Argdispatch Documentation

Release 0.2.0

Louis Paternault

Aug 28, 2017

Contents

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

CHAPTER 1

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.

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.

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`.

CHAPTER 2

Module documentation

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

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*.

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.

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.

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 `ImportError` exception.
- `argdispatch.`**`IGNORE`**
  Silently ignore this module.

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.

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.

Parameters **ondouble** – See *Subcommands defined twice*. Default is *DOUBLE*.

- Function subcommand

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

_SubCommandsDispatch.**add_module**(*module*, *command=None*, ***, *path=None*, *help=None*, *ondouble=ERROR*, *onerror=RAISE*, *forcepackage=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** (*str*) – Module or package to use. If a package, the `__main__` submodule is used.
- **command** (*str*) – Name of the subcommand. If `None`, the module name is used.
- **path** (*iterable*) – Iterator on paths in which module has to been searched for. If *None*, use `sys.path`. This arguments *replaces* `sys.path`: if you want to extend it, use `sys.path + ["my/custom", "path"]`.
- **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*.
- **forcepackage** – Raise error if parameter *module* is not a package (this error may be ignored if parameter *onerror* is *IGNORE*). Default is *False*.

_SubCommandsDispatch.**add_submodules**(*module*, ***, *path=None*, *ondouble=IGNORE*, *onerror=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** (*str*) – Module to use.
- **path** (*iterable*) – Iterator on paths in which module has to been searched for. See *add_module()* for more information.
- **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=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=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=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 3

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 4

Download and install

See the main project page for instructions, and changelog.

Python Module Index

a

Index

Symbols

A

D

E

I

R