scriptconfig Documentation

Release 0.7.11

Kitware Inc. Jon Crall

Nov 16, 2023

PACKAGE LAYOUT

CHAPTER

ONE

SCRIPTCONFIG

The goal of `scriptconfig` is to make it easy to be able to define a CLI by **simply defining a dictionary**. Thie enables you to write simple configs and update from CLI, kwargs, and/or json.

The pattern is simple:

1. Create a class that inherits from *scriptconfig.Config*
2. Create a class variable dictionary named `default`
3. The keys are the names of your arguments, and the values are the defaults.
4. Create an instance of your config object. If you pass `cmdline=True` as an argument, it will autopopulate itself from the command line.

Here is an example for a simple calculator program:

```
import scriptconfig as scfg


class MyConfig(scfg.Config):
    'The docstring becomes the CLI description!'
    default = {
        'num1': 0,
        'num2': 1,
        'outfile': './result.txt',
    }


def main():
    config = MyConfig(cmdline=True)
    result = config['num1'] + config['num2']
    with open(config['outfile'], 'w') as file:
        file.write(str(result))


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

If the above is written to a file *calc.py*, it can be be called like this.

```
python calc.py --num1=3 --num2=4 --outfile=/dev/stdout
```

It is possible to gain finer control over the CLI by specifying the values in `default` as a *scriptconfig.Value*, where you can specify a help message, the expected variable type, if it is a positional variable, alias parameters for the

command line, and more.

The important thing that gives scriptconfig an edge over things like `argparse` is that it is trivial to disable the `cmdline` flag and pass explicit arguments into your function as a dictionary. Thus you can write you scripts in such a way that they are callable from Python or from a CLI via with an API that corresponds 1-to-1!

A more complex example version of the above code might look like this

```
import scriptconfig as scfg


class MyConfig(scfg.Config):
    '''
    The docstring becomes the CLI description!
    '''
    default = {
        'num1': scfg.Value(0, type=float, help='first number to add', position=1),
        'num2': scfg.Value(1, type=float, help='second number to add', position=2),
        'outfile': scfg.Value('./result.txt', help='where to store the result',␣
↪position=3),
    }


def main(cmdline=1, **kwargs):
    '''
    Example:
        >>> # This is much easier to test than argparse code
        >>> kwargs = {'num1': 42, 'num2': 23, 'outfile': 'foo.out'}
        >>> cmdline = 0
        >>> main(cmdline=cmdline, **kwargs)
        >>> with open('foo.out') as file:
        >>>     assert file.read() == '65'
    '''
    config = MyConfig(cmdline=True, data=kwargs)
    result = config['num1'] + config['num2']
    with open(config['outfile'], 'w') as file:
        file.write(str(result))


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

This code can be called with positional arguments:

```
python calc.py 33 44 /dev/stdout
```

The help text for this program (via `python calc.py --help`) looks like this:

```
usage: MyConfig [-h] [--num1 NUM1] [--num2 NUM2] [--outfile OUTFILE] [--config CONFIG] [-
↪-dump DUMP] [--dumps] num1 num2 outfile

The docstring becomes the CLI description!

positional arguments:
  num1                first number to add
```

(continues on next page)

(continued from previous page)

```
  num2                second number to add
  outfile             where to store the result

optional arguments:
  -h, --help          show this help message and exit
  --num1 NUM1         first number to add (default: 0)
  --num2 NUM2         second number to add (default: 1)
  --outfile OUTFILE   where to store the result (default: ./result.txt)
  --config CONFIG     special scriptconfig option that accepts the path to a on-disk␣
↪configuration file, and loads that into this 'MyConfig' object. (default: None)
  --dump DUMP         If specified, dump this config to disk. (default: None)
  --dumps             If specified, dump this config stdout (default: False)
```

Note that keyword arguments are always available, even if the argument is marked as positional. This is because a scriptconfig object always reduces to key/value pairs — i.e. a dictionary.

See the *scriptconfig.config* module docs for details and examples on getting started as well as getting_started docs

CHAPTER

TWO

SCRIPTCONFIG

2.1 scriptconfig package

2.1.1 Submodules

2.1.1.1 scriptconfig._ubelt_repr_extension module

scriptconfig._ubelt_repr_extension.**_register_ubelt_repr_extensions**()

2.1.1.2 scriptconfig.argparse_ext module

Argparse Extensions

class scriptconfig.argparse_ext.**BooleanFlagOrKeyValAction**(*option_strings*, *dest*, *default=None*, *required=False*, *help=None*)

Bases: _StoreAction

An action that allows you to specify a boolean via a flag as per usual or a key/value pair.

This helps allow for a flexible specification of boolean values:

--flag > {'flag': True}

–flag=1 > {'flag': True} –flag True > {'flag': True} –flag True > {'flag': True} –flag False > {'flag': False} –flag 0 > {'flag': False} –no-flag > {'flag': False} –no-flag=0 > {'flag': True} –no-flag=1 > {'flag': False}

Example

```
>>> from scriptconfig.argparse_ext import *  # NOQA
>>> import argparse
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('-f', '--flag', action=BooleanFlagOrKeyValAction)
>>> print(parser.format_usage())
>>> print(parser.format_help())
>>> import shlex
>>> # Map the CLI arg string to what value we would expect to get
>>> variants = {
>>>     # Case1: you either specify the flag, or you don't
>>>     '': None,
```

(continues on next page)

(continued from previous page)

```
>>>     '--flag': True,
>>>     '--no-flag': False,
>>>     # Case1: You specify the flag as a key/value pair
>>>     '--flag=0': False,
>>>     '--flag=1': True,
>>>     '--flag True': True,
>>>     '--flag False': False,
>>>     # Case1: You specify the negated flag as a key/value pair
>>>     # (you probably shouldn't do this)
>>>     '--no-flag 0': True,
>>>     '--no-flag 1': False,
>>>     '--no-flag=True': False,
>>>     '--no-flag=False': True,
>>> }
>>> for args, want in variants.items():
>>>     args = shlex.split(args)
>>>     ns = parser.parse_known_args(args=args)[0].__dict__
>>>     print(f'args={args} -> {ns}')
>>>     assert ns['flag'] == want
```

format_usage()

_mark_parsed_argument(*parser*)

class scriptconfig.argparse_ext.**CounterOrKeyValAction**(*option_strings*, *dest*, *default=None*, *required=False*, *help=None*)

Bases: *BooleanFlagOrKeyValAction*

Extends :BooleanFlagOrKeyValAction: and will increment the value based on the number of times the flag is specified.

FIXME:
Can we get -ffff to work right?

Example

```
>>> from scriptconfig.argparse_ext import *  # NOQA
>>> import argparse
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('-f', '--flag', action=CounterOrKeyValAction)
>>> print(parser.format_usage())
>>> print(parser.format_help())
>>> import shlex
>>> # Map the CLI arg string to what value we would expect to get
>>> variants = {
>>>     # Case1: you either specify the flag, or you don't
>>>     '': None,
>>>     '--flag': True,
>>>     '--no-flag': False,
>>>     # Case1: You specify the flag as a key/value pair
>>>     '--flag=0': False,
>>>     '--flag=1': True,
```

(continues on next page)

(continued from previous page)

```
>>>     '--flag True': True,
>>>     '--flag False': False,
>>>     # Case1: You specify the negated flag as a key/value pair
>>>     # (you probably shouldn't do this)
>>>     '--no-flag 0': True,
>>>     '--no-flag 1': False,
>>>     '--no-flag=True': False,
>>>     '--no-flag=False': True,
>>>     # Multiple flag specification cases
>>>     '--flag --flag --flag': 3,
>>>     # An explicit set overwrites previous increments
>>>     '--flag --flag --flag --flag=0': 0,
>>>     # An increments modify previous explicit settings
>>>     '--flag=3 --flag --flag --flag': 6,
>>> }
>>> for args, want in variants.items():
>>>     args = shlex.split(args)
>>>     ns = parser.parse_known_args(args=args)[0].__dict__
>>>     print(f'args={args} -> {ns}')
>>>     assert ns['flag'] == want
```

class scriptconfig.argparse_ext.**RawDescriptionDefaultsHelpFormatter**(*prog*, *indent_increment=2*, *max_help_position=24*, *width=None*)

Bases: RawDescriptionHelpFormatter, ArgumentDefaultsHelpFormatter

group_name_formatter

alias of str

_concise_option_strings(*action*)

_format_action_invocation(*action*)

Custom mixin to reduce clutter from accepting fuzzy hyphens

_rich_format_action_invocation(*action*)

Mirrors _format_action_invocation but for rich-argparse

class scriptconfig.argparse_ext.**CompatArgumentParser**(**args*, ***kwargs*)

Bases: ArgumentParser

For Python 3.6-3.8 compatibility where the exit_on_error flag does not exist.

parse_known_args(*args=None*, *namespace=None*)

_parse_optional(*arg_string*)

Allow "_" or "-" on the CLI.

https://stackoverflow.com/questions/53527387/make-argparse-treat-dashes-and-underscore-identically

_get_option_tuples(*option_string*)

2.1.1.3 scriptconfig.cli module

scriptconfig.cli.**quick_cli**(*default*, *name=None*)

Quickly create a CLI

New in 0.5.2

Example

```
>>> # SCRIPT
>>> import scriptconfig as scfg
>>> default = {
>>>     'fpath': scfg.Path(None),
>>>     'modnames': scfg.Value([]),
>>> }
>>> config = scfg.quick_cli(default)
>>> print('config = {!r}'.format(config))
```

2.1.1.4 scriptconfig.config module

Write simple configs and update from CLI, kwargs, and/or json.

The `scriptconfig` provides a simple way to make configurable scripts using a combination of config files, command line arguments, and simple Python keyword arguments. A script config object is defined by creating a subclass of `Config` with a `default` dict class attribute. An instance of a custom `Config` object will behave similar a dictionary, but with a few conveniences.

Note:

- This class implements the old-style legacy Config class, new applications should favor using DataConfig instead, which has simpler boilerplate.

To get started lets consider some example usage:

Example

```
>>> import scriptconfig as scfg
>>> # In its simplest incarnation, the config class specifies default values.
>>> # For each configuration parameter.
>>> class ExampleConfig(scfg.Config):
>>>     __default__ = {
>>>         'num': 1,
>>>         'mode': 'bar',
>>>         'ignore': ['baz', 'biz'],
>>>     }
>>> # Creating an instance, starts using the defaults
>>> config = ExampleConfig()
>>> # Typically you will want to update default from a dict or file.  By
>>> # specifying cmdline=True you denote that it is ok for the contents of
>>> # `sys.argv` to override config values. Here we pass a dict to `load`.
```

(continues on next page)

(continued from previous page)

```
>>> kwargs = {'num': 2}
>>> config.load(kwargs, cmdline=False)
>>> assert config['num'] == 2
>>> # The `load` method can also be passed a json/yaml file/path.
>>> import tempfile
>>> config_fpath = tempfile.mktemp()
>>> open(config_fpath, 'w').write('{"num": 3}')
>>> config.load(config_fpath, cmdline=False)
>>> assert config['num'] == 3
>>> # It is possible to load only from CLI by setting cmdline=True
>>> # or by setting it to a custom sys.argv
>>> config.load(cmdline=['--num=4', '--mode' ,'fiz'])
>>> assert config['num'] == 4
>>> assert config['mode'] == 'fiz'
>>> # You can also just use the command line string itself
>>> config.load(cmdline='--num=4 --mode fiz')
>>> assert config['num'] == 4
>>> assert config['mode'] == 'fiz'
>>> # Note that using `config.load(cmdline=True)` will just use the
>>> # contents of sys.argv
```

Todo:

- [ ] Handle Nested Configs?
- [ ] Integrate with Hyrda
- [x] Dataclass support - See DataConfig

class scriptconfig.config.**Config**(*data=None*, *default=None*, *cmdline=False*, *_dont_call_post_init=False*)

Bases: NiceRepr, *DictLike*

Base class for custom configuration objects

A configuration that can be specified by commandline args, a yaml config file, and / or a in-code dictionary. To use, define a class variable named `__default__` and passing it to a dict of default values. You can also use special `Value` classes to denote types. You can also define a method `__post_init__`, to postprocess the arguments after this class receives them.

Basic usage is as follows.

Create a class that inherits from this class.

Assign the "__default__" class-level variable as a dictionary of options

The keys of this dictionary must be command line friendly strings.

The values of the "defaults dictionary" can be literal values or instances of the *scriptconfig.Value* class, which allows for specification of default values, type information, help strings, and aliases.

You may also implement `__post_init__` (function with that takes no args and has no return) to postprocess your results after initialization.

When creating an instance of the class the defaults variable is used to make a dictionary-like object. You can override defaults by specifying the `data` keyword argument to either a file path or another dictionary. You can also specify `cmdline=True` to allow the contents of `sys.argv` to influence the values of the new object.

An instance of the config class behaves like a dictionary, except that you cannot set keys that do not already exist (as specified in the defaults dict).

Key Methods:

- dump - dump a json representation to a file
- dumps - dump a json representation to a string
- argparse - create an `argparse.ArgumentParser` object that is defined by the defaults of this config.
- load - rewrite the values based on a filepath, dictionary, or command line contents.

Variables

- **_data** – this protected variable holds the raw state of the config object and is accessed by the dict-like
- **_default** – this protected variable maintains the default values for this config.
- **epilog** (*str*) – A class attribute that if specified will add an epilog section to the help text.

Example

```
>>> # Inherit from `Config` and assign `__default__`
>>> import scriptconfig as scfg
>>> class MyConfig(scfg.Config):
>>>     __default__ = {
>>>         'option1': scfg.Value((1, 2, 3), tuple),
>>>         'option2': 'bar',
>>>         'option3': None,
>>>     }
>>> # You can now make instances of this class
>>> config1 = MyConfig()
>>> config2 = MyConfig(default=dict(option1='baz'))
```

Parameters

- **data** (*object*) – filepath, dict, or None
- **default** (*dict | None*) – overrides the class defaults
- **cmdline** (*bool | List[str] | str | dict*) – If False, then no command line information is used. If True, then sys.argv is parsed and used. If a list of strings that used instead of sys.argv. If a string, then that is parsed using shlex and used instead

  of sys.argv.

  If a dictionary grants fine grained controls over the args passed to *Config._read_argv()*. Can contain:

  – strict (bool): defaults to False

  – argv (List[str]): defaults to None

  – special_options (bool): defaults to True

  – autocomplete (bool): defaults to False

  Defaults to False.

Note: Avoid setting `cmdline` parameter here. Instead prefer to use the `cli` classmethod to create a command line aware config instance..

`classmethod cli`(*data=None*, *default=None*, *argv=None*, *strict=True*, *cmdline=True*, *autocomplete='auto'*)

Create a commandline aware config instance.

Calls the original "load" way of creating non-dataclass config objects. This may be refactored in the future.

Parameters

- **data** (*dict | str | None*) – Values to update the configuration with. This can be a regular dictionary or a path to a yaml / json file.
- **default** (*dict | None*) – Values to update the defaults with (not the actual configuration). Note: anything passed to default will be deep copied and can be updated by argv or data if it is specified. Generally prefer to pass directly to data instead.
- **cmdline** (*bool*) – Defaults to True, which creates and uses an argparse object to interact with the command line. If set to False, then the argument parser is bypassed (useful for invoking a CLI programatically with kwargs and ignoring sys.argv).
- **argv** (*List[str]*) – if specified, ignore sys.argv and parse this instead.
- **strict** (*bool*) – if True use `parse_args` otherwise use `parse_known_args`. Defaults to True.
- **autocomplete** (*bool | str*) – if True try to enable argcomplete.

`classmethod demo`()

Create an example config class for test cases

CommandLine

```
xdoctest -m scriptconfig.config Config.demo
xdoctest -m scriptconfig.config Config.demo --cli --option1 fo
```

Example

```
>>> from scriptconfig.config import *
>>> self = Config.demo()
>>> print('self = {}'.format(self))
self = <DemoConfig({...'option1': ...}...)...>...
>>> self.argparse().print_help()
>>> # xdoc: +REQUIRES(--cli)
>>> self.load(cmdline=True)
>>> print(ub.urepr(self, nl=1))
```

`getitem`(*key*)

Dictionary-like method to get the value of a key.

Parameters

key (*str*) – the key

Returns

the associated value

Return type
Any

`setitem`(*key*, *value*)

Dictionary-like method to set the value of a key.

Parameters

- **key** (*str*) – the key
- **value** (*Any*) – the new value

`delitem`(*key*)

`keys`()

Dictionary-like keys method

Yields
str

`update_defaults`(*default*)

Update the instance-level default values

Parameters
default (*dict*) – new defaults

`load`(*data=None*, *cmdline=False*, *mode=None*, *default=None*, *strict=False*, *autocomplete=False*, *_dont_call_post_init=False*)

Updates the configuration from a given data source.

Any option can be overwritten via the command line if `cmdline` is truthy.

Parameters

- **data** (*PathLike | dict*) – Either a path to a yaml / json file or a config dict
- **cmdline** (*bool | List[str] | str | dict*) – If False, then no command line information is used. If True, then sys.argv is parsed and used. If a list of strings that used instead of sys.argv. If a string, then that is parsed using shlex and used instead

  of sys.argv.

  If a dictionary grants fine grained controls over the args passed to *Config._read_argv()*. Can contain:

  – strict (bool): defaults to False

  – argv (List[str]): defaults to None

  – special_options (bool): defaults to True

  – autocomplete (bool): defaults to False

  Defaults to False.
- **mode** (*str | None*) – Either json or yaml.
- **cmdline** (*bool | List[str] | str*) – If False, then no command line information is used. If True, then sys.argv is parsed and used. If a list of strings that used instead of sys.argv. If a string, then that is parsed using shlex and used instead of sys.argv. Defaults to False.
- **default** (*dict | None*) – updated defaults. Note: anything passed to default will be deep copied and can be updated by argv or data if it is specified. Generally prefer to pass directly to data instead.

- **strict** (*bool*) – if True an error will be raised if the command line contains unknown arguments.
- **autocomplete** (*bool*) – if True, attempts to use the autocomplete package if it is available if reading from sys.argv. Defaults to False.

Note: if cmdline=True, this will create an argument parser.

Example

```
>>> # Test load works correctly in cmdline True and False mode
>>> import scriptconfig as scfg
>>> class MyConfig(scfg.Config):
>>>     __default__ = {
>>>         'src': scfg.Value(None, help=('some help msg')),
>>>     }
>>> data = {'src': 'hi'}
>>> self = MyConfig(data=data, cmdline=False)
>>> assert self['src'] == 'hi'
>>> self = MyConfig(default=data, cmdline=True)
>>> assert self['src'] == 'hi'
>>> # In 0.5.8 and previous src fails to populate!
>>> # This is because cmdline=True overwrites data with defaults
>>> self = MyConfig(data=data, cmdline=True)
>>> assert self['src'] == 'hi', f'Got: {self}'
```

Example

```
>>> # Test load works correctly in strict mode
>>> import scriptconfig as scfg
>>> class MyConfig(scfg.Config):
>>>     __default__ = {
>>>         'src': scfg.Value(None, help=('some help msg')),
>>>     }
>>> data = {'src': 'hi'}
>>> cmdlinekw = {
>>>     'strict': True,
>>>     'argv': '--src=hello',
>>> }
>>> self = MyConfig(data=data, cmdline=cmdlinekw)
>>> cmdlinekw = {
>>>     'strict': True,
>>>     'special_options': False,
>>>     'argv': '--src=hello --extra=arg',
>>> }
>>> import pytest
>>> with pytest.raises(SystemExit):
>>>     self = MyConfig(data=data, cmdline=cmdlinekw)
```

Example

```
>>> # Test load works correctly with required
>>> import scriptconfig as scfg
>>> class MyConfig(scfg.Config):
>>>     __default__ = {
>>>         'src': scfg.Value(None, help=('some help msg'), required=True),
>>>     }
>>> cmdlinekw = {
>>>     'strict': True,
>>>     'special_options': False,
>>>     'argv': '',
>>> }
>>> import pytest
>>> with pytest.raises(Exception):
...   self = MyConfig(cmdline=cmdlinekw)
```

Example

```
>>> # Test load works correctly with alias
>>> import scriptconfig as scfg
>>> class MyConfig(scfg.Config):
>>>     __default__ = {
>>>         'opt1': scfg.Value(None),
>>>         'opt2': scfg.Value(None, alias=['arg2']),
>>>     }
>>> config1 = MyConfig(data={'opt2': 'foo'})
>>> assert config1['opt2'] == 'foo'
>>> config2 = MyConfig(data={'arg2': 'bar'})
>>> assert config2['opt2'] == 'bar'
>>> assert 'arg2' not in config2
```

_normalize_alias_key(*key*)

normalizes a single aliased key

_normalize_alias_dict(*data*)

Parameters
data (*dict*) – dictionary with keys that could be aliases

Returns
keys are normalized to be primary keys.

Return type
dict

_build_alias_map()

_read_argv(*argv=None*, *special_options=True*, *strict=False*, *autocomplete=False*)

Example

```
>>> import scriptconfig as scfg
>>> class MyConfig(scfg.Config):
>>>     description = 'my CLI description'
>>>     __default__ = {
>>>         'src':  scfg.Value(['foo'], position=1, nargs='+'),
>>>         'dry':  scfg.Value(False),
>>>         'approx':  scfg.Value(False, isflag=True, alias=['a1', 'a2']),
>>>     }
>>> self = MyConfig()
>>> # xdoctest: +REQUIRES(PY3)
>>> # Python2 argparse does a hard sys.exit instead of raise
>>> import sys
>>> if sys.version_info[0:2] < (3, 6):
>>>     # also skip on 3.5 because of dict ordering
>>>     import pytest
>>>     pytest.skip()
>>> self._read_argv(argv='')
>>> print('self = {}'.format(self))
>>> self = MyConfig()
>>> self._read_argv(argv='--src [,]')
>>> print('self = {}'.format(self))
>>> self = MyConfig()
>>> self._read_argv(argv='--src [,] --a1')
>>> print('self = {}'.format(self))
self = <MyConfig({'src': ['foo'], 'dry': False, 'approx': False})>
self = <MyConfig({'src': [], 'dry': False, 'approx': False})>
self = <MyConfig({'src': [], 'dry': False, 'approx': True})>
```

```
>>> self = MyConfig()
>>> self._read_argv(argv='p1 p2 p3')
>>> print('self = {}'.format(self))
>>> self = MyConfig()
>>> self._read_argv(argv='--src=p4,p5,p6!')
>>> print('self = {}'.format(self))
>>> self = MyConfig()
>>> self._read_argv(argv='p1 p2 p3 --src=p4,p5,p6!')
>>> print('self = {}'.format(self))
self = <MyConfig({'src': ['p1', 'p2', 'p3'], 'dry': False, 'approx': False})>
self = <MyConfig({'src': ['p4', 'p5', 'p6!'], 'dry': False, 'approx': False})>
self = <MyConfig({'src': ['p4', 'p5', 'p6!'], 'dry': False, 'approx': False})>
```

```
>>> self = MyConfig()
>>> self._read_argv(argv='p1')
>>> print('self = {}'.format(self))
>>> self = MyConfig()
>>> self._read_argv(argv='--src=p4')
>>> print('self = {}'.format(self))
>>> self = MyConfig()
>>> self._read_argv(argv='p1 --src=p4')
>>> print('self = {}'.format(self))
```

(continues on next page)

(continued from previous page)

```
self = <MyConfig({'src': ['p1'], 'dry': False, 'approx': False})>
self = <MyConfig({'src': ['p4'], 'dry': False, 'approx': False})>
self = <MyConfig({'src': ['p4'], 'dry': False, 'approx': False})>
```

```
>>> special_options = False
>>> parser = self.argparse(special_options=special_options)
>>> parser.print_help()
>>> x = parser.parse_known_args()
```

dump(*stream=None*, *mode=None*)

Write configuration file to a file or stream

Parameters

- **stream** (*FileLike | None*) – the stream to write to
- **mode** (*str | None*) – can be ‘yaml’ or ‘json’ (defaults to ‘yaml’)

dumps(*mode=None*)

Write the configuration to a text object and return it

Parameters
mode (*str | None*) – can be ‘yaml’ or ‘json’ (defaults to ‘yaml’)

Returns
str - the configuration as a string

property _description

property _epilog

property _prog

_parserkw()

Generate the kwargs for making a new argparse.ArgumentParser

port_to_dataconf()

Helper that will write the code to express this config as a DataConfig.

CommandLine

```
xdoctest -m scriptconfig.config Config.port_to_dataconf
```

Example

```
>>> import scriptconfig as scfg
>>> self = scfg.Config.demo()
>>> print(self.port_to_dataconf())
```

classmethod _write_code(*entries*, *name='MyConfig'*, *style='dataconf'*, *description=None*)

classmethod port_click(*click_main*, *name='MyConfig'*, *style='dataconf'*)

Example

@click.command() @click.option('–dataset', required=True, type=click.Path(exists=True), help='input dataset') @click.option('–deployed', required=True, type=click.Path(exists=True), help='weights file') def click_main(dataset, deployed):

...

classmethod port_argparse(*parser*, *name='MyConfig'*, *style='dataconf'*)

Generate the corresponding scriptconfig code from an existing argparse instance.

Parameters

- **parser** (*argparse.ArgumentParser*) – existing argparse parser we want to port
- **name** (*str*) – the name of the config class
- **style** (*str*) – either 'orig' or 'dataconf'

Returns

code to create a scriptconfig object that should work similarly to the existing argparse object.

Return type

str

Note: The correctness of this function is not guarenteed. This only works perfectly in simple cases, but in complex cases it may not produce 1-to-1 results, however it will provide a useful starting point.

Todo:

- [X] Handle "store_true".
- [ ] Argument groups.
- [ ] Handle mutually exclusive groups

Example

```
>>> import scriptconfig as scfg
>>> import argparse
>>> parser = argparse.ArgumentParser(description='my argparse')
>>> parser.add_argument('pos_arg1')
>>> parser.add_argument('pos_arg2', nargs='*')
>>> parser.add_argument('-t', '--true_dataset', '--test_dataset', help='path to␣
↪the groundtruth dataset', required=True)
>>> parser.add_argument('-p', '--pred_dataset', help='path to the predicted␣
↪dataset', required=True)
>>> parser.add_argument('--eval_dpath', help='path to dump results')
>>> parser.add_argument('--draw_curves', default='auto', help='flag to draw␣
↪curves or not')
>>> parser.add_argument('--score_space', default='video', help='can score in␣
↪image or video space')
>>> parser.add_argument('--workers', default='auto', help='number of parallel␣
↪scoring workers')
```

(continues on next page)

(continued from previous page)

```
>>> parser.add_argument('--draw_workers', default='auto', help='number of␣
↪parallel drawing workers')
>>> group1 = parser.add_argument_group('mygroup1')
>>> group1.add_argument('--group1_opt1', action='store_true')
>>> group1.add_argument('--group1_opt2')
>>> group2 = parser.add_argument_group()
>>> group2.add_argument('--group2_opt1', action='store_true')
>>> group2.add_argument('--group2_opt2')
>>> mutex_group3 = parser.add_mutually_exclusive_group()
>>> mutex_group3.add_argument('--mgroup3_opt1')
>>> mutex_group3.add_argument('--mgroup3_opt2')
>>> text = scfg.Config.port_argparse(parser, name='PortedConfig', style=
↪'dataconf')
>>> print(text)
>>> # Make an instance of the ported class
>>> vals = {}
>>> exec(text, vals)
>>> cls = vals['PortedConfig']
>>> self = cls(**{'true_dataset': 1, 'pred_dataset': 1})
>>> recon = self.argparse()
>>> print('recon._actions = {}'.format(ub.urepr(recon._actions, nl=1)))
```

port_to_argparse()

Attempt to make code for a nearly-equivalent argparse object.

This code only handles basic cases. Some of the scriptconfig magic is dropped so we dont need to rely on custom actions.

The idea is that sometimes we can't depend on scriptconfig, so it would be nice to be able to translate an existing scriptconfig class to the nearly equivalent argparse code.

SeeAlso:

Config.argparse() - creates a real argparse object

Returns

code to construct a similar argparse object

Return type

str

CommandLine

```
xdoctest -m scriptconfig.config Config.port_to_argparse
```

Example

```
>>> import scriptconfig as scfg
>>> class SimpleCLI(scfg.DataConfig):
>>>     data = scfg.Value(None, help='input data', position=1)
>>> self_or_cls = SimpleCLI()
>>> text = self_or_cls.port_to_argparse()
>>> print(text)
>>> # Test that the generated code is executable
>>> ns = {}
>>> exec(text, ns, ns)
>>> parser = ns['parser']
>>> args1 = parser.parse_args(['foobar'])
>>> assert args1.data == 'foobar'
>>> # Looks like we cant do positional or key/value easilly
>>> #args1 = parser.parse_args(['--data=blag'])
>>> #print('args1 = {}'.format(ub.urepr(args1, nl=1)))
```

property namespace

Access a namespace like object for compatibility with argparse

Returns
argparse.Namespace

to_omegaconf()

Creates an omegaconfig version of this.

Return type
omegaconf.OmegaConf

Example

```
>>> # xdoctest: +REQUIRES(module:omegaconf)
>>> import scriptconfig
>>> self = scriptconfig.Config.demo()
>>> oconf = self.to_omegaconf()
```

argparse(*parser=None*, *special_options=False*)

construct or update an argparse.ArgumentParser CLI parser

Parameters

- **parser** (*None | argparse.ArgumentParser*) – if specified this parser is updated with options from this config.
- **special_options** (*bool, default=False*) – adds special scriptconfig options, namely: –config, –dumps, and –dump.

Returns
a new or updated argument parser

Return type
argparse.ArgumentParser

CommandLine

```
xdoctest -m scriptconfig.config Config.argparse:0
xdoctest -m scriptconfig.config Config.argparse:1
```

Todo: A good CLI spec for lists might be

In the case where `key` ends with and =, assume the list is # given as a comma separated string with optional square brakets at # each end.

–key=[f]

In the case where `key` does not end with equals and we know # the value is supposd to be a list, then we consume arguments # until we hit the next one that starts with '–' (which means # that list items cannot start with – but they can contains # commas)

FIXME:

- In the case where we have an nargs='+' action, and we specify the option with an =, and then we give position args after it there is no way to modify behavior of the action to just look at the data in the string without modifying the ArgumentParser itself. The action object has no control over it. For example *–foo=bar baz biz* will parse as *[baz, biz]* which is really not what we want. We may be able to overload ArgumentParser to fix this.

Example

```
>>> # You can now make instances of this class
>>> import scriptconfig
>>> self = scriptconfig.Config.demo()
>>> parser = self.argparse()
>>> parser.print_help()
>>> # xdoctest: +REQUIRES(PY3)
>>> # Python2 argparse does a hard sys.exit instead of raise
>>> ns, extra = parser.parse_known_args()
```

Example

```
>>> # You can now make instances of this class
>>> import scriptconfig as scfg
>>> class MyConfig(scfg.Config):
>>>     __description__ = 'my CLI description'
>>>     __default__ = {
>>>         'path1':  scfg.Value(None, position=1, alias='src'),
>>>         'path2':  scfg.Value(None, position=2, alias='dst'),
>>>         'dry':  scfg.Value(False, isflag=True),
>>>         'approx':  scfg.Value(False, isflag=False, alias=['a1', 'a2']),
>>>     }
```

(continues on next page)

(continued from previous page)

```
>>> self = MyConfig()
>>> special_options = True
>>> parser = None
>>> parser = self.argparse(special_options=special_options)
>>> parser.print_help()
>>> self._read_argv(argv=['objection', '42', '--path1=overruled!'])
>>> print('self = {!r}'.format(self))
```

Example

```
>>> # Test required option
>>> import scriptconfig as scfg
>>> class MyConfig(scfg.Config):
>>>     __description__ = 'my CLI description'
>>>     __default__ = {
>>>         'path1':  scfg.Value(None, position=1, alias='src'),
>>>         'path2':  scfg.Value(None, position=2, alias='dst'),
>>>         'dry':  scfg.Value(False, isflag=True),
>>>         'important':  scfg.Value(False, required=True),
>>>         'approx':  scfg.Value(False, isflag=False, alias=['a1', 'a2']),
>>>     }
>>> self = MyConfig(data={'important': 1})
>>> special_options = True
>>> parser = None
>>> parser = self.argparse(special_options=special_options)
>>> parser.print_help()
>>> self._read_argv(argv=['objection', '42', '--path1=overruled!', '--
↪important=1'])
>>> print('self = {!r}'.format(self))
```

Example

```
>>> # Is it possible to the CLI as a key/val pair or an exist bool flag?
>>> import scriptconfig as scfg
>>> class MyConfig(scfg.Config):
>>>     __default__ = {
>>>         'path1':  scfg.Value(None, position=1, alias='src'),
>>>         'path2':  scfg.Value(None, position=2, alias='dst'),
>>>         'flag':  scfg.Value(None, isflag=True),
>>>     }
>>> self = MyConfig()
>>> special_options = True
>>> parser = None
>>> parser = self.argparse(special_options=special_options)
>>> parser.print_help()
>>> print(self._read_argv(argv=[], strict=True))
>>> # Test that we can specify the flag as a pure flag
>>> print(self._read_argv(argv=['--flag']))
>>> print(self._read_argv(argv=['--no-flag']))
```

(continues on next page)

(continued from previous page)

```
>>> # Test that we can specify the flag with a key/val pair
>>> print(self._read_argv(argv=['--flag', 'TRUE']))
>>> print(self._read_argv(argv=['--flag=1']))
>>> print(self._read_argv(argv=['--flag=0']))
>>> # Test flag and positional
>>> self = MyConfig()
>>> print(self._read_argv(argv=['--flag', 'TRUE', 'SUFFIX']))
>>> self = MyConfig()
>>> print(self._read_argv(argv=['PREFIX', '--flag', 'TRUE']))
>>> self = MyConfig()
>>> print(self._read_argv(argv=['--path2=PREFIX', '--flag', 'TRUE']))
```

Example

```
>>> # Test groups
>>> import scriptconfig as scfg
>>> class MyConfig(scfg.Config):
>>>     __description__ = 'my CLI description'
>>>     __default__ = {
>>>         'arg1':  scfg.Value(None, group='a'),
>>>         'arg2':  scfg.Value(None, group='a', alias='a2'),
>>>         'arg3':  scfg.Value(None, group='b'),
>>>         'arg4':  scfg.Value(None, group='b', alias='a4'),
>>>         'arg5':  scfg.Value(None, mutex_group='b', isflag=True),
>>>         'arg6':  scfg.Value(None, mutex_group='b', alias='a6'),
>>>     }
>>> self = MyConfig()
>>> parser = self.argparse()
>>> parser.print_help()
>>> print(self.port_argparse(parser))
>>> import pytest
>>> import argparse
>>> with pytest.raises(SystemExit):
>>>     self._read_argv(argv=['--arg6', '42', '--arg5', '32'])
>>> # self._read_argv(argv=['--arg6', '42', '--arg5']) # Strange, this does not␣
↪cause an mutex error
>>> self._read_argv(argv=['--arg6', '42'])
>>> self._read_argv(argv=['--arg5'])
>>> self._read_argv(argv=[])
```

default = {}

normalize()

overloadable function called after each load

scriptconfig.config.**define**(*default={}*, *name=None*)

Alternate method for defining a custom Config type

2.1.1.5 scriptconfig.dataconfig module

The new way to declare configurations.

Similar to the old-style Config objects, you simply declare a class that inherits from *scriptconfig.DataConfig* (or is wrapped by `scriptconfig.datconf()`) and declare the class variables as the config attributes much like you would write a dataclass.

Creating an instance of a `DataConfig` class works just like a regular dataclass, and nothing special happens. You can create the argument parser by using the :func:`DataConfig.cli` classmethod, which works similarly to the old-style *scriptconfig.Config* constructor.

The following is the same top-level example as in *scriptconfig.config*, but using `DataConfig` instead. It works as a drop-in replacement.

Example

```
>>> import scriptconfig as scfg
>>> # In its simplest incarnation, the config class specifies default values.
>>> # For each configuration parameter.
>>> class ExampleConfig(scfg.DataConfig):
>>>     num = 1
>>>     mode = 'bar'
>>>     ignore = ['baz', 'biz']
>>> # Creating an instance, starts using the defaults
>>> config = ExampleConfig()
>>> # Typically you will want to update default from a dict or file.  By
>>> # specifying cmdline=True you denote that it is ok for the contents of
>>> # `sys.argv` to override config values. Here we pass a dict to `load`.
>>> kwargs = {'num': 2}
>>> config.load(kwargs, cmdline=False)
>>> assert config['num'] == 2
>>> # The `load` method can also be passed a json/yaml file/path.
>>> import tempfile
>>> config_fpath = tempfile.mktemp()
>>> open(config_fpath, 'w').write('{"num": 3}')
>>> config.load(config_fpath, cmdline=False)
>>> assert config['num'] == 3
>>> # It is possbile to load only from CLI by setting cmdline=True
>>> # or by setting it to a custom sys.argv
>>> config.load(cmdline=['--num=4', '--mode' ,'fiz'])
>>> assert config['num'] == 4
>>> assert config['mode'] == 'fiz'
>>> # You can also just use the command line string itself
>>> config.load(cmdline='--num=4 --mode fiz')
>>> assert config['num'] == 4
>>> assert config['mode'] == 'fiz'
>>> # Note that using `config.load(cmdline=True)` will just use the
>>> # contents of sys.argv
```

Notes

https://docs.python.org/3/library/dataclasses.html

scriptconfig.dataconfig.**dataconf**(*cls*)

Aims to be similar to the dataclass decorator

Note: It is currently recommended to extend from the *DataConfig* object instead of decorating with `@dataconf`. These have slightly different behaviors and the former is more well-tested.

Example

```
>>> from scriptconfig.dataconfig import *  # NOQA
>>> import scriptconfig as scfg
>>> @dataconf
>>> class ExampleDataConfig2:
>>>     chip_dims = scfg.Value((256, 256), help='chip size')
>>>     time_dim = scfg.Value(3, help='number of time steps')
>>>     channels = scfg.Value('*:(red|green|blue)', help='sensor / channel code')
>>>     time_sampling = scfg.Value('soft2')
>>> cls = ExampleDataConfig2
>>> print(f'cls={cls}')
>>> self = cls()
>>> print(f'self={self}')
```

Example

```
>>> from scriptconfig.dataconfig import *  # NOQA
>>> import scriptconfig as scfg
>>> @dataconf
>>> class PathologicalConfig:
>>>     default0 = scfg.Value((256, 256), help='chip size')
>>>     default = scfg.Value((256, 256), help='chip size')
>>>     keys = [1, 2, 3]
>>>     __default__ = {
>>>         'argparse': 3.3,
>>>         'keys': [4, 5],
>>>     }
>>>     default = None
>>>     time_sampling = scfg.Value('soft2')
>>>     def foobar(self):
>>>         ...
>>> self = PathologicalConfig(1, 2, 3)
>>> print(f'self={self}')
```

FIXME: xdoctest problem. Need to be able to simulate a module global scope # Example: # >>> # Using inheritance and the decorator lets you pickle the object # >>> from scriptconfig.dataconfig import * # NOQA # >>> import scriptconfig as scfg # >>> @dataconf # >>> class PathologicalConfig2(scfg.DataConfig): # >>> default0 = scfg.Value((256, 256), help='chip size') # >>> default2 = scfg.Value((256, 256), help='chip size') # >>> #keys = [1, 2, 3] : Too much # >>> __default__3 = { # >>> 'argparse': 3.3, # >>> 'keys2': [4, 5], # >>> }

>>> default2 = None # >>> time_sampling = scfg.Value('soft2') # >>> config = PathologicalConfig2() # >>> import pickle # >>> serial = pickle.dumps(config) # >>> recon = pickle.loads(serial) # >>> assert 'locals' not in str(PathologicalConfig2)

class scriptconfig.dataconfig.**MetaDataConfig**(*name*, *bases*, *namespace*, **args*, ***kwargs*)

Bases: `MetaConfig`

This metaclass allows us to call *dataconf* when a new subclass is defined without the extra boilerplate.

class scriptconfig.dataconfig.**DataConfig**(**args*, ***kwargs*)

Bases: *Config*

Valid options: []

Parameters

- ***args** – positional arguments for this data config
- ****kwargs** – keyword arguments for this data config

classmethod legacy(*cmdline=False*, *data=None*, *default=None*, *strict=False*)

Calls the original "load" way of creating non-dataclass config objects. This may be refactored in the future.

classmethod parse_args(*args=None*, *namespace=None*)

Mimics argparse.ArgumentParser.parse_args

classmethod parse_known_args(*args=None*, *namespace=None*)

Mimics argparse.ArgumentParser.parse_known_args

default = {}

classmethod _register_main(*func*)

Register a function as the main method for this dataconfig CLI

2.1.1.6 scriptconfig.dict_like module

Defines *DictLike* which is a mixin class that makes it easier for objects to duck-type dictionaries.

class scriptconfig.dict_like.**DictLike**

Bases: `object`

An inherited class must specify the `getitem`, `setitem`, and `keys` methods.

A class is dictionary like if it has:

`__iter__`, `__len__`, `__contains__`, `__getitem__`, `items`, `keys`, `values`, `get`,

and if it should be writable it should have: `__delitem__`, `__setitem__`, `update`,

And perhaps: `copy`,

`__iter__`, `__len__`, `__contains__`, `__getitem__`, `items`, `keys`, `values`, `get`,

and if it should be writable it should have: `__delitem__`, `__setitem__`, `update`,

And perhaps: `copy`,

Example

from scriptconfig.dict_like import DictLike class DuckDict(DictLike):

def __init__(self, _data=None):

if _data is None:
_data = {}

self._data = _data

def getitem(self, key):
return self._data[key]

def keys(self):
return self._data.keys()

self = DuckDict({1: 2, 3: 4}) print(f'self._data={self._data}') cast = dict(self) print(f'cast={cast}') print(f'self={self}')

`getitem`(*key*)

Parameters
key (*Any*) – the key

Returns
the associated value

Return type
Any

`setitem`(*key*, *value*)

`delitem`(*key*)

`keys`()

Yields
str

`items`()

`values`()

`copy`()

`asdict`()

`to_dict`()

`update`(*other*)

`iteritems`()

`itervalues`()

`iterkeys`()

`get`(*key*, *default=None*)

2.1.1.7 scriptconfig.file_like module

class scriptconfig.file_like.**FileLike**(*path_or_file*, *mode='r'*)

Bases: object

Allows input to be a path or a file object

2.1.1.8 scriptconfig.modal module

The scriptconfig ModalCLI

This module defines a way to group several smaller scriptconfig CLIs into a single parent CLI that chooses between them "modally". E.g. if we define two configs: do_foo and do_bar, we use ModalCLI to define a parent program that can run one or the other. Let's make this more concrete.

CommandLine

```
xdoctest -m scriptconfig.modal __doc__:0
```

Example

```
>>> import scriptconfig as scfg
>>> #
>>> class DoFooCLI(scfg.DataConfig):
>>>     __command__ = 'do_foo'
>>>     option1 = scfg.Value(None, help='option1')
>>>     #
>>>     @classmethod
>>>     def main(cls, cmdline=1, **kwargs):
>>>         self = cls.cli(cmdline=cmdline, data=kwargs)
>>>         print('Called Foo with: ' + str(self))
>>> #
>>> class DoBarCLI(scfg.DataConfig):
>>>     __command__ = 'do_bar'
>>>     option1 = scfg.Value(None, help='option1')
>>>     #
>>>     @classmethod
>>>     def main(cls, cmdline=1, **kwargs):
>>>         self = cls.cli(cmdline=cmdline, data=kwargs)
>>>         print('Called Bar with: ' + str(self))
>>> #
>>> #
>>> class MyModalCLI(scfg.ModalCLI):
>>>     __version__ = '1.2.3'
>>>     foo = DoFooCLI
>>>     bar = DoBarCLI
>>> #
>>> modal = MyModalCLI()
>>> MyModalCLI.main(argv=['do_foo'])
>>> #MyModalCLI.main(argv=['do-foo'])
>>> MyModalCLI.main(argv=['--version'])
>>> try:
```

(continues on next page)

(continued from previous page)

```
>>>     MyModalCLI.main(argv=['--help'])
>>> except SystemExit:
>>>     print('prevent system exit due to calling --help')
```

class scriptconfig.modal.**MetaModalCLI**(*name*, *bases*, *namespace*, **args*, ***kwargs*)

Bases: type

A metaclass to help minimize boilerplate when defining a ModalCLI

class scriptconfig.modal.**ModalCLI**(*description=''*, *sub_clis=None*, *version=None*)

Bases: object

Contains multiple scriptconfig.Config items with corresponding *main* functions.

CommandLine

```
xdoctest -m scriptconfig.modal ModalCLI
```

Example

```
>>> from scriptconfig.modal import *  # NOQA
>>> import scriptconfig as scfg
>>> self = ModalCLI(description='A modal CLI')
>>> #
>>> @self.register
>>> class Command1Config(scfg.Config):
>>>     __command__ = 'command1'
>>>     __default__ = {
>>>         'foo': 'spam'
>>>     }
>>>     @classmethod
>>>     def main(cls, cmdline=1, **kwargs):
>>>         config = cls(cmdline=cmdline, data=kwargs)
>>>         print('config1 = {}'.format(ub.urepr(dict(config), nl=1)))
>>> #
>>> @self.register
>>> class Command2Config(scfg.DataConfig):
>>>     __command__ = 'command2'
>>>     foo = 'eggs'
>>>     baz = 'biz'
>>>     @classmethod
>>>     def main(cls, cmdline=1, **kwargs):
>>>         config = cls.cli(cmdline=cmdline, data=kwargs)
>>>         print('config2 = {}'.format(ub.urepr(dict(config), nl=1)))
>>> #
>>> parser = self.argparse()
>>> parser.print_help()
...
A modal CLI
...
commands:
```

(continues on next page)

(continued from previous page)

```
  {command1,command2}  specify a command to run
    command1           argparse CLI generated by scriptconfig...
    command2           argparse CLI generated by scriptconfig...
>>> self.run(argv=['command1'])
config1 = {
    'foo': 'spam',
}
>>> self.run(argv=['command2', '--baz=buz'])
config2 = {
    'foo': 'eggs',
    'baz': 'buz',
}
```

CommandLine

```
xdoctest -m scriptconfig.modal ModalCLI:1
```

Example

```
>>> # Declarative modal CLI (new in 0.7.9)
>>> import scriptconfig as scfg
>>> class MyModalCLI(scfg.ModalCLI):
>>>     #
>>>     class Command1(scfg.DataConfig):
>>>         __command__ = 'command1'
>>>         foo = scfg.Value('spam', help='spam spam spam spam')
>>>         @classmethod
>>>         def main(cls, cmdline=1, **kwargs):
>>>             config = cls.cli(cmdline=cmdline, data=kwargs)
>>>             print('config1 = {}'.format(ub.urepr(dict(config), nl=1)))
>>>     #
>>>     class Command2(scfg.DataConfig):
>>>         __command__ = 'command2'
>>>         foo = 'eggs'
>>>         baz = 'biz'
>>>         @classmethod
>>>         def main(cls, cmdline=1, **kwargs):
>>>             config = cls.cli(cmdline=cmdline, data=kwargs)
>>>             print('config2 = {}'.format(ub.urepr(dict(config), nl=1)))
>>> #
>>> MyModalCLI.main(argv=['command1'])
>>> MyModalCLI.main(argv=['command2', '--baz=buz'])
```

Example

```
>>> # Declarative modal CLI (new in 0.7.9)
>>> import scriptconfig as scfg
>>> class MyModalCLI(scfg.ModalCLI):
>>>     ...
>>> #
>>> @MyModalCLI.register
>>> class Command1(scfg.DataConfig):
>>>     __command__ = 'command1'
>>>     foo = scfg.Value('spam', help='spam spam spam spam')
>>>     @classmethod
>>>     def main(cls, cmdline=1, **kwargs):
>>>         config = cls.cli(cmdline=cmdline, data=kwargs)
>>>         print('config1 = {}'.format(ub.urepr(dict(config), nl=1)))
>>> #
>>> @MyModalCLI.register
>>> class Command2(scfg.DataConfig):
>>>     __command__ = 'command2'
>>>     foo = 'eggs'
>>>     baz = 'biz'
>>>     @classmethod
>>>     def main(cls, cmdline=1, **kwargs):
>>>         config = cls.cli(cmdline=cmdline, data=kwargs)
>>>         print('config2 = {}'.format(ub.urepr(dict(config), nl=1)))
>>> #
>>> MyModalCLI.main(argv=['command1'])
>>> MyModalCLI.main(argv=['command2', '--baz=buz'])
```

property sub_clis

classmethod register(*cli_cls*)

Parameters
cli_cli (*scriptconfig.Config*) – A CLI-aware config object to register as a sub CLI

_build_subcmd_infos()

_parserkw()

Generate the kwargs for making a new argparse.ArgumentParser

argparse(*parser=None*, *special_options=Ellipsis*)

Builds a new argparse object for this ModalCLI or extends an existing one with it.

build_parser(*parser=None*, *special_options=Ellipsis*)

Builds a new argparse object for this ModalCLI or extends an existing one with it.

classmethod main(*argv=None*, *strict=True*, *autocomplete='auto'*)

Execute the modal CLI as the main script

classmethod run(*argv=None*, *strict=True*, *autocomplete='auto'*)

Execute the modal CLI as the main script

2.1.1.9 scriptconfig.smartcast module

scriptconfig.smartcast.**smartcast**(*item*, *astype=None*, *strict=False*, *allow_split=False*)

Converts a string into a standard python type.

In many cases this is a simple alternative to *eval*. However, the syntax rules use here are more permissive and forgiving.

The *astype* can be specified to provide a type hint, otherwise we try to cast to the following types in this order: int, float, complex, bool, none, list, tuple.

Parameters

- **item** (*str | object*) – represents some data of another type.
- **astype** (*type | None*) – if None, try infer what the best type is, if astype == 'eval' then try to return *eval(item)*, Otherwise, try to cast to this type. Default to None.
- **strict** (*bool*) – if True raises a TypeError if conversion fails. Default to False.
- **allow_split** (*bool*) – if True will interpret strings with commas as sequences. Defaults to True.

Returns

some item

Return type

object

Raises

TypeError – if we cannot determine the type

Example

```
>>> # Simple cases
>>> print(repr(smartcast('?')))
>>> print(repr(smartcast('1')))
>>> print(repr(smartcast('1,2,3')))
>>> print(repr(smartcast('abc')))
>>> print(repr(smartcast('[1,2,3,4]')))
>>> print(repr(smartcast('foo.py,/etc/conf.txt,/baz/biz,blah')))
'?'
1
[1, 2, 3]
'abc'
[1, 2, 3, 4]
['foo.py', '/etc/conf.txt', '/baz/biz', 'blah']
```

```
>>> # Weird cases
>>> print(repr(smartcast('[1],2,abc,4')))
['[1]', 2, 'abc', 4]
```

Example

```
>>> assert smartcast('?') == '?'
>>> assert smartcast('1') == 1
>>> assert smartcast('1.0') == 1.0
>>> assert smartcast('1.2') == 1.2
>>> assert smartcast('True') is True
>>> assert smartcast('false') is False
>>> assert smartcast('None') is None
>>> assert smartcast('1', str) == '1'
>>> assert smartcast('1', eval) == 1
>>> assert smartcast('1', bool) is True
>>> assert smartcast('[1,2]', eval) == [1, 2]
```

Example

```
>>> def check_typed_value(item, want, astype=None):
>>>     got = smartcast(item, astype)
>>>     assert got == want and isinstance(got, type(want)), (
>>>         'Cast {!r} to {!r}, but got {!r}'.format(item, want, got))
>>> check_typed_value('?', '?')
>>> check_typed_value('1', 1)
>>> check_typed_value('1.0', 1.0)
>>> check_typed_value('1.2', 1.2)
>>> check_typed_value('True', True)
>>> check_typed_value('None', None)
>>> check_typed_value('1', 1, int)
>>> check_typed_value('1', True, bool)
>>> check_typed_value('1', 1.0, float)
>>> check_typed_value(1, 1.0, float)
>>> check_typed_value(1.0, 1.0)
>>> check_typed_value([1.0], (1.0,), 'tuple')
```

2.1.1.10 scriptconfig.value module

scriptconfig.value.**normalize_option_str**(*s*)

class scriptconfig.value.**Value**(*value=None*, *type=None*, *help=None*, *choices=None*, *position=None*, *isflag=False*, *nargs=None*, *alias=None*, *required=False*, *short_alias=None*, *group=None*, *mutex_group=None*, *tags=None*)

Bases: NiceRepr

You may set any item in the config's default to an instance of this class. Using this class allows you to declare the desired default value as well as the type that the value should be (Used when parsing sys.argv).

Variables

- **value** (*Any*) – A float, int, etc...
- **type** (*type | None*) – the "type" of the value. This is usually used if the value specified is not the type that *self.value* would usually be set to.
- **parsekw** (*dict*) – kwargs for to argparse add_argument

- **position** (*None | int*) – if an integer, then we allow this value to be a positional argument in the argparse CLI. Note, that values with the same position index will cause conflicts. Also note: positions indexes should start from 1.
- **isflag** (*bool*) – if True, args will be parsed as booleans. Default to False.
- **alias** (*List[str] | None*) – other long names (that will be prefixed with '–') that will be accepted by the argparse CLI.
- **short_alias** (*List[str] | None*) – other short names (that will be prefixed with '-') that will be accepted by the argparse CLI.
- **group** (*str | None*) – Impacts display of underlying argparse object by grouping values with the same type together. There is no other impact.
- **mutex_group** (*str | None*) – Indicates that only one of the values in a group should be given on the command line. This has no impact on python usage.
- **tags** (*Any*) – for external program use

CommandLine

```
xdoctest -m /home/joncrall/code/scriptconfig/scriptconfig/value.py Value
xdoctest -m scriptconfig.value Value
```

Example

```
>>> self = Value(None, type=float)
>>> print('self.value = {!r}'.format(self.value))
self.value = None
>>> self.update('3.3')
>>> print('self.value = {!r}'.format(self.value))
self.value = 3.3
```

update(*value*)

cast(*value*)

copy()

_to_value_kw()

Used in port-to-dataconf and port-to-argparse

classmethod _from_action(*action*, *actionid_to_groupkey*, *actionid_to_mgroupkey*, *pos_counter*)

Used in port_argparse

Example

import argparse from scriptconfig.value import * # NOQA action = argparse._StoreAction('foo', 'bar', default=3) value = Value._from_action(action, {}, {}, 0)

action = argparse._CountAction('foo', 'bar') value = Value._from_action(action, {}, {}, 0)

class scriptconfig.value.**Flag**(*value=False*, ***kwargs*)

Bases: *Value*

Exactly the same as a Value except isflag default to True

class scriptconfig.value.**Path**(*value=None*, *help=None*, *alias=None*)

Bases: *Value*

Note this is mean to be used only with scriptconfig.Config. It does NOT represent a pathlib object.

cast(*value*)

class scriptconfig.value.**PathList**(*value=None*, *type=None*, *help=None*, *choices=None*, *position=None*, *isflag=False*, *nargs=None*, *alias=None*, *required=False*, *short_alias=None*, *group=None*, *mutex_group=None*, *tags=None*)

Bases: *Value*

Can be specified as a list or as a globstr

FIXME:
will fail if there are any commas in the path name

Example

```
>>> from os.path import join
>>> path = ub.modname_to_modpath('scriptconfig', hide_init=True)
>>> globstr = join(path, '*.py')
>>> # Passing in a globstr is accepted
>>> assert len(PathList(globstr).value) > 0
>>> # Smartcast should separate these
>>> assert len(PathList('/a,/b').value) == 2
>>> # Passing in a list is accepted
>>> assert len(PathList(['/a', '/b']).value) == 2
```

cast(*value=None*)

scriptconfig.value.**_value_add_argument_to_parser**(*value*, *_value*, *self*, *parser*, *key*, *fuzzy_hyphens=0*)

POC for a new simplified way for a value to add itself as an argument to a parser.

Parameters

- **value** (*Any*) – the unwrapped default value
- **_value** (*Value*) – the value metadata

scriptconfig.value.**_value_add_argument_kw**(*value*, *_value*, *self*, *key*, *fuzzy_hyphens=0*)

TODO: resolve with *_value_add_argument_to_parser()*. This just creates one or more kwargs for add_argument. (Depending on how many variants of the argument we want).

Parameters

- **value** (*Any*) – the unwrapped default value

- **_value** (*Value*) – the value metadata

Returns
special keys to the method name, args, kwargs invocations.

Return type
Dict[str, Tuple[str, Tuple, Dict]]

scriptconfig.value.**_resolve_alias**(*name*, *_value*, *fuzzy_hyphens*)

scriptconfig.value.**scfg_isinstance**(*item*, *cls*)

use instead isinstance for scfg types when reloading

Parameters

- **item** (*object*) – instance to check
- **cls** (*type*) – class to check against

Returns
bool

scriptconfig.value.**_maker_smart_parse_action**(*self*)

class scriptconfig.value.**CodeRepr**

Bases: str

2.1.2 Module contents

2.1.2.1 ScriptConfig

The goal of `scriptconfig` is to make it easy to be able to define a CLI by **simply defining a dictionary**. Thie enables you to write simple configs and update from CLI, kwargs, and/or json.

The pattern is simple:

1. Create a class that inherits from *scriptconfig.Config*
2. Create a class variable dictionary named `default`
3. The keys are the names of your arguments, and the values are the defaults.
4. Create an instance of your config object. If you pass `cmdline=True` as an argument, it will autopopulate itself from the command line.

Here is an example for a simple calculator program:

```
import scriptconfig as scfg


class MyConfig(scfg.Config):
    'The docstring becomes the CLI description!'
    default = {
        'num1': 0,
        'num2': 1,
        'outfile': './result.txt',
    }
```

(continues on next page)

(continued from previous page)

```
def main():
    config = MyConfig(cmdline=True)
    result = config['num1'] + config['num2']
    with open(config['outfile'], 'w') as file:
        file.write(str(result))


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

If the above is written to a file *calc.py*, it can be be called like this.

```
python calc.py --num1=3 --num2=4 --outfile=/dev/stdout
```

It is possible to gain finer control over the CLI by specifying the values in `default` as a *scriptconfig.Value*, where you can specify a help message, the expected variable type, if it is a positional variable, alias parameters for the command line, and more.

The important thing that gives scriptconfig an edge over things like `argparse` is that it is trivial to disable the `cmdline` flag and pass explicit arguments into your function as a dictionary. Thus you can write you scripts in such a way that they are callable from Python or from a CLI via with an API that corresponds 1-to-1!

A more complex example version of the above code might look like this

```
import scriptconfig as scfg


class MyConfig(scfg.Config):
    '''
    The docstring becomes the CLI description!
    '''
    default = {
        'num1': scfg.Value(0, type=float, help='first number to add', position=1),
        'num2': scfg.Value(1, type=float, help='second number to add', position=2),
        'outfile': scfg.Value('./result.txt', help='where to store the result',␣
↪position=3),
    }


def main(cmdline=1, **kwargs):
    '''
    Example:
        >>> # This is much easier to test than argparse code
        >>> kwargs = {'num1': 42, 'num2': 23, 'outfile': 'foo.out'}
        >>> cmdline = 0
        >>> main(cmdline=cmdline, **kwargs)
        >>> with open('foo.out') as file:
        >>>     assert file.read() == '65'
    '''
    config = MyConfig(cmdline=True, data=kwargs)
    result = config['num1'] + config['num2']
    with open(config['outfile'], 'w') as file:
        file.write(str(result))
```

(continues on next page)

(continued from previous page)

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

This code can be called with positional arguments:

```
python calc.py 33 44 /dev/stdout
```

The help text for this program (via `python calc.py --help`) looks like this:

```
usage: MyConfig [-h] [--num1 NUM1] [--num2 NUM2] [--outfile OUTFILE] [--config CONFIG] [-
↪-dump DUMP] [--dumps] num1 num2 outfile

The docstring becomes the CLI description!

positional arguments:
  num1                first number to add
  num2                second number to add
  outfile             where to store the result

optional arguments:
  -h, --help          show this help message and exit
  --num1 NUM1         first number to add (default: 0)
  --num2 NUM2         second number to add (default: 1)
  --outfile OUTFILE   where to store the result (default: ./result.txt)
  --config CONFIG     special scriptconfig option that accepts the path to a on-disk␣
↪configuration file, and loads that into this 'MyConfig' object. (default: None)
  --dump DUMP         If specified, dump this config to disk. (default: None)
  --dumps             If specified, dump this config stdout (default: False)
```

Note that keyword arguments are always available, even if the argument is marked as positional. This is because a scriptconfig object always reduces to key/value pairs — i.e. a dictionary.

See the *scriptconfig.config* module docs for details and examples on getting started as well as getting_started docs

class scriptconfig.**Config**(*data=None*, *default=None*, *cmdline=False*, *_dont_call_post_init=False*)

Bases: NiceRepr, *DictLike*

Base class for custom configuration objects

A configuration that can be specified by commandline args, a yaml config file, and / or a in-code dictionary. To use, define a class variable named `__default__` and passing it to a dict of default values. You can also use special `Value` classes to denote types. You can also define a method `__post_init__`, to postprocess the arguments after this class receives them.

Basic usage is as follows.

Create a class that inherits from this class.

Assign the "__default__" class-level variable as a dictionary of options

The keys of this dictionary must be command line friendly strings.

The values of the "defaults dictionary" can be literal values or instances of the *scriptconfig.Value* class, which allows for specification of default values, type information, help strings, and aliases.

You may also implement `__post_init__` (function with that takes no args and has no return) to postprocess your results after initialization.

When creating an instance of the class the defaults variable is used to make a dictionary-like object. You can override defaults by specifying the `data` keyword argument to either a file path or another dictionary. You can also specify `cmdline=True` to allow the contents of `sys.argv` to influence the values of the new object.

An instance of the config class behaves like a dictionary, except that you cannot set keys that do not already exist (as specified in the defaults dict).

Key Methods:

- dump - dump a json representation to a file
- dumps - dump a json representation to a string
- argparse - create an `argparse.ArgumentParser` object that is defined by the defaults of this config.
- load - rewrite the values based on a filepath, dictionary, or command line contents.

Variables

- **`_data`** – this protected variable holds the raw state of the config object and is accessed by the dict-like
- **`_default`** – this protected variable maintains the default values for this config.
- **`epilog`** (*str*) – A class attribute that if specified will add an epilog section to the help text.

Example

```
>>> # Inherit from `Config` and assign `__default__`
>>> import scriptconfig as scfg
>>> class MyConfig(scfg.Config):
>>>     __default__ = {
>>>         'option1': scfg.Value((1, 2, 3), tuple),
>>>         'option2': 'bar',
>>>         'option3': None,
>>>     }
>>> # You can now make instances of this class
>>> config1 = MyConfig()
>>> config2 = MyConfig(default=dict(option1='baz'))
```

Parameters

- **data** (*object*) – filepath, dict, or None
- **default** (*dict | None*) – overrides the class defaults
- **cmdline** (*bool | List[str] | str | dict*) – If False, then no command line information is used. If True, then sys.argv is parsed and used. If a list of strings that used instead of sys.argv. If a string, then that is parsed using shlex and used instead

  of sys.argv.

  If a dictionary grants fine grained controls over the args passed to *Config._read_argv()*. Can contain:

  – strict (bool): defaults to False

  – argv (List[str]): defaults to None

- special_options (bool): defaults to True
- autocomplete (bool): defaults to False

Defaults to False.

Note: Avoid setting `cmdline` parameter here. Instead prefer to use the `cli` classmethod to create a command line aware config instance..

`classmethod cli`(*data=None*, *default=None*, *argv=None*, *strict=True*, *cmdline=True*, *autocomplete='auto'*)

Create a commandline aware config instance.

Calls the original "load" way of creating non-dataclass config objects. This may be refactored in the future.

Parameters

- **data** (*dict | str | None*) – Values to update the configuration with. This can be a regular dictionary or a path to a yaml / json file.
- **default** (*dict | None*) – Values to update the defaults with (not the actual configuration). Note: anything passed to default will be deep copied and can be updated by argv or data if it is specified. Generally prefer to pass directly to data instead.
- **cmdline** (*bool*) – Defaults to True, which creates and uses an argparse object to interact with the command line. If set to False, then the argument parser is bypassed (useful for invoking a CLI programatically with kwargs and ignoring sys.argv).
- **argv** (*List[str]*) – if specified, ignore sys.argv and parse this instead.
- **strict** (*bool*) – if True use `parse_args` otherwise use `parse_known_args`. Defaults to True.
- **autocomplete** (*bool | str*) – if True try to enable argcomplete.

`classmethod demo`()

Create an example config class for test cases

CommandLine

```
xdoctest -m scriptconfig.config Config.demo
xdoctest -m scriptconfig.config Config.demo --cli --option1 fo
```

Example

```
>>> from scriptconfig.config import *
>>> self = Config.demo()
>>> print('self = {}'.format(self))
self = <DemoConfig({...'option1': ...}...)...>...
>>> self.argparse().print_help()
>>> # xdoc: +REQUIRES(--cli)
>>> self.load(cmdline=True)
>>> print(ub.urepr(self, nl=1))
```

`getitem`(*key*)

Dictionary-like method to get the value of a key.

Parameters
key (*str*) – the key

Returns
the associated value

Return type
Any

`setitem`(*key*, *value*)

Dictionary-like method to set the value of a key.

Parameters

- **key** (*str*) – the key
- **value** (*Any*) – the new value

`delitem`(*key*)

`keys`()

Dictionary-like keys method

Yields
str

`update_defaults`(*default*)

Update the instance-level default values

Parameters
default (*dict*) – new defaults

`load`(*data=None*, *cmdline=False*, *mode=None*, *default=None*, *strict=False*, *autocomplete=False*, *_dont_call_post_init=False*)

Updates the configuration from a given data source.

Any option can be overwritten via the command line if `cmdline` is truthy.

Parameters

- **data** (*PathLike | dict*) – Either a path to a yaml / json file or a config dict
- **cmdline** (*bool | List[str] | str | dict*) – If False, then no command line information is used. If True, then sys.argv is parsed and used. If a list of strings that used instead of sys.argv. If a string, then that is parsed using shlex and used instead

  of sys.argv.

  If a dictionary grants fine grained controls over the args passed to *Config._read_argv()*. Can contain:

  – strict (bool): defaults to False

  – argv (List[str]): defaults to None

  – special_options (bool): defaults to True

  – autocomplete (bool): defaults to False

  Defaults to False.

- **mode** (*str | None*) – Either json or yaml.

- **cmdline** (*bool | List[str] | str*) – If False, then no command line information is used. If True, then sys.argv is parsed and used. If a list of strings that used instead of sys.argv. If a string, then that is parsed using shlex and used instead of sys.argv. Defaults to False.
- **default** (*dict | None*) – updated defaults. Note: anything passed to default will be deep copied and can be updated by argv or data if it is specified. Generally prefer to pass directly to data instead.
- **strict** (*bool*) – if True an error will be raised if the command line contains unknown arguments.
- **autocomplete** (*bool*) – if True, attempts to use the autocomplete package if it is available if reading from sys.argv. Defaults to False.

Note: if cmdline=True, this will create an argument parser.

Example

```
>>> # Test load works correctly in cmdline True and False mode
>>> import scriptconfig as scfg
>>> class MyConfig(scfg.Config):
>>>     __default__ = {
>>>         'src': scfg.Value(None, help=('some help msg')),
>>>     }
>>> data = {'src': 'hi'}
>>> self = MyConfig(data=data, cmdline=False)
>>> assert self['src'] == 'hi'
>>> self = MyConfig(default=data, cmdline=True)
>>> assert self['src'] == 'hi'
>>> # In 0.5.8 and previous src fails to populate!
>>> # This is because cmdline=True overwrites data with defaults
>>> self = MyConfig(data=data, cmdline=True)
>>> assert self['src'] == 'hi', f'Got: {self}'
```

Example

```
>>> # Test load works correctly in strict mode
>>> import scriptconfig as scfg
>>> class MyConfig(scfg.Config):
>>>     __default__ = {
>>>         'src': scfg.Value(None, help=('some help msg')),
>>>     }
>>> data = {'src': 'hi'}
>>> cmdlinekw = {
>>>     'strict': True,
>>>     'argv': '--src=hello',
>>> }
>>> self = MyConfig(data=data, cmdline=cmdlinekw)
>>> cmdlinekw = {
>>>     'strict': True,
>>>     'special_options': False,
```

(continues on next page)

(continued from previous page)

```
>>>     'argv': '--src=hello --extra=arg',
>>> }
>>> import pytest
>>> with pytest.raises(SystemExit):
>>>     self = MyConfig(data=data, cmdline=cmdlinekw)
```

Example

```
>>> # Test load works correctly with required
>>> import scriptconfig as scfg
>>> class MyConfig(scfg.Config):
>>>     __default__ = {
>>>         'src': scfg.Value(None, help=('some help msg'), required=True),
>>>     }
>>> cmdlinekw = {
>>>     'strict': True,
>>>     'special_options': False,
>>>     'argv': '',
>>> }
>>> import pytest
>>> with pytest.raises(Exception):
...   self = MyConfig(cmdline=cmdlinekw)
```

Example

```
>>> # Test load works correctly with alias
>>> import scriptconfig as scfg
>>> class MyConfig(scfg.Config):
>>>     __default__ = {
>>>         'opt1': scfg.Value(None),
>>>         'opt2': scfg.Value(None, alias=['arg2']),
>>>     }
>>> config1 = MyConfig(data={'opt2': 'foo'})
>>> assert config1['opt2'] == 'foo'
>>> config2 = MyConfig(data={'arg2': 'bar'})
>>> assert config2['opt2'] == 'bar'
>>> assert 'arg2' not in config2
```

_normalize_alias_key(*key*)

normalizes a single aliased key

_normalize_alias_dict(*data*)

Parameters

data (*dict*) – dictionary with keys that could be aliases

Returns

keys are normalized to be primary keys.

Return type

dict

_build_alias_map()

_read_argv(*argv=None*, *special_options=True*, *strict=False*, *autocomplete=False*)

Example

```
>>> import scriptconfig as scfg
>>> class MyConfig(scfg.Config):
>>>     description = 'my CLI description'
>>>     __default__ = {
>>>         'src':  scfg.Value(['foo'], position=1, nargs='+'),
>>>         'dry':  scfg.Value(False),
>>>         'approx':  scfg.Value(False, isflag=True, alias=['a1', 'a2']),
>>>     }
>>> self = MyConfig()
>>> # xdoctest: +REQUIRES(PY3)
>>> # Python2 argparse does a hard sys.exit instead of raise
>>> import sys
>>> if sys.version_info[0:2] < (3, 6):
>>>     # also skip on 3.5 because of dict ordering
>>>     import pytest
>>>     pytest.skip()
>>> self._read_argv(argv='')
>>> print('self = {}'.format(self))
>>> self = MyConfig()
>>> self._read_argv(argv='--src [,]')
>>> print('self = {}'.format(self))
>>> self = MyConfig()
>>> self._read_argv(argv='--src [,] --a1')
>>> print('self = {}'.format(self))
self = <MyConfig({'src': ['foo'], 'dry': False, 'approx': False})>
self = <MyConfig({'src': [], 'dry': False, 'approx': False})>
self = <MyConfig({'src': [], 'dry': False, 'approx': True})>
```

```
>>> self = MyConfig()
>>> self._read_argv(argv='p1 p2 p3')
>>> print('self = {}'.format(self))
>>> self = MyConfig()
>>> self._read_argv(argv='--src=p4,p5,p6!')
>>> print('self = {}'.format(self))
>>> self = MyConfig()
>>> self._read_argv(argv='p1 p2 p3 --src=p4,p5,p6!')
>>> print('self = {}'.format(self))
self = <MyConfig({'src': ['p1', 'p2', 'p3'], 'dry': False, 'approx': False})>
self = <MyConfig({'src': ['p4', 'p5', 'p6!'], 'dry': False, 'approx': False})>
self = <MyConfig({'src': ['p4', 'p5', 'p6!'], 'dry': False, 'approx': False})>
```

```
>>> self = MyConfig()
>>> self._read_argv(argv='p1')
>>> print('self = {}'.format(self))
>>> self = MyConfig()
>>> self._read_argv(argv='--src=p4')
```

(continues on next page)

(continued from previous page)

```
>>> print('self = {}'.format(self))
>>> self = MyConfig()
>>> self._read_argv(argv='p1 --src=p4')
>>> print('self = {}'.format(self))
self = <MyConfig({'src': ['p1'], 'dry': False, 'approx': False})>
self = <MyConfig({'src': ['p4'], 'dry': False, 'approx': False})>
self = <MyConfig({'src': ['p4'], 'dry': False, 'approx': False})>
```

```
>>> special_options = False
>>> parser = self.argparse(special_options=special_options)
>>> parser.print_help()
>>> x = parser.parse_known_args()
```

dump(*stream=None*, *mode=None*)

Write configuration file to a file or stream

Parameters

- **stream** (*FileLike | None*) – the stream to write to
- **mode** (*str | None*) – can be ‘yaml’ or ‘json’ (defaults to ‘yaml’)

dumps(*mode=None*)

Write the configuration to a text object and return it

Parameters

mode (*str | None*) – can be ‘yaml’ or ‘json’ (defaults to ‘yaml’)

Returns

str - the configuration as a string

property _description

property _epilog

property _prog

_parserkw()

Generate the kwargs for making a new argparse.ArgumentParser

port_to_dataconf()

Helper that will write the code to express this config as a DataConfig.

CommandLine

```
xdoctest -m scriptconfig.config Config.port_to_dataconf
```

Example

```
>>> import scriptconfig as scfg
>>> self = scfg.Config.demo()
>>> print(self.port_to_dataconf())
```

classmethod _write_code(*entries*, *name='MyConfig'*, *style='dataconf'*, *description=None*)

classmethod port_click(*click_main*, *name='MyConfig'*, *style='dataconf'*)

Example

@click.command() @click.option('–dataset', required=True, type=click.Path(exists=True), help='input dataset') @click.option('–deployed', required=True, type=click.Path(exists=True), help='weights file') def click_main(dataset, deployed):

...

classmethod port_argparse(*parser*, *name='MyConfig'*, *style='dataconf'*)

Generate the corresponding scriptconfig code from an existing argparse instance.

Parameters

- **parser** (*argparse.ArgumentParser*) – existing argparse parser we want to port
- **name** (*str*) – the name of the config class
- **style** (*str*) – either 'orig' or 'dataconf'

Returns

code to create a scriptconfig object that should work similarly to the existing argparse object.

Return type

str

Note: The correctness of this function is not guarenteed. This only works perfectly in simple cases, but in complex cases it may not produce 1-to-1 results, however it will provide a useful starting point.

Todo:

- [X] Handle "store_true".
- [ ] Argument groups.
- [ ] Handle mutually exclusive groups

Example

```
>>> import scriptconfig as scfg
>>> import argparse
>>> parser = argparse.ArgumentParser(description='my argparse')
>>> parser.add_argument('pos_arg1')
>>> parser.add_argument('pos_arg2', nargs='*')
>>> parser.add_argument('-t', '--true_dataset', '--test_dataset', help='path to␣
↪the groundtruth dataset', required=True)
>>> parser.add_argument('-p', '--pred_dataset', help='path to the predicted␣
↪dataset', required=True)
>>> parser.add_argument('--eval_dpath', help='path to dump results')
>>> parser.add_argument('--draw_curves', default='auto', help='flag to draw␣
↪curves or not')
>>> parser.add_argument('--score_space', default='video', help='can score in␣
↪image or video space')
>>> parser.add_argument('--workers', default='auto', help='number of parallel␣
↪scoring workers')
>>> parser.add_argument('--draw_workers', default='auto', help='number of␣
↪parallel drawing workers')
>>> group1 = parser.add_argument_group('mygroup1')
>>> group1.add_argument('--group1_opt1', action='store_true')
>>> group1.add_argument('--group1_opt2')
>>> group2 = parser.add_argument_group()
>>> group2.add_argument('--group2_opt1', action='store_true')
>>> group2.add_argument('--group2_opt2')
>>> mutex_group3 = parser.add_mutually_exclusive_group()
>>> mutex_group3.add_argument('--mgroup3_opt1')
>>> mutex_group3.add_argument('--mgroup3_opt2')
>>> text = scfg.Config.port_argparse(parser, name='PortedConfig', style=
↪'dataconf')
>>> print(text)
>>> # Make an instance of the ported class
>>> vals = {}
>>> exec(text, vals)
>>> cls = vals['PortedConfig']
>>> self = cls(**{'true_dataset': 1, 'pred_dataset': 1})
>>> recon = self.argparse()
>>> print('recon._actions = {}'.format(ub.urepr(recon._actions, nl=1)))
```

port_to_argparse()

Attempt to make code for a nearly-equivalent argparse object.

This code only handles basic cases. Some of the scriptconfig magic is dropped so we dont need to rely on custom actions.

The idea is that sometimes we can't depend on scriptconfig, so it would be nice to be able to translate an existing scriptconfig class to the nearly equivalent argparse code.

SeeAlso:

Config.argparse() - creates a real argparse object

Returns

code to construct a similar argparse object

Return type
str

CommandLine

```
xdoctest -m scriptconfig.config Config.port_to_argparse
```

Example

```
>>> import scriptconfig as scfg
>>> class SimpleCLI(scfg.DataConfig):
>>>     data = scfg.Value(None, help='input data', position=1)
>>> self_or_cls = SimpleCLI()
>>> text = self_or_cls.port_to_argparse()
>>> print(text)
>>> # Test that the generated code is executable
>>> ns = {}
>>> exec(text, ns, ns)
>>> parser = ns['parser']
>>> args1 = parser.parse_args(['foobar'])
>>> assert args1.data == 'foobar'
>>> # Looks like we cant do positional or key/value easilly
>>> #args1 = parser.parse_args(['--data=blag'])
>>> #print('args1 = {}'.format(ub.urepr(args1, nl=1)))
```

property namespace

Access a namespace like object for compatibility with argparse

Returns
argparse.Namespace

to_omegaconf()

Creates an omegaconfig version of this.

Return type
omegaconf.OmegaConf

Example

```
>>> # xdoctest: +REQUIRES(module:omegaconf)
>>> import scriptconfig
>>> self = scriptconfig.Config.demo()
>>> oconf = self.to_omegaconf()
```

argparse(*parser=None*, *special_options=False*)

construct or update an argparse.ArgumentParser CLI parser

Parameters

- **parser** (*None | argparse.ArgumentParser*) – if specified this parser is updated with options from this config.

- **special_options** (*bool, default=False*) – adds special scriptconfig options, namely: –config, –dumps, and –dump.

Returns
a new or updated argument parser

Return type
argparse.ArgumentParser

CommandLine

```
xdoctest -m scriptconfig.config Config.argparse:0
xdoctest -m scriptconfig.config Config.argparse:1
```

Todo: A good CLI spec for lists might be

In the case where `key` ends with and =, assume the list is # given as a comma separated string with optional square brakets at # each end.

–key=[f]

In the case where `key` does not end with equals and we know # the value is supposd to be a list, then we consume arguments # until we hit the next one that starts with '–' (which means # that list items cannot start with – but they can contains # commas)

FIXME:

- In the case where we have an nargs='+' action, and we specify the option with an =, and then we give position args after it there is no way to modify behavior of the action to just look at the data in the string without modifying the ArgumentParser itself. The action object has no control over it. For example *–foo=bar baz biz* will parse as *[baz, biz]* which is really not what we want. We may be able to overload ArgumentParser to fix this.

Example

```
>>> # You can now make instances of this class
>>> import scriptconfig
>>> self = scriptconfig.Config.demo()
>>> parser = self.argparse()
>>> parser.print_help()
>>> # xdoctest: +REQUIRES(PY3)
>>> # Python2 argparse does a hard sys.exit instead of raise
>>> ns, extra = parser.parse_known_args()
```

Example

```
>>> # You can now make instances of this class
>>> import scriptconfig as scfg
>>> class MyConfig(scfg.Config):
>>>     __description__ = 'my CLI description'
>>>     __default__ = {
>>>         'path1':  scfg.Value(None, position=1, alias='src'),
>>>         'path2':  scfg.Value(None, position=2, alias='dst'),
>>>         'dry':  scfg.Value(False, isflag=True),
>>>         'approx':  scfg.Value(False, isflag=False, alias=['a1', 'a2']),
>>>     }
>>> self = MyConfig()
>>> special_options = True
>>> parser = None
>>> parser = self.argparse(special_options=special_options)
>>> parser.print_help()
>>> self._read_argv(argv=['objection', '42', '--path1=overruled!'])
>>> print('self = {!r}'.format(self))
```

Example

```
>>> # Test required option
>>> import scriptconfig as scfg
>>> class MyConfig(scfg.Config):
>>>     __description__ = 'my CLI description'
>>>     __default__ = {
>>>         'path1':  scfg.Value(None, position=1, alias='src'),
>>>         'path2':  scfg.Value(None, position=2, alias='dst'),
>>>         'dry':  scfg.Value(False, isflag=True),
>>>         'important':  scfg.Value(False, required=True),
>>>         'approx':  scfg.Value(False, isflag=False, alias=['a1', 'a2']),
>>>     }
>>> self = MyConfig(data={'important': 1})
>>> special_options = True
>>> parser = None
>>> parser = self.argparse(special_options=special_options)
>>> parser.print_help()
>>> self._read_argv(argv=['objection', '42', '--path1=overruled!', '--
↪important=1'])
>>> print('self = {!r}'.format(self))
```

Example

```
>>> # Is it possible to the CLI as a key/val pair or an exist bool flag?
>>> import scriptconfig as scfg
>>> class MyConfig(scfg.Config):
>>>     __default__ = {
>>>         'path1':  scfg.Value(None, position=1, alias='src'),
>>>         'path2':  scfg.Value(None, position=2, alias='dst'),
>>>         'flag':  scfg.Value(None, isflag=True),
>>>     }
>>> self = MyConfig()
>>> special_options = True
>>> parser = None
>>> parser = self.argparse(special_options=special_options)
>>> parser.print_help()
>>> print(self._read_argv(argv=[], strict=True))
>>> # Test that we can specify the flag as a pure flag
>>> print(self._read_argv(argv=['--flag']))
>>> print(self._read_argv(argv=['--no-flag']))
>>> # Test that we can specify the flag with a key/val pair
>>> print(self._read_argv(argv=['--flag', 'TRUE']))
>>> print(self._read_argv(argv=['--flag=1']))
>>> print(self._read_argv(argv=['--flag=0']))
>>> # Test flag and positional
>>> self = MyConfig()
>>> print(self._read_argv(argv=['--flag', 'TRUE', 'SUFFIX']))
>>> self = MyConfig()
>>> print(self._read_argv(argv=['PREFIX', '--flag', 'TRUE']))
>>> self = MyConfig()
>>> print(self._read_argv(argv=['--path2=PREFIX', '--flag', 'TRUE']))
```

Example

```
>>> # Test groups
>>> import scriptconfig as scfg
>>> class MyConfig(scfg.Config):
>>>     __description__ = 'my CLI description'
>>>     __default__ = {
>>>         'arg1':  scfg.Value(None, group='a'),
>>>         'arg2':  scfg.Value(None, group='a', alias='a2'),
>>>         'arg3':  scfg.Value(None, group='b'),
>>>         'arg4':  scfg.Value(None, group='b', alias='a4'),
>>>         'arg5':  scfg.Value(None, mutex_group='b', isflag=True),
>>>         'arg6':  scfg.Value(None, mutex_group='b', alias='a6'),
>>>     }
>>> self = MyConfig()
>>> parser = self.argparse()
>>> parser.print_help()
>>> print(self.port_argparse(parser))
>>> import pytest
>>> import argparse
```

(continues on next page)

(continued from previous page)

```
>>> with pytest.raises(SystemExit):
>>>     self._read_argv(argv=['--arg6', '42', '--arg5', '32'])
>>> # self._read_argv(argv=['--arg6', '42', '--arg5']) # Strange, this does not␣
↪cause an mutex error
>>> self._read_argv(argv=['--arg6', '42'])
>>> self._read_argv(argv=['--arg5'])
>>> self._read_argv(argv=[])
```

default = {}

normalize()

overloadable function called after each load

class scriptconfig.**DataConfig**(**args*, ***kwargs*)

Bases: *Config*

Valid options: []

Parameters

- ***args** – positional arguments for this data config
- ****kwargs** – keyword arguments for this data config

classmethod legacy(*cmdline=False*, *data=None*, *default=None*, *strict=False*)

Calls the original "load" way of creating non-dataclass config objects. This may be refactored in the future.

classmethod parse_args(*args=None*, *namespace=None*)

Mimics argparse.ArgumentParser.parse_args

classmethod parse_known_args(*args=None*, *namespace=None*)

Mimics argparse.ArgumentParser.parse_known_args

default = {}

classmethod _register_main(*func*)

Register a function as the main method for this dataconfig CLI

class scriptconfig.**Path**(*value=None*, *help=None*, *alias=None*)

Bases: *Value*

Note this is mean to be used only with scriptconfig.Config. It does NOT represent a pathlib object.

cast(*value*)

class scriptconfig.**PathList**(*value=None*, *type=None*, *help=None*, *choices=None*, *position=None*, *isflag=False*, *nargs=None*, *alias=None*, *required=False*, *short_alias=None*, *group=None*, *mutex_group=None*, *tags=None*)

Bases: *Value*

Can be specified as a list or as a globstr

FIXME:
will fail if there are any commas in the path name

Example

```
>>> from os.path import join
>>> path = ub.modname_to_modpath('scriptconfig', hide_init=True)
>>> globstr = join(path, '*.py')
>>> # Passing in a globstr is accepted
>>> assert len(PathList(globstr).value) > 0
>>> # Smartcast should separate these
>>> assert len(PathList('/a,/b').value) == 2
>>> # Passing in a list is accepted
>>> assert len(PathList(['/a', '/b']).value) == 2
```

cast(*value=None*)

class scriptconfig.**Value**(*value=None*, *type=None*, *help=None*, *choices=None*, *position=None*, *isflag=False*, *nargs=None*, *alias=None*, *required=False*, *short_alias=None*, *group=None*, *mutex_group=None*, *tags=None*)

Bases: NiceRepr

You may set any item in the config's default to an instance of this class. Using this class allows you to declare the desired default value as well as the type that the value should be (Used when parsing sys.argv).

Variables

- **value** (*Any*) – A float, int, etc...
- **type** (*type | None*) – the "type" of the value. This is usually used if the value specified is not the type that *self.value* would usually be set to.
- **parsekw** (*dict*) – kwargs for to argparse add_argument
- **position** (*None | int*) – if an integer, then we allow this value to be a positional argument in the argparse CLI. Note, that values with the same position index will cause conflicts. Also note: positions indexes should start from 1.
- **isflag** (*bool*) – if True, args will be parsed as booleans. Default to False.
- **alias** (*List[str] | None*) – other long names (that will be prefixed with '–') that will be accepted by the argparse CLI.
- **short_alias** (*List[str] | None*) – other short names (that will be prefixed with '-') that will be accepted by the argparse CLI.
- **group** (*str | None*) – Impacts display of underlying argparse object by grouping values with the same type together. There is no other impact.
- **mutex_group** (*str | None*) – Indicates that only one of the values in a group should be given on the command line. This has no impact on python usage.
- **tags** (*Any*) – for external program use

CommandLine

```
xdoctest -m /home/joncrall/code/scriptconfig/scriptconfig/value.py Value
xdoctest -m scriptconfig.value Value
```

Example

```
>>> self = Value(None, type=float)
>>> print('self.value = {!r}'.format(self.value))
self.value = None
>>> self.update('3.3')
>>> print('self.value = {!r}'.format(self.value))
self.value = 3.3
```

update(*value*)

cast(*value*)

copy()

_to_value_kw()

Used in port-to-dataconf and port-to-argparse

classmethod _from_action(*action*, *actionid_to_groupkey*, *actionid_to_mgroupkey*, *pos_counter*)

Used in port_argparse

Example

import argparse from scriptconfig.value import * # NOQA action = argparse._StoreAction('foo', 'bar', default=3) value = Value._from_action(action, {}, {}, 0)

action = argparse._CountAction('foo', 'bar') value = Value._from_action(action, {}, {}, 0)

scriptconfig.**dataconf**(*cls*)

Aims to be similar to the dataclass decorator

Note: It is currently recommended to extend from the *DataConfig* object instead of decorating with `@dataconf`. These have slightly different behaviors and the former is more well-tested.

Example

```
>>> from scriptconfig.dataconfig import *  # NOQA
>>> import scriptconfig as scfg
>>> @dataconf
>>> class ExampleDataConfig2:
>>>     chip_dims = scfg.Value((256, 256), help='chip size')
>>>     time_dim = scfg.Value(3, help='number of time steps')
>>>     channels = scfg.Value('*:(red|green|blue)', help='sensor / channel code')
>>>     time_sampling = scfg.Value('soft2')
>>> cls = ExampleDataConfig2
```

(continues on next page)

(continued from previous page)

```
>>> print(f'cls={cls}')
>>> self = cls()
>>> print(f'self={self}')
```

Example

```
>>> from scriptconfig.dataconfig import *  # NOQA
>>> import scriptconfig as scfg
>>> @dataconf
>>> class PathologicalConfig:
>>>     default0 = scfg.Value((256, 256), help='chip size')
>>>     default = scfg.Value((256, 256), help='chip size')
>>>     keys = [1, 2, 3]
>>>     __default__ = {
>>>         'argparse': 3.3,
>>>         'keys': [4, 5],
>>>     }
>>>     default = None
>>>     time_sampling = scfg.Value('soft2')
>>>     def foobar(self):
>>>         ...
>>> self = PathologicalConfig(1, 2, 3)
>>> print(f'self={self}')
```

FIXME: xdoctest problem. Need to be able to simulate a module global scope # Example: # >>> # Using inheritance and the decorator lets you pickle the object # >>> from scriptconfig.dataconfig import * # NOQA # >>> import scriptconfig as scfg # >>> @dataconf # >>> class PathologicalConfig2(scfg.DataConfig): # >>> default0 = scfg.Value((256, 256), help='chip size') # >>> default2 = scfg.Value((256, 256), help='chip size') # >>> #keys = [1, 2, 3] : Too much # >>> __default__3 = { # >>> 'argparse': 3.3, # >>> 'keys2': [4, 5], # >>> } # >>> default2 = None # >>> time_sampling = scfg.Value('soft2') # >>> config = PathologicalConfig2() # >>> import pickle # >>> serial = pickle.dumps(config) # >>> recon = pickle.loads(serial) # >>> assert 'locals' not in str(PathologicalConfig2)

scriptconfig.**define**(*default={}*, *name=None*)

Alternate method for defining a custom Config type

scriptconfig.**quick_cli**(*default*, *name=None*)

Quickly create a CLI

New in 0.5.2

Example

```
>>> # SCRIPT
>>> import scriptconfig as scfg
>>> default = {
>>>     'fpath': scfg.Path(None),
>>>     'modnames': scfg.Value([]),
>>> }
>>> config = scfg.quick_cli(default)
>>> print('config = {!r}'.format(config))
```

class scriptconfig.**Flag**(*value=False*, ***kwargs*)

Bases: *Value*

Exactly the same as a Value except isflag default to True

class scriptconfig.**ModalCLI**(*description=''*, *sub_clis=None*, *version=None*)

Bases: object

Contains multiple scriptconfig.Config items with corresponding *main* functions.

CommandLine

```
xdoctest -m scriptconfig.modal ModalCLI
```

Example

```
>>> from scriptconfig.modal import *  # NOQA
>>> import scriptconfig as scfg
>>> self = ModalCLI(description='A modal CLI')
>>> #
>>> @self.register
>>> class Command1Config(scfg.Config):
>>>     __command__ = 'command1'
>>>     __default__ = {
>>>         'foo': 'spam'
>>>     }
>>>     @classmethod
>>>     def main(cls, cmdline=1, **kwargs):
>>>         config = cls(cmdline=cmdline, data=kwargs)
>>>         print('config1 = {}'.format(ub.urepr(dict(config), nl=1)))
>>> #
>>> @self.register
>>> class Command2Config(scfg.DataConfig):
>>>     __command__ = 'command2'
>>>     foo = 'eggs'
>>>     baz = 'biz'
>>>     @classmethod
>>>     def main(cls, cmdline=1, **kwargs):
>>>         config = cls.cli(cmdline=cmdline, data=kwargs)
>>>         print('config2 = {}'.format(ub.urepr(dict(config), nl=1)))
>>> #
>>> parser = self.argparse()
>>> parser.print_help()
...
A modal CLI
...
commands:
  {command1,command2}  specify a command to run
    command1           argparse CLI generated by scriptconfig...
    command2           argparse CLI generated by scriptconfig...
>>> self.run(argv=['command1'])
config1 = {
```

(continues on next page)

(continued from previous page)

```
    'foo': 'spam',
}
>>> self.run(argv=['command2', '--baz=buz'])
config2 = {
    'foo': 'eggs',
    'baz': 'buz',
}
```

CommandLine

```
xdoctest -m scriptconfig.modal ModalCLI:1
```

Example

```
>>> # Declarative modal CLI (new in 0.7.9)
>>> import scriptconfig as scfg
>>> class MyModalCLI(scfg.ModalCLI):
>>>     #
>>>     class Command1(scfg.DataConfig):
>>>         __command__ = 'command1'
>>>         foo = scfg.Value('spam', help='spam spam spam spam')
>>>         @classmethod
>>>         def main(cls, cmdline=1, **kwargs):
>>>             config = cls.cli(cmdline=cmdline, data=kwargs)
>>>             print('config1 = {}'.format(ub.urepr(dict(config), nl=1)))
>>>     #
>>>     class Command2(scfg.DataConfig):
>>>         __command__ = 'command2'
>>>         foo = 'eggs'
>>>         baz = 'biz'
>>>         @classmethod
>>>         def main(cls, cmdline=1, **kwargs):
>>>             config = cls.cli(cmdline=cmdline, data=kwargs)
>>>             print('config2 = {}'.format(ub.urepr(dict(config), nl=1)))
>>> #
>>> MyModalCLI.main(argv=['command1'])
>>> MyModalCLI.main(argv=['command2', '--baz=buz'])
```

Example

```
>>> # Declarative modal CLI (new in 0.7.9)
>>> import scriptconfig as scfg
>>> class MyModalCLI(scfg.ModalCLI):
>>>     ...
>>> #
>>> @MyModalCLI.register
>>> class Command1(scfg.DataConfig):
```

(continues on next page)

(continued from previous page)

```
>>>     __command__ = 'command1'
>>>     foo = scfg.Value('spam', help='spam spam spam spam')
>>>     @classmethod
>>>     def main(cls, cmdline=1, **kwargs):
>>>         config = cls.cli(cmdline=cmdline, data=kwargs)
>>>         print('config1 = {}'.format(ub.urepr(dict(config), nl=1)))
>>> #
>>> @MyModalCLI.register
>>> class Command2(scfg.DataConfig):
>>>     __command__ = 'command2'
>>>     foo = 'eggs'
>>>     baz = 'biz'
>>>     @classmethod
>>>     def main(cls, cmdline=1, **kwargs):
>>>         config = cls.cli(cmdline=cmdline, data=kwargs)
>>>         print('config2 = {}'.format(ub.urepr(dict(config), nl=1)))
>>> #
>>> MyModalCLI.main(argv=['command1'])
>>> MyModalCLI.main(argv=['command2', '--baz=buz'])
```

property sub_clis

classmethod register(*cli_cls*)

Parameters
cli_cli (*scriptconfig.Config*) – A CLI-aware config object to register as a sub CLI

_build_subcmd_infos()

_parserkw()

Generate the kwargs for making a new argparse.ArgumentParser

argparse(*parser=None*, *special_options=Ellipsis*)

Builds a new argparse object for this ModalCLI or extends an existing one with it.

build_parser(*parser=None*, *special_options=Ellipsis*)

Builds a new argparse object for this ModalCLI or extends an existing one with it.

classmethod main(*argv=None*, *strict=True*, *autocomplete='auto'*)

Execute the modal CLI as the main script

classmethod run(*argv=None*, *strict=True*, *autocomplete='auto'*)

Execute the modal CLI as the main script

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex

PYTHON MODULE INDEX

S

INDEX

Symbols

A

B

C

D

F

G

I

K

L

M

N