Argdispatch Documentation

Release 1.3.1

Louis Paternault

Oct 03, 2023

CONTENTS

This module is a drop-in replacement for `argparse`, dispatching `subcommand` calls to functions, modules or executables.

CHAPTER

ONE

RATIONALE

If your parser has less than five subcommands, you can parse them with `argparse`. If you have more, you still can, but you will get a huge, unreadable code. This module makes this easier by dispatching subcommand calls to functions, modules or executables.

1.1 Example 1 : Manually define subcommands

For instance, consider the following code for `mycommand.py`:

```
import sys
from argdispatch import ArgumentParser

def foo(args):
    """A function associated to subcommand `foo`."""
    print("Doing interesting stuff")
    sys.exit(1)

if __name__ == "__main__":
    parser = ArgumentParser()
    subparser = parser.add_subparsers()

    subparser.add_function(foo)
    subparser.add_module("bar")
    subparser.add_executable("baz")

     parser.parse_args()
```

With this simple code:

- `mycommand.py foo -v --arg=2` is equivalent to the python code `foo(['-v', '--arg=2'])`;
- `mycommand.py bar -v --arg=2` is equivalent to `python -m bar -v --arg=2`;
- `mycommand.py baz -v --arg=2` is equivalent to `baz -v --arg=2`.

Then, each function, module or executable does whatever it wants with the arguments.

1.2 Example 2 : Automatic subcommand definition

With programs like git, if a `git-foo` executable exists, then calling `git foo --some=arguments` is equivalent to `git-foo --some=arguments`. The following code, in `myprogram.py` copies this behaviour:

```
import sys
from argdispatch import ArgumentParser

if __name__ == "__main__":
    parser = ArgumentParser()
    subparser = parser.add_subparsers()

    subparser.add_submodules("myprogram")
    subparser.add_prefix_executables("myprogram-")

    parser.parse_args()
```

With this program, given that executable `myprogram-foo` and python module `myprogram.bar.__main__.py` exist:

- `myprogram foo -v --arg=2` is equivalent to `myprogram-foo -v --arg=2`;
- `myprogram bar -v --arg=2` is equivalent to `python -m myprogram.bar -v --arg=2`.

1.3 Example 3 : Defining subcommands with entry points

Now that your program is popular, people start writing plugins. Great! You want to allow them to add subcommands to your program. To do so, simply use this code:

```
import sys
from argdispatch import ArgumentParser

if __name__ == "__main__":
    parser = ArgumentParser()
    subparser = parser.add_subparsers()

    # You probably should only have one of those.
    subparser.add_entrypoints_functions("myprogram.subcommand.function")
    subparser.add_entrypoints_modules("myprogram.subcommand.module")

    parser.parse_args()
```

With this code, plugin writers can add lines like those in their `setup.cfg`:

```
[options.entry_points]
myprogram.subcommand.function =
    foo = mypluginfoo:myfunction
myprogram.subcommand.module =
    bar = mypluginbar
```

Then, given than function `myfunction()` exists in module `mypluginfoo`, and than module `mypluginbar` exists:

- `myprogram foo -v --arg=2` is equivalent to the python code `myfunction(['-v', '--arg=2'])`;
- `myprogram bar -v --arg=2` is equivalent to `python -m mypluginbar -v --arg=2`.

CHAPTER

TWO

MODULE DOCUMENTATION

A replacement for *argparse* dispatching subcommand calls to functions, modules or executables.

2.1 Parsing arguments

class argdispatch.**ArgumentParser**

Create a new *ArgumentParser* object.

There is no visible changes compared to `argparse.ArgumentParser`. For internal changes, see *Advanced usage*.

2.2 Adding subcommands

Adding subcommands to your program starts the same way as with argparse: one has to call `ArgumentParser.add_subparsers()`, and then call one of the methods of the returned object. With `argparse`, this object only have one method *add_parser()*. This module adds several new methods.

2.2.1 Subcommands defined twice

Most of the methods creating subcommands accept an *ondouble* arguments, which tells what to do when adding a subcommand that already exists:

- argdispatch.**ERROR**

  Raise an `AttributeError` exception;

- argdispatch.**IGNORE**

  The new subcommand is silently ignored;

- argdispatch.**DOUBLE**

  The new subcommand is added to the parser, and `argparse` deals with it. This does not seem to be documented, but it seems that the parser then contains two subcommands with the same name.

  Deprecated since version 1.3.0: Python3.11 raises an error if two subparsers have the same name.

2.2.2 Import errors

When using methods *add_module()* and *add_submodules()*, modules are imported. But some modules can be impossible to import because of errors. Both these methods have the argument `onerror` to define what to do with such modules:

- `argdispatch.RAISE`

  Raise an exception (propagate the exception raised by the module).

- `argdispatch.IGNORE`

  Silently ignore this module.

2.2.3 Return value

Unfortunately, different methods make `ArgumentParser.parse_args()` return different types of values. The two possible behaviours are illustrated below:

```
>>> from argdispatch import ArgumentParser
>>> def add(args):
...     print(int(args[0]) + int(args[1]))
...
>>> parser = ArgumentParser()
>>> subparsers = parser.add_subparsers()
>>> parser1 = subparsers.add_parser("foo")
>>> parser1.add_argument("--arg")
_StoreAction(
        option_strings=['--arg'], dest='arg', nargs=None, const=None, default=None,
        type=None, choices=None, help=None, metavar=None,
        )
>>> subparsers.add_function(add)
>>> parser.parse_args("foo --arg 3".split())
Namespace(arg='3')
>>> parser.parse_args("add 3 4".split())
7
```

The `NameSpace(...)` is the object *returned* by `parse_args()`, while the `7` is *printed* by function, and the interpreter then exits (by calling `sys.exit()`).

Call to `parse_args()`, when parsing a subcommand defined by:

- legacy method *add_parser()*, returns a `Namespace` (this method is (almost) unchanged compared to `argparse`);
- new methods do not return anything, but exit the program with `sys.exit()`.

Thus, we do recommand not to mix them, to make source code easier to read, but technically, it is possible.

2.2.4 Subcommand definition

Here are all the *_SubCommandsDispatch* commands to define subcommands.

- Legacy subcommand

  _SubCommandsDispatch.**add_parser**(**args*, ***kwargs*)

  Add a subparser, and return an *ArgumentParser* object.

  This is the same method as the original argparse, excepted that an ondouble argument has been added.

  > **Warning:** Depending of value of the *ondouble* argument, this method may return a *ArgumentParser* object, or *None*.
  >
  > If argument *ondouble* is *IGNORE*, and the command name is already defined, this function returns nothing (*None*). Otherwise, it returns an *ArgumentParser* object.

  Parameters
  ondouble – See *Subcommands defined twice*. Default is *ERROR* (default was *DOUBLE* before version 1.3.0).
  Returns
  A *ArgumentParser* object, or *None*.
  Raises
  A ValueError exception, if argument *ondouble* is *ERROR*, and command name already exists.

  Changed in version 1.3.0: Before version 1.3.0, default value for ondouble was *DOUBLE*. It is now *ERROR*.

- Function subcommand

  _SubCommandsDispatch.**add_function**(*function*, *command=None*, ***, *help=None*, *ondouble=_Constants.ERROR*)

  Add a subcommand matching a python function.
  Parameters
  - **function** – Function to use.
  - **command** (*str*) – Name of the subcommand. If None, the function name is used.
  - **help** (*str*) – A brief description of what the subcommand does. If *None*, use the first non-empty line of the function docstring.
  - **ondouble** – See *Subcommands defined twice*. Default is *ERROR*.

  This function is approximatively called using:

  ```
  sys.exit(function(args))
  ```

  It must either return something which will be transimtted to sys.exit(), or directly exit using sys.exit(). If it raises an exception, this exception is not catched by *argdispatch*.

- Module subcommands

  Those methods are compatible with PEP 420 namespace packages.

  _SubCommandsDispatch.**add_module**(*module*, *command=None*, ***, *help=None*, *ondouble=_Constants.ERROR*, *onerror=_Constants.RAISE*, *forcemain=False*)

  Add a subcommand matching a python module.

When such a subcommand is parsed, `python -m module` is called with the remaining arguments.

Parameters

- **module** – Module or package to use. If a package, the `__main__` submodule is used.

  This argument can either be a string or an already imported module. Both cases are shown in the following examble:

  ```
  import foo

  parser = ArgumentParser()
  subparser = parser.add_subparsers()

  # Argument `foo` is a module.
  subparser.add_module(foo)

  # Argument `bar` is a string.
  subparser.add_module("bar")
  ```

  Note that the only way to import a *relative* module is by importing it yourself, then passing the module as argument to this method.
- **command** (*str*) – Name of the subcommand. If `None`, the module name is used.
- **help** (*str*) – A brief description of what the subcommand does. If *None*, use the first non-empty line of the module docstring, only if the module is not a package. Otherwise, an empty message is used.
- **ondouble** – See *Subcommands defined twice*. Default is *ERROR*.
- **onerror** – See *Import errors*. Default is *RAISE*.
- **forcemain** – Raise error if parameter *module* is not a package containing a *__main__* module (this error may be ignored if parameter *onerror* is *IGNORE*). Default is *False*.

`_SubCommandsDispatch.`**add_submodules**(*module*, *, *ondouble=_Constants.IGNORE*, *onerror=_Constants.IGNORE*)

Add subcommands matching *module*'s submodules.

The modules that are used as subcommands are submodules of *module* (without recursion), that themselves contain a `__main__` submodule.

Parameters

- **module** – Module to use. It can either a string or a module (see *add_module()*).
- **ondouble** – See *Subcommands defined twice*. Default is *IGNORE*.
- **onerror** – See *Import errors*. Default is *IGNORE*.

- Entry points subcommands

  Those methods deal with setuptools entry points.

  `_SubCommandsDispatch.`**add_entrypoints_modules**(*group*, *, *ondouble=_Constants.IGNORE*, *onerror=_Constants.IGNORE*, *forcemain=False*)

  Add modules listed in entry points *group* as subcommands.

  Parameters

  - **group** (*str*) – The entry point group listing the functions to be used as subcommands.
  - **ondouble** – See *Subcommands defined twice*. Default is *IGNORE*.
  - **onerror** – See *Import errors*. Default is *IGNORE*.

- **forcemain** – Raise error if parameter *module* is not a package containing a *__main__* module (this error may be ignored, and the faulty module ignored as well, if parameter *onerror* is *IGNORE*). Default is *False*.

_SubCommandsDispatch.**add_entrypoints_functions**(*group*, *, *ondouble=_Constants.IGNORE*, *onerror=_Constants.IGNORE*)

Add functions listed in entry points *group* as subcommands.

Parameters

- **group** (*str*) – The entry point group listing the functions to be used as subcommands.
- **ondouble** – See *Subcommands defined twice*. Default is *IGNORE*.
- **onerror** – See *Import errors*. Default is *IGNORE*.

- Executable subcommands

_SubCommandsDispatch.**add_executable**(*executable*, *command=None*, *, *help=None*, *ondouble=_Constants.ERROR*)

Add a subcommand matching a system executable.

Parameters

- **executable** (*str*) – Name of the executable to use.
- **command** (*str*) – Name of the subcommand. If `None`, the executable is used.
- **help** (*str*) – A brief description of what the subcommand does. If *None*, use an empty help.
- **ondouble** – See *Subcommands defined twice*. Default is *ERROR*.

_SubCommandsDispatch.**add_pattern_executables**(*pattern*, *, *path=None*, *ondouble=_Constants.IGNORE*)

Add all the executables in path matching the regular expression.

If *pattern* contains a group named *command*, this is used as the subcommand name. Otherwise, the executable name is used.

Parameters

- **pattern** (*str*) – Regular expression defining the executables to add as subcommand.
- **path** (*iterable*) – Iterator on paths in which executable has to been searched for. If *None*, use the `PATH` environment variable. This arguments *replaces* the `PATH` environment variable: if you want to extend it, use `":".join(["my/custom", "path", os.environ.get("PATH", "")])`.
- **ondouble** – See *Subcommands defined twice*. Default is *IGNORE*.

_SubCommandsDispatch.**add_prefix_executables**(*prefix*, *, *path=None*, *ondouble=_Constants.IGNORE*)

Add all the executables starting with `prefix`

The subcommand name used is the executable name, without the prefix.

Parameters

- **prefix** – Common prefix of all the executables to use as subcommands.
- **path** (*iterable*) – Iterator on paths in which executable has to been searched for. See *add_pattern_executables()* for more information.
- **ondouble** – See *Subcommands defined twice*. Default is *IGNORE*.

CHAPTER

THREE

ADVANCED USAGE

This module works by subclassing two `argparse` classes:

- *`_SubCommandsDispatch`* is a subclass of `argparse._SubParsersAction`;
- *`ArgumentParser`* is a subclass of `argparse.ArgumentParser`.

The class doing all the job is *`_SubCommandsDispatch`*.

class argdispatch.**_SubCommandsDispatch**(**args*, ***kwargs*)

Object returned by the `argparse.ArgumentParser.add_subparsers()` method.

Its methods `add_*()` are used to add subcommands to the parser.

The only thing changed in *`ArgumentParser`* is `ArgumentParser.add_subparsers()`, which enforces argument `action=_SubCommandsDispatch` for parent method `argparse.ArgumentParser.add_subparsers()`. Thus, it is also possible to use this module as following:

```
import argparse
import argdispatch

parser = argparse.ArgumentParser(...)
subparsers = parser.add_subparsers(action=argdispatch._SubCommandsDispatch)
...
```

CHAPTER

FOUR

DOWNLOAD AND INSTALL

See the main project page for instructions, and changelog.

PYTHON MODULE INDEX

INDEX

Symbols

A

D

E

I

M

R