Russ Python Tips and Techniques

opencircuits.com/Russ_Python_Tips_and_Techniques

Contents

What

These are Tips and Techniques I have found useful in my Python programming User:Russ_hensel. I would be interested in your feedback. Let me know if you think the tip is useful/wrong/something everyone knows. A good set of this material relates to applications with a GUI. They are often not beginner techniques, but neither are they all advanced ones. Just starting Feb 2017 check history tab above to see if progress is being made. If you find a tip not finished or working feel free to contact me.

Ones Still To Be Written

- when division by 0 is useful
- code is multiple files - scratch file
- moving to 3.6 use fstrings
- example files with functions Python Example Code
- when you need a reference and when not
- dict based case statements
- choose names for cut and paste
- mark your code with unusual string * zzz in your code

Move these to a good place in the outline

FStrings

FStrings are incredibly nice. New in 3.6, use it (or 3.7....) They also are reputed to be really fast. Use them. At a later time I may have a bunch of examples, but for now just the basic statement:

```
joe  = "Joe Smith Jones Norton"
msg  = f"Joe's name is actually {joe}"
print( msg 0
```

Spyder has a nice feature for code navigation: the "Outline Tab". This tab outlines the code nesting classes, methods.... and expands and contracts the tree much like the code folding show in the editor. The outline tab works both to help you navigate to a place in the code, or to show you where in the code the cursor currently lies. What is less obvious is that comments can also mark a section. I have not found documentation for this but here is what I have found. A comment like "# ------ frame building methods --" will create an outline section whose indentation in the outline depends on indentation of the comment. The space and the "-" characters are important. Start by using exactly what is shown here, and experiment to see how much you can change it before it disappears from the outline. I often use this to group a large number of similar methods that can then be folded out of view.

msg

A name I use over and over again for messages, often like:

```
msg  = "this is a message"
print( msg )
```

Why two lines instead of one? Because I often change it to include logging or sending to the GUI. It might then become:

```
msg  = "this is a message"
print( msg )
AppGlobal.logger.debug( msg )
```

I find this a nice way to manage code.

#rint

If you use print for debugging you may want to scan your code of "print" to remove unwanted statements. I do not delete them because sometimes I want them again for more debugging. I change "print" to "#rint" to comment out the print but to stop it form showing

up on searches.

Make Code Searchable

I like generic names to enable cut and paste but this runs against the idea of searchable code. The idea is that unusual names are easy to search for. It also helps if names are descriptive. Why searchable? So you can find the code you want to work on. One way to be both searchable and generic is to take instance references and make local (or the reverse) For example consider this tkinter code:

```
      a_list     = list(self.sort_order_dict.keys())
      a_widget   =  ttk.Combobox( a_frame, values = a_list,  state='readonly')
      a_widget.grid( row = lrow + 1, column = lcol, rowspan = 1, columnspan = 3,  sticky = E + W
+ N + S )
      a_widget.set( a_list[0] )
      self.sort_order_widget = a_widget
      lcol   += 3
```

The name a_widget is generic, but to see what it "really" is you have the none generic name "self.sort_order_widget"

If you do not see an easy way to do this with names use a comment. Using zzzz.... is a short term way of doing this.

Dict Based Case Statements

Python does not have a case statement, but it does have if....elif which can be used and is only a little more verbose. However if you have more than a few cases it can be cumbersome. Using a dict gives a very compact and fast case statement. Suppose you want to have a case statement based on strings:

```
# set up, ideally only done onece, perhaps at module or class level
dispatch_dict   = { "one":    print_one,
                    "two":    print_two,
                    "three":  print_three,
                    }

# set a case for an example call:
key = "two"

# the slightly verbose case statement
function      = dispatch_dict[ key ]
function()
```

Parameters for Applications

There are various ways of storing start up parameters for applications. In the old days almost all windows programs had a "ini" file. Linux had/has "config" files. In both cases these are text files that are easily edited with a simple text editor. More recently xml files (which i tend to hate) have come into vogue. Finally some applications give you a gui for editing configuration values and then save them any way they want, perhaps in the "registry".

Using a gui is the easiest way for the user in particular the naive user. They are also a lot of work to implement and maintain. So I favor a flat, non-xml file. But what file? For Python my answere is a python.py file that has pretty much one class, Parameters, that is implemented as a singleton (in what ever way you want to implement singletons). These are easy to implement, easy to maintain, and very flexible. They can also do clever things like detecting what operating system they are running on and make automatic adjustments. There are no issues of converting type into strings for the file. Comments are easily included. Loading can print the progress of initialization.

I will put a brief sample here soon so you can see what a file looks like.

code comming here

Logging

I use the logging class that is part of standard python. I provide a button to view and edit the file. Why invent your own method. Here is some code:

code comming

Restart

When an application (here I am thinking about GUI applications) starts there can be a considerable delay as all the modules are imported. Typically a restart is done after an exception or when parameters are changed. So I include a restart button on the GUI to quickly restart the application. In my case applications are usually started by constructing some application class. So what I do is break up the __init__ method into 2 parts, only the second needs to be run to restart the application. Here is some sample code:

```
def __init__(self ):
    """
    create the application, its GUI is called self.gui which is
    created in restart
    """
    ..... more code ......

    self.restart( )    # continue in restart

# -------------------------------------------------------
def restart(self ):
    """
    use to restart the app without ending it
    parameters will be reloaded and the gui rebuilt
    call from a button on the GUI
    args: zip
    ret:  zip ... all sided effects
    """

    self.no_restarts    += 1 # if we want to keep track
    if not( self.gui is None ):  # determines if a restart or the start
        # !!  # need to shut down other thread
        self.gui.root.destroy()
        reload( parameters )

    self.logger         = None

    .... and so on ......
```

Names

I have heard it said that proper naming is the hardest problem in programming. Not sure if true, but it is not easy. So here is my take:

> Of course be consistent. It may help to make a vocabulary list of the terms you prefer. I have found that the values often come in pairs:

```
start_date / stop_date  or low_temp / high_temp ( or hi_temp ) or start_date / end_date
then some style are:
date_start / date_stop
```

Note that the stop_date and end_date is an easy place for confusion. So decide on your style and stick to it.

- Do not be afraid of long names but consistent abbreviations are fine (keep a list).
- I generally think that methods should be named with verb like names: os_open_file.
- I find that people use the name **file** for both the **file name** (often string like) and for the **file handle** (the thing you get when you open a file), so my variables for file names are like:

```
input_file_name ( sometimes input_fn )
```

> Lists and other iteratables often get a name to indicate that: so a list of names might be called something like student_names
>
> Sometimes generic names are preferable especially if there scope is just a few lines. I am willing to iterate over a list of numbers with i, but I prefer ix and even then I fell lazy. If I have a list of names as above I might itterate with the singular, but usually i prefix it with i_ so:

```
for i_name in student_names
   print( i_name )
```

Extreme programming trys to make the code so obvious that it is easy to tell what the code is doing without comments. If not taken too far I think this is a great idea. It requires really good names. But most of my comments do not tell what the code is doing, they tell either why the code is doing something or what the purpose is of doing what ever. But that is not all:

> Comments are left to indicate where the code might be made better:
>
> - More careful error management.
> - Where I am thinking about refactoring
> - To make up for deliberately generic names -- ?? I need more explanation here
> - To provide a search string so I can find a place in the code
> - Where a little trap may lurk in the code:

For example most methods return at the end, but sometimes you discover in your processing that you are done
early. You could put some sort of if then structure in, but now you indent one level more making the
code somewhat more difficult. So just return. In these cases I may put a comment at the beginning of the method:

```
# beware of early return
```

I try to indicate that there may be an early return, but refactoring may remove it and leave the comment in place

> Of course no comment should be considered 100% reliable

Copy and Paste

There are lots of arguments against copy and paste, but used carefully I think that it is extremely useful. But used carefully.

One of the biggest arguments against it is that you may be using code that you do not really understand. This can be true, but the same is true when you use someone else's module.

So I have a procedure depending on where I copy from:

> My own code:
> - I only take working code.
> - I reread it and make appropriate changes.
> - I own the results, if it does not work properly I am to blame
> - I write code in a generic way to make it easy to copy and paste.
>
> Refactoring:
> - Of course when I refactor I use copy and paste, because basically your are just rearranging code you already have that works.
> - Many people use git to make sure they do not loose stuff in translation. I just use file copies, I have my own methods
> - Sleep that is zzzzzzz: I put zzzz or similar around the place where I am making changes. It throws a quick syntax method if I foolishly try to run the code and is easy to find as zzzz is not anything I would normally use in code. Sometimes I copy a lot of junk in between a pair of zzz so I have all the names I need in easy view. Then I do the new code, remove the junk and finally the pair of zzzz'a.

Other peoples code:

- I put there code in an example file.
- I change the code to my style and comment it, often keeping a url to the source.
- I run it to make sure it does what I want.
- I may make it generic in a way to make copying easier.
- I save the example.

Python Code Example Files

When I learn a new thing in Python I often keep an example file. I can then use it for reference and/or copy code out of it for other uses. I have a few "tricks". See: **Python Code Example Files**

Complex Tkinker Windows

I found it really easy to get confused in coding Tkinter because it is not a visual interface. One of my goal in coding is to make the code create the gui from top to bottom and from left to right. That way the layout of code and gui are easier to match up in my mind. This has a benefit of making it relatively easy to:

- Cut and paste code to rearrange placement in the gui.
- Cut/paste/modify code to build new widgets.

So here are some tips:

- I do a sketch of what I want on paper.
- The GUI is built in a class.
- I break the sketch into Tkinter frames each of which will layout into the window in some fairly simple scheme, often the pack layout will work. When this is not enough I use grid.
- When using grids I always have variables for the row and column positions. These are typically managed with statements like lrow += 1. Coding this way make movement of the code much easier. Placement methods may also be used to manage the row and column variables.
- Each individual frame is built in its own method, it takes the parent frame as an argument and returns the frame that it builds. The returned frame is then placed in its parent.

- References to the widgets are often not needed once the GUI is built, so local variable are often used. Using a local variable makes it easier to copy/paste. I try to always use the name a_widget, thus code that places or otherwise manipulates the widget in a generic way (like placement) works well in a copy paste. If I need a reference, I typically will do that on the last line of the code for the widget. For widgets that need references you can use an instance variable of the class, or in some cases it it useful to store them in a dictionary.
- Building helper classes for either building the widgets, or placing them or both is often useful.
- You can see these techniques in GUI class of my various application documented on this wiki.
- See: **Python Buttons Switch Case** for some tricks on building Tkinter frames with lots of buttons linked to functions.

Threading with Tkinker

Tkinker wants to own the processing of the application. If you make a button start a long running operation then the GUI blocks and becomes unresponsive. So you need some sort of threading. Here are some tips.

Polling withing Tkinker

You can poll in the Tkinter event loop and call any function (no arguments I think) using a Tkinter function. Use this to run something short. You can only run it once this way.

```
    self.gui.root.after( self.parameters.gt_delta_t, self.polling )  # self.gui.root = Tkinter root, .1 = .1 sec, self.polling = name of the function
                                                                     # ( note no () which is a function call
```

To overcome the fact that it only runs once, ask for it again at the end of the function.

```
def polling( self, ):
    ..... do something quick

    self.gui.root.after( .1, self.polling )
```

If you function throws an exception it will stop running. Try Except works for me:

```
def polling( self, ):

    try:
        ..... do something quick
    except Exception, ex_arg:
        .... I log the error
    finally:
        self.gui.root.after( .1, self.polling )
```

Use Threading

Use real threading. I inherit from threading.Thread and implement a override of the run method. Use threading mechanisms to "message" back and fourth to the gui, Tkinter, thread. Seems to work fine.

Calling across Threads

If you have objects in 2 different threads calling across the threads can be very dangerous. I have worked out a method that I find useful/safe/easy. It does make some assumptions.

> Both objects/threads have something resembling polling where checks can be made at a convient time to see if the other thread wants attention.

Step 1, Set Up Queues

```
    self.queue_to_helper      = Queue.Queue( 20 )  # send from tkinker mainloop to helper here
    self.queue_fr_helper      = Queue.Queue( 20 )  # send back from helper thread to tkinker mainloop here
```

So how does this help making calls across the threads? What I am doing in SmartTerminal (could change, see the code) is to always pass a tuple of 3 parts:

```
( a_string, a_function, another_tuple_of_arguments_to_a_function )
```

if a_string is "call" then the queue item is for a function call (other cases not discussed here). Then the function (remember functions are first class objects math.sin(x) call to a function, math sin a function) is combined with the arguments to make the call: a_function(arguments_to_a_function). Almost, it will not work because argument_to_a_function is itself a tuple. And even if the tuple has five elements you are only passing one argument, the tuple, to the function. You need to unpack the arguments. How do you do that:

```
a_function( *another_tuple_of_arguments_to_a_function  )
```

Here is the code with a little more context (for more see SmartTerminal smart_terminal_helper.HelperThread.polling) but not actual production code.

```
.........
( action, function, function_args ) = self.rec_from_queue()
if action == "call":
  function( *function_args )
  .........
```

more coming.....

Starting a Tkinter App

These applications start fine but often remain in the background with their window behind other windows. It would be good to bring it to the top of your windows. This code, may not be the best, seems to work at least on the Windows.

```
root.attributes("-topmost", True)
root.attributes("-topmost", False)  # if left to True window always on top
```

You need to get this to run once after the mainloop is started. You can do this with **root.after()** Some details left to you.

Simple Techniques

These are techniques I use that I have not often seen documented.

Create a Local Reference

We often use "dotted" notation to reference non-local variables. If the variable is used several times we may want to create a local reference:

```
graph_live_time_zero   = self.parameters.graph_live_time_zero
if    graph_live_time_zero   = 1:
    .....
elif  graph_live_time_zero   = 2:
    .....
else:
    ......
```

instead of:

```
if    self.parameters.graph_live_time_zero = 1:
    .....
elif  self.parameters.graph_live_time_zero = 2:
    .....
else:
    ......
```

Why:

- Well Python searches for local references first, they are faster so your code is faster.
- If something about the non-local reference changes, then changing the code is easier/less error prone.