Some time ago I was working on a simple python script. What the script did is not very important for this article. What is important, is the way it parsed arguments, and the way I managed to improve it.

All below examples look similar to that script, however I cut most of the code, and changed the sensitive information, which I cannot publish.

The main ideas for the options management are:

- The script reads all config values from a config file, which is a simple ini file.
- The script values can be overwritten by the command line values.
- There are special command line arguments, which don't exist in the config file like:
  - –help - shows help in command line
  - –create-config - creates a new config file with default values
  - –config - the path to the config file which should be used
- If there is no value for a setting in the config file, and in the command line arguments, then a default value should be taken.
- The option names in the configuration file, and the command line, must be the same. If there is `repo-branch` in the ini file, then there must be `--repo-branch` in the command line. However the variable where it will be stored in python will be named `repo_branch`, as we cannot use `-` in the variable name.

The Basic Implementation

The basic config file is:

```
[example]
repo-branch = another
```

The basic implementation was:

```
#!/usr/bin/env python

import sys
import argparse
import ConfigParser

import logging
logger = logging.getLogger("example")
logger.setLevel(logging.DEBUG)

ch = logging.StreamHandler()
formatter = logging.Formatter('%(asctime)s - %(name)s : %(lineno)s - %(levelname)s - %(message)s')
ch.setFormatter(formatter)
logger.addHandler(ch)

class Options:

    def __init__(self, args):
        self.parser = argparse.ArgumentParser(description="Example script.")
        self.args = args

        self.parser.add_argument("--create-config",
```

```
                                dest="create_config",
                                help="Create configuration file with default values")

        self.parser.add_argument("--config",
                                dest="config",
                                default="/tmp/example.cfg",
                                help="Path to example.cfg")

        self.parser.add_argument("--repo-branch",
                                dest="repo_branch",
                                default="something",
                                help="git branch OR git tag from which to build")

        self.options = self.parser.parse_args()
        print "repo-branch from command line is: {}".format(self.options.repo_branch)

        # Here comes the next about 20 arguments

    def get_options(self):
        return self.options

    def get_parser(self):
        return self.parser

class UpgradeService():

    def __init__(self, options):
        if not options:
            exit(1)
        self.options = options
        if self.options.config:
            self.config_path = self.options.config
            self.init_config_file()
        self.init_options()

    def init_config_file(self):
        """ This function is to process the values provided in the config file """

        self.config = ConfigParser.RawConfigParser()
        self.config.read(self.config_path)

        self.repo_branch = self.config.get('example', 'repo-branch')

        # HERE COMES OVER 20 LINES LIKE THE ABOVE

        print "repo-branch from config is: {}".format(self.repo_branch)

    def init_options(self):
        """ This function is to process the command line options.
            Command line options always override the values given in the config file.
        """
        if self.options.repo_branch:
            self.repo_branch = self.options.repo_branch
```

```
        # HERE COMES OVER 20 LINES LIKE THE TWO ABOVE

    def run(self):
        pass

if __name__ == "__main__":
    options = Options(sys.argv).get_options()
    upgrade_service = UpgradeService(options)

    print "repo-branch value to be used is: {}".format(upgrade_service.repo_branch)
    upgrade_service.run()
```

The main idea of this code was:

- All the command line arguments parsing is done in the `Options` class.
- The `UpgradeService` class reads the ini file.
- The values from the `Options` class and the ini file are merged into the `UpgradeService` fields. So a config option like `repo-branch` will be stored in the `upgrade_service.repo_branch` field.
- The `upgrade_service.run()` method does all the script's magic, however this is not important here.

This way I can run the script with:

- `./example.py` - which will read the config file from `/tmp/example.cfg`, and the `repo_branch` should contain `another`.
- `./example.py --config=/tmp/a.cfg` - which will read the config from the `/tmp/a.cfg`.
- `./example.py --help` - which will show the help (this is automatically supported by the `argparse` module).
- `./example.py --repo-branch=1764` – and the `repo_branch` variable should contain `1764`.

The Problems

First of all, there is lots of repeated code. We have each option name mentioned in the command line arguments parser, also in the config file. What's more the variable name where we store the value is repeated a couple of times, with repeated logic, which is not the same for all options.

This makes the code hard to update, when we change an option name, or want to add a new one.

Another thing is a simple typo bug. There is no check if an option in the config file is a proper one. When a user, by a mistake, writes in the config file `repo_branch` instead of `repo-branch`, it will be ignored.

The Bug

One question: can you spot the bug in the code?

The problem is that the script reads the config file. Then overwrites all the values with the command line ones. What if there is no command line argument for `--repo-branch`? Then the default one will be used, and it will overwrite the config one.

```
./example.py --config=../example.cfg
repo-branch from command line is: something
repo-branch from config is: another
repo-branch value to be used is: something
```

Fixing Time

I tried to implement a better solution, it should fix the bug, inform user about bad config values, be easier to change later, and give the same result: the values should be used as `UpgradeService` fields.

The class `Option` is not that bad. I left it, however I moved all the default values to another dictionary. So now, if there is no command line option e.g. for `--repo-branch` then the `repo_branch` field in the object returned by the method `Options.get_options()` will be `None`.

After the changes, this part of the code is:

```
DEFAULT_VALUES = dict(
    config="/tmp/example.cfg",
    repo_branch="something",
)

class Options:

    def __init__(self, args):
        self.parser = argparse.ArgumentParser(description="Example script.")
        self.args = args

        self.parser.add_argument("--create-config",
                                 dest="create_config",
                                 help="Create configuration file with default values")

        self.parser.add_argument("--config",
                                 dest="config",
                                 help="Path to example.cfg")

        self.parser.add_argument("--repo-branch",
                                 dest="repo_branch",
                                 help="git branch OR git tag from which to build")

        self.options = self.parser.parse_args()
        print "repo-branch from command line is: {}".format(self.options.repo_branch)

        # Here comes the next about 20 arguments

    def get_options(self):
        return self.options

    def get_parser(self):
        return self.parser
```

So I have a dictionary with the default values. If I would have a dictionary with the config values, and a dictionary with the command line ones, then it would be quite easy to merge, and compare.

Get Command Line Options Dictionary

First let's make a dictionary with the command line values. This can be made with a simple:

```
def parse_args():
  return Options(sys.argv).get_options().__dict__
```

However there are two things to remember:

- There is the command `--create-config` which should be supported, and this is the best place to do it.
- The arguments returned by the `__dict__`, will have underscores in the names, instead of dashes.

So let's add creation of the new config file:

```
def parse_args():
    """ Parses the command line arguments, and returns dictionary with all of them.

    The arguments have dashes in the names, but they are stored in fields with underscores.

    :return: arguments
    :rtype: dictionary
    """
    options = Options(sys.argv).get_options()
    result = options.__dict__
    logger.debug("COMMAND LINE OPTIONS: {}".format(result))

    if options.create_config:
        logger.info("Creating configuration file at: {}".format(options.create_config))
        with open(options.create_config, "w") as c:
            c.write("[{}]\n".format("example"))
            for key in sorted(DEFAULT_VALUES.keys()):
                value = DEFAULT_VALUES[key]
                c.write("{}={}\n".format(key, value or ""))
        exit(0)
    return result
```

The above function first gets the options from an `Options` class object, then converts it to a dictionary. If there is the option `create_config` set, then it creates the config file. If not, this function returns the dictionary with the values.

Get Config File Dictionary

The config file converted to a dictionary is also quite simple. However what we can get is a dictionary with keys like they are written in the config file. These will contain dashes like `repo-branch`, but in the other dictionaries we have dashes like `repo_branch`, I will also convert the keys to dashes.

```
CONFIG_SECTION_NAME = "example"
def read_config(fname, section_name=CONFIG_SECTION_NAME):
    """ Reads a configuration file.

    Here the field names contain the dashes, in args parser we have underscores.
    So additionally I will convert the dashes to underscores here.

    :param fname: name of the config file
    :return: dictionary with the config file content
    :rtype: dictionary
    """
    config = ConfigParser.RawConfigParser()
    config.read(fname)
```

```
    result = {key.replace('-','_'):val for key, val in config.items(section_name)}
    logger.info("Read config file {}".format(fname))
    logger.debug("CONFIG FILE OPTIONS: {}".format(result))
    return result
```

And yes, I'm using dictionary comprehension there.

Merging Time

Now I have two dictionaries with values.

- The DEFAULT_VALUES.
- The config values, returned by the `read_config` function.
- The command line values, returned by the `parse_args` function.

And I need to merge them. Merging cannot be done automatically, as I need to:

- Get the DEFAULT_VALUES.
- Overwrite or add values read from the config file.
- Overwrite or add values from command line, but only if the values are not `None`, which is a default value when an argument it not set.
- At the end I want to return an object. So I can call the option with `settings.branch_name` instead of the `settings['branch_name']`.

For merging I created this generic function, it can merge the `first` with the `second` dictionary, and can use the `default` values for the initial dictionary.

At the end it uses the `namedtuple` to get a nice object with fields' names taken from the keys, and filled with the merged dictionary values.

```
def merge_options(first, second, default={}):
    """
    This function merges the first argument dictionary with the second.
    The second overrides the first.
    Then it merges the default with the already merged dictionary.

    This is needed, because if the user will set an option `a` in the config file,
    and will not provide the value in the command line options configuration,
    then the command line default value will override the config one.

    With the three-dictionary solution, the algorithm is:
    * get the default values
    * update with the values from the config file
    * update with the command line options, but only for the values
      which are not None (all not set command line options will have None)

    As it is easier and nicer to use the code like:
        options.path
    then:
        options['path']
    the merged dictionary is then converted into a namedtuple.
```

```
    :param first: first dictionary with options
    :param second: second dictionary with options
    :return: object with both dictionaries merged
    :rtype: namedtuple
    """
    from collections import namedtuple
    options = default
    options.update(first)
    options.update({key:val for key,val in second.items() if val is not None})
    logger.debug("MERGED OPTIONS: {}".format(options))
    return namedtuple('OptionsDict', options.keys())(**options)
```

Dictionary Difference

The last utility function I need is something to compare dictionaries. I think it is a great idea to inform the user that he has a strange option name in the config file. Let's assume, that:

- The main list of the options is the argparse option list.
- The config file can contain less options, but cannot contain options which are not in the argparse list.
- There are some options which can be in the command line, but cannot be in the config file, like `--create-config`.

The main idea behind the function is to conver the keys for the dictionaries to sets, and then make a difference of the sets. This must be done for the settings names in both directions:

- `config.keys` - `commandline.keys` - if the result is not an empty set, then it is an error
- `commandline.keys` - `config.keys` - if the result is not an empty set, then we should just show some information about this

The below function returns a tuple like (first-second, second-first). There is also the third argument, it is a list of the keys which we should ignore, like the `create_config` one.

```
def dict_difference(first, second, omit_keys=[]):
    """
    Calculates the difference between the keys of the two dictionaries,
    and returns a tuple with the differences.

    :param first:     the first dictionary to compare
    :param second:    the second dictionary to compare
    :param omit_keys: the keys which should be omitted,
                      as for example we know that it's fine that one dictionary
                      will have this key, and the other won't

    :return: The keys which are different between the two dictionaries.
    :rtype: tuple (first-second, second-first)
    """
    keys_first = set(first.keys())
    keys_second = set(second.keys())
    keys_f_s = keys_first - keys_second - set(omit_keys)
    keys_s_f = keys_second - keys_first - set(omit_keys)

    return (keys_f_s, keys_s_f)
```

Build The Options

And now the end. The main function for building the options, which will use all the above code. This function:

- Gets a dictionary with command line options from the `parse_args` function.
- Finds the path to the config file (from the command line, or from the default value).
- Reads the dictionary with config file options from the `read_config` function.
- Calculates the differences between the dictionaries using the `dict_difference` function.
- Prints information about the options which can be set in the config file, but are not.
- Prints information about the options which are in the config file, and cannot be.
- If there are any options which cannot be in the config file, the script exits with error code.
- Then it merges all three dictionaries using the function `merge_options`, and returns the

```
    """
    Builds an object with the merged opions from the command line arguments,
    and the config file.

    If there is an option in command line which doesn't exist in the config file,
    then the command line default value will be used. That's fine, the script
    will just print an info about that.

    If there is an option in the config file, which doesn't exist in the command line,
    then it looks like an error. This time the script will show this as error information,
    and will exit.

    If there is the same option in the command line, and the config file,
    then the command line one overrides the config one.
    """

    options = parse_args()
    config = read_config(options['config'] or DEFAULT_VALUES['config'])

    (f, s) = dict_difference(options, config, COMMAND_LINE_ONLY_ARGS)
    if f:
        for o in f:
            logger.info("There is an option, which is missing in the config file,"
                        "that's fine, I will use the value: {}".format(DEFAULT_VALUES[o]))
    if s:
        logger.error("There are options, which are in the config file, but are not supported:")
        for o in s:
            logger.error(o)
        exit(2)

    merged_options = merge_options(config, options, DEFAULT_VALUES)
    return merged_options
```

Other Changes

There are some additional changes. I had to add a list with the command line argumets, which are fine to be omitted in the config file:

```
COMMAND_LINE_ONLY_ARGS = ["create_config"]
```

The `UpgradeService` class is much simpler now:

```
class UpgradeService():

    def __init__(self, options):
        if not options:
            exit(1)
        self.options = options

    def run(self):
        pass
```

The runner part also changed a little bit:

```
if __name__ == "__main__":
    options = build_options()
    upgrade_service = UpgradeService(options)

    print "repo-branch value to be used is: {}".format(upgrade_service.options.repo_branch)
    upgrade_service.run()
```

Full Code

The code for the two implementations can be found on github:

- Initial version
- Improved version