Coma

Release 1.0.1

Francois Roewer-Despres

May 23, 2022

TUTORIALS

Configurable **co**mmand **ma**nagement for humans.

CHAPTER

ONE

INTRODUCTION

Commands (also known as sub-commands) are the `commit` and `pull` part of `git commit` and `git pull` (whereas `git` is the program). Commands can be seen as command-line meta-arguments: They drastically affect not only the behavior of the program, but also what additional command-line line arguments and flags are accepted.

`coma` goes one step further. With it, commands determine which configuration files are loaded, enabling command-specific configs that don't affect the whole program.

CHAPTER

TWO

KEY FEATURES

`coma` makes it easy to build configurable command-based programs in Python by:

- Removing the boilerplate of argparse while retaining full `argparse` interoperability and customizability for complex use cases.
- Providing a comprehensive set of hooks to easily tweak, replace, or extend `coma`.
- Integrating with omegaconf's extremely rich and powerful configuration management features.

CHAPTER

THREE

INSTALLATION

```
pip install coma
```

CHAPTER

FOUR

GETTING STARTED

Excited? Jump straight into the short *introductory tutorial*.

CHAPTER

FIVE

WHY COMA?

Why choose `coma` over another `omegaconf`-based solution, like hydra? `hydra`, specifically, has the following limitations (all of which are features that `coma` supports!):

- **No commands.** Related functionality must be implemented as separate programs.
- **No command-line arguments or flags.** All program data must be provided through configs.
- **No parallel/independent configs.** All configs must be hierarchical.

If these limitations aren't deal-breakers for you, then, by all means, use `hydra` (or any other framework). `hydra`, in particular, certainly has a wonderfully rich feature set (including config groups, which inspired its name).

5.1 Introduction

`coma` makes it easy to build configurable command-based programs in Python.

5.1.1 Commands

Let's dive in with a classic `Hello World!` program:

Listing 1: main.py

```python
import coma

if __name__ == "__main__":
    coma.register("greet", lambda: print("Hello World!"))
    coma.wake()
```

Now, let's run this program:

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

Note: The meat of working with `coma` is the *`register()`* function. It has two required parameters:

name the name of a command (`greet` in this example)

command the command itself (an anonymous function in this example)

Note: The *wake()* function should always follow the last call to *register()*. Calling *wake()* tells coma that all commands have been *register()*ed. Coma will then attempt to invoke whichever one was specified on the command line. In this example, greet was specified on the command line and so the *register()*ed command with that name was invoked.

In addition to anonymous functions, command can be any Python function:

```
import coma

def cmd():
    print("Hello World!")

if __name__ == "__main__":
    coma.register("greet", cmd)
    coma.wake()
```

or any Python class with a no-parameter run() method:

```
import coma

class Cmd:
    def run(self):
        print("Hello World!")

if __name__ == "__main__":
    coma.register("greet", Cmd)
    coma.wake()
```

5.1.2 Multiple Commands

coma is intended to manage multiple commands as part of building complex programs. Let's extend our previous example:

Listing 2: main.py

```
import coma

if __name__ == "__main__":
    coma.register("greet", lambda: print("Hello World!"))
    coma.register("leave", lambda: print("Goodbye!"))
    coma.wake()
```

This *register()*s two commands. By calling each in turn, we induce different program behavior:

```
$ python main.py greet
Hello World!
$ python main.py leave
Goodbye!
```

5.1.3 Configurations

Commands alone are great, but `omegaconf` integration is what makes `coma` truly powerful. The simplest way to create an `omegaconf` config object is with a plain dictionary:

Listing 3: main.py

```
import coma

if __name__ == "__main__":
    coma.register("greet", lambda cfg: print(cfg.message), {"message": "Hello World!"})
    coma.wake()
```

Note: The command now takes one positional argument (`cfg` in this example). It will be bound to the config object if the command is invoked.

Note: If the command is a Python class, it is the **constructor** that should have a positional config argument, not the `run()` method:

```
import coma

class Cmd:
    def __init__(self, cfg):
        self.cfg = cfg

    def run(self):
        print(self.cfg.message)

if __name__ == "__main__":
    coma.register("greet", Cmd, {"message": "Hello World!"})
    coma.wake()
```

This separation between initialization and execution is done so that stateful commands can be initialized based on config attributes, which is typically more straightforward than delaying part of the initialization until `run()` is called.

The program essentially runs as before:

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

The only difference is that, by default, `coma` serializes the config object to a YAML file in the current working directory:

```
$ ls
dict.yaml
main.py
```

By default, `coma` uses the config object's `type`'s name (`dict` in this example) to create an **identifier** for the config, and this identifier is then used derive a default file name. The default identifier can be overridden by explicitly identifying the config object using a keyword argument:

Listing 4: main.py

```
import coma

if __name__ == "__main__":
    coma.register("greet", lambda cfg: print(cfg.message),
                  greet={"message": "Hello World!"})
    coma.wake()
```

Now the config will be serialized to `greet.yaml`:

```
$ rm dict.yaml
$ python main.py greet
Hello World!
$ ls
greet.yaml
main.py
```

Config files can be used to hardcode attribute values that override the default config attribute values. For example, changing `greet.yaml` to:

Listing 5: greet.yaml

```
message: hardcoded message
```

leads to the following program execution:

```
$ python main.py greet
hardcoded message
```

Note: See *here* for full details on configuration files.

Config attribute values can also be overridden on the command line using `omegaconf`'s dot-list notation:

```
$ python main.py greet message="New Message"
New Message
```

Note: See *here* for full details on command line overrides.

Note: Serialized configs override default configs and command line-based configs override *both* serialized and default configs: `default < serialized < command line`.

`coma` supports any valid `omegaconf` config object. In particular, structured configs are useful for enabling runtime validation:

Listing 6: main.py

```
from dataclasses import dataclass
```

(continues on next page)

(continued from previous page)

```
import coma

@dataclass
class Config:
    message: str = "Hello World!"

if __name__ == "__main__":
    coma.register("greet", lambda cfg: print(cfg.message), Config)
    coma.wake()
```

Note: Because `Config` has `type` name `config`, it will be serialized to `config.yaml`.

5.1.4 Multiple Configurations

coma enables commands to take an arbitrary number of independent configs:

Listing 7: main.py

```
from dataclasses import dataclass

import coma

@dataclass
class Greeting:
    message: str = "Hello"

@dataclass
class Receiver:
    entity: str = "World!"

if __name__ == "__main__":
    coma.register("greet", lambda g, r: print(g.message, r.entity), Greeting, Receiver)
    coma.wake()
```

Note: In this example, the command now takes two positional arguments. Each will be bound (in the given order) to the supplied config objects if the command is invoked.

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

Multiple configs are often useful in practice to separate otherwise-large configs into smaller components, especially if some components are shared between commands:

Listing 8: main.py

```
from dataclasses import dataclass

import coma
```

(continues on next page)

(continued from previous page)

```
@dataclass
class Greeting:
    message: str = "Hello"

@dataclass
class Receiver:
    entity: str = "World!"

if __name__ == "__main__":
    coma.register("greet", lambda g, r: print(g.message, r.entity), Greeting, Receiver)
    coma.register("leave", lambda r: print("Goodbye", r.entity), Receiver)
    coma.wake()
```

Note: Configs need to be uniquely identified per-command, but not across commands, so it is perfectly acceptable for both `greet` and `leave` to share the `Receiver` config. To disable this sharing (so that each command has its own serialized copy of the config), use unique identifiers:

```
coma.register("greet", ..., Greeting, greet=Receiver)
coma.register("leave", ..., leave=Receiver)
```

We invoke both commands in turn as before:

```
$ python main.py greet
Hello World!
$ python main.py leave
Goodbye World!
```

5.1.5 Next Steps

You now have a solid foundation for writing Python programs with configurable commands!

For more advanced use cases, `coma` offers many additional features, including:

- Customizing the underlying `argparse` objects.
- Adding command line arguments and flags to your program.
- Registering global configurations that are applied to every command.
- Using hooks to tweak, replace, or extend `coma`'s default behavior.
- And more!

Check out the other tutorials to learn more.

5.2 Core UI

Four functions make up the core coma user interface:

5.2.1 Initiate

The first step to using coma is always to *initiate()* a new coma.

initiate() is always called exactly once, before any calls to *register()*. Calling *initiate()* explicitly is required only to initiate a coma with non-default parameters (coma implicitly calls *initiate()* with default parameters otherwise).

argparse Overrides

By default, coma creates an ArgumentParser with default parameters. However, *initiate()* can optionally accept a custom ArgumentParser:

Listing 9: main.py

```
import argparse

import coma

if __name__ == "__main__":
    coma.initiate(parser=argparse.ArgumentParser(description="My Program␣
↪description."))
    coma.register("greet", lambda: print("Hello World!"))
    coma.wake()
```

Now, let's run this program with the -h flag to see the result:

```
$ python main.py -h
usage: main.py [-h] {greet} ...

My Program description.

positional arguments:
  {greet}

optional arguments:
  -h, --help  show this help message and exit
```

You can also provide keyword arguments to override the default parameter values to the internal ArgumentParser.add_subparsers() call through the subparsers_kwargs parameter to *initiate()*:

```
coma.initiate(subparsers_kwargs=dict(help="sub-command help"))
```

Global Configs

Configs can be *initiate()*d globally to all commands or *register()*ed locally to a specific command.

Let's revisit the second of the *Multiple Configurations* examples from the *introductory tutorial* to see the difference:

Listing 10: main.py

```
from dataclasses import dataclass

import coma

@dataclass
class Greeting:
    message: str = "Hello"

@dataclass
class Receiver:
    entity: str = "World!"

if __name__ == "__main__":
    coma.register("greet", lambda g, r: print(g.message, r.entity), Greeting,␣
↪Receiver)
    coma.register("leave", lambda r: print("Goodbye", r.entity), Receiver)
    coma.wake()
```

Notice how, in the original example, the `Receiver` config is *register()*ed (locally) to both commands. Instead, we can *initiate()* a coma with this config so that it is (globally) supplied to all commands:

Listing 11: main.py

```
from dataclasses import dataclass

import coma

@dataclass
class Greeting:
    message: str = "Hello"

@dataclass
class Receiver:
    entity: str = "World!"

if __name__ == "__main__":
    coma.initiate(Receiver)
    coma.register("greet", lambda r, g: print(g.message, r.entity), Greeting)
    coma.register("leave", lambda r: print("Goodbye", r.entity))
    coma.wake()
```

This produces the same overall effect, while being more DRY.

Note: Configs need to be uniquely identified per-command, but not across commands.

Note: Each command parameter will be bound (in the given order) to the supplied config objects if the command is invoked. In this example, because `Receiver` is now supplied first instead of second to `greet`, the order of parameters to `greet` had to be swapped: `g, r` becomes `r, g`. See below for a nexample of how to prevent this.

Global Hooks

`coma`'s behavior can be easily tweaked, replaced, or extended using hooks. These are covered in great detail *in their own tutorial*. Here, the emphasis is on the difference between global and local hooks: As with configs, hooks can be *`initiate()`*d globally to affect `coma`'s behavior towards all commands or *`register()`*ed locally to only affect `coma`'s behavior towards a specific command.

Let's revisit the *previous example*. Recall that the order of parameters to `greet` had to be swapped: `g, r` became `r, g`. Suppose we want to prevent this change. To do so, we can force `coma` to bind configs to parameters differently by writing a custom `init_hook`:

Listing 12: main.py

```
from dataclasses import dataclass

import coma

@dataclass
class Greeting:
    message: str = "Hello"

@dataclass
class Receiver:
    entity: str = "World!"

@coma.hooks.hook
def custom_init_hook(command, configs):
    return command(*reversed(list(configs.values())))

if __name__ == "__main__":
    coma.initiate(Receiver, init_hook=custom_init_hook)
    coma.register("greet", lambda g, r: print(g.message, r.entity), Greeting)
    coma.register("leave", lambda r: print("Goodbye", r.entity))
    coma.wake()
```

The details of how the hook is defined aren't important for the moment. The point is that `coma`'s default behavior regarding config binding has been replaced from positional matching to anti-positional matching, which is sufficient in this simple example.

5.2.2 Register

The meat of working with `coma` is to *`register()`* new commands with an *`initiate()`*d coma.

The main use cases for *`register()`*, including command names, command objects, config objects, config identifiers, and *`register()`*ing multiple commands have all been covered in the *introductory tutorial*. Here, the emphasis is on local `argparse` overrides and local hooks as additional use cases.

`argparse` Overrides

By default, `coma` uses ArgumentParser.add_subparsers().add_parser() to create a new ArgumentParser with default parameters for each *`register()`*ed command. However, you can provide keyword arguments to override the default parameter values in the internal `add_parser()` call through the `parser_kwargs` parameter to *`register()`*.

For example, suppose you want to add command aliases. This can be achieved through the `aliases` keyword:

Listing 13: main.py

```
import coma

if __name__ == "__main__":
    coma.register("greet", lambda: print("Hello World!"),
                  parser_kwargs=dict(aliases=["gr"]))
    coma.wake()
```

With this alias, `greet` can now be invoked with just `gr`:

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

Local Hooks

`coma`'s behavior can be easily tweaked, replaced, or extended using hooks. These are covered in great detail *in their own tutorial*. Here, the emphasis is on the difference between global and local hooks: As with configs, hooks can be *`initiate()`*d globally to affect `coma`'s behavior towards all commands or *`register()`*ed locally to only affect `coma`'s behavior towards a specific command.

Let's see how a few local hooks can easily inject additional behavior into a program:

Listing 14: main.py

```
import coma

parser_hook = coma.hooks.parser_hook.factory("--dry-run", action="store_true")

@coma.hooks.hook
def pre_run_hook(known_args):
    if known_args.dry_run:
        print("Early exit!")
        quit()

if __name__ == "__main__":
```

(continues on next page)

(continued from previous page)

```
    coma.register("greet", lambda: print("Hello World!"),
                  parser_hook=parser_hook, pre_run_hook=pre_run_hook)
    coma.wake()
```

In this example, we locally *register()*ed a `parser_hook` that adds a new `--dry-run` flag to the command line as well as a `pre_run_hook` that exits the program early (before the command is actually executed) if the flag is given on the command line:

```
$ python main.py greet
Hello World!
$ python main.py greet --dry-run
Early exit!
```

Note: `coma` provides **factory functions** for some of the more common hooks. In this example, we used *coma.hooks.parser_hook.factory()*, which simply creates a function that in turn relays the provided parameters to the add_argument() method of the underlying ArgumentParser bound to this command.

Warning: Local hooks are **appended** to the list of global hooks. Local hooks **do not** override global hooks. To override a global hook, use *register()* in conjunction with *forget()*. See *here* for details.

5.2.3 Forget

forget() is a context manager designed for more advanced use cases. It enables you to selectively forget global configs or hooks from an *initiate()*d coma.

Forgetting Global Hooks

`coma`'s behavior can be easily tweaked, replaced, or extended using hooks. These are covered in great detail *in their own tutorial*. Here, the emphasis is on the difference between global and local hooks. Hooks can be *initiate()*d globally to affect `coma`'s behavior towards all commands or *register()*ed locally to only affect `coma`'s behavior towards a specific command.

Warning: Local hooks are **appended** to the list of global hooks. Local hooks **do not** override global hooks. When a local hook is being *register()*ed, the corresponding global hook can only be replaced using *forget()*.

For example, suppose we have a class-based command with a `handle()` method instead of the `run()` method that `coma` expects by default:

```
class HandleCommand:
    def handle(self):
        print("Hello Handle!")
```

In order to use this command, we need to tell `coma` to:

- Stop looking for `run()`. We will *forget()* the existing global hook that does this.

- Start looking for `handle()` instead. We will *`register()`* a new local hook to do this.

Listing 15: main.py

```
import coma

class HandleCommand:
    def handle(self):
        print("Hello Handle!")

class RunCommand:
    def run(self):
        print("Hello Run!")

if __name__ == "__main__":
    with coma.forget(run_hook=True):
        coma.register("handle", HandleCommand,
                      run_hook=coma.hooks.run_hook.factory("handle"))
    coma.register("run", RunCommand)
    coma.wake()
```

In this example, we locally *`register()`*ed a `run_hook` that tells `coma` to call `handle()` and we used the *`forget()`* context manager to get `coma` to temporarily forget its default `run_hook`, which attempts to call `run()` instead.

Note: `coma` provides **factory functions** for some of the more common hooks. In this example, we used *`coma.hooks.run_hook.factory()`*, which simply creates a function that in turn calls the provided attribute (in this case, `"handle"`) of the command object.

Because *`forget()`* is a context manager, any commands registered outside its context are unaffected. In this example, `RunCommand` still functions normally.

```
$ python main.py handle
Hello Handle!
$ python main.py run
Hello Run!
```

Forgetting Global Configs

As with hooks, configs can be *`initiate()`*d globally to all commands or *`register()`*ed locally to a specific command.

Let's revisit the second of the *Multiple Configurations* examples from the *introductory tutorial* to see how we can implement it differently with *`forget()`*:

Listing 16: main.py

```
from dataclasses import dataclass

import coma

@dataclass
```

(continues on next page)

(continued from previous page)

```
class Greeting:
    message: str = "Hello"

@dataclass
class Receiver:
    entity: str = "World!"

if __name__ == "__main__":
    coma.register("greet", lambda g, r: print(g.message, r.entity), Greeting,␣
↪Receiver)
    coma.register("leave", lambda r: print("Goodbye", r.entity), Receiver)
    coma.wake()
```

Notice how, in the original example, the `Receiver` config is *`register()`*ed (locally) to both commands. Instead, we can *`initiate()`* a coma with both configs so that they are (globally) supplied to all commands, then *`forget()`* the `Greeting` config just for the `leave` command:

Listing 17: main.py

```
from dataclasses import dataclass

import coma

@dataclass
class Greeting:
    message: str = "Hello"

@dataclass
class Receiver:
    entity: str = "World!"

if __name__ == "__main__":
    coma.initiate(Greeting, Receiver)
    coma.register("greet", lambda g, r: print(g.message, r.entity))
    with coma.forget("greeting"):
        coma.register("leave", lambda r: print("Goodbye", r.entity))
    coma.wake()
```

Notice that *`forget()`* takes the config identifier (in this case, we used the default identifier, which is `"greeting"`), not the config itself.

Note: Configs need to be uniquely identified per-command, but not across commands.

5.2.4 Wake

The main use case for *wake()* has been covered in the *introductory tutorial*. Specifically, after all commands have been *register()*ed, *wake()* is used to wake from a coma.

The only additional use case is simulating command line arguments using `args` and `namespace`, which are simply passed to ArgumentParser.parse_known_args():

Listing 18: main.py

```
import coma

if __name__ == "__main__":
    coma.register("greet", lambda: print("Hello World!"))
    coma.wake(["greet"])
```

Running this program without providing command line arguments works because *wake()* is simulating `greet` as a command line argument:

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

5.3 Hooks

Hooks make it easy to tweak, replace, or extend `coma`.

5.3.1 Introduction

Hooks make it easy to tweak, replace, or extend `coma`.

Types of Hooks

At the highest level, hooks belong to one of two types:

parser Parser hooks are called when a command is *register()*ed and are meant to add_arguments() to the underlying ArgumentParser bound to the command.

invocation Invocation hooks are called by *wake()* if, and only if, the command to which they are globally *initiate()*d or locally *register()*ed is invoked.

Invocation hooks further belong to one of three sub-types:

config A config hook is meant to initialize or affect the config objects that are globally *initiate()*d or locally *register()*ed to a command.

init An init hook is meant to instantiate or affect the command itself.

> **Warning:** For function-based commands, the function is internally wrapped in another object, and it is this wrapper object that an init hook receives.

run A run hook is meant to execute or surround the execution of the command.

For all three hook sub-types above (**config**, **init**, and **run**), each is further split into three sub-sub-types:

pre A pre hook is called immediately *before* a **main hook** as a way to add additional behavior.

main A main hook is generally meant to perform the bulk of the work.

post A post hook is called immediately *after* a **main hook** as a way to add additional behavior.

Altogether, there are 9 **invocation hooks**.

Hook Pipeline

As stated above, **parser hooks** are called at the time a command is *register()*ed, whereas the **invocation hooks** are called if, and only if, the corresponding command is invoked.

The following keywords are used to *initiate()*, *register()*, and/or *forget()* hooks:

Type	Sub-Type	Sub-Sub-Type	Keyword
parser	N/A	N/A	`parser_hook`
invocation	config	pre	`pre_config_hook`
		main	`config_hook`
		post	`post_config_hook`
	init	pre	`pre_init_hook`
		main	`init_hook`
		post	`post_init_hook`
	run	pre	`pre_run_hook`
		main	`run_hook`
		post	`post_run_hook`

The **invocation hook pipeline** consists of calling all the **invocation hooks**, in the order listed here, one immediately following the other, with no other code in between. In other words, the invocation hooks make up the entirety of the hook pipeline.

Default Hook Pipeline

Rather than being hard-coded, `coma`'s default behavior is, almost entirely, a result of having certain specific default hooks *initiate()*d. The upshot is that there is almost no part of `coma`'s default behavior that cannot be tweaked, replaced, or extended through clever use of hooks.

The default hooks are:

parser The default `parser_hook` is *coma.hooks.parser_hook.default()*. This hook uses add_argument() to add, for each config, a parser argument of the form `--{config_id}-path` where `{config_id}` is the config's identifier. This enables an explicit file path to the serialized config to be specified on the command line.

pre config N/A

main config The default `config_hook` is *coma.hooks.config_hook.default()*. This hook does a lot of the heaving lifting for manifesting `coma`'s default behavior regarding configs. In short, for each config, this hook:

- Attempts to load the config from file. This can interact with the default `parser_hook`.
- If the config file isn't found, a config object with default attribute values is instantiated, and the default config object is serialized.

Note: YAML is used for serialization by default (since it is the only format that `omegaconf` supports), but `coma` also natively supports JSON. See *here* for full details on configuration files.

post config The default `post_config_hook` is *`coma.hooks.post_config_hook.default()`*. This hook is responsible for overriding config attribute values with any that are specified on the command line in `omegaconf`'s dot-list notation. See *here* for full details on command line overrides.

pre init N/A

main init The default `init_hook` is *`coma.hooks.init_hook.default()`*. This hook instantiates the command object by invoking it with all configs given, in order, as positional arguments.

post init N/A

pre run N/A

main run The default `run_hook` is *`coma.hooks.run_hook.default()`*. This hook calls the command object's `run()` method with no parameters.

post run N/A

Note: For each of the default hooks, **factory functions** are provided that can create new variations on these defaults. For example, *`coma.hooks.run_hook.factory()`* can be used to change the command execution method name from `run()` to something else. See *here* to explore all factory options.

Note: If you are finding that the factory functions for the **parser hook**, **main config hook**, and/or **post config hook** are insufficient, consider making use of the many config-related utilities found *here* to help you in writing your own custom hooks.

Global and Local Hooks

Hooks can be *`initiate()`*d globally to affect `coma`'s behavior towards all commands or *`register()`*ed locally to only affect `coma`'s behavior towards a specific command.

Warning: Local hooks are **appended** to the list of global hooks. Local hooks **do not** override global hooks. To override a global hook, use *`register()`* in conjunction with *`forget()`*. See *here* for details.

5.3.2 Protocols

`coma` uses a *hook pipeline*, both to implement its default behavior and to enable customization. To make this work, the various *types of hook* must each follow a pre-defined protocol (i.e., function signature) for both their parameters and their return value.

The hook protocols are fairly similar across all hook types, but there are a number of variations depending on the type. We begin by enumerating the shared aspects of the various protocols.

Function Signature

All protocols require the corresponding hook function signatures to define positional-or-keyword parameters. For example, we can define and invoke some (hypothetical) hook function as:

```
def some_hook(a, b, c):
    ...

some_hook(a=..., b=..., c=...)
```

This is fine because the hook function parameters are defined as positional-or-keyword. These requirement is needed because hook parameters are bound positionally by keyword.

Note: Technically, the requirement is that the hook function parameters are not positional-only, not variadic (positional or keyword), and not keyword-only. In practice, that means they have to be positional-or-keyword.

In addition to the above, **all** protocol parameters must be:

- **Defined** in the hook function's signature.
- **Ordered** correctly in the hook function's signature.
- **Named** (i.e., spelled) correctly in the hook function's signature.

Protocol Parameters

generic_protocol(*name*, *parser*, *known_args*, *unknown_args*, *command*, *configs*, *result*)

Here, we list all possible protocol parameters, in the order in which they should be defined in the hook function's signature.

Note: Not every type of hook uses every protocol parameter. The protocols specific to each hook type are listed *below*. **None of these specific differences affect the parameter ordering or naming shown here.**

Parameters

- **name** (*str*) – The name given to a command when it is *register()*ed.
- **parser** (*argparse.ArgumentParser*) – The `ArgumentParser` created to add command line arguments for a specific command when it is *register()*ed and subsequently parse actual command line arguments if the command is invoked on the command line.
- **known_args** (*argparse.Namespace*) – The `Namespace` object (i.e., first object) returned by parse_known_args() if a specific command is invoked on the command line.
- **unknown_args** (*List[str]*) – The `list` object (i.e., second object) returned by parse_known_args() if a specific command is invoked on the command line.
- **command** (*Union[Callable, Any]*) – The command object itself that was *register()*ed if it is invoked on the command line.

  Note: If the *register()*ed object was a class, it is left unchanged. If the *register()*ed object was a function, it is implicitly wrapped in another object. The *main init hook* is meant to instantiate `command`.

> **Warning:** Never make decisions based on the `type` of `command`, since it may be implicitly wrapped. Instead, use `name`, which is guaranteed to be unique across all *`register()`*ed commands.

- **`configs`** (*`Dict[str, Any]`*) – A dictionary of identifier-configuration pairs representing all configs (both global and local) bound to a specific command if it is invoked on the command line.

  > **Note:** Before the *main config hook*, the values in the `configs` dictionary are assumed to be uninitialized config objects. Afterwards, they are assumed to be initialized config objects.

- **`result`** (*`Any`*) – The value returned from executing the command if it is invoked on the command line.

Returns Some protocols return values; others do not. See *below* for details on each protocol.

Return type *Any*

`@hook` Decorator

For many hooks, only a subset of the corresponding protocol parameters are needed to implement their logic. It can therefore be cumbersome to define a function with multiple unused parameters just to satisfy the hook protocol. The `@hook` decorator (link: *`hook()`*) solves this problem, as it allows hook functions to be defined with a subset of the protocol parameters. For example:

```
@coma.hooks.hook
def name_hook(name):
    ...
```

defines a hook that only requires the command's `name` and ignores all other protocol parameters.

> **Note:** The `@hook` decorator only alleviates the requirement that all protocol parameters are defined in the hook function's signature. Other requirements, such as having the correct ordering and spelling of parameters, remain active.

`sequence()` Function

Each *type of hook* must be implemented as a single function. However, it is often beneficial to decompose a large hook function into a series of smaller ones. These component functions must then be wrapped with a higher-order function that executes them in order, while binding all parameters using keywords.

While this wrapping can always be done manually, a convenience wrapper, *`sequence()`*, can be used when all hooks share the exact same function signature (or are wrapped in the `@hook` decorator) to abstract away some of the minutiae. Compare:

```
wrapper = coma.hooks.sequence(
    coma.hooks.parser_hook.factory("-a", type=int, default=123),
    coma.hooks.parser_hook.factory("-b", type=int, default=456),
)

coma.register(..., parser_hook=wrapper)
```

with:

```
@coma.hooks.hook
def wrapper(parser):
    coma.hooks.parser_hook.factory("-a", type=int, default=123)(parser=parser)
    coma.hooks.parser_hook.factory("-b", type=int, default=456)(parser=parser)

coma.register(..., parser_hook=wrapper)
```

The former isn't shorter, but it removes the minutiae of adding `(parser=parser)` to each wrapped hook function and removes the need to decorate the wrapper function with the `@hook` decorator.

Specific Protocols

Here, we list the specific protocol and intended semantics for each *type of hook*. See *Protocol Parameters* for details on each parameter.

Parser

parser_hook_protocol(*name*, *parser*, *command*, *configs*)

Semantics This protocol adds command line arguments using `parser`.

Returns The return value of a parser hook (if any) is always ignored.

Return type None

Pre Config

pre_config_hook_protocol(*name*, *known_args*, *unknown_args*, *command*, *configs*)

Semantics This protocol is the first invocation hook to be executed.

Returns The return value of a pre config hook (if any) is always ignored.

Return type None

Config

config_hook_protocol(*name*, *known_args*, *unknown_args*, *command*, *configs*)

Semantics The values in the `configs` dictionary represent uninitialized config objects. This protocol initializes them and returns them **in the same order**.

Returns The return value of a config hook is an initialized configs dictionary.

Return type *Dict*[str, *Any*]

Post Config

`post_config_hook_protocol`(*name*, *known_args*, *unknown_args*, *command*, *configs*)

Semantics This protocol takes the initialized configs objects and returns these same objects (possibly modified in some way) **in the same order**.

Returns The return value of post config hooks is the configs dictionary.

Return type *Dict*[str, *Any*]

Pre Init

`pre_init_hook_protocol`(*name*, *known_args*, *unknown_args*, *command*, *configs*)

Semantics This protocol's hook is executed after all the config hooks and before the main init hook.

Returns The return value of a pre init hook (if any) is always ignored.

Return type None

Init

`init_hook_protocol`(*name*, *known_args*, *unknown_args*, *command*, *configs*)

Semantics This protocol instantiates `command` using the `configs`, returning the resulting instance object.

Note: If the *`register()`*ed command object was a class, it was left unchanged. If the *`register()`*ed command object was a function, it was implicitly wrapped in another object. Either way, `command` acts as though it is a class object that can be instantiated.

Returns The return value of an init hook is an instantiated command object.

Return type *Any*

Post Init

`post_init_hook_protocol`(*name*, *known_args*, *unknown_args*, *command*, *configs*)

Semantics This protocol takes the instantiated command object and returns the same object (possibly modified in some way).

Returns The return value of a post init hook is the instantiated command object.

Return type *Any*

Pre Run

pre_run_hook_protocol(*name*, *known_args*, *unknown_args*, *command*, *configs*)

Semantics This protocol's hook is executed after all the config and init hooks and before the main run hook.

Returns The return value of a pre run hook (if any) is always ignored.

Return type None

Run

run_hook_protocol(*name*, *known_args*, *unknown_args*, *command*, *configs*)

Semantics This protocol executes the instantiated `command` object, then returns the resulting value.

Returns The return value of a run hook is the value resulting from executing the instantiated command object.

Return type *Any*

Post Run

post_run_hook_protocol(*name*, *known_args*, *unknown_args*, *command*, *configs*, *result*)

Semantics This protocol is the last invocation hook to be executed.

Returns The return value of a post run hook (if any) is always ignored.

Return type None

5.4 Examples

Examples of `coma` use cases that aren't covered elsewhere:

5.4.1 Command Line Arguments

Program-Level Arguments

Using program-level command line arguments is as an easy way to inject additional behavior into a program. This example is similar to the one seen *here*. The main difference is using global hooks instead of local hooks to avoid repeating the hook registration for the new `leave` command:

Listing 19: main.py

```
import coma

parser_hook = coma.hooks.parser_hook.factory("--dry-run", action="store_true")

@coma.hooks.hook
def pre_run_hook(known_args):
```

(continues on next page)

(continued from previous page)

```
    if known_args.dry_run:
        print("Early exit!")
        quit()

if __name__ == "__main__":
    coma.initiate(parser_hook=parser_hook, pre_run_hook=pre_run_hook)
    coma.register("greet", lambda: print("Hello World!"))
    coma.register("leave", lambda: print("Goodbye World!"))
    coma.wake()
```

In this example, the `parser_hook` adds a new `--dry-run` flag to the command line. This flag is used by the `pre_run_hook` to exit the program early (before a command is actually executed) if the flag is given on the command line. Because these are global hooks, this behavior is present regardless of the command that is invoked:

```
$ python main.py greet
Hello World!
$ python main.py leave
Goodbye World!
$ python main.py greet --dry-run
Early exit!
$ python main.py leave --dry-run
Early exit!
```

Command-Level Arguments

Using command-level command line arguments is as an easy way to give a command additional data or modifiers that, for whatever reason, don't belong in a dedicated config object:

Listing 20: main.py

```
import coma

parser_hook = coma.hooks.sequence(
    coma.hooks.parser_hook.factory("a", type=int),
    coma.hooks.parser_hook.factory("-b", default=coma.SENTINEL),
)

@coma.hooks.hook
def init_hook(known_args, command):
    if known_args.b is coma.SENTINEL:
        return command(known_args.a)
    else:
        return command(known_args.a, known_args.b)

if __name__ == "__main__":
    with coma.forget(init_hook=True):
        coma.register("numbers", lambda a, b=456: print(a, b),
                      parser_hook=parser_hook, init_hook=init_hook)
    coma.register("greet", lambda: print("Hello World!"))
    coma.wake()
```

Here, `greet` acts in accordance with `coma`'s default behaviour, whereas `numbers` is defined quite differently. First, we define a *`sequence()`* for the `parser_hook` made up of *`factory()`* calls, each of which simply passes its arguments to the underlying parser object. Next, we define a custom `init_hook` that is aware of how to instantiate this non-standard command object. Finally, we *`forget()`* the default `init_hook`, which doesn't know how to handle non-standard commands.

With these definitions, we can invoke the program's commands as follows:

```
$ python main.py greet
Hello World!
$ python main.py numbers 123
123 456
$ python main.py numbers 123 -b 321
123 321
```

Using `coma.SENTINEL`

In the *previous example*, we used `coma`'s convenience sentinel object, *`coma.SENTINEL`*. Another way to implement the same functionality would be:

Listing 21: main.py

```
import coma

parser_hook = coma.hooks.sequence(
    coma.hooks.parser_hook.factory("a", type=int),
    coma.hooks.parser_hook.factory("-b", default=456),
)

@coma.hooks.hook
def init_hook(known_args, command):
    return command(known_args.a, known_args.b)

if __name__ == "__main__":
    with coma.forget(init_hook=True):
        coma.register("numbers", lambda a, b=456: print(a, b),
                      parser_hook=parser_hook, init_hook=init_hook)
    coma.register("greet", lambda: print("Hello World!"))
    coma.wake()
```

In terms of final program behavior, these two versions of the program are essentially identical, yet the version without the sentinel is shorter. The tradeoff is that the sentinel allows the default value of `b` to be defined only once, rather than twice, which can be less error-prone.

Note: It would also be possible to define the default value of `b` only once (in the `parser_hook`):

```
coma.hooks.parser_hook.factory("-b", default=456)
...
coma.register(..., lambda a, b: print(a, b), ...)
```

The leads to another tradeoff: The full command definition is now spread out in the code, which can obscure the fact that `b` has a default value.

On-the-Fly Hook Redefinition

Command line arguments can also be used to redefine hooks on the fly. In this example, we have two configs, both of which define the same `x` attribute. We then define a new `-e` flag, which is used to toggle the `exclusive` parameter of *override_factory()*. In short, the presence of this flag prevents any command line override involving `x` from overriding more than one config attribute:

Listing 22: main.py

```
from dataclasses import dataclass

import coma

@dataclass
class Config1:
    x: int

@dataclass
class Config2:
    x: int

excl = coma.hooks.parser_hook.factory("-e", dest="excl", action="store_true")

@coma.hooks.hook
def post_config_hook(known_args, unknown_args, configs):
    override = coma.config.cli.override_factory(exclusive=known_args.excl)
    multi_cli = coma.hooks.post_config_hook.multi_cli_override_
↪factory(override)
    return multi_cli(unknown_args=unknown_args, configs=configs)

if __name__ == "__main__":
    coma.initiate(Config1, Config2, post_config_hook=post_config_hook)
    coma.register("multiply", lambda c1, c2: print(c1.x * c2.x), parser_
↪hook=excl)
    coma.wake()
```

Without the `-e` flag, we can use `x` on the command line to override *both* configs at once:

```
$ python main.py multiply x=3
9
```

This lets `multiply` is essentially act as `square`. We can prevent this by setting the `-e` flag:

```
$ python main.py multiply x=3
...
ValueError: Non-exclusive override: override: x=3 ; matched configs (possibly␣
↪others too): ['config1', 'config2']
```

Note: See *here* for additional details on this example.

5.4.2 Command Line Config Overrides

Prefixing Overrides

Command line config overrides can sometimes clash. In this example, we have two configs, both of which define the same `x` attribute:

Listing 23: main.py

```
from dataclasses import dataclass

import coma

@dataclass
class Config1:
    x: int

@dataclass
class Config2:
    x: int

if __name__ == "__main__":
    coma.register("multiply", lambda c1, c2: print(c1.x * c2.x), Config1,␣
↪Config2)
    coma.wake()
```

By default, `coma` enables the presence of `x` on the command line to override *both* configs at once:

```
$ python main.py multiply x=3
9
```

This lets `multiply` is essentially act as `square`. To prevent this, we can override a specific config by *prefixing the override* with its identifier:

```
$ python main.py multiply config1:x=3 config2:x=4
12
```

Note: See *here* for an alternative way to prevent these clashes.

By default, `coma` also supports prefix abbreviations: A prefix can be abbreviated as long as the abbreviation is unambiguous (i.e., matches only one config identifier):

Listing 24: main.py

```
from dataclasses import dataclass

import coma

@dataclass
class Config1:
    x: int

@dataclass
```

(continues on next page)

(continued from previous page)

```
class Config2:
    x: int

if __name__ == "__main__":
    coma.register("multiply", lambda c1, c2: print(c1.x * c2.x),
                  some_long_identifier=Config1, another_long_
↪identifier=Config2)
    coma.wake()
```

This is enables convenient shorthands for command line overrides:

```
$ python main.py multiply some_long_identifier:x=3 another_long_identifier:x=4
12
$ python main.py multiply s:x=3 a:x=4
12
```

Capturing Superfluous Overrides

For rapid prototyping, it is often beneficial to capture superfluous command line overrides. These can then be transferred to a proper config object once the codebase is solidifying. In this example, we name this superfluous config `extras`:

Listing 25: main.py

```
import coma


if __name__ == "__main__":
    coma.initiate(
        extras={},
        init_hook=coma.hooks.init_hook.positional_factory("extras"),
        post_run_hook=coma.hooks.hook(
            lambda configs: print("extras =", configs["extras"])
        ),
    )
    coma.register("greet", lambda: print("Hello World!"))
    coma.wake()
```

This works because, as a plain `dict`, `extras` will accept any *non-prefixed* arguments given on the command line:

```
$ python main.py greet
Hello World!
extras = {}
$ python main.py greet foo=1 bar=baz
Hello World!
extras = {'foo': 1, 'bar': 'baz'}
```

Note: We redefined the `init_hook` using *`positional_factory()`*. This factory *skips* the given config identifiers when instantiating the command. Without this hook redefinition, the `lambda` defining the

command would need to accept 1 positional argument to accommodate `extras`.

Note: We added a new `post_run_hook`. This hook is simply added to print out the attributes of the `extras` config after the command is executed.

5.4.3 Config Serialization

By default, `coma` favors YAML over JSON for its config serialization, since omegaconf only supports YAML. However, `coma` does natively support JSON as well.

Default Behavior

To illustrate the default behavior, let's revisit an example from the *introductory tutorial*:

Listing 26: main.py

```
from dataclasses import dataclass

import coma

@dataclass
class Config:
    message: str = "Hello World!"

if __name__ == "__main__":
    coma.register("greet", lambda cfg: print(cfg.message), Config)
    coma.wake()
```

Because `Config` has `type` name `config`, it will be serialized to `config.yaml` by default:

```
$ python main.py greet
Hello World!
$ cat config.yaml
message: Hello World!
```

We can force `coma` to serialize to JSON by specifying an explicit file path for `Config`:

```
$ python main.py greet --config-path config.json
Hello World!
$ cat config.json
{
    "message": "Hello World!"
}
```

Note: By default, `coma` automatically adds the `--config-path` flag through the default `parser_hook` of *`initiate()`*. Specifically, a flag of the form `--{config_id}-path` is added for each global and local config, where `{config_id}` is the corresponding config identifier.

Now we have two competing config files. Let's modify each one to distinguish them:

Listing 27: config.yaml

```yaml
message: Hello YAML!
```

Listing 28: config.json

```json
{
    "message": "Hello JSON!"
}
```

Now, if we run the program, we see that YAML is favored:

```
$ python main.py greet
Hello YAML!
```

But we can still force `coma` to use JSON instead:

```
$ python main.py greet --config-path config.json
Hello JSON!
```

If we specify a file path without an extension, `coma` will again favor YAML:

```
$ python main.py greet --config-path config
Hello YAML!
```

Finally, if we delete the YAML file while keeping the JSON file, `coma` will *ignore the existing JSON file* (and create a new YAML file instead) unless explicitly given a JSON file extension:

```
$ rm config.yaml
$ python main.py greet --config-path config
Hello World!
$ python main.py greet --config-path config.json
Hello JSON!
```

In summary, by default `coma` natively *supports* JSON, but YAML always takes *precedence*.

Favoring JSON

We can reverse `coma`'s default preference by setting JSON as the default file extension through the `config_hook` of *`initiate()`*:

Listing 29: main.py

```python
from dataclasses import dataclass

import coma

@dataclass
class Config:
    message: str = "Hello World!"

if __name__ == "__main__":
    coma.initiate(
```

(continues on next page)

(continued from previous page)

```
        config_hook=coma.hooks.config_hook.multi_load_and_write_factory(
            default_ext=coma.config.io.Extension.JSON
        )
    )
    coma.register("greet", lambda cfg: print(cfg.message), Config)
    coma.wake()
```

First, let's ensure that both YAML and JSON config files exist and are differentiated:

Listing 30: config.yaml

```
message: Hello YAML!
```

Listing 31: config.json

```
{
    "message": "Hello JSON!"
}
```

Now, when running the program, we see that JSON is favored in all cases, unless a YAML file extension is explicitly provided:

```
$ python main.py greet
Hello JSON!
$ python main.py greet --config-path config
Hello JSON!
$ python main.py greet --config-path config.json
Hello JSON!
$ python main.py greet --config-path config.yaml
Hello YAML!
```

5.4.4 Fitting coma to Existing Code

In general, there are at least four ways to modify `coma` to fit the interface of an existing codebase. We highlight these options using the following example:

```
class StartCommand:
    def __init__(self):
        self.foo = "bar"

    def start(self):
        print(f"foo = {self.foo}")
```

In this example, we suppose that an existing command-like class has a `start()` method instead of the default `run()` method.

Redefining Hooks

The first option is redefining the `run_hook` of *`initiate()`* to call `start()` instead of `run()`:

Listing 32: main.py

```
import coma

class StartCommand:
    def __init__(self):
        self.foo = "bar"

    def start(self):
        print(f"foo = {self.foo}")

if __name__ == "__main__":
    coma.initiate(run_hook=coma.hooks.run_hook.factory("start"))
    coma.register("start", StartCommand)
    coma.wake()
```

The program now runs as expected:

```
$ python main.py start
foo = bar
```

Warning: Internally, **function-based** commands will still be wrapped in a class that defines a `run()` method, regardless of any `run_hook` redefinition. As such, it is generally safer, if more verbose, to locally redefine the `run_hook` using *`register()`* and a *`forget()`* context manager:

Listing 33: main.py

```
import coma

class StartCommand:
    def __init__(self):
        self.foo = "bar"

    def start(self):
        print(f"foo = {self.foo}")

if __name__ == "__main__":
    with coma.forget(run_hook=True):
        coma.register("start", StartCommand,
                      run_hook=coma.hooks.run_hook.factory("start"))
    coma.wake()
```

This ensures that other commands are not affected. See *here* for details on using *`forget()`*.

Wrapping with Functions

The second option is wrapping `StartCommand` in a function-based command:

Listing 34: main.py

```
import coma

class StartCommand:
    def __init__(self):
        self.foo = "bar"

    def start(self):
        print(f"foo = {self.foo}")

if __name__ == "__main__":
    coma.register("start", lambda: StartCommand().start())
    coma.wake()
```

The benefit of this approach is in its simplicity. The drawback is the loss of separation between command initialization and execution.

Wrapping with Classes

The third option is wrapping the incompatible `StartCommand` in a compatible class-based command:

Listing 35: main.py

```
import coma

class StartCommand:
    def __init__(self):
        self.foo = "bar"

    def start(self):
        print(f"foo = {self.foo}")

class WrapperCommand(StartCommand):
    def run(self):
        self.start()

if __name__ == "__main__":
    coma.register("start", WrapperCommand)
    coma.wake()
```

The benefit of this approach is that it maintains the separation between command initialization and execution. The drawback is that it is slightly more verbose than the function-based wrapper.

Adding Interface Elements

The fourth option is adding missing interface elements (in this case, an attribute) to `StartCommand`:

Listing 36: main.py

```
import coma

class StartCommand:
    def __init__(self):
        self.foo = "bar"

    def start(self):
        print(f"foo = {self.foo}")

if __name__ == "__main__":
    StartCommand.run = StartCommand.start
    coma.register("start", StartCommand)
    coma.wake()
```

For simple cases, this option is often the most succinct.

5.5 coma

Configurable command management for humans.

5.5.1 coma.hooks

Hook utilities, factories, and defaults.

coma.hooks.parser_hook

Parser hook utilities, factories, and defaults.

factory(**names_or_flags*, ***kwargs*) → Callable[[...], None]

Factory for creating a parser hook that adds an `argparse` argument.

Creates a parser hook that add an argument to the `ArgumentParser` of the hook protocol.

Example:

```
coma.initiate(..., parser_hook=factory('-l', '--lines', type=int))
```

Parameters

- ***names_or_flags** – Passed to add_argument()
- ****kwargs** – Passed to add_argument()

Returns A parser hook

single_config_factory(*config_id: str*, **names_or_flags*, ***kwargs*) → Callable[[...], None]

Factory for creating a parser hook that adds a single config file path argument.

If no arguments are provided, the following defaults are used for add_argument():

```
from coma.config import default_default, default_flag, default_help
names_or_flags = [default_flag(config_id)]
kwargs = {
    "type": str,
    "metavar": "FILE",
    "dest": default_dest(config_id)
    "default": default_default(config_id),
    "help": default_help(config_id),
}
```

Any of these defaults can be overridden by providing alternative arguments. Additional arguments beyond these can also be provided.

Example:

```
@dataclass
class Config:
    ...

cfg_id = default_id(Config)
parser_hook = single_config_factory(cfg_id, metavar=cfg_id.upper())
coma.register(..., parser_hook=parser_hook)
```

Parameters

- **config_id** (*str*) – A config identifier
- ***names_or_flags** – Passed to add_argument()
- ****kwargs** – Passed to add_argument()

Returns A parser hook

See also:

- *single_load_and_write_factory()*

multi_config(*parser: argparse.ArgumentParser*, *configs: Dict[str, Any]*) → None

Parser hook for adding all config file path arguments.

Equivalent to calling *single_config_factory()* for each config in `configs`.

Automatically adds file path arguments for all `configs` using `parser`.

Example:

```
@dataclass
class Config:
    ...

coma.initiate(..., parser_hook=multi_config)
```

Parameters

- **parser** – The parser parameter of the parser hook protocol
- **configs** – The configs parameter of the parser hook protocol

See also:

- *single_config_factory()*

default(*parser: argparse.ArgumentParser*, *configs: Dict[str, Any]*) → None

Default parser hook.

An alias for *multi_config()*.

coma.hooks.config_hook

Config hook utilities, factories, and defaults.

single_load_and_write_factory(*config_id: str*, ***, *parser_attr_name: Optional[str] = None*, *default_file_path: Optional[str] = None*, *default_ext: coma.config.io.Extension = Extension.YAML*, *raise_on_fnf: bool = False*, *write_on_fnf: bool = True*, *resolve: bool = False*) → Callable[[...], Dict[str, Any]]

Factory for creating a config hook that initializes a config object.

The created config hook has the following behaviour:

First, an attempt is made to load the config object corresponding to `config_id` from file.

Note: If a file path is provided as a command line argument (assuming the presence of *multi_config()* or equivalent functionality), that path is used. Otherwise, `default_file_path` is used as a default. If `default_file_path` is `None`, a sensible default is derived from `config_id` instead.

In any case, if the provided or derived file path has no file extension, `default_ext` is used as a default extension.

If loading the file fails due to a `FileNotFoundError`, then:

If `raise_on_fnf` is `True`, the error is re-raised.

If `raise_on_fnf` is `False`, a config object with default values is initializes, and then:

If `write_on_fnf` is `True`, the newly-initialized config object with default values is written to the file.

If `resolve` is `True`, the underlying `omegaconf` handler attempts to resolve variable interpolation before writing.

The created config hook raises:

KeyError If `config_id` does not match any known config identifier

ValueError If the file extension is not supported. See *Extension* for supported types.

FileNotFoundError If `raise_on_fnf` is `True` and the config file was not found

Others As may be raised by the underlying `omegaconf` handler

Example

Fail fast when encountering a `FileNotFoundError`:

```
coma.initiate(..., config_hook=single_factory(..., raise_on_fnf=True))
```

Parameters

- **config_id** (*str*) – A config identifier
- **parser_attr_name** (*str*) – The `known_args` attribute representing this config's file path parser argument. If `None`, a sensible default is derived from *default_dest()*.
- **default_file_path** (*str*) – An optional default value for the config file path. If `None`, a sensible default is derived from *default_default()*.
- **default_ext** (`coma.config.io.Extension`) – The extension to use when the provided file path lacks one
- **raise_on_fnf** (*bool*) – If `True`, raises a `FileNotFoundError` if the config file was not found. If `False`, a config object with default values is initialized instead of failing outright.
- **write_on_fnf** (*bool*) – If the config file was not found and `raise_on_fnf` is `False`, then `write_on_fnf` indicates whether to write the config object to the provided file
- **resolve** (*bool*) – If about to write a config object to file, then `resolve` indicates whether the underlying `omegaconf` handler attempts to resolve variable interpolation beforehand

Returns A config hook

See also:

- *default_dest()*
- *default_default()*
- *single_config_factory()*

multi_load_and_write_factory(*, *default_ext:* coma.config.io.Extension = *Extension.YAML*, *raise_on_fnf:* *bool* = *False*, *write_on_fnf:* *bool* = *True*, *resolve:* *bool* = *False*) → Callable[[...], Dict[str, Any]]

Factory for creating a config hook that is a sequence of single factory calls.

Equivalent to calling *single_load_and_write_factory()* for each config with the other arguments passed along. See *single_load_and_write_factory()* for details.

Returns A config hook

default(*known_args*, *configs: Dict[str, Any]*) → Dict[str, Any]

Default config hook function.

An alias for calling *multi_load_and_write_factory()* with default arguments.

coma.hooks.post_config_hook

Post config hook utilities, factories, and defaults.

single_cli_override_factory(*config_id: str*, *cli_override: typing.Callable = <function override>*) → Callable[[...], Dict[str, Any]]

Factory for creating a post config hook that overrides a config's attributes.

Overriding with command line arguments is achieved by calling `cli_override`, which is *override()* by default. Slight alternatives can be created using *override_factory()*. Alternatively, a custom function can also be used.

Example

Change separator to "~":

```
coma.initiate(..., post_config_hook=override_factory(sep="~"))
```

Parameters

- **config_id** (*str*) – A config identifier
- **cli_override** (*Callable*) – Function to override config attributes with command line arguments

Returns A post config hook

See also:

- *override_factory()*
- *single_load_and_write_factory()*

multi_cli_override_factory(*cli_override: typing.Callable = <function override>*) → Callable[[...], Dict[str, Any]]

Factory for creating a post config hook that overrides attributes of all configs.

Equivalent to calling *single_cli_override_factory()* for each config with `cli_override` passed along. See *single_cli_override_factory()* for details.

default(*unknown_args: List[str]*, *configs: Dict[str, Any]*) → Dict[str, Any]

Default post config hook.

An alias for calling *multi_cli_override_factory()* with default arguments.

coma.hooks.init_hook

Init hook utilities, factories, and defaults.

positional_factory(**skips: str*) → Callable

Factory for creating an init hook that instantiates a command with some configs.

Instantiates the command object by invoking it with all configs given as positional arguments.

Note: This works because the hook protocol assumes the configs dictionary is insertion-ordered.

Parameters ***skips** (*str*) – Undesired configs can be skipped by providing the appropriate config identifiers

Returns An init hook

keyword_factory(**skips: str*, *force: bool = False*) → Callable

Factory for creating an init hook that instantiates a command with some configs.

Instantiates the command object by invoking it with all configs given as keyword arguments based on matching parameter names in the command's function signature.

Parameters

- ***skips** (*str*) – Undesired configs can be skipped by providing the appropriate config identifiers
- **force** (*bool*) – For all un-skipped configs, whether to forcibly pass them to the command object, even if no parameter names in the command's function signature match a particular config identifier. In this case, `TypeError` will be raised unless the command's function signature includes variadic keyword arguments.

Returns An init hook

default(*command: Callable*, *configs: Dict[str, Any]*) → Any

Default init hook.

An alias for calling *coma.hooks.init_hook.positional_factory()* with default arguments.

coma.hooks.run_hook

Run hook utilities, factories, and defaults.

factory(*attr_name: str = 'run'*) → Callable

Factory for creating a run hook that executes a command.

Example:

```
class Command:
    def start(self):
        ...

with coma.forget(run_hook=True):
    coma.register("cmd", Command, run_hook=factory("start"))
```

Parameters **attr_name** (*str*) – The name of the command attribute to call to execute it

Returns A run hook

default(*command*)

Default init hook function.

An alias for calling *coma.hooks.run_hook.factory()* with default arguments.

Module Attributes

General hook utilities.

hook(*fn: Callable[[...], coma.hooks.utils._T]*) → Callable[[...], coma.hooks.utils._T]

Decorator for `coma` hooks.

Enables hook definitions with only a subset of all protocol parameters.

Example:

```
@hook
def parser_hook(parser):  # "parser" is a subset of the parser hook protocol
    ...
```

Parameters **fn** (*Callable*) – Any function that implements a subset of a hook protocol

Returns A wrapped version of the function that is protocol-friendly

sequence(*hook_: Callable*, **hooks: Callable*, *return_all: bool = False*) → Callable

Wraps a sequence of hooks into a single function.

Equivalent to calling all given hooks one at a time in sequence while passing them all the same parameters. The hooks, therefore, need to have compatible call signatures. The best way to achieve this is to decorate each hook with the `@hook` decorator and ensuring all hooks subset the same hook protocol.

Example

Replace:

```
@coma.hooks.hook
def wrapper(parser):
    coma.hooks.parser_hook.factory("-a", default=123)(parser=parser)
    coma.hooks.parser_hook.factory("-b", default=456)(parser=parser)

coma.register(..., parser_hook=wrapper)
```

with:

```
wrapper = coma.hooks.sequence(
    coma.hooks.parser_hook.factory("-a", default=123),
    coma.hooks.parser_hook.factory("-b", default=456),
)

coma.register(..., parser_hook=wrapper)
```

Parameters

- **hook** (*Callable*) – The first hook in the sequence
- ***hooks** (*Callable*) – The remaining hooks in the sequence
- **return_all** (*bool*) – Whether to return all values or the last

Returns If `return_all` is `False`, the wrapper returns the value of the last hook. If `return_all` is `True`, the wrapper returns the value of all hooks in a list.

See also:

- *hook()*

5.5.2 coma.config

All config utilities.

coma.config.cli

Utilities for overriding config attributes with command line arguments.

override(*config_id: str*, *configs: Dict[str, Any]*, *unknown_args: List[str]*, ***, *sep: str = ':'*, *exclusive: bool = False*, *raise_on_no_matches: bool = True*, *raise_on_many_matches: bool = True*, *raise_on_many_seps: bool = True*, *raise_on_empty_split: bool = True*) → Any

Overrides a config's attribute values with command line arguments.

Similar to from_dotlist() followed by merge(), but with additional features.

Specifically, since `coma` commands accept an arbitrary number of configs, config attributes' names may end up clashing when using pure `omegaconf` dot-list notation. To resolve these clashes, a prefix notation is introduced.

Prefix Notation

For a config with identifier `config_id`, any `omegaconf` dot-list notation can be prefixed with `config_id` followed by `sep` to uniquely link the override to the corresponding config.

In addition, an attempt is made to match **all non-prefixed** arguments in dot-list notation to the config corresponding to `config_id`. These shared config overrides **are not consumed**, and so can be used to override multiple configs without duplication. However, this powerful feature can also be error prone. To disable it, set `exclusive` to `True`. This raises a `ValueError` if shared overrides match more than one config.

Note: If the config is not structured, `omegaconf` will happily add any attributes to it. To prevent this, ensure that the config is structured (using structured() or set_struct()).

Finally, prefixes can be abbreviated as long as the abbreviation is unambiguous (i.e., matches a unique config identifier).

Examples

Resolving clashing dot-list notations with (abbreviated) prefixes:

Listing 37: main.py

```
@dataclass
class Person:
    name: str

@dataclass
class School:
    name: str
```

(continues on next page)

(continued from previous page)

```
class AddStudent:
    def __init__(self, person, school):
        ...

    def run(self):
        ...


...
coma.register("add_student", AddStudent, Person, School)
```

Invoking on the command line:

```
$ python main.py add_student p:name="..." s:name="..."
```

Parameters

- **config_id** (*str*) – A config identifier for the config to target
- **configs** (*Dict[str, Any]*) – A dictionary of (id-config) pairs
- **unknown_args** (*List[str]*) – Remainder (second return value) of parse_known_args()
- **sep** (*str*) – The prefix separation token to use
- **exclusive** (*bool*) – Whether shared overrides should match at most one config
- **raise_on_no_matches** (*bool*) – Whether to raise or suppress a `ValueError` if a prefix does not match any known config identifier
- **raise_on_many_matches** (*bool*) – Whether to raise or suppress a `ValueError` if a prefix ambiguous (i.e., matches more than one config identifier)
- **raise_on_many_seps** (*bool*) – Whether to raise or suppress a `ValueError` if more than one `sep` token is found within a single override argument
- **raise_on_empty_split** (*bool*) – Whether to raise or suppress a `ValueError` if no split is achieved. This can only happen if `sep` is `None` and one of the arguments consists entirely of whitespace.

Returns A new config object that uses command line arguments to overrides the attributes of the config object originally corresponding to `config_id`

Raises

- **KeyError** – If `config_id` does not match any known config identifier
- **ValueError** – Various. See the `raise_on_*` above.
- **Others** – As may be raised by the underlying `omegaconf` handler

override_factory(*, *sep: str = ':'*, *exclusive: bool = False*, *raise_on_no_matches: bool = True*, *raise_on_many_matches: bool = True*, *raise_on_many_seps: bool = True*, *raise_on_empty_split: bool = True*) → Callable

Factory for creating slight variations of *override()*.

coma.config.io

Utilities for serializing configs to file.

class Extension(*value*)

Supported config serialization file extensions:

Value	Meaning
YAML	".yaml"
YML	".yml"
JSON	".json"

maybe_add_ext(*file_path: str*, *ext:* coma.config.io.Extension) → str

If `file_path` lacks a file extension, appends `ext`.

Parameters

- **file_path** (*str*) – Any file path
- **ext** (*coma.config.io.Extension*) – An extension to possibly append

Returns A file path with an extension if one was lacking

is_json_ext(*file_path: str*) → bool

Returns whether `file_path` has a JSON-like file extension.

Parameters **file_path** (*str*) – Any file path

Returns Whether `file_path` has a JSON-like file extension

is_yaml_ext(*file_path: str*, *, *strict: bool = False*) → bool

Returns whether `file_path` has a YAML-like file extension.

Parameters

- **file_path** (*str*) – Any file path
- **strict** (*bool*) – Whether to match `Extension.YAML` exactly or also allow matching against other valid YAML-like file extensions

Returns Whether `file_path` has a YAML-like file extension

is_yml_ext(*file_path: str*, *, *strict: bool = False*) → bool

Returns whether `file_path` has a YAML-like file extension.

Parameters

- **file_path** (*str*) – Any file path
- **strict** (*bool*) – Whether to match `Extension.YML` exactly or also allow matching against other valid YAML-like file extensions

Returns Whether `file_path` has a YAML-like file extension

is_ext(*file_path: str*, *which:* coma.config.io.Extension, **alts:* coma.config.io.Extension, *strict: bool = False*) → bool

Returns whether `file_path` has a file extension from a specific set.

Parameters

- **file_path** (*str*) – Any file path
- **which** (*coma.config.io.Extension*) – The primary file extension to test against

- ***alts** (coma.config.io.Extension) – A set of alternative file extensions to test against
- **strict** (*bool*) – Whether to match `which` exactly or also allow matching against any extensions in `alts`

Returns Whether `file_path` has a file extension from a specific set

load(*config: Any*, *file_path: Optional[str] = None*) → Any

Initializes a config object and possibly updates its attributes from file.

Initializes a default config object from `config` using `omegaconf`. If `file_path` is not `None`, attempts to also load a config object from file. If that succeeds, then attempts to update the default config object's attributes with attributes of the config object loaded from file.

Parameters

- **config** (*Any*) – Any config type or object to create a default config
- **file_path** (*str*) – An optional file path from which default attributes can be updated

Returns A new config object, possibly updated from file

Raises

- **ValueError** – If `file_path` has an unsupported file extension
- **IOError** – If there are issues relating to reading from `file_path`
- **Others** – As may be raised by the underlying `omegaconf` handler

dump(*config: Any*, *file_path: str*, ***, *resolve: bool = False*) → None

Serializes a config to file.

Parameters

- **config** (*Any*) – Any valid `omegaconf` config object to serialize
- **file_path** (*str*) – A file path for serializing `config`
- **resolve** (*bool*) – Whether the underlying `omegaconf` handler should resolve variable interpolation in the configuration

Raises

- **ValueError** – If `file_path` has an unsupported file extension
- **IOError** – If there are issues relating to writing to `file_path`
- **Others** – As may be raised by the underlying `omegaconf` handler

Module Attributes

General config utilities.

default_id(*config: Any*) → str

Returns the default identifier of `config`.

The default identifier is derived from `config`'s `type` name.

Parameters **config** – Any valid `omegaconf` config

Returns The default identifier of `config`

default_dest(*config_id: str*) → str

Returns the default file path parser argument destination of `config_id`.

Returns the default value for the `dest` keyword argument to add_argument() that will define the file path parser argument corresponding to `config_id`. This will also be the attribute of the `namespace` return object (first return value) of parse_known_args().

Parameters **config_id** – A config identifier

Returns The default file path parser argument attribute of `config_id`

default_default(*config_id: str*) → str

Returns the default file path parser argument default value for `config_id`.

Returns the default value for the `default` keyword argument to add_argument() that will define the file path parser argument corresponding to `config_id`.

Parameters **config_id** – A config identifier

Returns The default file path parser argument default value for `config_id`

default_flag(*config_id: str*) → str

Returns the default file path parser argument flag value for `config_id`.

Returns the default value for the `names_or_flags` variadic argument to add_argument() that will define the file path parser argument corresponding to `config_id`.

Parameters **config_id** – A config identifier

Returns The default file path parser argument flag value for `config_id`

default_help(*config_id: str*) → str

Returns the default file path parser argument help value for `config_id`.

Returns the default value for the `help` keyword argument to add_argument() that will define the file path parser argument corresponding to `config_id`.

Parameters **config_id** – A config identifier

Returns The default file path parser argument help value for `config_id`

to_dict(**configs: Union[Any, Tuple[str, Any]]*) → Dict[str, Any]

Converts configs provided in raw format to dictionary format.

`configs` should be of the form <conf> or (<id>, <conf>), where <conf> represents a config and <id> is any identifier for the config. If <id> is omitted, an identifier is derived from <conf>'s `type` name using *default_id()*. That is, specifying just <conf> is a shorthand for (default_id(<conf>), <conf>).

Note: For each *register()*ed command, both global and local config identifiers need to be unique for that command.

Returns

Configs as a dictionary with <id> keys and <conf> values.

Note: The dictionary is guaranteed to be insertion-ordered (even in Python < 3.7).

See also:

- *default_id()*
- *initiate()*
- *register()*

5.5.3 Module Attributes

SENTINEL = <object object>

A convenient sentinel for general use.

initiate(**configs: typing.Any*, *parser: typing.Optional[argparse.ArgumentParser] = None*, *parser_hook: typing.Optional[typing.Callable] = <function multi_config>*, *pre_config_hook: typing.Optional[typing.Callable] = None*, *config_hook: typing.Optional[typing.Callable] = <function multi_load_and_write_factory.<locals>._hook>*, *post_config_hook: typing.Optional[typing.Callable] = <function multi_cli_override_factory.<locals>._hook>*, *pre_init_hook: typing.Optional[typing.Callable] = None*, *init_hook: typing.Optional[typing.Callable] = <function positional_factory.<locals>._hook>*, *post_init_hook: typing.Optional[typing.Callable] = None*, *pre_run_hook: typing.Optional[typing.Callable] = None*, *run_hook: typing.Optional[typing.Callable] = <function factory.<locals>._hook>*, *post_run_hook: typing.Optional[typing.Callable] = None*, *subparsers_kwargs: typing.Optional[dict] = None*, ***id_configs: typing.Any*) → None

Initiates a coma.

Starts up `coma` with an optional argument parser, optional global configs, optional global hooks, and optional subparsers keyword arguments.

Note: Any optional configs and/or hooks are applied **globally** to every *register()*ed command, unless explicitly forgotten using the *forget()* context manager.

Configs can be provided with or without an identifier. In the latter case, an identifier is derived automatically. See *to_dict()* for additional details.

Example:

```
@dataclass
class Config1:
    ...

@dataclass
class Config2:
    ...
coma.initiate(Config1, a_non_default_id=Config2, pre_run_hook=...)
```

Parameters

- ***configs** (*Any*) – Global configs with default identifiers
- **parser** (*argparse.ArgumentParser*) – Top-level `ArgumentParser`. If `None`, an `ArgumentParser` with default parameters is used.
- **parser_hook** (*Callable*) – An optional global parser hook
- **pre_config_hook** (*Callable*) – An optional global pre config hook
- **config_hook** (*Callable*) – An optional global config hook
- **post_config_hook** (*Callable*) – An optional global post config hook

- **pre_init_hook** (*Callable*) – An optional global pre init hook
- **init_hook** (*Callable*) – An optional global init hook
- **post_init_hook** (*Callable*) – An optional global post init hook
- **pre_run_hook** (*Callable*) – An optional global pre run hook
- **run_hook** (*Callable*) – An optional global run hook
- **post_run_hook** (*Callable*) – An optional global post run hook
- **subparsers_kwargs** (*Dict[str, Any]*) – Keyword arguments to pass along to ArgumentParser.add_subparsers()
- ****id_configs** (*Any*) – Global configs with explicit identifiers

Raises **KeyError** – If config identifiers are not unique

See also:

- *forget()*
- *register()*
- *to_dict()*

register(*name: str*, *command: Callable*, **configs: Any*, *parser_hook: Optional[Callable] = None*, *pre_config_hook: Optional[Callable] = None*, *config_hook: Optional[Callable] = None*, *post_config_hook: Optional[Callable] = None*, *pre_init_hook: Optional[Callable] = None*, *init_hook: Optional[Callable] = None*, *post_init_hook: Optional[Callable] = None*, *pre_run_hook: Optional[Callable] = None*, *run_hook: Optional[Callable] = None*, *post_run_hook: Optional[Callable] = None*, *parser_kwargs: Optional[dict] = None*, ***id_configs: Any*) → None

Registers a command that might be invoked upon waking from a coma.

Registers a command with ArgumentParser.add_subparsers().add_parser(), along with providing optional local configs and optional local hooks.

Note: Any provided local configs are **appended** to the list of global configs (rather than replacing them). See *initiate()* and *forget()* for more details.

Configs can be provided with or without an identifier. In the latter case, an identifier is derived automatically. See *to_dict()* for additional details.

Examples

Register function-based command with no configurations:

```
coma.register("cmd", lambda: ...)
```

Register function-based command with configurations:

```
@dataclass
class Config:
    ...
coma.register("cmd", lambda cfg: ..., Config)
```

Register class-based command with explicit configuration identifier:

```
@dataclass
class Config:
    ...
class Command:
    def __init__(self, cfg):
        ...
    def run(self):
        ...
coma.register("cmd", Command, a_non_default_id=Config)
```

Parameters

- **name** (*str*) – Any (unique) valid command name according to `argparse`
- **command** (*Callable*) – A command class or function
- ***configs** (*Any*) – Local configs with default identifiers
- **parser_hook** (*Callable*) – An optional local parser hook
- **pre_config_hook** (*Callable*) – An optional local pre config hook
- **config_hook** (*Callable*) – An optional local config hook
- **post_config_hook** (*Callable*) – An optional local post config hook
- **pre_init_hook** (*Callable*) – An optional local pre init hook
- **init_hook** (*Callable*) – An optional local init hook
- **post_init_hook** (*Callable*) – An optional local post init hook
- **pre_run_hook** (*Callable*) – An optional local pre run hook
- **run_hook** (*Callable*) – An optional local run hook
- **post_run_hook** (*Callable*) – An optional local post run hook
- **parser_kwargs** (*Dict[str, Any]*) – Keyword arguments to pass along to the constructor of the sub-`ArgumentParser` created for `command`
- ****id_configs** (*Any*) – Local configs with explicit identifiers

Raises

- **ValueError** – If `name` is not unique
- **KeyError** – If config identifiers are not unique

See also:

- *initiate()*
- *forget()*
- *wake()*
- *to_dict()*

forget(**config_ids: str, parser_hook: bool = False, pre_config_hook: bool = False, config_hook: bool = False, post_config_hook: bool = False, pre_init_hook: bool = False, init_hook: bool = False, post_init_hook: bool = False, pre_run_hook: bool = False, run_hook: bool = False, post_run_hook: bool = False*) → Iterator[None]

Temporarily forget selected global configs or hooks while in a coma.

A context manager that enables *register()*ing commands while selectively forgetting global configs or hooks.

Example:

```
with coma.forget(...):
    coma.register(...)
```

Note: Configs are referenced by identifier whereas hooks are referenced by type. For configs that were *initiate()*d or *register()*ed without an explicit identifier, the automatically-derived identifier can be retrieved programmatically using *default_id()*.

Parameters

- ***config_ids** (*str*) – Identifiers of global configs to temporarily forget
- **parser_hook** (*bool*) – Whether to ignore the global parser hook (if any)
- **pre_config_hook** (*bool*) – Whether to ignore the global pre config hook (if any)
- **config_hook** (*bool*) – Whether to ignore the global config hook (if any)
- **post_config_hook** (*bool*) – Whether to ignore the global post config hook (if any)
- **pre_init_hook** (*bool*) – Whether to ignore the global pre init hook (if any)
- **init_hook** (*bool*) – Whether to ignore the global init hook (if any)
- **post_init_hook** (*bool*) – Whether to ignore the global post init hook (if any)
- **pre_run_hook** (*bool*) – Whether to ignore the global pre run hook (if any)
- **run_hook** (*bool*) – Whether to ignore the global run hook (if any)
- **post_run_hook** (*bool*) – Whether to ignore the global post run hook (if any)

Returns A generator yielding a single `None`

Raises `KeyError` – If any provided config identifier does not match any known config

See also:

- *initiate()*
- *register()*
- *default_id()*

wake(*args=None*, *namespace=None*) → None

Wakes from a coma.

Parses command line arguments and invokes the appropriate command using the *register()*ed hooks.

Example

Use `sys.argv` as source of command line arguments.

```
coma.wake()
```

Simulate command line arguments.

```
coma.wake(args=...)
```

Parameters

- **args** – Passed to ArgumentParser.parse_known_args()
- **namespace** – Passed to ArgumentParser.parse_known_args()

See also:

- *register()*

PYTHON MODULE INDEX

C

INDEX

L

M

O

P

R

S

T

W