List[str]]

The parsed CLI tokens that were initially fed into the *App*.

unused_tokens: Optional[List[str]]

Leftover tokens after parsing is complete.

target: Optional[Callable]

The python function associated with the command being parsed.

cli2parameter: Optional[Dict[str, Tuple[inspect.Parameter, Any]]]

Dictionary mapping CLI strings to python parameters.

parameter2cli: Optional[ParameterDict]

Dictionary mapping function parameters to possible CLI tokens.

command_chain: Optional[List[str]]

List of command that lead to `target`.

app: Optional['App']

The Cyclopts application itself.

console: Optional['Console']

Rich console to display runtime errors.

exception cyclopts.**UnusedCliTokensError**(*, *msg=None*, *verbose=True*, *root_input_tokens=None*, *unused_tokens=None*, *target=None*, *cli2parameter=None*, *parameter2cli=None*, *command_chain=None*, *app=None*, *console=None*)

Bases: *CycloptsError*

Not all CLI tokens were used as expected.

exception cyclopts.**MissingArgumentError**(*, *msg=None*, *verbose=True*, *root_input_tokens=None*, *unused_tokens=None*, *target=None*, *cli2parameter=None*, *parameter2cli=None*, *command_chain=None*, *app=None*, *console=None*, *parameter*, *tokens_so_far*)

Bases: *CycloptsError*

A parameter had insufficient tokens to be populated.

parameter: Parameter

The parameter that failed to parse.

tokens_so_far: List[str]

The tokens that were parsed so far for this Parameter.

exception cyclopts.**CommandCollisionError**

A command with the same name has already been registered to the app.

5.13 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*.

5.13.1 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.

```
@app.command
def foo(name: str, *favorite_numbers: int):
    print(f"{name}'s favorite numbers are: {favorite_numbers}")
```

```
$ 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)
```

5.13.2 Kwargs (Variable Keywords)

A variable number of keyword arguments consume all remaining CLI tokens starting with `--`. Individual values are converted to the annotated type. As with normal python `**kwargs`, the keywords are limited to python identifiers. Most prominently, no spaces allowed. Keyword name-conversion is the *same as commands*.

```
@app.command
def add(**country_to_capitols):
    for country, capitol in country_to_capitols.items():
        print(f"Adding {country} with capitol {capitol}.")
```

```
$ my-script add --united-states="Washington, D.C." --canada=Ottawa
Adding united_states with capitol Washington, D.C..
Adding canada with capitol Ottawa.
```

5.14 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.

5.14.1 __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}!")


def main():
    app()


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

```
$ python -m mypackage World
Hello World!
```

5.14.2 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. We'll investigate the setuptools solution, and the poetry solution.

Setuptools

`setup.py` is a script at the root of your project that gets executed upon installation. `setup.cfg` and `pyproject.toml` are two other alternatives that are supported.

The following are all equivalent, **but should not be used at the same time**. It is important that the function specified **takes no arguments**.

```
# 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",
        ]
    },
)
```

```
# pyproject.toml
[project.scripts]
my-package = "mypackage.__main__:main"
```

```
# setup.cfg
[options.entry_points]
console_scripts =
    my-package = mypackage.__main__:main
```

All of these represent the same thing: create 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`. The setuptools entrypoint documentation goes into further detail.

Poetry

Poetry is a tool for dependency management and packaging in Python (and what Cyclopts uses). The syntax is very similar to setuptools:

```
# pyproject.toml

[tool.poetry.scripts]
my-package = "mypackage.__main__:main"
```

5.15 App Calling & Return Values

In this section, we'll take a closer look at the *App.__call__()* method.

5.15.1 Input Command

Typically, a Cyclopts app looks something like:

```
app = cyclopts.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.

5.15.2 Return Value

The app invocation returns the value of the called command.

```
app = cyclopts.App()


@app.command
def foo(a: int, b: int, c: int):
    return a + b + c


return_value = app("foo 1 2 3")
print(f"The return value was: {return_value}.")
# The return value was: 6.
```

If you decide you want each command to return an exit code, you could invoke your app like:

```
if __name__ == "__main__":
    sys.exit(app())
```

5.15.3 Exception Handling and Exiting

For the most part, Cyclopts is hands-off when it comes to 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. This can be disabled by specifying `exit_on_error=False` when calling the app. At the same time, you may want to set `print_error=False` to disable the printing of the formatted exception.

```
app("this-is-not-a-registered-command")
print("this will not be printed since cyclopts exited.")
# ╭─ Error ──────────────────────────────────────────────────────────────────────╮
# │ Unable to interpret valid command from "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 "<stdin>", line 1, in <module>
# File "/cyclopts/cyclopts/core.py", line 318, in __call__
#   command, bound = self.parse_args(tokens)
#                    ^^^^^^^^^^^^^^^^^^^^^^^
# File "/cyclopts/cyclopts/core.py", line 281, in parse_args
#   command, bound, unused_tokens = self.parse_known_args(tokens)
#                                   ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
# File "/cyclopts/cyclopts/core.py", line 246, in parse_known_args
#   raise InvalidCommandError(unused_tokens=unused_tokens)
# cyclopts.exceptions.InvalidCommandError: Unable to interpret valid command from "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 `InvalidCommandError` is raised the same as a normal python exception.

5.16 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!

5.16.1 Meta Sub App

Typically, a Cyclopts application is launched by calling the *App* object:

```
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.

```python
from cyclopts import App, Group, Parameter
from typing_extensions 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
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 is mostly a normal Cyclopts app; the only thing special about it is that it will be additionally scanned when generate 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.                                      │
╰──────────────────────────────────────────────────────────────────────────────╯
```

5.16.2 Meta Commands

If you want a command to circumvent `my_app_launcher`, add it as you would any other command to the meta app.

```python
@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`.

5.16.3 Custom Command Invocation

The core logic of `App.__call__()` method is the following:

```python
def __call__(self, tokens=None, **kwargs):
    tokens = normalize_tokens(tokens)
    command, bound = 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 all subsequent commands. This might be useful to share an expensive-to-create object amongst commands in a single session; see *Command Chaining*.

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

app = App()


class User:
    def __init__(self, name):
        self.name = name


@app.command
def create(
    age: int,
```

(continues on next page)

(continued from previous page)

```
    *,
    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):
    user_obj = User(user)
    command, bound = app.parse_args(tokens)
    return command(*bound.args, **bound.kwargs, user_obj=user_obj)


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. The annotated parameter **must** be a keyword-only parameter.

5.17 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.

```
import cyclopts
import itertools

app = cyclopts.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):
    # tokens is ``["foo", "123", "AND", "foo", "456", "AND", "bar", "--flag"]``
    delimiter = "AND"
```

(continues on next page)

(continued from previous page)

```
    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
```

5.18 Pydantic

Suppose you want to use pydantic to manage your validation logic instead of Cyclopts. Pydantic offers a `validate_call()` decorator that performs validation checks on invocation.

```
from cyclopts import App
from pydantic import validate_call, PositiveInt

app = App()


@app.command
@validate_call
def foo(value: PositiveInt):
    print(value)


app()
```

```
$ python my-script.py foo 10
10
$ python my-script.py foo -1
╭─ Error ──────────────────────────────────────────────────────────────────────╮
│ 1 validation error for foo                                                   │
│ 0                                                                            │
│   Input should be greater than 0 [type=greater_than,                         │
│ input_value=-1, input_type=int]                                              │
│     For further information visit                                            │
│ https://errors.pydantic.dev/2.5/v/greater_than                               │
╰──────────────────────────────────────────────────────────────────────────────╯
```

A benefit of this approach is that calling your decorated function from python code will still perform these validation checks. Pydantic's types are aliases for `Annotated[python_type, additional_metadata]`, so they are naturally compatible with Cyclopts.

5.19 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):

```python
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[*_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.
```

Not bad, but let's see how this would look with autoregistry.

```python
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[*_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. AutoRegistry is a great tool for converting string CLI options into functional objects.

5.20 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


@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",
↪"gnwmanager"])
    res = subprocess.run([sys.executable, "-m", "gnwmanager", "--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}.")
```

`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.

5.21 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.
```

(continues on next page)

(continued from previous page)

```
    """


@app.command
def bar(p1):
    """Bar Docstring.

    Parameters
    ----------
    p1: str
        Bar's first parameter.
    """


# 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:

```
@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.                               │
╰──────────────────────────────────────────────────────────────╯
```

5.22 Using `pyproject.toml`

Let's create a CLI tool that is configurable via the user's `pyproject.toml`. Think tools like Poetry, Ruff, and Codespell.

5.22.1 Base Application

For this example, we'll be creating a simple CLI named `compact` that can compress data with either the `zip` or the `lzma` algorithm.

```
from cyclopts import App, Parameter
from cyclopts.types import ExistingFile, File
from typing import Annotated, Literal, Optional
import lzma
import zlib

app = App(name="compact", help="Data compression tool.")


@app.command
def compress(src: ExistingFile, dst: Optional[File] = None, *, method: Literal["lzma",
↪"zip"] = "zip"):
    """Compress a file."""
    data = src.read_bytes()
    if method == "lzma":
        out = lzma.compress(data)
    elif method == "zip":
        out = zlib.compress(data)
    else:
```

(continues on next page)

(continued from previous page)

```
        raise NotImplementedError
    dst = dst or src.with_suffix(src.suffix + "." + method)
    dst.write_bytes(out)


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

This application works as-is, but it may be useful for the caller to set the default `method` field from their `pyproject.toml`. More precisely, we want to be able to use the following configuration:

```
# pyproject.toml
[tool.compact.compress]
method = "lzma"
```

5.22.2 Customizing Launch

First, we need to hook into the application launch process. In Cyclopts, this is done with the *Meta App* (an app that launches an app). The most basic meta-app is:

```
@app.meta.default
def main(*tokens: Annotated[str, Parameter(show=False, allow_leading_hyphen=True)]):
    app(tokens)


if __name__ == "__main__":
    app.meta()  # Call app.meta() instead of app()
```

For our purposes, we'll want to dive into the Cyclopts machinery a little further.

```
@app.meta.default
def main(*tokens: Annotated[str, Parameter(show=False, allow_leading_hyphen=True)]):
    command, bound = app.parse_args(tokens)
    return command(*bound.args, **bound.kwargs)
```

`command` is the actual python function to-be-executed. `bound` is a `BoundArguments` object containing all the parsed & converted CLI arguments. It follows that `command(*bound.args, **bound.kwargs)` would execute the function with all of our supplied arguments.

5.22.3 Reading in `pyproject.toml`

We now have an appropriate place to read `pyproject.toml` from the current working directory. Add the following to the beginning of the `main` meta-app function:

```
import tomli  # Or you can just use ``toml`` in Python >=3.11

try:
    with open("pyproject.toml", "rb") as f:
        config = tomli.load(f)["tool"]["compact"]  # Reads the [tool.compact] table.
except (FileNotFoundError, KeyError):
    config = {}
```

`config` will be empty if `pyproject.toml` does not exist, or if it does not contain a `[tool.compact]` table.

Note: Many applications search the current working directory for `pyproject.toml`, and will fallback to searching parenting directories until a `pyproject.toml` is found. Here's a snippet for that:

```
from pathlib import Path

def find_pyproject() -> Path:
    """Searches current directory, then parenting directories until a pyproject.toml is␣
↪found."""
    for parent in Path("pyproject.toml").absolute().parents:
        if (candidate := parent / path.name).exists():
            return candidate
    raise FileNotFoundError("Cannot find a pyproject.toml")
```

5.22.4 Getting the command string

We want to dynamically parse what sub-table of the config we need to access based on the command being executed. The *parse_commands()* method returns a bunch of data; the first returned element is a list of strings containing the parsed command names.

```
command_names, _, _ = app.parse_commands(tokens)
```

If we invoked our program:

```
$ python compact.py compress foo.bin
```

Then the resulting `command_names` would be `["compress"]`.

We can now access the config for this specific subcommand:

```
for command_name in command_names:
    config = config.get(command_name, {})
```

5.22.5 Updating bound arguments

Finally, we need to set these values as defaults to the `bound.arguments` dictionary. We don't want to simply update the dictionary, as that would mean our toml-configured values would overwrite CLI-provided values. Using `dict.setdefault()` will only set values for previously non-existent keys.

```
# Update the bound arguments for unset keys:
for key, value in config.items():
    bound.arguments.setdefault(key, value)
```

Warning: You are responsible to correctly interpreting/coercing data types from non-cli sources to the correct type. E.g. A value from `toml` may be a string, but the function might be expecting a `Path` object.

5.22.6 Final Code

Putting it all together, here's the complete copy/pastable example code:

```python
from cyclopts import App, Parameter
from cyclopts.types import ExistingFile, File
from typing import Annotated, Literal, Optional
import tomli
import lzma
import zlib

app = App(name="compact", help="Data compression tool.")


@app.command
def compress(src: ExistingFile, dst: Optional[File] = None, *, method: Literal["lzma",
↪"zip"] = "zip"):
    """Compress a file."""
    data = src.read_bytes()
    if method == "lzma":
        out = lzma.compress(data)
    elif method == "zip":
        out = zlib.compress(data)
    else:
        raise NotImplementedError
    dst = dst or src.with_suffix(src.suffix + "." + method)
    dst.write_bytes(out)


@app.meta.default
def main(*tokens: Annotated[str, Parameter(show=False, allow_leading_hyphen=True)]):
    try:
        with open("pyproject.toml", "rb") as f:
            config = tomli.load(f)["tool"]["compact"]
    except (FileNotFoundError, KeyError):
        config = {}

    # The main Cyclopts parsing/conversion
    command, bound = app.parse_args(tokens)

    # Get the config dictionary for the specified command.
    command_names, _, _ = app.parse_commands(tokens)
    for command_name in command_names:
        config = config.get(command_name, {})

    # Update the bound arguments for unset keys:
    for key, value in config.items():
        bound.arguments.setdefault(key, value)

    # Actual function execution.
    return command(*bound.args, **bound.kwargs)
```

(continues on next page)

(continued from previous page)

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

5.23 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:** • *`App.name`* - Optional name of application or sub-command. **Cyclopts has more user-friendly default features:** • Equivalent `no_args_is_help=True`. • 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()`* *`@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(...)]` `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.*
`Optional[List] = None`	When no CLI argument is specified, Typer passes in an empty list `[]`. *Cyclopts will not bind an argument,* resulting in the default `None`.

5.23.1 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.

5.24 Typer Comparison

Much of Cyclopts was inspired by the excellent Typer library. Despite it's popularity, Typer has some traits that I (and others) find less than ideal. Part of this stems from Typer's age, with it's first release in late 2019, soon after Python 3.8's release. Because of this, most of it's 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 it's own internal parsing strategy, gaining complete control over the process.

This section was written about the current version of Typer: `v0.9.0`.

5.24.1 Argument vs Option

In Typer, the actual difference between the `Argument` and `Option` classes aren't very clear. Generally, it can be said that:

- Arguments are **required** and provided as positional arguments.
- Options are **optional** and are provided as keyword arguments, generally 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:

```
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.

5.24.2 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
```

5.24.3 Choices

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`.

```
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()
```

(continues on next page)

(continued from previous page)

```
@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**.

```
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
```

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. Unfortuneately, Typer has not provided support, despite a feature request dating back to early 2020 Cyclopts has builtin support for `Literal`, see *Coercion Rules - 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
```

5.24.4 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:

```
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:

```
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.

```
@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.
    """
    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.

```
cyclopts_app = cyclopts.App()


@cyclopts_app.command
def foo():
    print("FOO")


cyclopts_app(["foo"])
# FOO
```

5.24.5 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:

```
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.
    """
    pass
```

```
$ my-script --help
─ Commands ─────────────────────────────────────────────────────────────────────
│ foo                 Foo Docstring.                                         │
────────────────────────────────────────────────────────────────────────────────

$ 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:

```python
cyclopts_app = cyclopts.App()


@cyclopts_app.command()
def foo(bar):
    """Foo Docstring.

    Parameters
    ----------
    bar: str
        Bar parameter docstring.
    """
    pass
```

```
$ 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!

5.24.6 Decorator Parentheses

A minor nitpick, but all of Typer's decorators require parentheses.

```python
# This doesn't work! Missing ()
@typer_app.command
def foo():
    pass
```

Cyclopts works with and without parentheses.

```python
# This works! Missing ()
@cyclopts_app.command
def foo():
    pass


@cyclopts_app.command()
```

(continues on next page)

(continued from previous page)

```
def bar():
    pass
```

5.24.7 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:

```
typer_app = typer.Typer()


@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`*.

```
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]
```

(continues on next page)

(continued from previous page)

```
cyclopts_app([])
# My favorite numbers are: [1, 2, 3]
cyclopts_app(["--empty-favorite-numbers"])
# My favorite numbers are: []
```

5.24.8 Flag Negation

For boolean parameters, Typer adds a `--no-MY-FLAG-NAME` to specify a `False` argument.

```
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:

```
@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.

```
@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.

```
cyclopts_app = cyclopts.App()
```

(continues on next page)

(continued from previous page)

```
@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
```

5.24.9 Help Defaults

In Typer's `--help` display, default values are unhelpfully shown for required arguments.

```
typer_app = typer.Typer()


@typer_app.command()
def compress(
```

(continues on next page)

(continued from previous page)

```
    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:

```
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:

```
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.
```

(continues on next page)

(continued from previous page)

```
    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]      │
# ───────────────────────────────────────────────────────────────────────────
```

5.24.10 Validation

Typer has builtin argument validation for certain type annotations.

```
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.

```
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,
```

(continues on next page)

(continued from previous page)

```
    # 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,
    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:

```
cyclopts_app = cyclopts.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
```

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))
```

5.24.11 Union/Optional Support

Currently, Typer does not support `Union` type annotations.

```
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.

```
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]`.

5.24.12 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:

```
typer_app = typer.Typer()


def version_callback(value: bool):
    if not value:
        return
    print("1.2.3")
    raise typer.Exit()

```

(continues on next page)

(continued from previous page)

```
@typer_app.callback()
def common(
    version: Annotated[
        bool,
        typer.Option(
            "--version",
            "-v",
            callback=version_callback,
            help="Print version.",
        ),
    ] = False,
):
    print("Callback body executed.")


print("Typer:")
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*.

```
cyclopts_app = cyclopts.App(version="1.2.3")
cyclopts_app(["--version"])
# 1.2.3
```

5.24.13 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.

5.25 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 it's 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.

```python
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:

```python
import cyclopts


app = cyclopts.App()


@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'>
```

(continues on next page)

(continued from previous page)

```
$ 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 and has some cool parlour tricks, but is not suitable for a serious CLI. Therefore, I wouldn't say Fire is a direct competitor to Cyclopts.

5.26 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.

5.26.1 Global State

Unlike Cyclopts or Typer, with `arguably` you directly jump into decorating functions:

```python
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:

```python
# library_using_arguably.py
import arguably


@arguably.command
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:

```python
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.

5.26.2 Subcommands

Arguably parses the command tree based on __ delimited function names.

```python
import arguably


@arguably.command
def ec2__start_instances(*instances):
    """Start instances.

    Args:
        *instances: {instance}s to start
    """
    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:

```python
from cyclopts import App

app = App()
app.command(ec2 := App(name="ec2"))


@ec2.command
def 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}")


@ec2.command
def stop_instances(*instances):
    """Stop instances.

    Args:
        *instances: {instance}s to stop
    """
    for inst in instances:
        print(f"Stopping {inst}")


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

```
$ python main.py ec2 --help
─ Commands ──────────────────────────────────────────────────────────────────
│ start-instances  start instances.                                        │
│ stop-instances   stop instances.                                         │
──────────────────────────────────────────────────────────────────────────────
```

INDEX

Symbols

A

C

D

E

F

G

H

I

L

M

N

P

R

S

T

U

V