Python Desk Top Applications

This is page about how I write my desktop applications, and so, naturally, how I would recommend they be writtern. It also documents the structure of most of the applications that I have documented on this wiki. Try them and let me know how you think they might be better.

See also: Python Smart Plug Technical

Contents

- 3 Coding Conventions
- 4 See Also

Overview

General Information

These notes are here so you can more easily modify my code and build your own.

Goals/Needs

I would like my applications to have the following characteristics:

- Runs on all PC like platforms, PC, PI, Linux -- self configurable to os .... as much as practical.
- Easily programmable for modifications.
- Easily configured, flexible
- Not too Ugly.
- GUI. Friendly
- Intuitive.
- Easy to Debug.
- Bug Free, robust.
- Easy, quick to program.
- Meaningful error messages.

Big Picture

Here is an overview of the general plan, details can be filled out by reading the example code in my projects (see category ..... below).

The overall architecture is called the model view controller or MVC.

The main program is the controller.

The view component is called GUI (typically gui.py with the class name GUI). It creates all the visible components, and relays user input to the controller. You can unplug the GUI object from the application and plug in new components. Don't like the GUI? You could modify mine, or you could make a modification and choose which one to use. This is sort of like a skin for an application.

Projects Here Done in this Style

Development Environment

- Windows 10 64bit
- Anaconda Spyder 4.x
- Python 3.7

Runtime Environments

Should Run in:

- Windows, Mac, Linux, Raspberry Pi with desktop support (not headless)
- Python 3.6 up.

I am now running in:

- Windows 10, Python 3.7
- Linux Mint
- Raspberry Pi Buster

I have had some issues with matplotlib in linux which are still unresolved.

Imported Libraries

Frequently or occasionally used:

- Tkinter for the GUI

- pyLogging for logging
- Second thread for processing without being blocked by/blocking the GUI
- Mathplotlib for graphing.
- SQLLite or MySQL for the database.
- os, sys, psutil, platform .. and pathlib for system access.

My Components

Some of my components (more information below)

- Global Singleton for app cohesion.
- Parameter file for easy customization.
- MVC component for application control.
- Multithreading, when required, most of code in its own class.
- Typically one class, **GUI**, for the graphical user interface.

My Style

I have written a good number of Graphical User Interface (GUI) applications and keep re using the parts that work well while slowly evolving the parts that do not work well or are hard to program, or just for some reason seem wrong.

The languages I have used in the past did not really have a concept of module, so I do not do much programming at the module level. Mostly I use modules to organize my classes, most modules have only a couple of classes at most.

My immediate prior language was Java which is very class oriented, as a result I probably overuse classes. But while a bit intelligent, and perhaps non zen, it works well enough.

Components and Functions

The Controllers for Applications

The class for the MVC controller is also the class you create to run the application: see the code at the bottom of the file, just run the file. Similar code is at the bottom of some of the other source files to make it convenient to run from those files, this is just to make it easier in development, the code in smart_plug.py is the model for what should be used. Sometimes the code at the bottom of other file may have code for testing objects in the file, it may not be maintained, may not work as intended.

The __init__ method is the initialization and "run" method for the application. Much of the code that would normally be in it has been moved to .restart which is used to restart the application when just the parameters have been changed. See the docstring there.

GUI

The view component is called GUI (typically gui.py with the class name GUI). It creates all the visible components, and relays user input to the controller. You can unplug the GUI object from the application and plug in new components. Don't like the GUI? You could modify mine, or you could make a modification and choose which one to use. This is sort of like a skin for an application.

Singletons

Most classes are used as singletons, although they may not be coded that way. There is little to stop programmers from creating multiple instances, but why would they. Keep in mind an application is not a library. So in addition to AppGlobal we have

- AppGlobal -- used as a class, never any instances
- GUI a single instance is the user interface.
- Main program, controller. A single instance.
- Helper thread Class a single instance which has most of the hellp thread code, or acts as a controller for the second thread and communicates with the "main" or GUI thread

AppGlobal

Global Values: Yes I know that globals are bad, but they can be useful. For example many class instances need to access the parameter file. This can be done using the singleton class AppGlobal. It has values at the class level (not instance) that are available simply by importing the class. Many values are originally defaulted to None, and are set to valid values as the application initializes and runs, starting, default values often come from parameters, or for instance variables from self as the instance is initialized. Some values that used to be shared through an instance of the application are now shared through AppGlobal. This code change is still in process. AppGlobal has now expanded to include globally useful functions.

RunningOn

A new class RunningOn has fairly recently be added to gather information about the platform the application is running on. This information is later used to adjust the application automatically when moved from computer to computer (including OS changes) without any code or parameter changes. It is a bit tricky because some of the functions it uses and value obtained vary from OS to OS. So to test it you need to "move it aroung" some. Still seems to work pretty well.

Variables in RunningOn may be use in parameters to branch based OS or computer name or TCP IP address. Occasionally values will be used in other code.

Paths and File Names

There are a couple of challenges with paths and file names:

- Are they relative or absolute?
- How to OS differences effect the code?

My current approach:

> For files that are "close" to the code, in your system for the application, I try to use file names that are relative to the installation location of the main Python application. To facilitate that the application tries to determine the startup location of the application and change the default directory to that location. Then notations like "./images/something.png" seem to work in the code. Since slashes seem to work in windows I try to avoid back-slashes.
>
> For system resources, like your editor, I use both bare filenames -- the system can often find them through environmental paths, or full file names. These are mostly in parameters.py so you can use what works for you.

Threads

The application runs two threads, one for the GUI and one where processing can occur (HelperThread) with out locking up the GUI. There are 2 queues that allow the threads to communicate. Multithreading is not used in the graphing application as of now.

The Tkinker Thread

Once Tkinker is started it runs its own mainloop. In order to receive however we need to check the rs232 port from time to time. This is done in SmartTerminal.polling() I call this thread the GUI thread or gt. The terminal is configured to have a second thread called the

Helper Thread or ht. In some cases the receiving of data is passed to the Helper Thread so where data is processed in addition to being posted to the GUI. See the section HelperThread for more info.

HelperThread

The application has a main thread running in a Tkinter mainloop. There is also a second thread called a "helper" running which makes some processing much easier. To make GUI mainloop responsive to both the GUI and its own processing it uses a pseudo event loop or a polling subroutine that is implemented in xxxxx. The frequency which polling occurs is set in parameters, the relatively low rate of 100 ms between calls (.1 sec) seems to give a perfectly responsive application in most cases. I have run it as fast as once every 10 ms. Have not tried to find a limit.

Helper Thread in Smart Terminal

Smart Terminal is fairly typical: HelperThread in smart_terminal_helper.py This class provides the support for a second thread of execution that does not block the main thread being run by Tinker. I call the two threads the GUI Thread (gt) and the Helper Thread (ht). It can get confusing keeping track of which method is running in which thread, I sometimes annotate them with gt and ht. The helper thread is started by running HelperThread.run() which pretty much just runs a polling task in HelperThread.polling(). HelperThread.polling() is an infinite loop, it uses sleep to set the polling rate. When used with the green house processing module, it may call a function there that is its own infinite loop. There are a lot of details here, I should write some more about it.

Polling

Both threads have method that perform polling for events often for items in their queue that may have been sent from the other thread. Info on a similar app in Python Smart Terminal Technical Details. Polling is also used in some single threaded applications. Parameters for polling are set in parameters.py

Parameters

This is a fairly big topic. It is how you or your user easily controls some aspect of the program.

Parameters (in parameters.py) this is pretty much a structure (that is all instance variables) that is instantiated early in the life of the application. It passes values, strings, numbers, objects to application elements that need them. The instance of parameters is

made globally available thru the AppGlobal class. Values in Parameters are treated as constants, read only. Much of the appearance and behavior of the application is controlled here.

The standard gui has a button to kick off editing of this file, the application may then be restarted (another button) with the new values.

There are a couple of meta parameters, including os_win, mode and computername which then may be used in conditionals later in parameters.py. Except for this sort of thing there is really not much "code" in parameters. You can change this code pretty much as much as you like, as long as you end up setting up values for the required parameters.

The code is extensively commented: use that for documentation.

Values are often set to None as a default, then later set to some other value. Or the value may be set several times in a row (this is an artifact of messing with the values); only the last value set has any meaning.

For more info see: Configuration Files For Python

DataBase

For databases I have used both SQLLite and versions of MySQL. See application for which one it uses. I may expand on this section later for now see the code.

Logging

This uses the standard Python logging class. Logging level and other logging details are controlled using the parameter file. Logger (in logger.py) and Parameters (in parameters.py). The controller creates one of each, and make them available to the other components. The other components can interact with them, and uses them respectively for logging events, and getting access to parameters (those aspects of the application that are particularly easy to change). I describe more of this in My Python Coding Conventions

System Editor

I often use the system editor, for example to view the log files. This has gotten more complicated over time (and is still a bit in flux now). The current approach is to give the system a list of possible editor programs (some hardcoded and some via parameters) and try each until one works or all are tried. The system then remembers the one that works. Code for this is mostly in AppGlobal with a bit in parameters.

Other Classes

For now the documentation, as far as it exists is in the source code. This probably will not change.

Coding Conventions

See: My Python Coding Conventions

See Also

Look at the categories below, especially the one Python Projects Python Control of Smart Plugs