# DeclarativeParser Documentation

***Release 0.1.1***

**MK**

**Nov 30, 2017**

Jump into the docs:


1 Parser 3

2 Constructor Parser 7

3 Utility types 9

4 Built on top of argparse 11

5 Nested parsers 13

6 Parallel parsers 15

7 Production pattern 17

8 Batteries included 19

9 Arguments deduction (typing, docstrings, kwargs) 21

10 Actions 23

11 Acknowledgements 25

12 Indices and tables 27

Python Module Index 29

Powerful like click, integrated like argparse, declarative as sqlalchemy. MIT licenced.

Features:

- *Built on top of argparse*
- *Nested parsers*
- *Parallel parsers*
- *Production pattern*
- *Batteries included*
- *Arguments deduction (typing, docstrings, kwargs)*
- *Actions*

CHAPTER 1

# Parser

use *declarative_parser.parser* module to create custom parsers and arguments.

**class Argument**(*name=None*, *short=None*, *optional=True*, *as_many_as: typing.Union[declarative_parser.parser.Argument, NoneType] = None*, ***kwargs*)

Defines argument for *Parser*.

In essence, this is a wrapper for `argparse.ArgumentParser.add_argument()`, so most options (type, help) which work in standard Python parser will work with Argument too. Additionally, some nice features, like automated naming are available.

Worth to mention that when used with *ConstructorParser*, *type* and *help* will be automatically deduced.

**Parameters**

- **name** – overrides deduced argument name
- **short** – a single letter to be used as a short name (e.g. "c" will enable using "-c")
- **optional** – by default True, provide False to make the argument required
- **as_many_as** (Optional[*Argument*]) – if provided, will check if len() of the produced value is equal to len() of the provided argument
- ****kwargs** – other keyword arguments which are supported by *argparse.add_argument()*

**class Parser**(*parser_name=None*, ***kwargs*)

Parser is a wrapper around Python built-in `argparse.ArgumentParser`.

Subclass the *Parser* to create your own parser.

Use help, description and epilog properties to adjust the help screen. By default help and description will be auto-generated using docstring and defined arguments.

Attach custom arguments and sub-parsers by defining class-variables with *Argument* and *Parser* instances.

Example:

```
class TheParser(Parser):
    help = 'This takes only one argument, but it is required'
```

```
    arg = Argument(optional=False, help='This is required')

class MyParser(Parser):
    description = 'This should be a longer text'

    my_argument = Argument(type=int, help='some number')
    my_sub_parser = TheParser()

    epilog = 'You can create a footer with this'

# To execute the parser use:

parser = MyParser()

# The commands will usually be `sys.argv[1:]`
commands = '--my_argument 4 my_sub_parser value'.split()

namespace = parser.parse_args(commands)

# `namespace` is a normal `argparse.Namespace`
assert namespace.my_argument == 4
assert namespace.my_sub_parser.arg == 'value'
```

Implementation details:

To enable behaviour not possible with limited, plain *ArgumentParser* (e.g. to dynamically attach a sub-parser, or to chain two or more sub-parsers together) the stored actions and sub-parsers are:

- not attached permanently to the parser,
- attached in a tricky way to enable desired behaviour,
- executed directly or in hierarchical order.

Class-variables with parsers will be deep-copied on initialization, so you do not have to worry about re-use of parsers.

Uses kwargs to pre-populate namespace of the *Parser*.

**Parameters** **parser_name** – a name used for identification of sub-parser

**attach_argument**(*argument*, *parser=None*)
Attach Argument instance to given (or own) argparse.parser.

**attach_subparsers**()
Only in order to show a nice help, really.

There are some issues when using subparsers added with the built-in add_subparsers for parsing. Instead subparsers are handled in a custom implementation of parse_known_args (which really builds upon the built-in one, just tweaking some places).

**bind_argument**(*argument*, *name=None*)
Bind argument to current instance of Parser.

**bind_parser**(*parser*, *name*)
Bind deep-copy of Parser with this instance (as a sub-parser).

**Parameters**

- **parser** (*Parser*) – parser to be bound as a sub-parser (must be already initialized)
- **name** – name of the new sub-parser

This method takes care of 'translucent' sub-parsers (i.e. parsers which expose their arguments and sub-parsers to namespace above), saving their members to appropriate dicts (lifted_args/parsers).

**description**
Longer description of the parser.

Description is shown when user narrows down the help to the parser with: `./run.py sub_parser_name -h`.

**epilog**
Use this to append text after the help message

**error**(*message*)
Raises SystemExit with status code 2 and shows usage message.

**help**
A short message, shown as summary on >parent< parser help screen.

Help will be displayed for sub-parsers only.

**parse_args**(*args=None*)
Same as *parse_known_args()* but all arguments must be parsed.

This is an equivalent of `argparse.ArgumentParser.parse_args()` although it does >not< support *namespace* keyword argument.

Comparing to *parse_known_args()*, this method handles help messages nicely (i.e. passes everything to `argparse`).

**Parameters** **args** (`Optional[Sequence[str]]`) – strings to parse, default is sys.argv[1:]

**parse_known_args**(*args*)
Parse known arguments, like `argparse.ArgumentParser.parse_known_args()`.

**Additional features (when compared to argparse implementation) are:**

- ability to handle multiple sub-parsers
- validation with *self.validate* (run after parsing)
- additional post-processing with *self.produce* (after validation)

**produce**(*unknown_args*)
Post-process already parsed namespace.

You can override this method to create a custom objects in the parsed namespace (e.g. if you cannot specify the target class with Argument(type=X), because X depends on two or more arguments).

You can chery-pick the arguments which were not parsed by the current parser (e.g. when some step of parsing depends on provided arguments), but please remember to remove those from *unknown_args* list.

Remember to operate on the provided list object (do not rebind the name with *unknown_args = []*, as doing so will have no effect: use *unknown_args.remove()* instead).

**validate**(*opts*)
Perform additional validation, using *Argument.validate*.

As validation is performed after parsing, all arguments should be already accessible in *self.namespace*. This enables testing if arguments depending one on another have proper values.

**action**(*method*)
Decorator for Action.

**Parameters** **method** – static or class method for use as a callback

**create_action**(*callback*, *exit_immediately=True*)
Factory for `argparse.Action`, for simple callback execution

**dedent_help**(*text*)
Dedent text by four spaces

**group_arguments**(*args*, *group_names*)
Group arguments into given groups + None group for all others

# CHAPTER 2

## Constructor Parser

**ClassParser**
alias of *ConstructorParser*

**class ConstructorParser**(*constructor*, *docstring_type='google'*, ***kwargs*)
Create a parser from an existing class, using arguments from __init__

as well as arguments and sub-parsers defined as class properties.

Example usage:

```
import argparse

class MyProgram:

    database = Argument(
        type=argparse.FileType('r'),
        help='Path to file with the database'
    )

    def __init__(self, threshold:float=0.05, database=None):
        # do some magic
        pass

parser = ConstructorParser(MyProgram)

options, remaining_unknown_args = parser.parse_known_args(unknown_args)

program = parser.constructor(**vars(options))
```

Initializes parser analyzing provided class constructor.

**Parameters**

- **constructor** – a class to use for parser auto-generation
- **docstring_type** – docstring convention used in *__init__* method of provided class; one of: google, numpy, rst

- **kwargs** – custom keyword arguments to be passed to Parser

**class FunctionParser**(*constructor*, ***kwargs*)

Create a parser from an existing function

as well as arguments and sub-parsers defined in function object.

Example usage:

```
def calc_exponent(base: float, exponent: int=2):
    return base ** exponent

parser = FunctionParser(calc_exponent)

commands = '2 --exponent 3'.split()
options = parser.parse_args(commands)
result = parser.constructor(**vars(options))

assert result == 2 * 2 * 2
```

Advanced usage:

```
def calc_exponent(base: float, exponent: int=2):
    return base ** exponent

# you can override definitions deduced from signature and
# docstring: just assign custom Argument on the function:
calc_exponent.exponent = Argument(short='n', type=int, default=2)

parser = FunctionParser(calc_exponent)

commands = '2 -n 3'.split()
options = parser.parse_args(commands)
result = parser.constructor(**vars(options))

assert result == 2 * 2 * 2
```

Initializes parser analyzing provided function.

**Parameters**

- **constructor** – a function to use for parser auto-generation
- **docstring_type** – docstring convention used in provided function one of: google, numpy, rst
- **kwargs** – custom keyword arguments to be passed to Parser

# CHAPTER 3

## Utility types

**class Range**(*string*)
Simplified slice with '-' as separator.

Handles only start and end, does not support negative numbers.

**item_type**
alias of `int`

**class StringHandlingMixin**(*string*)
Turn string provided on initialization into *data_type*.

**require_separator**
If True and the string has no separator ArgumentTypeError will be raised.

**separator**
Separator for split operation

**dsv**(*value_type*, *delimiter=', '*)
Delimiter Separated Values

**n_tuple**(*n*)
Factory for n-tuples.

**one_of**(**types*)
Create a function which attempts to cast input to any of provided types.

The order of provided *types* is meaningful - if two types accept given input value, the first one on list will be used. Types should be able to accept a string (if correct) as input value for their constructors.

# CHAPTER 4

## Built on top of argparse

The basic API of the DeclarativeParser is compatible with argparse, so you do not need to learn from start.

This is the arparse way:

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("square", help="display a square of a given number")
args = parser.parse_args()
print(args.square**2)
```

This is the declarative way:

```
from declarative_parser import Parser, Argument

class MyParser(Parser):
    square = Argument(help='display a square of a given number')

parser = MyParser()
args = parser.parse_args()
print(args.square**2)
```

CHAPTER 5

# Nested parsers

DeclarativeParser allows you to nest parsers one in another, just like 'git commit' or 'git push':

```
class TheParser(Parser):
    help = 'This takes only one argument, but it is required'

    arg = Argument(optional=False, help='This is required')

class MyParser(Parser):
    description = 'This should be a longer text'

    my_argument = Argument(type=int, help='some number')
    my_sub_parser = TheParser()

    epilog = 'You can create a footer with this'
```

To execute the parser use:

```
parser = MyParser()

# The commands will usually be `sys.argv[1:]`
commands = '--my_argument 4 my_sub_parser value'.split()

namespace = parser.parse_args(commands)
```

Resultant *namespace* is a normal `argparse.Namespace`

```
assert namespace.my_argument == 4
assert namespace.my_sub_parser.arg == 'value'
```

CHAPTER 6

# Parallel parsers

You can have multiple sub-parsers on the same level, like:

```
supported_formats = ['png', 'jpeg', 'gif']

class InputOptions(Parser):
    path = Argument(type=argparse.FileType('rb'), optional=False)
    format = Argument(default='png', choices=supported_formats)

class OutputOptions(Parser):
    format = Argument(default='jpeg', choices=supported_formats)
    scale = Argument(type=int, default=100, help='Rescale image to % of original size
↪')

class ImageConverter(Parser):
    description = 'This app converts images'

    verbose = Argument(action='store_true')
    input = InputOptions()
    output = OutputOptions()

parser = ImageConverter()

commands = '--verbose input image.png output --format gif --scale 50'.split()

namespace = parser.parse_args(commands)

assert namespace.input.format == 'png'
assert namespace.output.format == 'gif'
```

As simple as it looks!

CHAPTER 7

# Production pattern

Do you want to introduce sophisticated behaviour to your parser, but keep the logic away from the core of your app? DeclarativeParser enables you to add "produce" method to each parser, which will transform the arguments namespace in a way you wish it to be done!

Have a look on this example of advanced file parsing:

```
from declarative_parser import Parser, Argument
from declarative_parser.types import Slice, Indices, Range, one_of


def slice_file(file, columns_selector=None, delimiter=None):
    pass

class FileSubsetFactory(Parser):
    """Parse user options and load desired part of given file.

     The files should come in Delimiter Separated Values format
     (like .csv or .tsv). The default delimiter is a tab character.

     To use only a subset of columns from given file,
     specify column numbers with --columns.
     """

    file = Argument(
        type=argparse.FileType('r'),
        optional=False
    )

    columns = Argument(
        # we want to handle either ":4", "5:" or even "1,2,3"
        type=one_of(Slice, Indices, Range),
        # user may (but do not have to) specify columns
        # to be extracted from given file(s).
        help='Columns to be extracted from files: '
             'either a comma delimited list of 0-based numbers (e.g. 0,2,3) '
```

```
            'or a range defined using Python slice notation (e.g. 3:10). '
            'Columns for each of files should be separated by space.'
    )

    delimiter = Argument(
        default='\t',
        help='Delimiter of the provided file(s). Default: tabulation mark.'
    )

    def produce(self, unknown_args=None):
        opts = self.namespace

        opts.file_subset = slice_file(
            opts.file,
            columns_selector=opts.columns.get_iterator if opts.columns else None,
            delimiter=opts.delimiter,
        )

        return opts
```

After parsing *file_subset* will become a part of your resultant namespace.

CHAPTER 8

# Batteries included

Powerful validation, additional types and more.

Do you want to allow user to provide distinct options for each of provided files, but not to validate the number of arguments every single time? No problem, just use *as_many_as=files*.

```
class AdvancedFileFactory(Parser):
    """Parse user options and load given file(s).

     To use only a subset of columns from files(s) specify column numbers
     (--columns) or column names (--names) of desired columns.
     """

    files = Argument(
        type=argparse.FileType('r'),
        # at least one file is always required
        nargs='+',
        optional=False
    )

    names = Argument(
        type=dsv(str),
        nargs='*',
        as_many_as=files,
        help='Names of columns to be extracted from the file. '
             'Names are determined from the first non-empty row. '
             'Use a comma to separate column names. '
             'Column names for each of files should be separated by space.'
    )

    columns = Argument(
        # we want to handle either ":4", "5:" or even "1,2,3"
        type=one_of(Slice, Indices, Range),
        # user may (but do not have to) specify columns
        # to be extracted from given file(s).
        nargs='*',
```

```
        as_many_as=files,
        help='Columns to be extracted from files: '
             'either a comma delimited list of 0-based numbers (e.g. 0,2,3) '
             'or a range defined using Python slice notation (e.g. 3:10). '
             'Columns for each of files should be separated by space.'
    )


    def produce(self, unknown_args=None):
        opts = self.namespace

        file_chunks = []

        for i, file_obj in enumerate(opts.files):

            file_chunks.append(slice_file(
                opts.file,
                names=opts.names[i] if opts.names else None,
                columns_selector=opts.columns[i].get_iterator if opts.columns else␣
↪None,
            ))

        opts.file_subset = merge_chunks(file_chunks)

        return opts
```

To further explore additional types, see: *Utility types*.

# CHAPTER 9

## Arguments deduction (typing, docstrings, kwargs)

What about automatic parser creation? You can use ClassParser of FunctionParser for that!

Just feed *constructor_parser.ClassParser* with your main class and it will take care of it. Arguments defined in your *__init__* and in body of your class (i.e. class variables) will be used to create a parser; Type annotations (as long as based on real types, not typing module) will be used to define types of your arguments; Default: from keyword arguments. Positional arguments will be always required. Docstring descriptions will be used to provide help for your arguments.

Following docstring formats are supported: Google, NumPy and reStructuredText, with the default being Google. To change the format, pass *docstring_type='numpy'* or *docstring_type='rst'* respectively.

When an argument is defined in both: *__init__* and `declarative_parser.Argument()` variable, the class variable overwrites the values from *__init__* .

```
import argparse
from declarative_parser import Argument
from declarative_parser.constructor_parser import ClassParser

class MyProgram:

    database = Argument(
        type=argparse.FileType('r'),
        help='Path to file with the database'
    )

    def __init__(self, text: str, threshold: float=0.05, database=None):
        """My program does XYZ.

        Arguments:
          threshold: a floating-point value defining threshold, default 0.05
          database: file object to the database if any
        """
        print(text, threshold, None)

parser = ClassParser(MyProgram)
```

```
options = parser.parse_args()
program = parser.constructor(**vars(options))
```

And it works quite intuitively:

```
$ ./my_program.py test --threshold 0.6
test 0.6 None
$ ./my_program.py test --threshold f
usage: my_program.py [-h] [--database DATABASE] [--threshold THRESHOLD] text {} ...
my_program.py: error: argument --threshold: invalid float value: 'f'
$ ./my_program.py --threshold 0.6
usage: my_program.py [-h] [--database DATABASE] [--threshold THRESHOLD] text {} ...
my_program.py: error: the following arguments are required: text
```

You could then implement *run* method and call *program.run()* to start you application.

Likewise, *constructor_parser.FunctionParser* will create a parser using your function:

```
def calc_exponent(base: float, exponent: int=2):
    return base ** exponent

parser = FunctionParser(calc_exponent)

commands = '2 --exponent 3'.split()
options = parser.parse_args(commands)
result = parser.constructor(**vars(options))

assert result == 2 * 2 * 2
```

# CHAPTER 10

# Actions

What if you only want to show licence of your program? or version? It there a need to write a separate logic? DeclarativeParser gives you utility decorator: *@action* which utilizes the power of `argparse.Action`, leaving behind the otherwise necessary boilerplate code.

```
__version__ = 2.0

import argparse
from declarative_parser import action
from declarative_parser.constructor_parser import ConstructorParser

class MyProgram:

    def __init__(self, threshold: float=0.05):
        """My program does XYZ.

        Arguments:
          threshold: a floating-point value, default 0.05
        """
        pass

    @action
    def version(options):
       print(__version__)

parser = ConstructorParser(MyProgram)

options = parser.parse_args()
program = parser.constructor(**vars(options))
```

The execution of an action will (by default) cause the program to exit immediately when finished. See following run as example:

```
$ ./my_program.py --version
2.0
$
```

CHAPTER 11

# Acknowledgements

This module was originally developed for https://github.com/kn-bibs/pathways-analysis project. Big thanks go to @hansiu, @sienkie and @pjanek for early feedback, inspiration and some valuable insights.

CHAPTER 12

# Indices and tables

- genindex
- modindex
- search

# Python Module Index

## d

declarative_parser.constructor_parser, 7
declarative_parser.parser, 3
declarative_parser.types, 9

# Index


## A

action() (in module declarative_parser.parser), 5
Argument (class in declarative_parser.parser), 3
attach_argument() (Parser method), 4
attach_subparsers() (Parser method), 4

## B

bind_argument() (Parser method), 4
bind_parser() (Parser method), 4

## C

ClassParser (in module declarative_parser.constructor_parser), 7
ConstructorParser (class in declarative_parser.constructor_parser), 7
create_action() (in module declarative_parser.parser), 5

## D

declarative_parser.constructor_parser (module), 7
declarative_parser.parser (module), 3
declarative_parser.types (module), 9
dedent_help() (in module declarative_parser.parser), 6
description (Parser attribute), 5
dsv() (in module declarative_parser.types), 9

## E

epilog (Parser attribute), 5
error() (Parser method), 5

## F

FunctionParser (class in declarative_parser.constructor_parser), 8

## G

group_arguments() (in module declarative_parser.parser), 6

## H

help (Parser attribute), 5

## I

item_type (Range attribute), 9

## N

n_tuple() (in module declarative_parser.types), 9

## O

one_of() (in module declarative_parser.types), 9

## P

parse_args() (Parser method), 5
parse_known_args() (Parser method), 5
Parser (class in declarative_parser.parser), 3
produce() (Parser method), 5

## R

Range (class in declarative_parser.types), 9
require_separator (StringHandlingMixin attribute), 9

## S

separator (StringHandlingMixin attribute), 9
StringHandlingMixin (class in declarative_parser.types), 9

## V

validate() (Parser method), 5