

Embedding a Python interpreter into your program

Alan J. Salmoni

Do you want to write your own Python interpreter that you can embed into your program rather than use IDLE? What, exactly, is the point of reinventing the wheel? Well, first of all, it's not reinventing it: it's re-designing it. This is a worthy task unless we prefer to use stone wheels.

While extending my statistics package, SalStat, I needed to embed an interpreter within the program. Primarily, this was for debugging purposes: with an interpreter, I could check on the state of variables and classes, and try to work out what was going on. But I found it such a useful tool to use as the main interface to the program that I extended it to include all sorts of weird functionality that may be fun. For example, instead of clicking through various screens to perform a range of descriptive statistics, I could script the entire thing.

```
for i in DO.varlist:
    descriptives([i], "sum", "mean", "stdev")
```

This article describes how you can write your very own Python interpreter, and how it can be modified.

How does an interpreter work? Put simply, you create some kind of text interface (this could be a console, a simple multi-line edit control within a Tkinter window, or a wxPython frame with Scintilla – which is what ours is), and run the program. With a bit of magic, instead of clicking on buttons etc, you can examine variables in situ, modify things, run some code, and do pretty much what ever you would like as long as it's valid Python. The best thing is that you don't have to resort to exec or eval statements and there is persistence of variables. You can also import whatever variables you want. This was important because the interpreter needs to access those variables of interest to the user (whether the user is the programmer who wishes to debug the program, or the end-user who wishes to use the interpreter to issue commands).

I should point out that running a program from IDLE can probably achieve the same things; but it's fun to run your own, and sometimes IDLE is a bit heavy and this code is probably easier to customise and integrate. In addition, you can embed a very capable interpreter using iPython. The difference with the way described here is that when you make changes to the variables in the interpreter, they propagate back to the rest of the program.

The Code Module

An interpreter is based around the built-in 'code' module which isn't very large (it has two classes) but is extremely useful. This module "provides facilities to implement read-eval-print loops in Python".

The simplest way to implement an interpreter is using the InteractiveConsole class. This works much like Python when invoked from the command line. It is by far the simplest way to do things, but I don't use it because it stops all other operations until the interpreter is closed down. There are ways around this, but I consider the use of the other class, the InteractiveInterpreter class to be a better solution.

But if you want the InteractiveConsole, it's quite easy. Try this from IDLE or a script:

```
import code
terp = code.InteractiveConsole() # `terp' is short for interpreter
terp.interact()
```

Wow. How clever. Here, I have managed to make a Python interpreter from a Python interpreter. Of course, I could have gone through all that drawn-out convoluted nonsense of typing just "python" at the command prompt instead (or even done nothing), so maybe not so clever. But remember that this can be called from within a program. One problem is that execution of the rest of the program halts until I kill the console. Still, it's a single import statement and two other statements to make it work. One fun thing to do might be to add arguments to the interact statement such as:

```
terp.interact("A kind of Python interpreter")
```

which brings up a different banner to the normal one. If you have an application and you want to advertise your organisation, you can insert what you want here. To either the InteractiveConsole or interact statement, you can also add local variables which will be discussed later.

We can get around this drawback by using the InteractiveInterpreter class which requires a little more work but is more flexible. First, we import the code module and then we have to put the InteractiveInterpreter class into the window. In this case, it's a wx styled text control (STC, also known as Scintilla which is great fun and extremely capable). I won't detail the STC here other than going through what is essential to setting up an interpreter.

First of all, we need to import the relevant modules: code for the interpreter, wx and wx.stc for the GUI widgets, sys to take stdout and stderr, and __main__ to deal with what objects are within the scope of the interpreter.

```
import wx # needed for the GUI
import wx.stc as stc # needed for the styled text control (Scintilla
component)
import code # where the interpreter is
import sys # needed for admin stuff
import __main__ # needed to import the variables you want to interact
with
```

Next, get the InteractiveInterpreter up and running. I did this by deriving a new class (called II) from this class.

```
class II(code.InteractiveInterpreter):
    def __init__(self, locals):
        code.InteractiveInterpreter.__init__(self, locals)

    def Runit(self, cmd):
        code.InteractiveInterpreter.runsource(self, cmd)
```

This class adds a new method called Runit which is not entirely necessary as runsource can be accessed directly. However, it is useful sometimes to derive classes so as to have custom functionality. What this class does is set up an interpreter and it can receive commands. The command output will be sent to stdout or stderr and these can be redirected to an appropriate place using the sys module (this is described later). When instantiated, this class requires an argument, locals. These are objects that should be visible to the interpreter and it is through this that you can define what should lie within the interpreter's scope.

So far, so good. You can type the above code into IDLE and instantiate the class. This will allow you to run simple code like this:

```
>>> x = II(None)
>>> x.Runit('print "hi"')
hi
```

which shows that the interpreter seems to work. These last two pieces of code are the core of the interpreter. However, for something more useful, you need to have some way of getting data in and out of this interpreter that is independent of IDLE, or in other words have a GUI with a customised Python interpreter running. For this paper, we are going to use a wxPython frame to hold the interpreter so we define a wx frame with a Scintilla component embedded:

```
class ecpintframe(wx.Frame):
    def __init__(self, *args, **kws):
        kws["size"] = (700,600)
        wx.Frame.__init__(self, *args, **kws)
        self.ed = PySTC(self, -1)
```

This is the class definition of the wx frame that holds the STC. This is basic wxPython stuff. Then we have some code to instantiate the wxFrame which will cause it to appear once we have defined the PyTSC class:

```
if __name__ == '__main__':
    Ecpint = wx.PySimpleApp(0)
    win = ecpintframe(None, -1, "EcPint - Interactive intepreter")
    win.Show()
    Ecpint.MainLoop()
```

Then we define the Scintilla styled text control to do the hard work for us.

```
class PySTC(stc.StyledTextCtrl):
```

```

def __init__(self, parent, ID, pos=(10,10), size=(700, 600),
style=0):
    stc.StyledTextCtrl.__init__(self, parent, ID, pos, size,
style)
    sys.stdout = self
    sys.stderr = self
    self.Bind(wx.EVT_KEY_DOWN, self.OnKeyPressed)
    KEY_RETURN = 13

def SetInter(self, interpreter):
    self.inter = interpreter

def write(self, ln):
    self.AppendTextUTF8('%s'%str(ln))
    self.GotoLine(self.GetLineCount())

def OnKeyPressed(self, event):
    self.changed = True # records what's been typed in
    char = event.GetKeyCode() # get code of keypress
    if char == 13:
        lnno = self.GetCurrentLine()
        ln = self.GetLine(lnno)
        self.cmd = self.cmd + ln + '\r\n'
        self.NewLine()
        self.cmd = self.cmd.replace('\r\n','\n')
        self.inter.Runit(self.cmd)
        self.cmd = ''
        self.lastpos = self.GetCurrentPos()
    event.Skip() # ensure keypress is shown

```

This subclasses from the `StyleTextCtrl` class and changes a few attributes like making it take `stdout` and `stderr` (in other words, console output will be routed to this control). The `SetInter` method sets something (mysteriously called ‘interpreter’) as an attribute of the class. This is a direct link to the interpreter so that text typed into the STC can be sent straight there to be run. The `write` method outputs the text to the STC and is necessary if you want `stdout` and/or `stderr` output to be written there (they always look for a `write` method). The lines `sys.stdout = self` and `sys.stderr = self` redirect `stdout` and `stderr` respectively to the STC. This is a neat trick that means that all interpreter output is sent to the STC. You can of course decide to have errors printed elsewhere if that meets your needs better. The `OnKeyPressed` method catches keypresses and checks to see if they are the return key. If so, then the interpreter assumes that the user wishes to run the command; and the `self.inter.Runit(self.cmd)` does just that by sending the line just typed in to the interpreter. As we have already shown, `stdout` goes to the STC so you can use this already for input and output.

If you run this code, you will find that it works! If you type ‘print “hi”’ into the editor, it should print “hi” just underneath – we have a working interpreter!

However, there are problems with code blocks longer than one line. It is possible to get the interpreter to handle blocks at a time but this needs careful preparation of the data which means altering how the STC deals with typed-in data. Change the `OnKeyPressed` method above for this one below:

```

def OnKeyPressed(self, event):

```

```

self.changed = True # records what's been typed in
char = event.GetKeyCode() # get code of keypress
if (self.GetCurrentPos() < self.lastpos) and (char < 314)
or (char > 317):
    pass
    # need to check for arrow keys in this
elif char == 13:
    lnno = self.GetCurrentLine()
    ln = self.GetLine(lnno)
    self.cmd = self.cmd + ln + '\r\n'
    self.NewLine()
    self.tabs = ln.count('\t')
    if (ln.strip() == '') or ((self.tabs < 1) and (':'
not in ln)):
        # record command in command list
        self.cmd = self.cmd.replace('\r\n','\n')
        # run command now
        self.inter.Runit(self.cmd)
        self.cmd = ''
        self.lastpos = self.GetCurrentPos()
    else:
        if ':' in ln:
            self.tabs = self.tabs + 1
            self.AppendText('\t' * self.tabs)
            # change cursor position now
            p = self.GetLineIndentPosition(lnno + 1)
            self.GotoPos(p)
        else:
            event.Skip() # ensure keypress is shown

```

What this does is check whether the code would be expecting another line (say you typed in ‘for x in range(5):’), and if so, it auto-indents using an extra tab for you. Purists will prefer spaces and this code can easily be changed to that if you wish. However, the above code will also indent if a colon is on the just-typed-in line so if there is a colon within a print statement, or a dictionary is set, then the code will be indented. This is far from a disaster and solved with backspacing; and you are encouraged to come up with a better method of knowing when to indent the next line.

Adding Local Variables

We can also add our own local variables. These are particularly useful for an embedded interpreter because they allow the interpreter to access your program’s variables and objects. These aren’t automatically included any of your program variables, functions, classes or methods and need to be explicitly specified. Adding them is easy. The interpreter has its own space which it calls `__main__` (the same as the module that we imported at the start). This module contains references to a base set of objects. If you open the interpreter and import `__main__`, you can see what is available by typing `dir(__main__)`. You should see objects concerning the interpreter (‘Ecpint’, ‘I’, ‘II’, ‘PySTC’, ‘code’ and others) as well as some others like `__builtins__` or `__doc__`. The one that doesn’t show up is the one of interest and this is `__main__.__dict__` which is a dictionary that keeps the name and reference of objects. When you use an object, it looks in this dictionary to find out the object’s reference, and can then call it. All we have to do is provide a reference to the objects we want to make available and provide a suitable name for them.

Let's assume you have an object called a `dataObject` which is instantiated using the name `DO`. We want that object to be visible to the interpreter (i.e., within the interpreter's scope). This is done with a simple one-line instruction:

```
__main__.__dict__["DO"] = DO
```

Of course, when this is issued, the `DO` needs to be visible. It could be passed (along with any other objects you want to be made visible) as an argument to the `Inter` class. Using this, you can make all of a program's objects available to the interpreter.

Interrupting Exceptions

The `InteractiveInterpreter` class also has a very useful method called `'showtraceback'`. This is called when an exception is raised and normally shows the traceback that occurred along with the exception. It is possible to interrupt this and have some fun.

```
def showtraceback(self):
    type, value, tb = sys.exc_info()
    if type == exceptions.NameError:
        cd = tb.tb_frame.f_locals["code"]
        print cd
    else:
        # follow this to catch all other errors!
        code.InteractiveInterpreter.showtraceback(self)
```

What this does is catch the exception. If the exception is a `NameError`, it grabs the code that caused the exception and prints it out. If it is not a `NameError`, it continues with the traceback as normal.

But why do this? My application imported data from databases and the names of columns / fields were not always valid Python variable names (sometimes including white space, starting with numerics and so on). It would have been possible to change the names and tell our users that they have to be more sensible with naming. The other alternative (which might get purists a bit annoyed!) was to use the name strings as variables.

This was done by watching our for attribute errors, catching them, and redirecting them to the object that holds the variables (in this case, as a list in the `dataObject` we dealt with earlier):

When doing this, remember to use the `'code.InteractiveInterpreter.showtraceback(self)'` so that all other errors will be shown.

Other Things

It is possible to build a restricted Python interpreter. In this case, you would simply compare the `self.cmd` of the `STC` against the list of keywords that are allowed so that only the valid ones are put through to the interpreter. Of course, it is entirely possible that a clever user will find a way around this so it's not a secure solution. However, it should work for most users if the aim is to reduce complexity.

The Scintilla component is extremely rich and a lot of options can be configured (so many that there could be another article just for them). Things like tooltips, auto-completion, line numbering, and syntax highlighting are all available.

Internationalisation

If your application is internationalised, you have a little thing to watch out for. That little thing is the underscore character, which tells the gettext module to treat the string as one for interpretation. The problem is that the interpreter uses the underscore for something else and this will always overwrite the gettext version. This means that when you set up an interpreter within an internationalised application, you need to substitute the interpreters 'magic' underscore with something else. This function needs to import the sys module (which has already been done) and the `__builtin__` module.

```
import sys, __builtin__

def newhook(val):
    if val is not None:
        __builtin__.__last__ = val
        sys.stdout.write('%r\n'%val)
```

Then put this code somewhere before the interpreter is instantiated:

```
sys.displayhook = newhook
```

This code came courtesy of Peter Otten from the comp.lang.python newsgroup.

Embedding iPython

iPython is an extremely powerful extension and can also be embedded easily into a program. Although a program's variables and objects can all be examined, this is a one-way process and any changes are not propagated back to the program.

```
from IPython.Shell import IPShellEmbed

ipshell = IPShellEmbed()

ipshell() # this call anywhere in your program will start IPython
```

Conclusion.

You can see that more programming is required for the InteractiveInterpreter class than the InteractiveConsole class, but it offers a lot more power. Most of the code for the InteractiveInterpreter class concerns the user interface like doing things like indentation properly. But once you have it in place, you can easily build an interpreter that would fit into almost any application. The biggest danger is that because it's so useful, you may be tempted to put an interactive interpreter in every program that you do!

aReference:

Otten, Peter[___peter___@web.de] "gettext and the interpreter" In [comp.lang.python] 4 April 2004.

Appendix

This is all the code together

```
import wx
import wx.stc as stc
import code
import sys
import __main__

#x = code.InteractiveConsole()
#x.interact("A kind of Python interpreter")

class II(code.InteractiveInterpreter):
    def __init__(self, locals):
        code.InteractiveInterpreter.__init__(self, locals)

    def Runit(self, cmd):
        code.InteractiveInterpreter.runsource(self, cmd)

class PySTC(stc.StyledTextCtrl):
    def __init__(self, parent, ID, pos=(10,10), size=(700, 600),
style=0):
        stc.StyledTextCtrl.__init__(self, parent, ID, pos, size,
style)
        sys.stdout = self
        sys.stderr = self
        self.Bind(wx.EVT_KEY_DOWN, self.OnKeyPressed)
        self.cmd = ''
        self.lastpos = self.GetCurrentPos()

    def SetInter(self, interpreter):
        self.inter = interpreter

    def write(self, ln):
        self.AppendTextUTF8('%s'%str(ln))
        self.GotoLine(self.GetLineCount())

    def OnKeyPressed(self, event):
        self.changed = True # records what's been typed in
        char = event.GetKeyCode() # get code of keypress
        if (self.GetCurrentPos() < self.lastpos) and (char <314)
or (char > 317):
            pass
            # need to check for arrow keys in this
        elif char == 13:
            """
            What to do if <enter> is pressed? It depends if
there are enough
            instructions
            """
            lnno = self.GetCurrentLine()
```

The Python Papers Volume 3 Issue 2

```
ln = self.GetLine(lnno)
self.cmd = self.cmd + ln + '\r\n'
self.NewLine()
self.tabs = ln.count('\t') #9
if (ln.strip() == '') or ((self.tabs < 1) and (':'
not in ln)):
    # record command in command list
    self.cmd = self.cmd.replace('\r\n','\n')
    # run command now
    self.inter.Runit(self.cmd)
    self.cmd = ''
    self.lastpos = self.GetCurrentPos()
else:
    if ':' in ln:
        self.tabs = self.tabs + 1
        self.AppendText('\t' * self.tabs)
        # change cursor position now
        p = self.GetLineIndentPosition(lnno + 1)
        self.GotoPos(p)
    else:
        event.Skip() # ensure keypress is shown

class ecpintframe(wx.Frame):
    def __init__(self, *args, **kwds):
        kwds["size"] = (700,600)
        wx.Frame.__init__(self, *args, **kwds)
        self.ed = PySTC(self, -1)

if __name__ == '__main__':
    Ecpint = wx.PySimpleApp(0)
    I = II(None)
    win = ecpintframe(None, -1, "EcPint - Interactive intepreter")
    win.Show()
    win.ed.SetInter(I)
    Ecpint.MainLoop()
```