ArgParseInator Documentation

Release 1.0.18

ellethee <luca800@gmail.com>

Jun 16, 2017

Contents

1 Getting Started **3**

2 Decorators and Function declarations **5**
- 2.1 Add arguments to parser (**ap_arg**) . . . 5
- 2.2 Add arguments to a function (**@arg**) . . . 5
- 2.3 Authorize commands (**@cmd_auth**) . . . 6
- 2.4 Function declarations . . . 7
- 2.5 Enable classes (**@class_args**) . . . 8

3 Within classes **9**
- 3.1 First way (commands container) . . . 10
- 3.2 Second way (sub-commands container) . . . 10
- 3.3 Shared arguments . . . 10
- 3.4 Sub commands container and command arguments . . . 11
- 3.5 Importing commands packages . . . 11

4 ArgParseInator object **13**
- 4.1 Parameters . . . 14
  - 4.1.1 add_output . . . 14
  - 4.1.2 argpi_name . . . 14
  - 4.1.3 args . . . 14
  - 4.1.4 auth_phrase . . . 15
  - 4.1.5 auto_exit . . . 15
  - 4.1.6 config . . . 15
  - 4.1.7 never_single . . . 16
  - 4.1.8 write_name . . . 16
  - 4.1.9 write_line_name . . . 16
  - 4.1.10 default_cmd . . . 16
  - 4.1.11 error . . . 17
  - 4.1.12 ff_prefix . . . 17
  - 4.1.13 msg_on_error_only . . . 17
  - 4.1.14 setup . . . 17
  - 4.1.15 argparse_args . . . 17
- 4.2 Methods . . . 18
- 4.3 `__argpi__`, `write()` and `writeln()` . . . 18

5 ArgParseInated object **19**

5.1 __preinator__ . . . 19

6 Events **21**
6.1 Examples . . . 22

7 Utils **23**
7.1 `init()` . . . 23
7.1.1 The StandAlone approch . . . 23
7.1.2 The SubModules approch . . . 24
7.1.3 The SubProjects approch . . . 24
7.1.4 The Skeleton folder . . . 25
7.2 `get_compiled()` . . . 25

8 Examples **27**
8.1 Modular multi-commands script with sub-commands . . . 27
8.1.1 Main script . . . 27
8.1.2 User commands . . . 28
8.1.3 Admin commands . . . 29
8.2 Silly configuration example . . . 30
8.3 Simple HTTP requests . . . 31
8.4 Simple HTTP requests sub-classing `ArgParseInated` object . . . 32
8.5 StandAlone Structure Example . . . 32
8.5.1 Let's create the structure . . . 33
8.5.2 The running script . . . 33
8.5.3 The __init__ module . . . 33
8.5.4 The command module . . . 34
8.6 SubModules structure example . . . 35
8.6.1 Let's create the structure . . . 35
8.6.2 The pets/main submodule . . . 36
8.6.3 The pets/main commands . . . 36
8.6.4 The dog submodule . . . 37
8.6.5 The dog commands . . . 37
8.6.6 The running script . . . 38

Warning: My English really sucks! So, if you wanna to correct this documentation or to report any error, I'll be very pleased to receive your suggestions.

email: luca800@gmail.com
documentation

Argparseinator is a silly, but useful, thing that permit you to easily add argparse's arguments and options to your script directly within functions and classes with the use of some decorators.

Contents:

CHAPTER 1

Getting Started

The use of ArgParseInator is very simple.

Decorate the functions and / or classes by setting the necessary parameters and putting into the __doc__string the function description.

```
from argparseinator import ArgParseInator
from argparseinator import arg


@arg("name", help="The name to print")
@arg('-s', '--surname', default='', help="optional surname")
def print_name(name, surname=""):
    """
    Will print the passed name.
    """
    print "Printing the name...", name, surname
```

Note: functions can have positional arguments and keyword argument to reflect the ArgParseInator arguments. Anyway there are some tricks that we will see later in the function declaration section.

Get the ArgParseInator instance passing some parameters if necessary. Then verify the commands passed to the script.

```
if __name__ == "__main__":
    ArgParseInator(description="Silly script").check_command()
```

Try out your script help

```
$ python apitest.py -h
```

Will output

```
usage: apitest.py [-h] [-s SURNAME] name
```

```
Silly script

    Will print the passed name.


positional arguments:
  name                  The name to print

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

Try out your script

```
$python apitest.py --surname=Smith John
```

Will output

```
Printing the name... John Smith
```

Note: If we have only one function decorated it will become the default command and Argparseinator appends the function description to the main parser's description.

CHAPTER 2

Decorators and Function declarations

- *Add arguments to parser (ap_arg)*
- *Add arguments to a function (@arg)*
- *Authorize commands (@cmd_auth)*
- *Function declarations*
- *Enable classes (@class_args)*
- *Within classes*

Add arguments to parser (ap_arg)

ap_arg is not a decorator, but a silly convenience function that returns a tuple with positional arguments and keyword arguments. Useful to pass list of arguments to the parser.

```
inator = ArgParseInator(args=[
    ap_arg('-O', '--option', help="Optional parameter")
])
```

Add arguments to a function (@arg)

@arg adds arguments to function in `ArgumentParser.add_argument()` style. Or just enable the function as `ArgParseInator` command if no parameters are passed.

If no parameters are passed to **@arg** and there are function arguments they become the command arguments and optional arguments. For optional arguments all the options can be specify using a dictionary and the argument becomes **–option_name** unless you use the `flag` and the `lflag` keys in the dictionary to specify the short and the long option name.

ArgParseInator will use the *function* **name** as command name unless you pass the special parameter **cmd_name** that can change the command name for the function. Or you can specify the name using the **function.__cmd_name__** form.

```
# Creates an ArgParseInator command without passing parameters to **@arg**
# decorator. But retrieves them from the function itself.
# it defines also the path option flags using **flag** and **lflag** keywords.
# and changes the command name with the function.__cmd_name__ form.

@arg()
def save(filename, overwrite={'action': 'store_true', 'default':False},
         path={'flag': '-p', 'lflag': 'pth', 'help': "the file path", 'default': ""}):
    """
    A Save Function.
    """
    # code here
save.__cmd_name__ = 'mysave'
```

```
# Creates a ArgParseInator command **with** parameters.

@arg('filename', help="the file name")
@arg('-p', '--path', help="the file path", default='')
@arg('--overwrite', help="overwrite the file", default=False)
def save(filename, path, overwrite):
    """
    A save function.
    """
    # code here
```

```
# Creates a ArgParseInator command **without** parameters.

@arg()
def foo_print():
    """
    Foo!!!
    """
    # code_here
```

```
# Creates a ArgParseInator command **without** parameters and the command name
# will be **foo**

@arg(cmd_name="foo")
def foo_print():
    """
    Foo!!!
    """
    # code_here
```

Authorize commands (@cmd_auth)

Sometimes our scripts can be potentially dangerous. So we would like to protect some commands with a *auth phrase*.

And here comes the **@cmd_auth** decorator. We can pass the **auth_phrase** parameter which will be used to check the authorization for the command or call it without the parameter, in this case it will use the global *auth_phrase* passed to the ArgParseInator instance.

```
# with global auth_phrase
@arg()
@cmd_auth()
def deleteall():
    # code_here

# with command specific auth_phrase
@arg()
@cmd_auth('im-sure')
def deleteall():
    # code_here
```

Note: **@cmd_auth** automatically adds the **–auth** optional argument to the top level parser.

Function declarations

When we declare a function that will then be decorated, we can use both types of parameters, positional and keyword. However, considering that the optional parameters are declared with the decorator, we can declare all parameters as positional without caring about the order.

```
@arg('name', help="The name")
@arg('-s', '--surname', help="The surname", default='')
def print_name(name, surname):
    # code_here
```

is the same as

```
@arg('name', help="The name")
@arg('-s', '--surname', help="The surname", default='')
def print_name(surname, name):
    # code_here
```

We can also refer to an argument declared in the top level parser or into the parent of the function (if it is inside a class).

```
@arg()
def print_foo(foo):
    # code_here

# We instantiate the class with a foo top level optional parameter.
inator = ArgParseInator(args=[ap_arg('--foo', help="Foo", default="bar")])
```

Last but not least the special parameter args. If ArgParseInator find it in the function declaration will assign to it the parse_args result.

```
@arg('name', help="the name")
@arg('--surname', help="the surname")
def print_args(args):
    print args.name, args.surname, args.foo
```

This can be useful if you don't know exactly what parameters you need or if you are lazy enough (like me) to avoid typing all the parameters in the function declaration.

Enable classes (@class_args)

We can enable classes to become commands container simply adding a **@class_args** decorator to the classes.

Once the class is enabled we can specify some class attributes that will modify the commands behavior:

- **__cmd_name__** set the command name
- **__arguments__** set extra arguments for the command.
- **__shared_arguments__** set arguments which will be shared by the class sub commands

```
@class_args
class Greetings(object):
    """
    Greeting command.
    """
    __cmd_name__ = 'greet'
    # code here
```

We will discuss the classes usage in the *Within classes* section.

CHAPTER 3

Within classes

We can also use classes for define our commands and sub commands. There are three properties that we can use within classes.

- `__cmd_name__`: Declare the command name. In this way the class becomes a Sub Commands container.
- `__arguments__`: Arguments list. When the class is a sub commands container we can set a list of attributes for the command itself.
- `__shared_arguments__`: Shared arguments list. In any case we can declare a list of arguments that will be shared by all commands or sub commands.
- `__no_share__`: Must be defined at function level and can bypass __shared_arguments__.

```
__shared_arguments__ = [ap_arg("name"), ap_arg("surname")]

@arg()
def mytime(self):
    pass
mytime.__no_share__ = True  # None of __shared_arguments__.

@arg()
def nick(self):
    pass
nick.__no_share__ = ['surname'] # all __shared_arguments__ but not␣
↪surname.
```

The way we declare the class using these properties changes the behavior of ArgParseInator. Always using the *Enable classes (@class_args)* decoration for the class of course.

- *First way (commands container)*
- *Second way (sub-commands container)*
- *Shared arguments*
- *Sub commands container and command arguments*

First way (commands container)

We can turn a class in a commands container using the `@class_args` decorator and using the `@arg` decorator with the class methods to turn them into commands

```
@class_args
class CommandsContainer(object):
    """Commands container class."""
    prefix = "The name is"

    @arg('name', help="The Name")
    def name(self, name):
        """Print the name."""
        print self.prefix, name
    ...
```

Second way (sub-commands container)

We can specify a **__cmd_name__** for the class so it will become the *command* and all decorated methods will become *sub-commands*.

```
@class_args
class SubCommandsContainer(object):
    """Sub Commands container class."""
    # our command will be dosub
    __cmd_name__ = 'dosub'
    prefix = "The name is"

    @arg('name', help="The Name")
    def name(self, name):
        """Print the name. """
        print self.prefix, name
    ...
```

Shared arguments

Whether we use the *First way (commands container)* or the *Second way (sub-commands container)* we can specify a **__shared_arguments__** with a list of arguments that will be added to all commands contained in the class.

```
@class_args
class CommandsContainer(object):
    """
    Commands Container class.
    """
    # share arguments with commands.
    __shared_arguments__ = [
        ap_arg('name', help="The Name"),
        ap_arg('--prefix', help="prefix string", default="The name is..")]

    @arg()
    def name(self, name, prefix):
        """
```

```
        Print the name.
        """
        print prefix, name
    ...
```

Sub commands container and command arguments

With the *Second way (sub-commands container)* we can also specify a command specific arguments list using the **__arguments__** attribute.

```
@class_args
class SubCommandsContainer(object):
    """
    Commands Container class.
    """
    # our command is dosub
    __cmd_name__ = 'dosub'
    # our command arguments
    __arguments__ = [
        ap_arg('--prefix', help="prefix string", default="The name is..")]
    # the sub command shared arguments.
    __shared_arguments__ = [ap_arg('name', help="The Name")]

    @arg()
    def name(self, name, prefix):
        """
        Print the name.
        """
        print prefix, name
    ...
```

Importing commands packages

A good way to keep our code ordered is to put modules under a sub folder which can become, for convenience, a package. So we can have our structure like this.

```
+- commands
|   +- admin.py
|   +- __init__.py
|   +- user.py
+- multicommand.py
```

And our multicommand.py should looks like this.

Listing 3.1: multicommand.py

```
from argparseinator import ArgParseInator
from commands import admin, user

ArgParseInator().check_command()
```

But if we want to add other command modules we have to import all of them. And to do this we should modify our multicommand.py script. Or we can use the `import_commands()` function which loads all modules in a package.

```
from argparseinator import ArgParseInator, import_commands
import_commands('commands')

ArgParseInator().check_command()
```

Deprecated since version 1.0.15: Use normal import instead. Possibly use *The StandAlone approch* or *The SubModules approch* or *The SubProjects approch*.

CHAPTER 4

ArgParseInator object

class argparseinator.**ArgParseInator**(*add_output=None*, *argpi_name=None*, *args=None*, *auth_phrase=None*, *auto_exit=None*, *config=None*, *default_cmd=None*, *error=None*, *ff_prefix=None*, *formatter_class=None*, *msg_on_error_only=None*, *never_single=None*, *setup=None*, *write_line_name=None*, *write_name=None*, ***argparse_args*)

Create a new *ArgParseInator* object. All parameters should be passed as keyword arguments.

- *add_output* - Enable the output option (default: `False`).
- *argpi_name* - Name for the global reference of the Argparseinator instance (default: `__argpi__`).
- *args* - List of argument to pass to the parser (default: `None`).
- *auth_phrase* - Global authorization phrase (default: `None`).
- *auto_exit* - if True ArgParseInator uses the functions return values as exit status code. If the return statment is missing or the return value is None then EXIT_OK will be used (default: `True`).
- *config* - Tuple containing config filename, config factory and optionally a config error handler.
- *default_cmd* - Name of the default command to set.
- *error* - Error handler to pass to parser.
- *ff_prefix* - fromfile_prefix_chars to pass to the parser.
- formatter_class - A class for customizing the help output.
- *msg_on_error_only* - if *auto_exit* is `True`, it outputs the command message only if there is an exception.
- *never_single* - Force to have one command, even it is the only one (default: `False`).
- *setup* - List of functions (having the ArgParseInator class as parameter) that will be executed after the arguments parsing and just before to execute the command.
- *write_name* - Name for the global function which calls *ArgParseInator.write()* (default: `write`).

- *write_line_name* - Name for the global function which calls *ArgParseInator.writeln()* (default: writeln).
- *write_mode* - default write mode when using *add_output* (default: wb).
- show_defaults - If True shows default values for parameters in help (default: True).
- *argparse_args* - All standard ArgumentParser parameters.

Parameters

add_output

Automatically append to the top level parser the -o --output optional argument. If the argument is passed the *ArgParseInator* write() and writeln() methods will write the output to the file.

```
from argparseinator import ArgParseInator, arg

@arg('string', help="String to write")
def write(string):
    """
    Wrtie a string
    """
    ArgParseInator().writeln(string)

ArgParseInator(add_output=True).check_command()
```

```
$ python script.py --output=filename.txt "Hello my name is Luca"
```

Will create a file named **filename.txt** containing the line **Hello my name is Luca**

argpi_name

The **Argparseinator** instance can be accessed globally via the name __argpi__. Anyway you can change the global name using this parameter.

args

Accepts a list of argument to pass to the top level parser. Every element of the list must be a tuple with positional args and keyword args. Something like this (('-o', '--option'), {'help': 'option', 'default': 'no option'}) but for convenience we use the ap_arg() which simplify things.

```
from argparseinator import ArgParseInator, arg, ap_arg

@arg('string', help="String to write")
def write(string, prefix):
    """
    print a string
    """
    print prefix, string

ArgParseInator(args=[
    ap_arg('-p', '--prefix', help="string prefix", default="Now Writing..")
]).check_command()
```

```
$ python script.py -h
```

will output

```
usage: script.py [-h] [-p PREFIX] string

    print a string


positional arguments:
  string                String to write

optional arguments:
  -h, --help            show this help message and exit
  -p PREFIX, --prefix PREFIX
                        string prefix
```

auth_phrase

Set a global authorization phrase to protect special commands. See *Authorize commands (@cmd_auth)*

auto_exit

If True ArgParseInator exits just executed the command using the returned value(s) as status code.

If the command function return only a numeric value it will be used as status code exiting the script if the command function returns a tuple with numeric and string value the string will be printed as message.

```
@arg()
def one():
    # will exit from script with status code 1
    return 1

@arg()
def two():
    # will exit from script with status code 2 and print the message
    # "Error"
    return 2, "Error."
```

config

It could happen that we need a configuration dictionary or something similare, usually loaded from a file. We can specify a dictionary with the configuration or a tuple to handle the configuration file and optionally a configuration error handler. It will be available as **self.cfg** if you use a subclass of ArgParseInated or globally using *__argpi__*.**cfg**

```
def cfg_factory(filename):
    """Configuration factory"""
    import yaml
    return yaml.load(filename)

def cfgname():
    """Prints name"""
```

```
    print __argpi__.cfg['name']

Argparseinator(config=('default.cfg', cfg_factory)).check_command()
```

never_single

When we have only one decorated function *ArgParseInator* automatically set it as default and adds all its arguments to the top level parser. We can also tell to *ArgParseInator* to keep it as a command by setting the **never_single** parameter to `True`.

```
from argparseinator import ArgParseInator, arg

@arg('string', help="String to write")
def write(string):
    """
    Write a string
    """
    print string

ArgParseInator().check_command()
```

```
$ python script.py "String to print"
String to print
```

```
ArgParseInator(never_single=True).check_command()
```

```
$ python script.py write "String to print"
String to print
```

write_name

Sets the name for the global shortcut `write()` (see *__argpi__, write() and writeln()*)

```
@arg()
def write_test():
    w("this is a test.")

ArgParseInator(write_name="w").check_command()
```

write_line_name

As write_name does, it sets the name for the global shortcut `writeln()` (see *__argpi__, write() and writeln()*)

default_cmd

When we have multiple commands we can set a default one to be used if *ArgParseInator* can't find a valid one in `sys.argv`

error

Usually if we need to handle argparse error we have to subclass the ArgumentParser and override the error method. With the *ArgParseInator* we can just pass the handler as `error` parameter.

```
def error_hander(self, message):
    """Error handler"""
    print "And the error is ...", message

ArgParseInator(error=error_hander).check_command()
```

ff_prefix

It's a shortcut for fromfile_prefix_chars. Note that if its value is True then it automatically uses the @ as fromfile_prefix_chars.

msg_on_error_only

if *auto_exit* is True, it outputs the command message only if there is an exception.

setup

A list or tuple of functions that will be executed just before executing the command, receives as parameter the ArgParseInator instance.

```
def setup_1(inator):
    """first setup"""
    inator.args.name = 'Luca'

def setup_2(inator):
    """second setup"""
    inator.args.name = inator.args.name.upper()

ArgParseInator(setup=[setup_1, setup_2]).check_command()
```

argparse_args

**argparse_args are all the parameters to pass to the `ArgumentParser`.

Note: The part below is copied from the argparse module page.

- prog - The name of the program (default: `sys.argv[0]`)
- usage - The string describing the program usage (default: generated from arguments added to parser)
- description - Text to display before the argument help (default: none)
- epilog - Text to display after the argument help (default: none)
- formatter_class - A class for customizing the help output
- prefix_chars - The set of characters that prefix optional arguments (default: '-')

- fromfile_prefix_chars - The set of characters that prefix files from which additional arguments should be read (default: `None`)
- argument_default - The global default value for arguments (default: `None`)
- add_help - Add a -h/–help option to the parser (default: `True`)

Methods

`ArgParseInator.`**`check_command`**`(`***new_attributes*`)`
Essentially executes the command doing these steps.

1. Create all the arguments parsers with arguments according with the decorators and classes.
2. Parse the arguments passed by the command line.
3. Try to execute the command.

`ArgParseInator.`**`write`**`(`**strings*`)`
Write to the output (stdout or file, see *add_output*). If more than a string is passed then it will be written space separated.

`ArgParseInator.`**`writeln`**`(`**strings*`)`
Exactly as *`ArgParseInator.write()`* but appends a newline at the end of the string.

`__argpi__`, `write()` and `writeln()`

Just before executing the command *`ArgParseInator`* it adds two global shortcuts for its methods *`ArgParseInator.write()`* and *`ArgParseInator.writeln()`* respectively `write()` and `writeln()` and the global reference to the *Singleton* instance as `__argpi__`.

So you can use `write()` or `writeln()` instead of `` `ArgParseInator().write()` ``, `` `ArgParseInator().writeln()` ``. And access directly to the Singleton instance using `__argpi__` instead of `` `ArgParseInator()` ``.

The two methods names can be changed via the *write_name* and *write_line_name* arguments and the global instance name via the *argpi_name* while instantiating the ArgParseInator.

CHAPTER 5

ArgParseInated object

class ArgParseInated(*parseinator*, ***new_attributes*)

This class is meant to create sub classes that automatically expose the `write()`, `writeln()` methods, the `args` and `cfg` attributes of the `ArgParseInator`. Plus expose all passed **new_attributes** which will passed by the `ArgParseInator.check_command()`. It has also the *__preinator__* method which is called just at the `__init__()` end. So we can use it to do some extra action before the command is execute.

```
class Greetings(ArgParseInated):
    """
    Greet somebody.
    """

    @arg()
    def ciao(self):
        """
        say ciao.
        """
        self.writeln(self.prefix, "ciao to", self.name)

ArgParseInator().check_command(prefix="We say", name="Luca")
```

Note: We can specify __cmd_name__, __arguments__, __shared_arguments__ to do some more trick, see *Within classes* .

__preinator__

The original name was **__prepare__** but I've changed it to avoid problems. It's just called at the end of the `__init__()` method and is intended to do some action before `ArgParseInator` executes the command.

```
class Greetings(ArgParseInated):
    """
    Greet somebody.
    """

    @arg("name", help="The name")
    def ciao(self, name):
        """
        say ciao.
        """
        self.writeln("We say ciao to", name)


    def __preinator__(self):
        if self.args.name.lower() == 'luca':
            self.args.name = "who? Nobody?"

ArgParseInator().check_command()
```

CHAPTER 6

Events

ArgParseInator can handle two simply events **before_execute** and **after_execute**. These events can be used on two levels: **class level** and **command level**. The **command level** event overrides the **class level** event. If the **before_execute** event returns something different than **None** or than a **3 elements tuple** which the first item is different than **None** will interrupt the process and return the **before_execute** value.

The two levels differs in declaration but have the same return type. Obvously the command level leacks the *_cmd_name* parameter.

```
# class level event
def event(self, _cmd_name, *args, **kwargs):
    pass

# command level event
def event(self, *args, **kwargs):
    pass

# simple breaking return
return 1

# 2 values breaking return
return 2, "I Will stop executions"

# 3 values NON breaking return
return None, ['new', 'positional', 'args'], {'new': 'keywords args'}

# 3 values breaking return
return 1, ['new', 'positional', 'args'], {'new': 'keywords args'}

# 3 values breaking tuple return
return (2, "I Will stop executions", ), ['new', 'positional', 'args'], {'new':
↪'keywords args'}
```

Examples

Some silly example.

```
from argparseinator import ArgParseInated
from argparseinator import arg, ap_arg
from argparseinator import class_args


@class_args
class Commands(ArgParseInated):
    """test commands"""

    # we can share the only argument
    __shared_arguments__ = [ap_arg("word", help="The word")]

    def before_execute(self, _cmd_name, *args, **kwargs):
        """Class Level Before execute event"""
        # if we are going to whisper we will replace the word.
        if _cmd_name == 'whisper':
            # return None as first element so we will continue the execution.
            return None, ['pst, pst'], kwargs

    @arg()
    def whisper(self, word):
        """ whisper a word """
        print "Whisper", word
        return 0

    @arg()
    def say(self, word):
        """ say a word """
        print "Say", word
        return 0

    @arg()
    def shout(self, word):
        """ shout a word """
        print "Shout", word
        return 0

    def shout_before_execute(self, *args, **kwargs):
        """Command Level Before execute event"""
        print "Please dont shout!"
        # we will break the execution with a message too
        return 1, "\nBe quiet\n"
    shout.before_execute = shout_before_execute
```

CHAPTER 7

Utils

ArgParseInator comes with few simply utils, the **init** function which create a project structure and the **get_compiled** function which returns the compiled parser.

`init()`

```
.. currentmodule:: argparseinator.__main__

.. argparse::
   :module: argparseinator.__main__
   :func: get_compiled
   :prog: python -margparseinator
```

The Argparse Inator project structures aims to forget about the main script and concentrate to the commands part of the project and simplify the distribution/integration/extendibility.

The StandAlone approch

This approch has a project structure like this:

```
project
+- project
|   +- commands.py
|   +- __init__.py
+- project.py
```

And you can run it in this way

```
$ python project/project.py --help
```

The SubModules approch

This approch has a project structure like this:

```
project/
+- project              |
|   +- commands.py      | - the *core* folder is optional
|   +- __init__.py      |
+- project.py
+- submod1
|   +- commands.py
|   +- __init__.py
+- submod2
|   +- commands.py
|   +- __init__.py
+- submod3
    +- commands.py
    +- __init__.py
```

And you can run it in this way

```
$ python project/project.py --help
$ python project/project.py project --help
$ python project/project.py submod1 --help
```

In this case you have a main script **project.py** which loads the submodule you asked for.

The **ArgParseInator** main call is done in the project/project/__init__.py, if is present, otherwise you must set it up in the **project.py** script.

The SubProjects approch

This approch has a project structure like this:

```
project/
+- project              |
|   +- __init__.py      | - the *core* folder is optional
|   +- utils.py         |
+- project.py
+- subpro1
|   +- commands.py
|   +- __init__.py
+- subpro2
|   +- commands.py
|   +- __init__.py
+- subpro3
    +- commands.py
    +- __init__.py
```

And you can run it in this way

```
$ python project/project.py --help
$ python project/project.py project --help
$ python project/project.py subpro1 --help
```

The **SubProjects** is mix. You have a **SubModules** structure but there isn't a **ArgParseInator** setup in the (optional) project/__init__.py.

Every SubProject have it's **Argparseinator** setup. And the optional project package/folder is used for shared modules.

The Skeleton folder

ArgParseInator uses it's own skeleton folder:

```
skeleton/
+- standalone
|   +- project
|   |   +- commands.py
|   |   +- __init__.py
|   +- project.py
+- submodules
|   +- project
|   |   +- commands.py
|   |   +- __init__.py
|   +- project.py
|   +- submodule
|       +- commands.py
|       +- __init__.py
+- subprojects
    +- project
    |   +- __init__.py
    |   +- utils.py
    +- project.py
    +- submodule
        +- commands.py
        +- __init__.py
```

Which contains the base structure for StandAlone, SubProjects ans SubModules strcutures.

You can copy and modify your own skeleton folder but keep in mind:

- Everything in the skeleton folder will be copied in the new project folder.
- Only *project.py* script and *project* filder will renamed to the new project name.
- Only the *submodule* will be renamed to the subproject/submodule folder.

Everything else will be copied as it is.

get_compiled()

Argparseinator can use the `get_compiled()` function and autodoc with the sphinx-argparse extension to quickly create documentation.

Is quite easy.

Listing 7.1: test.py

```
from argparseinator import ArgParseInator, arg
from argparseinator import get_compiled

@arg("word", help="The word")
def say(self, word):
    """ say a word """
    print "say", word
```

```
    return 0

ArgParseInator().check_command()
```

Listing 7.2: test/index.rst

```
.. currentmodule:: test

.. automodule:: test

.. argparse::
   :module: test
   :func: get_compiled
   :prog: test
```

CHAPTER 8

Examples

Some example that shoud help you or could inspire you.

Modular multi-commands script with sub-commands

This script is a little more complicated anyway not so much. It will be modular. That means it will load commands at run time. We decided to create a script that can executes user commands and admin commands.

For convenience we will put the admin part in a module named `admin.py` and the user part in a module named `user.py`. And our modules will reside in the **commands subfolder**. (Actually we can have many modules into the commands sub folder)

Download `this example zip`.

- *Main script*
- *User commands*
- *Admin commands*

Main script

In the main script we will just import all the commands, pass all desired arguments to the ArgParseInator and finally call the check_command method.

Listing 8.1: multicommand.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
"""
    Modular multi commands with subcommands example
"""
# If we specify here out version ArgParseInator will include it for us.
__version__ = "1.2.3"
```

```
import os
from argparseinator import ArgParseInator, import_commands


if __name__ == "__main__":
    # Let's import commands for the script which resides into the commands
    # package.
    import_commands('commands')
    # Ok, Istantiate ArgParseInator
    ArgParseInator(
        # Enable the --output if we want to write the result on file.
        add_output=True,
        # We will automatically exit from the script with the command return
        # code and message if any
        auto_exit=True,
        # Set up the phrase for commands that needs authorization
        auth_phrase="ok",
        # And set the default command for when we don't explicitally pass it
        default_cmd="user",
    # And finally tell to ArgParseInator to check command
    ).check_command()
```

Note: We also defined the **__version__** at module level, ArgParseInator will handle it for us.

User commands

In this module we define our user commands. Its intended as example. It doesn't really something useful.

Listing 8.2: commnds/user.py

```
# -*- coding: utf-8 -*-
"""
    User commands for multi commands example.
"""
from argparseinator import class_args, arg, ArgParseInated


@class_args
class User(ArgParseInated):
    """
    User commands.
    """
    __cmd_name__ = "user"

    @arg()
    def files(self):
        """
        List files.
        """
        self.writeln("Listing files...")
        # listing files code.
        return 0, "Files listed\n"

    @arg('name', help="Name to greet")
```

```
    def greet(self, name):
        """
        Greeting command.
        """
        self.writeln('Greeting person...')
        self.writeln('Ciao', name)
        return 0, "person greeted\n"
```

Admin commands

And here we have our Administrations command. As you can see we used the *Authorize commands (@cmd_auth)* decorator for the `useradd()` and `userdelete()` commands without the parameter, so `ArgParseInator` will use the global *auth_phrase*. For the `deleteallusers()` command, instead, we used a specif auth_phrase.

Listing 8.3: commands/admin.py

```
# -*- coding: utf-8 -*-
"""
    Administation commands for multi commands example.
"""
from argparseinator import class_args, arg, cmd_auth, ArgParseInated, ap_arg


class UserExistsError(Exception):
    """Fake user error"""
    pass

UserNotExistsError = UserExistsError


@class_args
class Admin(ArgParseInated):
    """Administrations commands."""
    # Our command name.
    __cmd_name__ = "admin"
    __shared_arguments__ = [
        ap_arg("-p", "--passwd", help="User parssword"),
        ap_arg("username", help="username"),
    ]

    @arg()
    def useradd(self, username, passwd):
        """Fake Create new user command."""
        self.writeln("Creating user", username, '...')
        try:
            # ... create user code.
            pass
        except UserExistsError:
            return 1, "User {} already exists".format(username)
        if passwd:
            # ... set password code.
            pass
        return 0, "User {} created".format(username)

    @arg()
    def userdelete(self, username):
```

```
        """Fake Delete user command."""
        self.writeln("Deleting user", username, '...')
        try:
            # ... delete user code.
            pass
        except UserNotExistsError:
            return 1, "User {} do not exists".format(username)
        return 0, "User {} deleted".format(username)

    @arg()
    @cmd_auth("yesiwantit")
    def deleteallusers(self):
        """Fake delete all users command"""
        self.writeln("Deleting ALL users...")
        # ... delete all users code
        return 0, "All users deleted"
```

Silly configuration example

This is a silly configuration example.

Download `this example zip`.

We have our configuration file written in yaml where we have defined username and password.

Listing 8.4: config.yaml

```
##
# yaml confiuration file
#
username: Luca         # username
password: password     # password
```

Listing 8.5: config.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
"""
    Configuration example.
"""
import sys
import yaml
from argparseinator import ArgParseInator, arg

def config_factory(self, filename):
    """Configuration handler"""
    return yaml.load(open(filename, 'rb'))

def config_error(self, error):
    """ Configuration error handler"""
    print error
    sys.exit(1)

@arg()
def name():
    """
```

```
        Prints configuration username and password
        using then __argpi__ global reference to the ArgParseInator instance
    """
    print __argpi__.cfg['username'], __argpi__.cfg['password']

if __name__ == "__main__":
    ArgParseInator(
        config=('config.yaml', config_factory, config_error)
    ).check_command()
```

Simple HTTP requests

This simple script uses the functions approach to send simple HTTP requests to a server.

Download `this example script`.

Listing 8.6: httprequest.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
"""
    Simple Http retrieving.
"""
from argparseinator import ArgParseInator, arg
import requests


@arg('url', help="url to retrieve")
def get(url):
    """Retrieve an url."""
    writeln("Getting data from url", url)
    response = requests.get(url)
    if response.status_code == 200:
        writeln(response.text)
    else:
        writeln(str(response.status_code), response.reason)


@arg('var', help="Post data in NAME=VALUE form", nargs="*")
@arg('url', help="url to retrieve")
def post(url, var):
    """Post data to an url."""
    data = {b[0]: b[1] for b in [a.split("=") for a in var]}
    writeln("Sending data to url", url)
    response = requests.post(url, data=data)
    if response.status_code == 200:
        writeln(response.text)
    else:
        writeln(str(response.status_code), response.reason)

if __name__ == "__main__":
    ArgParseInator(
        add_output=True,
        default_cmd="get",
    ).check_command()
```

Simple HTTP requests sub-classing `ArgParseInated` object

Let's do the same thing but by inheriting the class from *ArgParseInated*. We can define the shared argument `url` and pass an instance of requests.Session to our HttpRequest class.

Download `this example script`.

Listing 8.7: httprequest2.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
"""
    Simple Http retrieving with ArgParseInated class.
"""
from argparseinator import ArgParseInator, ArgParseInated
from argparseinator import arg, ap_arg, class_args
import requests


@class_args
class HttpRequest(ArgParseInated):
    """Silly Http retrieving."""

    __shared_arguments__ = [ap_arg('url', help="url to retrieve")]

    @arg()
    def get(self, url):
        """Retrieve an url."""
        self.writeln("Getting data from url", url)
        response = self.session.get(url)
        if response.status_code == 200:
            self.writeln(response.text)
        else:
            self.writeln(str(response.status_code), response.reason)

    @arg('var', help="Posta data in NAME=VALUE form", nargs="*")
    def post(self, url, var):
        """Post data to an url."""
        data = {b[0]: b[1] for b in [a.split("=") for a in var]}
        self.writeln("Sending data to url", url)
        response = self.session.post(url, data=data)
        if response.status_code == 200:
            self.writeln(response.text)
        else:
            self.writeln(str(response.status_code), response.reason)

if __name__ == "__main__":
    ArgParseInator(
        add_output=True,
    ).check_command(session=requests.Session())
```

StandAlone Structure Example

StandAlone project example.

Let's create the structure

```
$ python -margparseinator -d ~/src/python -D"Greets people" greets
```

```
~/src/python/greets
+- greets
|   +- commands.py
|   +- __init__.py
+- greets.py
```

The running script

As you can see the script loads the submodules/package dinamically at runtime.

Listing 8.8: greets.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
"""
Standard launcher
"""
import sys
from os.path import basename, splitext
from importlib import import_module
sys.modules[__name__] = import_module(splitext(basename(sys.argv[0]))[0])

if __name__ == "__main__":
    __argpi__.check_command()
```

The __init__ module

As you can see we can define here everything we need to run the main script.

Listing 8.9: greets/__init__.py

```
# -*- coding: utf-8 -*-
"""
======
Greets
======

Greets people
"""
from argparseinator import ArgParseInator
# we import the get_compiled function in case of sphinx.
from argparseinator import get_compiled
import yaml
import greets.commands
__version__ = "0.0.1"

# We don't really need a configuration but it's for example propouse.
def cfg_factory(filename):
    """Config Factory"""
    try:
        # try to load config as yaml file.
```

```
        with open(filename, 'rb') as stream:
            return yaml.load(stream)
    except StandardError as error:
        # In case of error we use the **__argpi__** builtin to exit from
        # script
        __argpi__.exit(1, str(error))


ArgParseInator(
    # Add the possibility of writing the output to file.
    add_output=True,
    # we will append to existing file.
    write_mode='ab',
    # Add the cfg_factory with the default config name to None.
    config=(None, cfg_factory,)
)
```

The command module

As you can see

Listing 8.10: greets/commands.py

```
# -*- coding: utf-8 -*-
"""
==============================
Commands :mod:`greets.commands`
==============================
"""
from argparseinator import ArgParseInated
from argparseinator import arg, ap_arg
from argparseinator import class_args


@class_args
class Commands(ArgParseInated):
    """Commands for greets"""

    # we have the same params for all the commands so we can share them
    __shared_arguments__ = [
        ap_arg("who", help="The who", nargs="?", default="World!"),
        ap_arg("-l", "--lang", default="en", help="Language")
    ]

    def __preinator__(self):
        # Actually we are not sure we have a configuration so is better
        # add some default.
        cfg = {
            'lang': 'en',
            'words_en': {
                'greet': 'Greetings',
                'hello': 'Hello',
                'bye': 'Goodbye',
            }
        }
        cfg.update(self.cfg)
```

```
        self.cfg = cfg

    def get_language(self, word, lang=None):
        """Get right word for configured language"""
        lang = lang or self.cfg.get('lang', 'en')
        # let's retrieve the word from configuration dict.
        try:
            return self.cfg['words_' + lang][word]
        except StandardError:
            return 'Do not know how to "{}" in "{}"'.format(word, lang)

    @arg()
    def greet(self, who, lang):
        """Greets"""
        # we added the add_output param so we use the writeln to greet on the
        # right output.
        writeln(self.get_language('greet', lang), who)

    @arg()
    def hallo(self, who, lang):
        """Hallo"""
        writeln(self.get_language('hello', lang), who)

    @arg()
    def bye(self, who, lang):
        """Bye"""
        writeln(self.get_language('bye', lang), who)
```

SubModules structure example

Let's create the structure

```
$ python -margparseinator -s -d ~/src/python/pets dog
$ python -margparseinator -s -d ~/src/python/pets cat
$ python -margparseinator -s -d ~/src/python/pets bird
# add a pets submodule for shared things
$ python -margparseinator -s -d ~/src/python/pets pets
```

```
~/src/python/pets
+- bird
|   +- commands.py
|   +- __init__.py
+- cat
|   +- commands.py
|   +- __init__.py
+- dog
|   +- commands.py
|   +- __init__.py
+- pets
|   +- commands.py
|   +- __init__.py
+- pets.py
```

The pets/main submodule

Listing 8.11: pets/__init__.py

```
# -*- coding: utf-8 -*-
"""
====
Pets
====

Manages various pets.
"""
from argparseinator import ArgParseInator
import pets.commands
__version__ = "0.0.1"

# We enable the output and set the default write_mode to append binary.
ArgParseInator(add_output=True, write_mode="ab")
```

The pets/main commands

Listing 8.12: pets/commands.py

```
# -*- coding: utf-8 -*-
"""
============================
Commands :mod:`pets.commands`
============================
"""
import os
from os.path import dirname
from importlib import import_module
from argparseinator import ArgParseInated
from argparseinator import arg
from argparseinator import class_args


@class_args
class Commands(ArgParseInated):
    """Commands for pets"""

    @arg()
    def allsays(self):
        """Asks all pets to say something"""
        # cycle in the main dir
        for name in os.listdir(dirname(dirname(__file__))):
            # try to import the module and say something.
            try:
                mod = import_module(name)
                mod.commands.Commands(__argpi__).say()
            except (ImportError, AttributeError):
                # just pass in case of these errors
                pass
    @arg()
    def allnames(self):
        """Asks all pets their names"""
```

```
        # cycle in the main dir
        for name in os.listdir(dirname(dirname(__file__))):
            # try to import the module and say something.
            try:
                mod = import_module(name)
                mod.commands.Commands(__argpi__).name()
            except (ImportError, AttributeError):
                # just pass in case of these errors
                pass
```

The dog submodule

Listing 8.13: dog/__init__.py

```
# -*- coding: utf-8 -*-
"""
===
Dog
===

Does the dog's things.
"""
from argparseinator import get_compiled
import dog.commands
__version__ = "0.0.1"
```

The dog commands

Listing 8.14: dog/commands.py

```
# -*- coding: utf-8 -*-
"""
============================
Commands :mod:`dog.commands`
============================
"""
from argparseinator import ArgParseInated
from argparseinator import arg
from argparseinator import class_args


@class_args
class Commands(ArgParseInated):
    """Commands for dog"""

    @arg("word", help="The word", nargs="?", default="dog")
    def say(self, word="dog"):
        """says the word"""
        writeln("I woof", word)

    @arg()
    def name(self):
        """The pet name"""
        writeln("Walle")
```

The running script

As you can see the script loads the submodules/package dinamically at runtime.

Listing 8.15: pets.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
"""
==============================
Generic ArgParseInator Launcer
==============================

Generic ArgParseInator Launcer.
"""
from __future__ import print_function
import sys
from glob import glob
from os.path import basename, splitext, join, dirname
from importlib import import_module
try:
    # we try to import the core module (pets/__init__.py in this case)
    MOD = import_module(splitext(basename(sys.argv[0]))[0])
except ImportError:
    MOD = None
try:
    # we try to import the wanted module
    sys.modules[__name__] = import_module(splitext(basename(sys.argv[1]))[0])
    # modify the sys.argv to set it as starting script or the startin command.
    sys.argv[0] = sys.argv.pop(1)
except (ImportError, IndexError):
    # if we have a import error will and we have found the core module we
    # will set is as the main module.
    if MOD:
        sys.modules[__name__] = MOD
    else:
        print("Valid module are:")
        print("\n".join(set([
            basename(dirname(d))
            for d in glob(join(dirname(sys.argv[0]), "*/__init__.py*"))])))
        sys.exit(1)
# we use the builtin __argpi__ object to check the command line.
__argpi__.check_command()
```

We can run the script in various ways

```
# Using the *core* module name *pets*
$ python pets.py pets allsays

# No module name uses the default pets commands.
$ python pets.py allsays

# both cases the result is
I woof dog
I miaow cat
I tweet bird

# Passing the submodule name
$ python pets.py dog say "I'm a coold dog!"
```

```
I woof I'm a cool dog

# submodules preserves the core module commands
$ python pets.py dog allsays

I woof dog
I miaow cat
I tweet bird
```

Index

A

ArgParseInated (built-in class), 19
ArgParseInator (class in argparseinator), 13

C

check_command() (argparseinator.ArgParseInator method), 18

W

write() (argparseinator.ArgParseInator method), 18
writeln() (argparseinator.ArgParseInator method), 18