Configuration Files For Python

opencircuits.com/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.

This page discusses both the general virtues of using a parameter file written in Python, and the particular implementation, including some details I use in my applications.

Contents

Why Configuration Files

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

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

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 this: **logging_id = system_parameter.logging_id**. It is very easy.

You may ask how does some 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 of the instance. You can get access to the class just by importing it, creating an instance servers no particular purpose and I never do it. 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: New Users Guide

This section of this page is for a new user to the parameters as used in my applications. Later you can read the documentation below for more advanced users. For new users there are two parts of parameters.py that you should probably read. The method parameters.new_user_mode() is normally where you should put your personal settings. This alone is not sufficient, because you need to know what settings to include and how to choose values. For this refer to the method parameters.default_mode, where a default is set for all values (I hope) and comments document the settings. Also refer to the written documentation for each application.

Remember that the parameter.py file has to be legal Python.

How: More Advanced

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

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

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

Set Values for Your Computer

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

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

Override Values

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

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.

Modes

A feature that has emerged over time is that of "Modes". Modes are simply a grouping of parameter values for some use of the application. The smart terminal, for example, has a mode for working with a particular arduino program. Say the arduino program is for monitoring a green house. Then I might create a mode called "Green House". This has become important enough that I display the mode in the title bar of the application.

Default Mode

There is one mode that is always set/called. This is the default mode. Typically it sets all parameters and has a comment to document the parameter. After it is called, you normally call the method for a mode of your own where you change the mode name and change any values from the default mode that you need to. If you just run with the default mode the

program should basically run, but some function may misbehave, and you may not find the configuration much to your liking. If you eliminate the default mode the undefined values will probably cause an exception and crash the program.

You are encouraged to create new mode method, following the pattern of the ones shipped, and use them rather than changing mode values from the value as shipped.

A special mode like method is plus_test_mode. It is designed for you to make short term additions to another mode for testing, instead of changing the mode name it simply adds " + test" to the mode, so "Green House" would become "Green House + test". Again this shows in the application title bar.

Choose Mode

Modes have become so important for me that the first method in parameters is called choose_mode. It then lists all the mode methods (except default which is called earlier than choose_mode) and the plus_test_mode with all but the mode you are using commented out. Then just un-comment the mode you want and make sure the unwanted modes are commented out. Un-comment **plus_test_mode** if you want to run a test.

Running On

Since the program may call the OS for operations such as text editing you may want to tweak settings based on the OS. To do this there is a method **running_on_tweaks**, it is called after the default mode and tweaks the values for the environment (including the computer name or tcpip address). Information about what you are running on is collected by parameters.running_on, so the value parameters.running_on.os_is_win will be True if you are running on Windows. To see what values are set see running_on.py. Change running_on_tweaks() to suit your situation. In it you may find some tweaks for the computers I use (by name), you can leave them in or delete them, since it is unlikely that you will have computers of the same name.

Conditionally Assign Values

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

Check the __init__ method to see

- Meet the syntactic requirements for class instance creation.
- Assign instance to AppGlobal
- Call a subroutine that defaults all value as best it can for typical situations.
- Call a subroutine that tweaks values according to the environment the program is running on.
- Call a subroutine at give a value to mode and sets the mode of operation that has the name self.mode. Typically called from choose_mode()
- Done

Code discipline is such that other code never modifies 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

- 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

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

This documentation is old, you can now give a list of editors for the system to try. More revisions coming.

You need a text editor suitable 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

> **Check out link to left "What links here" This will, in part, link to projects using this type of configuration**
>
> **Param — Param 1.9.0 documentation**
>
> **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