Configuration Files For Python

This is an article started by Russ Hensel, see "http://www.opencircuits.com/index.php?title=Russ_hensel#About My Articles" About My Articles for a bit of info. The page is only partly finished.

Contents [hide]

1 Why Configuration Files
2 Configuration in .py Files
2.1 How: The Basics
2.2 How: More Advanced
2.2.1 Data Types
2.2.2 Do a Little Math
2.2.3 Set Values for Your Computer
2.2.4 Override Values
2.2.5 Group Values
2.2.6 Conditionally Assign Values
2.2.7 My Overall Structure
2.3 Why Advantages/Features
2.3.1 Why Not
3 Editing and Editors
4 Other Links

Why Configuration Files [edit]

Most larger programs should have configuration files:

- Program becomes more flexible without reprogramming.
- Users can inject their own preferences.
- The environment changes, requiring that the program adapt.

There are a large number of formats for configuration files, some are accessed only through wizards and they may have a "secret" format. Some configuration files are not even really files but instead are entries in a data base. But most are stored in some sort of human readable text format and can be edited with a straight ahead text editor.

My SmartTerminal program now has over 50 different parameters that control its use in a variety of different applications. I have used a parameter file for: Python Control of Smart Plugs, Python Smart Terminal, Python Smart Terminal Graph, and other programs.

Configuration in .py Files [edit]

I have decided for my use that configuring everything in a single python, .py, file is the best solution for me (and I think for many of you) I will describe how I do it and then will give some of the reasons why I think the method is so very useful.

How: The Basics [edit]

No matter what the application I put the configuration in a file called parameters.py and use it to define/create a class called Parameters. It is full of instance variables like self.logging_id = "MyLoggingId". Any part of my system that wants to know a parameter value takes the instance of Parameters created at startup and accesses its instance value logging_id = system_parameter.logging_id. It is very easy.

You may ask how does that part of the system get the instance of of parameters? The best way is probably through a global singleton. It is more or less what I do. There seem to be a host of methods of implementing singletons. I use a little recommended one but one that I find more than adequate: I define a class and make the global variables be variables of the class not the instance. You can get access to the class just by importing it, creating an instance servers no particular purpose. So the global class, AppGlobal, is defined something like this (in a file app_global.py)

```
class AppGlobal( object ):

    controller              = None
    parameters              = None
```

This AppGlobal object has only 2 variables, both undefined = None initially.

Then parameters.py looks something like this:

```
from app_global import AppGlobal

class Parameters( object ):

    def __init__(self,  ):

        AppGlobal.parameters    = self
        self.logging_id         = "MyLoggingId"
        self.timeout_sec        = 3
        ............
```

A class instance that needs to use a parameter uses code like:

```
from app_global import AppGlobal

    .............

    timeout     = AppGlobal.parameters.timeout_sec
    ............
```

If you are asking why Parameters is not all defined at a Class level instead of instance level it is because I did not think of it then, I am now but have not changed the code so far (requires more thought).

How: More Advanced [edit]

The more advanced uses also point out some of the advantages of this method, you may have already noticed one. In most configuration files all data types are string. There needs to be some reading of the file and conversion. So you need to write (or get a library) to do that part. For Python you can think that either your are allowed types "joe" is a string 10. is a float, or that they are strings and the Python interpreter is the conversion program.

Data Types [edit]

One of my configuration values was most useful a some sort of dict so:

```
        self.hour_chime_dict   = {  1:"2", 2:"3", 3:"2", 4:"2",  5:"2", 6:"3", 7:"2",  8:"3", 9:"2", 10:"3", 11:"2", 12:"2" }
```

or something from a library

```
        self.bytesize         = serial.EIGHTBITS     # Possible values: FIVEBITS, SIXBITS, SEVENBITS, EIGHTBITS
        self.logging_level    = logging.INFO
```

values can even be functions in your own code.

Do a Little Math [edit]

```
        self.ht_delta_t        = 100/1000.        # /1000 so you can think of value as ms
```

Set Values for Your Computer [edit]

```
        self.computername   = ( str( os.getenv( "COMPUTERNAME" ) ) ).lower()
```

Now that can be used in parameters.py to conditionally set other values.

Override Values [edit]

It is easy to set a parameter value and then change your mind later in the code:

```
       self.name   = "sue"
       ...........
       self.name   = "Susan"
```

I use this by setting all parameters to some default value (using subroutine grouping also discussed on this page) and then overriding them later for the particular situation.

Group Values [edit]

I usually want to group values so that I can assign then all at once. Physically grouping them is of course the first level of doing this but since this is a class I use instance functions to do the grouping. Often one group is used for one purpose one group for another. I comment out a whole group by commenting out a call to it it. Here is a sample (all inside parameters.py):

```
    ......
    # choose one:
    self.dbLPi()
    #self.dbLocal()

    ......

    return

    def dbRemote( self, ):
        self.connect            = "Remote"
        self.db_host            = '192.168.0.0'
        self.db_port            = 3306
        self.db_db              = 'env_data'
        ..............

    def dbLocal( self, ):
        self.connect            = "Local"
        self.db_host            = '127.0.0.1'
        self.db_port            = 3306
        self.db_db              = 'local_data'
        ....
```

At some later point I may not remember well which grouping I used so I often give them names as in self.connect = "Remote" in the example.

Conditionally Assign Values [edit]

Sometimes I call these meta parameters, they control the way other parameters are set. Commonly I use:

- mode (self.mode in the parameter file)
- os_win
- computername

For example I can move the parameters.py file between OS's without tweaking it manually:

```
        if  self.os_win:
            self.port       = "COM5"
        else:
            self.port       = "/dev/ttyUSB0"
```

My Overall Structure [edit]

- Meet the syntactic requirements for class creation.
- Assign instance to AppGlobal
- Call a subroutine that defaults all value as best it can (including getting OS and computer name)
- Call a subroutine that tweaks values according to OS
- Call a subroutine that tweaks values according to computer name
- Call a subroutine at give a value to mode and sets the mode of operation that has the name self.mode
- Done

Code discipline is such that other code never touches these values again (except for some cute little monkey patching that I currently do not use and do not want to explain).

Why Advantages/Features [edit]

- Text file based, easy to edit, works in all OS's.
- Early in development use before features are add in a GUI.
- Can use programmatic features as it is in executable code. (if overdone can be confusing)
- Easy to create parameters of any type (*.ini files, for example, typically create strings that may need conversion to be useful)
- Can detect and respond to environment like OS or computer name.
- Easy to have multiple configurations (modes) in one file.
- and so many more..... really?

Why Not [edit]

Worth some consideration, may be important to you (but not for me in my situation)

- Need to keep legal Python Syntax.
- User may find too complex, mess up.
- Since it is programmatic you can be tempted to get too clever.
- May be less than secure.
- May not be what users are used to.
- Not your shops standard.
- Not XML. (actually for me this is a plus, I find XML hard to read)

Editing and Editors [edit]

You need a text editor sutiable for .py files to manage the parameter file (parameters.py) This includes most text editors. I particularity like:

- notepad++
- geany

You can also use the editor that comes in many python development environments, the simplest of which may be Idle. But there are many many others. If you are reading this you probably have some experience.

Once you configure an editor in parameters.py you can edit from the <Edit Parms> in the GUI (see below)

When editing there are couple of gotchas to watch out for.

- Python cares about capitalization, use the capitalization indicated in the default files and the example code.
- Python also cares a lot about how lines are indented. Do not change the indentation from the sample files, and always indent using spaces (not tabs. most text editors will use spaces automatically for .py files, even if you use the tab key)

Other Links [edit]

- Check out link to left "What links here"
- **Param — Param 1.9.0 documentation**
- **Smart Terminal Parameter Examples** documents how I used this approach in the **Python Smart Terminal** You can also link to the full code.
- **RecentChanges - Python Wiki**
- **14.2. configparser — Configuration file parser — Python 3.4.8 documentation**
- **Configuration files in Python · Martin Thoma**
- **4 Ways to manage the configuration in Python – Hacker Noon**
- **parsing - What's the best practice using a settings file in Python? - Stack Overflow**

Categories: Arduino/RaspberryPi | Python | SmartTerminal | Python SmartPlug