ConfigMaster.py

Contents

Overview

This page documents a python module that can be used to facilitate handling of configuration files and command line arguments. The goal is to create a 'tdrp' like system for Python.

In particular ConfigMaster provides:

- Separation of parameter definition from script logic (i.e. tdrp's paramdef file).
  - A separate paramdef file is optional (constrast examples 1 & 2 below)).
- Ability to easily generate a default parameter file (i.e --print_params capability). The documentation included in the file helps the new user to understand the use of the program and customize the parameters.
- Encapsulation of parameter handling logic (contained in the ConfigMaster class)
- Simple default behavior. If no parameter file is used, the default parameter values are used.
  - Overriding of missing parameters with default values.
- Warning if you give a parameter in your config file that it doesn't expect.
- Automatic command line override for configuration parameters.
  - Only for int, long, float, string, and bool types currently
- Self-documenting. All parameter documentation is done in the paramdef, therefore the documentation and code are co-located, making it easy to keep the documentation up-to-date.
- Support for environment variables in the parameter files.
  - Other python code can also be included in the param files - date/time logic, file system logic, etc.
- Simple. Minimal coding required. Most work is done in the the paramdef file.

Repository Location

ConfigMaster for Python2.x is in cvs at libs/python/src/ConfigMaster

ConfigMaster for Python3 is in github at https://github.com/NCAR/ConfigMaster

Installation

Check out ConfigMaster.py from cvs and put it in a location that is in your $PYTHONPATH.

Examples

Example 1 - No separate paramdef

This is the general case, that I recommend for most stand-alone scripts.

cvs location: libs/python/src/ConfigMaster/example1

This script defines some basic configuration options in a triple-quoted string called defaultParams.

To use the ConfigMaster, you create an instance of it, pass it your default params, and then call it's init method with a simple description of your script. (I use the docstring as the description).

Now all of your parameters are available in a ConfigMaster dictionary:

```
#!/usr/bin/env python
'''
A simple script to show how ConfigMaster works.
'''

from ConfigMaster import ConfigMaster
def main():
    p = ConfigMaster()
    p.setDefaultParams(defaultParams)
    p.init(__doc__)

    print "Using these parameters"
    p.printParams()

        if p.opt["debug"]:
        print "\nDEBUG: Using forecast hour: {}".format(p.opt["forecastHour"])

defaultParams = """
####################################
## GENERAL CONFIGURATION
####################################

## debug ##
# Flag to output debugging information
debug = False

# Forecast hour
forecastHour = 3

# Email Address
emailAddy = "prestop@ucar.edu"
"""

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

Now you can get a usage statement which shows the command line options generated for you:

```
$ ./example1.py -h
usage: example1.py [-h] [-c CONFIG] [-p] [--debug] [--emailAddy EMAILADDY]
                   [--forecastHour FORECASTHOUR]
A simple script to show how ConfigMaster works.
optional arguments:
  -h, --help            show this help message and exit
  -c CONFIG, --config CONFIG
                        The configuration file.
  -p, --print_params    Generate a sample configuration file.
  --debug               Set debug to True
  --no-debug            Set debug to False
  --emailAddy EMAILADDY
                        Overide the param file value of emailAddy
  --forecastHour FORECASTHOUR
                        Overide the param file value of forecastHour
```

You can create a default config file:

```
$ ./example1.py -p
#!/usr/bin/env python

#####################################
## GENERAL CONFIGURATION
#####################################

## debug ##
# Flag to output debugging information
debug = False

# Forecast hour
forecastHour = 3

# Email Address
emailAddy = "prestop@ucar.edu"
```

You can modify that config file and use it to control your program's behavior.

Note that your config file must end in .py and it must not have any other periods in the name.

```
$ ./example1.py -p > example1_params.py
$ $EDITOR example1_params.py
# edit config file to change debug to True and forecastHour to 5

$ ./example1.py -c example1_params.py
Using these parameters
debug: True
emailAddy: prestop@ucar.edu
forecastHour: 5

DEBUG: Using forecast hour: 5
```

You can override default parameters on the command line:

```
$ ./example1.py --debug
Using these parameters
debug: True
emailAddy: prestop@ucar.edu
forecastHour: 3

DEBUG: Using forecast hour: 3
```

Command line options also override config file options: (**note: the .py is optional when passing the name of the config file via the command line**)

```
$ ./example1.py -c example1_params --forecastHour 6
Using these parameters
debug: True
emailAddy: prestop@ucar.edu
forecastHour: 6

DEBUG: Using forecast hour: 6
```

Try modifying the config file to accidentally misspell a parameter

```
$ ./example1.py -c example1_params.py
ERROR: Invalid parameter in configuration file example1_params.py: forecasthour
```

Example 2 - Using a separate paramdef

This can be useful when you want to have a set of parameters to share between multiple scripts.

cvs location: libs/python/src/ConfigMaster/example2

This example isolates the parameter definition into it's own file

Create a parameter definition file. By convention I name this <script>_pdef.py. Python requires it to end it .py, and it cannot have any other periods in the filename.

Note: this _pdef.py file must also be in a location in your $PYTHONPATH

Here is the pdef:

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

from ConfigMaster import ConfigMaster

class Params(ConfigMaster):
  defaultParams = """
#!/usr/bin/env python

####################################
## GENERAL CONFIGURATION
####################################

## debug ##
# Flag to output debugging information
debug = False

# Forecast hour
forecastHour = 3

# Email Address
emailAddy = "prestop@ucar.edu"

"""
```

Then you import your derived class and use that instead of ConfigMaster:

```
import example2_pdef as P
def main():
    p = P.Params()
```

Example 3 - More advanced config file options

This example shows how to do things like have conditional options, use environment variables, and use python in your config file.

Everything is the same as example 1, except a more complicated defaultParams:

```
defaultParams = """
import os
import datetime
####################################
## GENERAL CONFIGURATION
####################################

## debug ##
# Flag to output debugging information
debug = False

# Forecast hour
if datetime.datetime.now().hour % 2 == 0:
  forecastHour = 4
else:
  forecastHour = 3

# Email Address
emailAddy = "prestop@ucar.edu"

dataDir = os.path.join(os.environ["HOME"],"data")

logFile = os.path.join(dataDir,"logs",datetime.datetime.now().strftime("%Y%m%d") + ".log")

"""
```

This example shows:

- how to use imported modules in your params
- how to use an environment variable (dataDir)
- how to set conditional parameters (forecastHour)
- how to use imported module methods like join(), now(), and strftime() (logFIle)

Example 4 - Adding additional command line options.

This isn't supported yet. See the ToDo list below.

Todo

- Add support for additional types for auto cmd line arguments. (i.e. lists, dictionaries, etc.)
- Add support for cmd line arguments that are not in the config file.
    - I think the way to handle this will be to add another method to the ConfigMaster class. addAdditionalArgs(), and I think it can just take (*args, **kwargs) and pass them to parser.add_arguments.
- Explicitly check for missing '.py' or extra '.' in config file name and give a better error message before exiting gracefully.
- Add support/examples for enumerated types
- Add support for quality control (e.g. min/max values & valid values (i.e. enumerated types))
- Parent/Child configurations - Add ability to include one configuration file in another. For example, several scripts might have the same parameter that should have the same value among different scripts. Can you put the shared bits into one configuration file, put the different bits into different configuration files, and then include one in the other?
    - Would the shared bit be in the parameter definition part, or in the runtime configuration part, or maybe both or either depending on the setup? I think ideally there would be the flexibility to do it any which way, but it does make things more complicated.
    - I think we can do this with the python built-in execfile()

Limitations

Overriding configuration options that are used to set other configuration options.

The default configuration definition is processed before any configuration files or command line overrides. This means that if you define a variable in your default configuration, and then use that variable when creating further variables, you cannot simply override the original variable and expect the later variables to also get updated. You need to override all subsequent variables. For example, if this is your default configuration definition:

```
DATA_DIR = "/d1/data"
RADAR_DATA_DIR = os.path.join(DATA_DIR,"radar")
```

If you override DATA_DIR, it will not affect RADAR_DATA_DIR, because it has already been set based upon the original DATA_DIR.

You can instead override both DATA_DIR and RADAR_DATA_DIR:

Alternatively you can move DATA_DIR to an environment variable:

```
DATA_DIR = os.environ["DATA_DIR"]
RADAR_DATA_DIR = os.path.join(DATA_DIR,"radar")
```

Now you can set DATA_DIR when you setup your environment and it will use whatever $DATA_DIR is set to when your script runs.

Developer Notes: How else could this be solved? Could we somehow reparse the default configuration logic to account for the new DATA_DIR value? Would having parent/child configuration files help?