KEL: C++/Python Integration

Mark Lutz

Preface for the Python workshop distribution.

Introduction.

The following paper chronicles some experiences I've had using Python as an embedded language, in C++ applications. At KaPRE, we're using Python as a general extension tool, for on-site customization and configuration of C++ libraries, and as an alternative to *home-grown* parsers and interpreters. Other uses (rapid-prototyping) are also foreseen. By integrating Python, we can accommodate on-site variations in our applications, without shipping C++ source code. Python's simplicity makes it ideal for end-user coding.

"*KEL*" is just a 'glue' layer, used to integrate our C++ framework with embedded Python code. It currently consists of 2 components:

- An embedded-call API (simplified access to Python run-time tools)
- A C++ extension module and Python stub-class, for using passed-in C++ objects.

One notable feature of our C++ framework is the ability to access C++ class instance members and methods by name (called '*generic-access*'). This makes for a somewhat unique integration strategy: complex C++ objects are passed in and out of Python as generic pointers, wrapped in an instance of a single Python stub-class.

By overloading Python operators and qualification (__getattr__) in the stub class, we're able to catch C++ object manipulations in embedded Python code, and route them back to a C++ extension module. We don't need to generate code for each exposed C++ class (but that scheme has some advantages; for instance, KEL supports C++ data members, but *methods* must be registered).

Roughly, there's at least 4 ways to structure embedded Python programs:

- As character strings
- As file references (module/function names)
- As registered Python callable-objects
- As UNIX scripts

KEL supports the first two of these (strings and file references), since C++ is "*on-top*": no Python code gets executed until an embedded action is fired. Because of this, it's inconvenient to register arbitrary Python objects. Of course, executed strings can still run scripts, register and call objects, import site-specific modules, etc., as needed.

Some experiences so far.

Since this paper was written (March), we've started applying its ideas. To date, embedded character strings of Python code seem to be the dominant integration structure, and Python global variables are preferred for communicating data to/from Python code:

- Because we're a database environment, it's convenient to attach code to persistent database objects as strings, and allow them to be edited using our normal editing tools. This provides a high-level of variability, and makes issues of code loading/reloading trivial. File references haven't becomes as important (so far), partly due to administration complications when Python files must be managed.
- In general, inputs to embedded code are sent to Python by binding global (module) variables, and outputs are fetched from global variables set by the embedded string. A more direct scheme using function calls is supported, but hasn't been utilized.

In retrospect, our use of embedded Python code is much simpler than anticipated. However, the present situation is based on KEL applications aimed at *end-users*, and will change quickly, if we apply KEL to more complex domains, such as general rapid-development under our C++ framework.

Disclaimer.

This paper was originally written for in-house distribution at KaPRE. All the code examples are completely *hypothetical*, and do not represent the structure of KaPRE's financial products. For instance, the "post" system validation code names fields that don't even exist in our persistent object schema.

Further, some of this paper's material has changed, since it was written in March:

- The embedded-call API is now a C++ 'module' (all-static class).
- Tkinter is now part of the 'KEL' system; this allows embedded code to put up simple GUI objects, without exposing our C++ GUI framework.

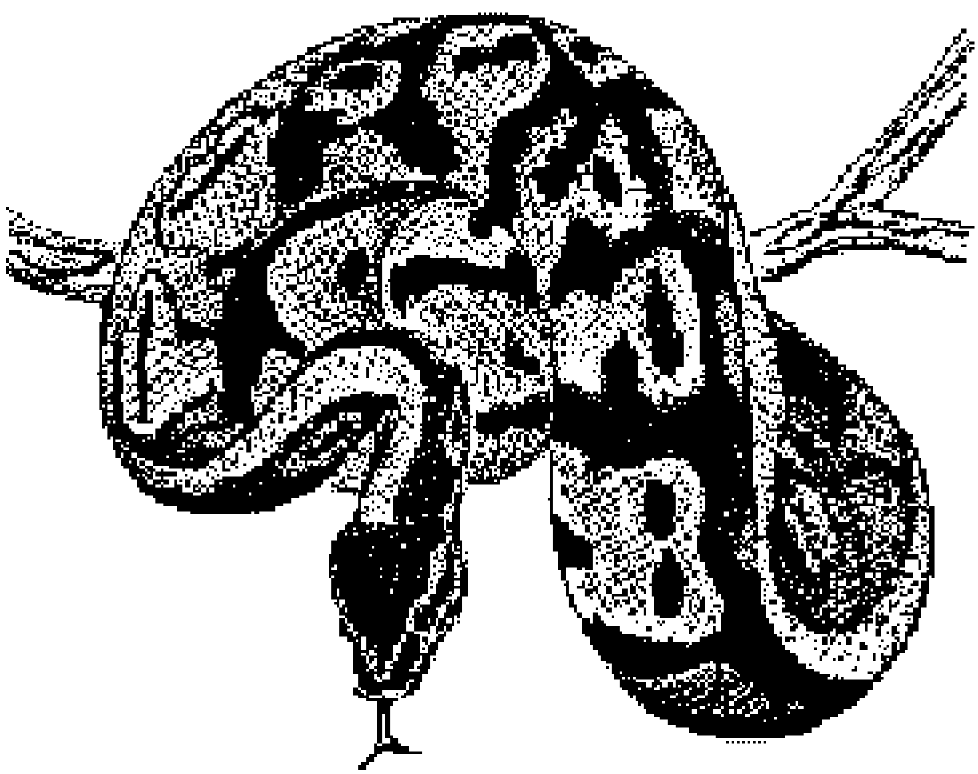
This was actually the second of three KEL papers. The third dealt with Tkinter. The first laid foundations for some of the ideas here; it included a code-generation proposal, which wasn't used. Unfortunately, it's 32 pages long, too big to include here. I've appended an example (again, hypothetical) from our account-generation-rule system, which is currently based on embedded Python code strings.

Other material (especially source-code for the wrapper class) would be helpful, but can't be released here. (Hey, we have to earn a living :-). Hopefully, nothing in this paper is too KaPRE-specific to be understandable.

On with the show...

KEL

And now for something completely different...



KEL: C++/Python Integration

Mark Lutz

Introduction.

This paper presents KEL's current C++/Python integration system. It supersedes much of the earlier "*KEL: An Overview*" paper's technical material. In particular, the prior paper's 'Appendix' presented an implementation scheme based on code-generation, which was not ultimately used. That paper's first half, dealing with KEL's *goals*, is still relevant.

This paper's main goal is to show how KEL can be used today, in KaPRE-based systems. Along the way, we'll also look at some of the implementation. We should note up-front that the current KEL implementation is intended to *evolve*, as we gain experience applying it; some of this paper may become arbitrarily out-of-date.

Functional summary.

As discussed in the 'overview' paper, KEL (the Kapre Extension Language) can serve in a variety of *roles* at KaPRE:

- as a mechanism for on-site customization (without shipping C++ source code)
- as a vehicle for implementing rapid-development and prototyping
- as a basis for unifying the language-based components in KaPRE products
- as a client/server integration facility (in conjunction with CORBA packages).

Ultimately, KEL's scope will depend on the extent to which it is employed at KaPRE. This is a matter of company policy, which we won't attempt to debate here; the 'overview' paper discusses KEL's potential benefits in some depth.

As also discussed earlier, *Python*, a very-high-level object-oriented language, is the basis for KEL. Integrating Python with C++ achieves KEL's goals by providing:

- a *dynamic* programming language, for situations in which a C++ compile/link is either impossible (on-site customization) or inconvenient (rapid-prototyping/development)
- a powerful but *simple* programming language, for situations in which C++ complexity can be a liability (prototyping, end-user coding)
- a generalized *language-tool*, for situations where we might otherwise need to build specialized parsers and interpreters.

The result is a highly flexible extension tool, which enhances our C++-based toolset.

When to use KEL.

Because of its flexibility, KEL's potential applicability is broad. Of course, there's no usage guidelines yet, but as a *rule-of-thumb*, if a component:

- may need to be altered on-site,
- is part of a rapid-prototyping effort,
- needs an easily-programmable interface,
- requires a language parser,
- or can't be conveniently implemented using our C++ tools,

then it's probably a good candidate for KEL. To date, we've identified a number of KEL application areas, including report-writer customization, an alternative implementation of 'N-amounts' equations, and various system-architecture components. A role as a rapid-development language in the planned 'application-editor' is also being explored.

In general, each KEL application area may have unique requirements, which will be addressed on a case-by-case basis. The current integration system provides *tools* to run Python code from C++; the *structure* of the integration may vary per KEL application.

When *not* to use KEL.

Given Python's ease-of-use, it might be tempting to use KEL in contexts where a C++ implementation would serve just as well, and perform *much* better. If a component does *not* match any of the above criteria, there's probably no reason to write it in KEL.

While some systems demand the dynamic-nature of a tool like Python, its poor run-time performance (shared by all dynamic languages) make it best suited for special-case, limited use. C++ and Python have distinct strengths and roles; a *hybrid* approach, where Python is used for 'front-end' components, leverages the benefits of both.

For instance, it's reasonable to *prototype* components in KEL, and later re-code them in C++. But KEL is not intended as a *delivery* medium for large, static components. Similarly, it's reasonable to expose C++ API's for use in Python. But if such an API doesn't present a *simplified* view of the corresponding C++ framework, its complexity may negate much of the advantage of using a simple OOP language such as Python.

Some quick examples.

Before we get mired in technical details, let's take a quick look at KEL in action. Here's a hypothetical, but typical scenario: suppose we have a validation in the '*post*' system, which we're unable to fully implement in C++, here at KaPRE.

It may be that we can't predict the sort of constraints that will exist at any particular user site. Or it may be that we're simply interested in making the application more powerful, by opening up strategic portions of its logic to the end-user. In any event, it's been decided that this particular validation is best not *hard-coded* in the C++ code of the libraries and executables we ship.

For such cases, we could elect to ship customers all the C++ source-code for our ‘post’ system, along with all the required C++ tools code, and *hope* that they can discover how and where to change a default validation. But such a policy is not only dangerous from a support perspective, it’s also hopelessly user-*un*friendly, given the relative complexity of our systems.

Now, suppose we’re fortunate enough to have a powerful, full-blown extension language such as Python at the ready, integrated with our C++ framework. Here’s how we might delegate the validation to user-customizable code:

File “postModule.C”

```
// C++ code in the kapre ‘post’ system

#include <KELembed.h>

int isValid, status;
AbstractAccount *acct;          // acct/comp are persistent objects
Company *comp;

status = KEL_run_function(                        // run a function
                  “postmodule”,                   // name-space/file
                  “accountValidation”,            // function-name
                  “i”, &isValid,                  // output/result
                  “(KiK)”, acct, type, comp);     // inputs/arguments

if (status < 0)
      error(KEL_LASTMSG);
if (isValid)
      // do passed activity
else
      // do failed activity
```

File “postmodule.py”

```
# customizable Python code

def accountValidation(account, type, company):
      if account.balance > 1000:                    # fetch member
            account.balance = 1000                  # set member
      if type < 2:
            return 0
      if account.info.balanceType not in company.balanceTypes:
            return 0
      for option in account.options:
            if option not in company.info[2]: return 0
      return 1
```


And that's (more or less) as complex as KEL extension use gets. Here's some notes on this example, before we get into a more formal description:

- ***The view from C++***

  In the C++ code of the 'post' system, we simply need to include the KEL header file, call the functions it defines, and link 'post' executables with Python's libraries. We don't need to know anything about Python internals or header-files, since data-conversion is handled for us automatically by '*printf*'-style format strings, and 'varargs' lists. We pass in 2 unconverted C++ objects and a C++ integer ('KiK'), and get back an 'int'; both primitive data-types, and unconverted C++ instances, can be passed from/to C++.

- ***The view from Python***

  In the Python source file, we might pre-code a *default* validation function, which the user can change on-site as appropriate. Alternatively, we could leave the validation function blank, as a *place-holder* for a user-defined extension. The Python source file gets shipped and installed with our products, along a standard install search path.

  In the Python function, C++ instances can be treated like normal Python data objects, and can be used in arithmetic, comparisons, etc. They may also be *qualified* arbitrarily deep: Python qualifications ("object.member") are translated into C++ dereference operations (member accesses) automatically, using generic-access. KEL also overloads Python subscripting and slicing to work on passed-in C++ collections, allows Python programs to create new C++ instances by class name, etc.

- ***Extensions as strings***

  In this example, we called a *function* coded in a module file. KEL also has interfaces for running Python expressions and statements, passed in as C character *strings*. Code strings can live in the database (changeable by users), be computed on demand, etc. And variables in expressions can be bound to values converted to Python form, just as easy as arguments in function calls. For instance:

```
stat = KEL_run_expression(
                "post",                          // name-space
                "Lower < Value < 999.99",        // an expression
                "i", &intres,                    // the result/output
                "Lower", "f", 100.99,            // set variables
                "Value", "f", acct.balance, NULL);
```

  KEL also allows us to bind global variables for use in expressions individually too:

```
stat = KEL_set_global("post", "Lower", "f", 100.99);
stat = KEL_set_global("post", "Value", "f", acct.balance);
stat = KEL_run_expression("post", "Value < 1000", "i", &intres, NULL);
```

• *User-defined KEL actions*

In the first example, we had a specific validation context in mind, and could assign it a name which was hard-coded in the C++ client. Since the module and function names are passed in as strings, the interface allows for more dynamic application. For instance, we could let the user schedule unforeseen KEL actions, by storing new function names in the database, creating lists of action names or expressions, etc. Here's an example:

```
for (i = 0; i < pobject.actions().count(); i++) {
      stat = KEL_run_function(
                  pobject.actions[i].modname(),        // user defined
                  pobject.actions[i].funcname(),
                  "s", &stringResult,
                  "(sKs)", "Nee", company, SPAM.as_string());

      if (!strcmp(stringResult, "stop"))
            break;
}
```

Of course, we could similarly evaluate lists of *strings*, stored in the database:.

```
stat = KEL_set_global(NULL, "UNITS", "i", acct.balance);
stat = KEL_set_global(NULL, "BALANCE", "f", 100.99);

for (i = 0; i < pobject.actions().count(); i++)
      stat = KEL_run_statements(NULL, pobject.actions[i], NULL);

stat = KEL_get_global(NULL, "UNITS", "i", &acct.balance);
```

Here, we bound some variables we wanted to give KEL access to, and used the default name-space (NULL), which is just the interactive command-line's name-space. We also ran the strings as *statements*, not expressions: there's no return value, but we fetched 'UNITS' as a result. The string list attached to 'pobject' might look something like this:

```
"if UNITS > 15: print 'adjusting units'; UNITS = 15"
"if BALANCE < 0 or BALANCE > 1000: print 'balance error'"
"import myModule; myModule.action(UNITS, BALANCE)"
"for x in range(1000):\n print 'shrubbery %i...' % x"
...etc.
```

• *Exposing other functionality to KEL code.*

In the original example, we passed in 2 C++ objects directly ('acct', 'comp'), and manipulated them in the Python function. This exploits the C++ functionality we get '*for free*' with KEL: the generic-access interface, and Python operator overloading. In short, KEL provides a C++ extension which calls-back to the C++ system, whenever C++ instances are manipulated in Python code.

We can expose additional C++ functionality to KEL code, by coding extensions in Python and/or C++, or installing packages available in the public-domain. Some additional KaPRE extensions may eventually become standard in KEL (general tools facilities, Tkinter, etc.); other application functionality can be exposed as needed. [It's also possible to register C++ class methods for direct use from Python; see ahead.]

For instance, in the example, we passed in a 'type' code, which the function compared to integer 2. In practice, 'type' might be an 'enum'; we can export its *symbolic* values by providing another Python module, which the KEL function can import and use:

File postdefs.py

```
FINANCIAL   = 1
BUDGET      = 2             # file installed on-site
SUMMARY     = 3

CompOpts    = 2

[Asset, Liability, Equity, Revenue, Expense] = range(5)   # 0..4
```

This module could also be written in C++, if we don't want to ship another Python file. It's also possible to expose entire API's for use in KEL code, as C++ (and/or Python) extension modules. For example, non-trivial KEL actions could make use of a general C++ tools API module. We won't go into the structure of C++ extension modules here (see Python's extensions manual), but they roughly take the form of:

File toolsmodule.C

- N static wrapper functions (convert to C++, call C++, convert to Python)

```
static object* error_func(object *self, object *args) { ... }
static object* warning_func(object *self, object *args) { ... }
```

- a static name/function registration table

```
{  {"error",     error_func},          // tools_table
   {"warning",   warning_func},
   {NULL, NULL} }
```

- an initialization function, called when the module is first imported in Python code.

```
extern "C"
void inittools() { initmodule("tools", tools_table);... }
```

Although modules export a *function*-based API, C++ *methods* can also be exported as functions, whose first argument is the subject C++ object; parallel Python stub classes can make the API appear object-oriented (see the prior paper's examples):

```
obj_method(void *obj, ...) { return ((class*)obj)->method(...); }
```


A C++ extension module can be added to Python just by compiling into a dynamically-loadable object file (*.so*), and dropping it in a directory on Python's module search path. It can also be statically bound-in to the executable, be built as combinations of Python and C++ logic, etc. Once built and installed, a C++ module acts exactly like one coded in Python. Here's our KEL validation function again, using these 2 extensions:

File postmodule.py

```
import tools                # our general tools C++ module
import postdefs             # our Python (or C++) constants module

def accountValidation(account, type, company):
    if account.balance > 1000:
        tools.warning("account '%s' balance" % account.nameString)
        account.balance = 1000
    if type not in [postdefs.FINANCIAL, postdefs.BUDGET]:
        tools.error('account type ' + type)
        return 0
    if account.info.balanceType not in company.balanceTypes:
        tools.error('account balance type')
        return 0
    for option in account.options:
        if option not in company.info[postdefs.CompOpts]:
            tools.error("account option:" + option)
            return 0
    return 1
```

As this example shows, KEL programs can become arbitrarily complex, depending on the application, and on how much C++ functionality is used. In the simplest case, KEL provides for generic-access to C++ members and methods; in pathological cases, we might expose large parts of the tools hierarchy, as generically-accessed methods, and/or additional C++ extension modules. These are application-dependent decisions.

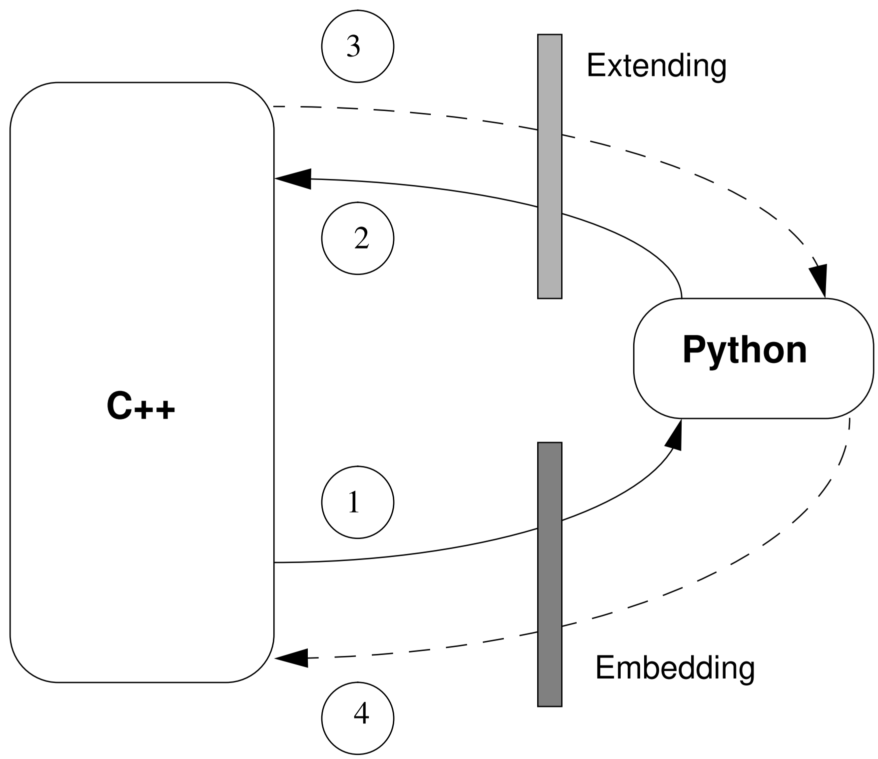
• *Other bells-and-whistles...*

KEL provides some other features that make life easier for both C++ and Python programmers, such as the ability to *debug* embedded Python code when it's called from C++, and the ability to automatically *reload* changed Python modules without stopping (or relinking!) the C++ client application. C++ can even spawn the Python *command-line* interpreter for interactive entry of code, etc.

For example, by setting a switch (or a '*spypoint*'), embedded KEL code calls are automatically routed to the '*pdb*' Python debugger. Execution stops in the application's '*stdio*' window, where programmers can step through Python code, set breakpoints, inspect variables, etc. The C++ caller continues with the return value, after the debugged embedded code exits.

The "big picture".

Let's move on to a more complete description of the KEL integration. From a high-level perspective, C++ and Python interact something like this:



In this picture, C++ and Python are depicted as separate systems; in reality, Python is bound-in to the C++ process/executable, so the distinction is only a logical one [more dynamic communication structures are possible, but not planned]. Some notation:

- solid lines represent argument passing
- dashed lines represent return values
- the lower portion represents calling embedded Python code from C++
- the upper part depicts what happens when Python code calls back to C++ to manipulate passed-in C++ instances, or calls other exposed C++ functionality.

KEL implements the 4 communication ports across language boundaries (the circles in the diagram). Control typically flows between languages like this:

(1) embedded Python call...	arguments converted from C++ to Python form
(2) C++ extension call...	arguments converted from Python to C++ form
(3) C++ extension return...	C++ return value converted to Python form
(4) embedded Python return...	Python return value converted to C++ form.

The conversion ports

Ports (1) and (4) take the form of the '*printf*'-style conversion specifications, in the embedded-call API's functions. The header file, 'KELembed.h' defines the functions used to invoke embedded Python operations.

Ports (2) and (3) are implemented in the C++ extension module KEL provides (and in any application-specific extensions). More precisely, KEL provides a Python class to represent ('wrap') passed-in C++ objects, and route manipulations to the C++ extension module. Objects are converted to and from C++ form, according to type signatures in registered C++ methods, or using a generic C++ type protocol (gType/gThing, LatentProtocol).

When *strings* are executed instead of *functions*, this picture changes slightly: then port (1) roughly depicts bindings to global variables, before or during an embedded call, rather than argument passing. Ports (2) through (4) behave the same as for functions.

Recursive control-flow

KEL uses both Python embedding and extending facilities, to implement a bidirectional control-flow:

- *C++ => Python*: embedded calls
- *Python => C++*: using C++ objects and extensions

Python execution occurs nested in at least 1 C++ execution level: '*main*' is in C++, and Python only runs *on-demand* from a C++ module. In general, there may any number of C++ and Python execution levels active at any time: it's possible that a call from Python to a C++ extension [port (2)] may trigger another recursive embedded-Python call [port (1)]... KEL (and Python) support arbitrarily deep nesting of embedded calls.

Recursive embedding also occurs inside KEL itself. For instance:

- debugged calls invoke the (python-coded) debugger's API internally
- passing-in C++ instances calls a wrapper class's constructor
- passing-out wrapped C++ instances calls a method in the wrapper class.

Basic architecture

The KEL system really just provides the '*glue*' which implements the embedding and extending ports on the diagram. Specifically, KEL consists of:

- a simplified embedded-call interface to Python
- an automatic mechanism for calling back to C++ generic access

plus a small number of patches to the Python language system. We'll look at both of these next. The embedded-call API is more critical to KEL clients, since it defines the view from C++. For KEL extension coders, the view from Python is essentially just Python's normal semantics; the extension layer is transparent.

The embedded-call interface.

To make it easier to run embedded-code, KEL implements a high-level API for using Python, on top of the existing Python run-time interface. Among the API's features:

- no python include files needed (Python's 'object*' is hidden)
- high-level interfaces for running functions by name
- high-level interfaces for running code strings
- simple access to module-level ('global') variables by name
- high-level interfaces for calling/accessing object members by name
- simple error message interface
- automatic data conversion to/from python form
- automatic dynamic debugging of embedded python code
- automatic dynamic reloading of python modules

The embedded-call interface is defined in source files:

- **KELembed.h**

  The external definition of the API (the only header file you need).

- **KELembed.c**

  The API implementation, on-top of low-level Python interfaces.

Some general usage notes:

Strings, Functions, and Objects.

The API supports running Python code using strings, functions, or methods of Python objects. For strings, it distinguishes between 'expressions' and 'statements' (no return value). Similarly, functions may be run as 'functions' or 'procedures' (no return value). Object method calls assume an object has been exported from Python directly (for instance, by calling a class constructor as a function). Variables may be accessed as 'globals' (which are at the top-level of some module) or 'members' of a Python object.

The main extension forms are strings and functions; from a functional perspective:

(KEL_set_global + KEL_run_statements + KEL_get_global) == KEL_run_function

Representing extensions a functions provides a more direct mapping for C++ inputs and results (via argument-lists and 'return' statements). However, functions may be less convenient in some cases, since they usually require an external Python code file ['def' statements can be run as strings outside a file, in a 'dummy' module, if needed].

When running statement strings, the '\n' character can be used to build-up compound statements (along with indentation if needed: N tabs '\t'). The ';' character can separate multiple simple statements, on a single line.

Modules and name-spaces.

Most of the API functions expect a module-name to be passed-in as their first argument. In Python, code objects always exist (and run) in a named name-space, called a 'module'. This is a convenient feature for KEL, since it allows us to partition functions, global-variables, etc., according to the calling application's needs (versus a single global scope).

In many cases, the module-name corresponds to a real Python source file, which gets loaded (imported) when any of its items are first accessed. This works well for functions, but doesn't really apply when running raw code strings: there may be no real python source file. For these cases, KEL clients can create a '*dummy*' module, which just becomes a name-space without a source-file. Use 'KEL_make_dummy_module()'.

In all cases, if the module-name is passed-in as 'NULL', it defaults to "__main__", which is the name of the interactive command-line's name-space (roughly). This allows for names that are truly 'global' to the entire process (not recommended). It also allows for interactive users to communicate directly with a KEL program: a KEL client can run the interactive *command-line*, and get/set user-visible variables in '__main__'.

Data conversions.

The API uses a '*printf*'-style conversion-code string to convert inputs/outputs to/from Python form automatically. For function arguments, and variable bindings, a list of 'varargs' is passed in, corresponding to the conversion code strings.

The 'printf'-style conversion system is really just a convenient wrapper around Python's standard conversion routines. Because of that, KEL supports all primitive-type conversions Python does [integers, floats, characters, strings, raw-objects; see python's extension manual, for an exhaustive list]. C++ callers can cast-down as needed to use these standard conversion codes.

For KEL, we've patched Python to support a 'K' conversion code, which allows C++ instances to be passed in unconverted. The 'K' code wraps/unwraps a C++ object pointer in a 'KapreThing' instance (see below). KEL will also eventually support other non-standard conversions; for instance, mappings between collections and Python lists, etc.

When using the embedded-call functions, function argument-list conversion strings always have surrounding parenthesis -- "(iKs)"; other conversion strings usually don't -- "i". Technically, we need to build a Python 'tuple' of values for argument lists, hence the parenthesis. Empty argument lists are just "()".

[Python modules can return *compound* values, as tuples (or lists, dictionaries, classes), and may expect compound values passed-in. For *inputs*, tuples can be passed using a possibly-nested parenthesized substring -- "(i(fs)K)". Alternatively, KEL will provide standard conversion functions, which can be invoked with the 'O&' convert code, using the varargs mechanism -- ..., "(O&)", convertFunc, source/target...). The 'K' convert code is just a convenience for 'O&', with 'PythonThing'/'CxxThing' converters.

Since *outputs* don't support a 'varargs' list of targets, compound outputs must be fetched with the 'O' convert code (raw Python object), and converted by calling 'KEL_unpack_object(<object>, <format-string>, targets...)'. Use of compound inputs and outputs in KEL will probably be rare, so we won't dwell on them here]

Dynamic debugging and reloading.

When *dynamic debugging* is enabled, control stops in the Python debugger (in the stdio window), when debugged embedded-code is called from C++ (functions, strings, or methods). Once stopped, all normal Python debugger interactions can be used (stepping, break, prints, etc.). Control resumes with the return value, when the debugged Python code returns/exits.

When *dynamic reloading* is enabled, the enclosing module file is reloaded each time one of the objects it contains is accessed (functions, global variables). This is particularly convenient for rapid-development/prototyping scenarios: KEL programmers can change extension module files while the calling C++ application remains active. Their changes will take effect immediately, without stopping or relinking the C++ executable.

Both debugging and reloading are available on a global basis (using bit-flags), or selectively (by calling functions to set/clear debugging and reloading for a particular module and function). For strings, dynamic debugging works, but dynamic reloading is only useful to reload global variable definitions (if there's a module file at all); since strings are passed-in each time they're run, there's no need to reload code.

Error-handling.

The embedded-API functions all return an integer status code; zero for success, -1 for failure (in keeping with Python conventions). On failure, the global '*KEL_LASTMSG*' holds a string describing the error. C++ programmers may also call Python's '*print_error()*' to print the Python stack at an error occurrence, '*err_get()*' to fetch a Python-specific string, etc. If the global '*KEL_VERBOSE*' is set, KEL will print KEL_LASTMSG and call print_error(), to display the error on 'stderr'.

C++ extension-module writers communicate errors back to Python, by returning NULL from the module function, and calling '*err_setstr()*', etc. to set Python's error message. See 'KELextend.C' for an example.

Compile/link requirements.

To build and link an executable that uses KEL/Python, use:

- the KELembed.h header file in the tools tree
- the .libs/.ldflags in the tools stuff/project/test/keltest directory

KEL initializes itself when first called, so there are no special 'main' considerations for C++. Python currently exists as 4 static libraries, and a separate command-line interpreter. Although Python can be built as a shareable library, its code-space requirements are negligible, and will only be incurred in executables that use KEL:

- A'C" executable using Python (keltest/testapp1.c) took 442K, when built without KaPRE libraries, and Python's command-line interpreter takes some 418K.
- A Python client executable with our C++ tools libraries took some 8.7M; adding basic GL libraries increased its size to 11.8M.

When we build a C++ executable which uses Python, we pull in our 'libKel' and Python's libraries, but replace a stubbed-out C++ extension module, with our generic-access module (KELextend.C; see below). It's important to use the '.libs' file's contents in the order in which it appears; we also need to pull in a tailored-version of a generated Python configuration file.

Running the Python command-line interpreter.

To use the python stand-alone interpreter, add the following to a '.login' or '.chsrc' file:

- alias python /development/python-1.1/python
- setenv PYTHONSTARTUP "~/.PythonInit"
- setenv PYTHONPATH ".:/development/python-1.1/Lib"

None of these are strictly needed; for instance, the default module search path ($PYTHONPATH) is as shown in the 'setenv' command, and the start-up file is optional.

Also note that the stand-alone interpreter is not currently built with any C++ extensions or libraries (since C++ objects aren't passed-in); when KEL runs the command-line, it actually uses a bound-in version, not the stand-alone interpreter (so C++ libraries are available). The stand-alone interpreter is useful for testing Python code outside KEL.

Shipping/installing Python code.

Python extensions can exist as strings, or functions in a module file. For strings, shipping isn't a concern: they live in C++ code, or in the database, etc. For module files, there's 3 ways to ship:

- **'.py' source-code files**

  This form is required for file-based extensions that are meant for on-site customization. Such files will be installed in a standard directory, which will be added to the default Python module search path.

- **'.pyc' pre-compiled files**

  This form is useful for Python-files which we don't want on-site users to view or change. These are roughly equivalent to '.o' object modules, but must also be installed in a directory on Python's module search-path (dynamically loaded).

- **frozen module files**

  Python has a feature which allows module-files to be 'frozen' as C object files, which can then be bound-in with the executable. In this case, there's no associated module file, and no search-path requirements.

Of these, the '.py' source-file medium will likely be most common. Unchangeable Python code will probably be rare (what good is an extension that users can't change?). Minimally, KEL clients will require 1 '.py' file, if they pass-in C++ objects to Python unconverted: the 'KEL.py' wrapper-class file we ship (see below).

The Embedded-call API functions

The current API definition is included at the end of this paper, in an *appendix* (despite my better judgement). It's extremely prone to change, expansion, etc., so take it with a reasonably large grain-of-salt. For a complete, and *up-to-date* definition of the embedded-call API, see file KELembed.h.

An implicit goal of the embedded-call API is to isolate as much Python interaction as possible, in a common place. C++ programmers can use low-level Python interfaces, but this is discouraged: if the API doesn't support an embedding need, the API should be changed, to make the addition generally available, make upgrading simpler, etc.

Of course, lower-level Python interfaces are necessary in some contexts. In particular, coders of additional *C++ extension modules* will need to make use of data-conversion routines minimally, and possibly much more. There's no good way to 'wrap' this functionality

The C++ extension interface.

To allow Python programs to use passed-in C++ class instances, KEL also provides the logic needed to route C++ object manipulation back to C++ logic, and allow C++ objects to be treated like normal Python objects, in KEL code.

The basic approach uses a Python wrapper class, and KaPRE++ generic-access, to expose C++ object internals to Python. There's 2 pieces to the *Python=>C++* interface:

- **KEL.py**

  A Python module, defining a stub class used to 'wrap' unconverted C++ instances, passed-in to (or created in) Python.

- **KELextend.C**

  A normal C/C++ extension module, with entry points for access to C++ services: generic-access, etc. The KELextend.C module is statically bound-in to the executable.

KEL.py uses Python *operator-overloading* to make C++ objects act like Python objects, and Python *meta-class-protocol* to intercept member accesses, and route them to the KELextend.C extension module. KELextend.C is only called from KEL.py logic: it's not meant to be used directly by Python or C++ programmers.

Since most of the KEL extension interface isn't intended for direct access, there's not much that needs to be documented on it here. See the source files for more information. Here, we'll take a look at a few random issues, and then take a better look at how KEL.py works, since this is the most unusual aspect of the scheme.

Data-member access

KEL (Python) code accesses data members of wrapped C++ objects using generic-access. This means that everything 'public', and visible in the generic-access sense, is accessible to Python. This includes most (all?) persistent objects.

To KEL coders, wrapped C++ objects act like generic Python objects. When one is qualified ("object.member"), C++ fetches the corresponding member. When one is indexed ("object.member[i]"), KEL verifies that the member is a collection, and calls C++ to do the indexing. When used in Python expressions, the C++ object is converted to Python form (usually).

Class method access

So far, we've only looked at C++ data members. KEL also has access to class methods, but only if they've been 'registered' to generic-access. This registration process involves setting up a static member equivalent, and registering a 'Fxn' variant to the class. We won't cover this process here, since C++ method-registration is considered beyond the scope of KEL, and is likely to change.

For now, given a method call in Python, “object.method(args)”, KEL will run a Fxn accessible by calling the ‘operation(“method”)’ method of the wrapped C++ object’s class. Installing the Fxn at the class is the C++ programmer’s responsibility, until a better policy emerges. Registered methods are sometimes an alternative to writing C++ extension modules for Python.

Creating C++ instances in KEL code

Python code can create new C++ instances by class-name, by calling the “KapreThing” class constructor, with the C++ class-name as its argument (“x= KapreThing(‘Account’)”). But only ‘default’ new instances can be constructed using the current generic-access interface: no C++ constructor arguments are used, and the Python creator must set members manually, after the instance is made (“x.balance=1000,...”). The Python ‘KapreThing’ wrapper owns the created C++ instance; it’s automatically deleted when the original KapreThing wrapper instance is no longer referenced.

Application-specific extensions.

As mentioned earlier, C++ programmers can expose additional functionality to Python, as registered generic-access methods, or C/C++ extension modules (using .so’s, or static linking). There’s also a growing library of Python extensions in the public domain, which can be used to make KEL code more powerful.

For instance, Python has an optional object-oriented interface to TCL’s Tk GUI library, ‘t*kpython*’. By binding this module (and Tk libraries) into Python executables, KEL programmers would have a means of rapidly building GUI interfaces on X, MS-Windows, and other platforms, apart from the KaPRE++ models hierarchy.

Similar extensions exist for database access (Sybase, Oracle,...), client/server protocols (ILU/CORBA), and more. When considering exposing portions of the KaPRE++ tools framework to KEL code, one should also consider functionally-similar, but simpler extensions which already exist.

KEL.py notes

The “KapreThing” wrapper-class defines a number of methods available to KEL coders. To fully understand its utility, we’d first need to document Python in general. Because this is beyond our scope here, the interested reader is referred to the “KEL.py” source file instead. Here’s some notes on how “KEL.py” is used by the KEL system.

- A Python class (“*KapreThing*”, in KEL.py) wraps an unconverted C++ instance. When we pass-in an unconverted C++ instances, we make a new Python “KapreThing”, passing in the C++ instance pointer to its constructor. The C++ pointer is saved as a member in the KapreThing, for later access to the wrapped C++ object. “KapreThing also knows how to create a *new* C++ instance.

- Python *meta-class-protocols* are used to intercept wrapped C++ object access, and route it to the KELextend.C C++ extension module. Python knows nothing about C++ object layout, member-names, etc. When a class instance is qualified ("object.member"), the special method "__getattr__(self, name)" is called, if the member does not exist: by redefining this, we intercept all KapreThing qualifications.

- Python *operator-overloading* is used to make KapreThing's look like normal python objects. They can be used in arithmetic expressions, comparisons, etc. For most purposes, KEL coders can treat them like other Python objects.In most cases, operators cast KapreThing's down to native Python objects.

- Python's 'for' loop iterator can also be applied to wrapped C++ instances: iteration just uses the overloaded index operator method ('__getitem__'), which creates a new KapreThing, for each loop iteration.

- Some operations on a KapreThing instance return another KapreThing (qualification, indexing, method calls...), while some convert down to a Python object (addition, slicing, etc.) thereby breaking the C++ connection.

- *Qualifications* ("object.member") are not evaluated immediately, but are recorded in a list on the KapreThing; a complete qualification list is evaluated only when the wrapped object is used in a Python context. The structure of generic-access currently requires us to have access to the last member name, for indexing and method calls.

- There is exactly 1 Python "*stub class*": KapreThing, in KEL.py. Because the KaPRE++ framework allows for generic member/method access by-name, this single class can represent all possible C++ classes. This is better than the code-generation scheme (in the prior paper), which requires 1 stub class per C++ class. However, it makes KEL heavily dependant on generic-access facilities, and the approach is not generally applicable to other frameworks (without generating code to simulate generic-access).

Local Python patches.

To support KEL, the Python language system was changed locally. All patches were minor, and there were relatively few (roughly a half-dozen). A file documenting what was changed is maintained at:

/development/python-1.1/KAPREMODS

More examples?...

See files '*testapp1.c*', and '*testapp2.C*' in the 'keltest' directory, for more examples of embedded-calls, and KEL extension code. These KEL test cases will be expanded periodically.

One interesting example of a complex Python application: KEL has been considered as an alternative implementation for the equations in the 'N-amounts' system. N-amounts 'dependency' links can be implemented by coding N-Amounts as simple Python functions. For instance, at an invoice header/detail, a list of function statements:

```
def A(): return B() + C()
def B(): return D() + 99
def C(): return B() + taxrate("Boulder")
D = 24
```

where 'taxrate' is coded in Python or C++ (in a dynamically-loaded .so). Function calls serve to recalculate dependant 'N-amounts'. Circular dependencies will loop, (but aren't trapped currently [?]).

By running these as statements, an N-amount can be recalculated, by calling the named function, after re-setting global variables. This system is still somewhat rough, but illustrates the sort of application-specific structuring some KEL clients may require.

Sources of additional information.

- **KEL use (source files):**

  /development/projects/stuff/project/src/kel/*
  the embedding/extension interfaces code, API header file,...

  /development/projects/stuff/project/test/keltest/*
  test programs, more examples, linkage specifications

  Also: see '*KEL: An Overview*' for background information.

- **Python use:**

  /development/python-1.1/Doc/POSTSCRIPT/*
  standard manuals/tutorial:
  tut.ps: a Python tutorial
  ref.ps: language reference
  lib.ps : library modules/services
  ext.ps: C embedding/extensions
  life-preserver.ps: tkpython GUI module manual

  Also: an 'alpha' version of a book tutorial section (see me).

- **Python code examples on-line:**

  /home/lutz/public/python/Apps/*
  shell-scripts, a class-based application framework,...

  /home/lutz/public/python/{Demo2, Gui}/*
  an expert system, command-line shell, set classes, Tkinter stuff...

  /development/python-1.1/{Lib, Demo}/*
  examples in the standard python distribution

  /development/projects/stuff/project/src/kel/KEL.py
  the wrapper class for passed-in kapre C++ objects

  /development/projects/stuff/project/test/keltest/*.py
  some simple embedded python code examples

Note: '/development' paths are prone to change (especially the python-1.1 path).

Caveats (room for growth).

- Generic-access is simpler, but not as general as a wrapper-code generation scheme:
    - KEL code can only create '*default*' new instances (and set members manually).
    - Indexing/method-calls need a member name: can't pass-in a 'Cltn' at the top-level.
    - Only works on 'generic-access' classes (under 'gType'/'gThing'; persistence ok).
    - Need to manually register class's *methods* for generic use (a better way?).
    - Without stub classes, *all* public members visible to generic-access are exposed: extra features needed to limit exposure.
    - Without stub-classes, there's no *documentation* on what members/methods are available to KEL code: need extra documentation.
    - Not as '*seamless*' as wrapper-code scheme (but more in synch with KaPRE++).
    - Not *symmetric*: Python uses C++ classes, but C++ can't use Python classes, as is.

- KELembed.c should be a C++ all-static class, not a C file (the only real 'C' module we need is the generated 'config.c', which doesn't compile under C++ in release 1.1).

- The next Python release (1.2) may make some of our local patches obsolete. It may also make some of the embedded-call API, and registration of new conversion codes (like our 'K') standard Python features. [Guido mentioned something about trying to reverse-engineer some of these features].

- A number of Python shortcomings were identified (again, may be fixed in 1.2):
    - Overloading '__cmp__' method only catches "KapreThing <cmp> KapreThing" cases. Mixed cases (1 operand not a "KapreThing" instance) don't use the '__cmp__' method. This will be fixed in 1.2. For now, KEL coders can cast-down manually in comparisons: "int(KT) > 1000", "999 < asPython(KT)", etc.
    - Python doesn't always automatically coerce class instances down to primitive types, even if '__coerce__', '__int__', etc., methods are overloaded. For instance, "KapreThing" instances used in range(), as indexes, etc., must be casted-down manually, as for comparisons.

- The stand-alone 'python' executable is built without any C++ libraries (cxx is a stub module). This may be an issue if/when a UI needs to *spawn* a command-line process, instead of calling it as a function.

- Some issues remain open: UI issues (spawning python shells, interactive boxes), using wrapped HashDict's like Python dictionaries,...

Appendix A: The embedded-call API (KELembed.h).

Warning: extremely prone to change...

```
//*****************************************************************************
 * execution-mode switches (bit patterns)
 *****************************************************************************/


#define KEL_DEBUG_FUNCTIONS     1
#define KEL_DEBUG_STATEMENTS    2
#define KEL_DEBUG_EXPRESSIONS   4
#define KEL_DEBUG_STRINGS       6
#define KEL_DEBUG_ALL           7

#define KEL_RELOAD_ALL          1

extern int KEL_RELOAD_MODE;
extern int KEL_DEBUG_MODE;

extern void KEL_set_debug(int modebits);
extern void KEL_clear_debug(int modebits);

extern void KEL_set_reload(int modebits);
extern void KEL_clear_reload(int modebits);



/*****************************************************************************
 * selective reload/debug
 *****************************************************************************/


extern void KEL_set_spypoint(char *modname, char *funcname);
extern void KEL_clear_spypoint(char *modname, char *funcname);

extern void KEL_set_reload_name(char *modname);
extern void KEL_clear_reload_name(char *modname);



/*****************************************************************************
 * message-handling
 *****************************************************************************/


extern int   KEL_VERBOSE;
extern char *KEL_LASTMSG;
```

```
/*****************************************************************************
 * converters for the 'O&' format code
 *****************************************************************************/


// pass C++ instance in/out directly (unconverted); same as 'K' code

extern void *
PythonThing(void *cxxObj);                    // C->Python

extern int
CxxThing(void *pythonObj, void **target);     // Python -> C


// convert C++ cltn <--> Python list

extern void *
PythonList(void *cxxObj);

extern int
CxxList(void *pythonObj, void **target);



/*****************************************************************************
 * create a dummy (non-file) module for get/set_global, strings
 *****************************************************************************/


extern int
KEL_make_dummy_module(char *modname);



/*****************************************************************************
 * run a python function (or class constructor, or method, or...)
 *****************************************************************************/


extern int
KEL_run_function( char *modname,    char *funcname,
                  char *resfmt,     void *cresult,
                  char *argfmt,     ... /* arg, arg... */ );


extern int
KEL_run_procedure( char *modname,   char *procname,
                   char *argfmt,    ... /* arg, arg... */ );
```

```
/*****************************************************************************
 * run a python expression string; bind (global) variables first;
 *****************************************************************************/

extern int
KEL_run_expression( char *modname,   char *expression,
                    char *resfmt,    void *cresult,
                    ...    /* "name", "fmt", val,...  NULL */ );

extern int
KEL_run_statements( char *modname,   char *statements,
                    ...    /* "name", "fmt", val,...  NULL */ );


/*****************************************************************************
 * get/set values of global (module) python variables
 *****************************************************************************/

extern int
KEL_get_global(char *modname,   char *varname,
               char *valfmt,    void *cval);

extern int
KEL_set_global(char *modname,   char *varname,
               char *valfmt,  ... /* cval(s) */);


/*****************************************************************************
 * run object's executable method, fetch/set attr (no module)
 *****************************************************************************/

extern int
KEL_run_method( void *pobject,   char *method,
                char *resfmt,    void *cresult,
                char *argfmt,    ... /* arg, arg... */ );

extern int
KEL_get_member( void *pobject,   char *member,
                char *valfmt,    void *cval);

extern int
KEL_set_member( void *pobject,   char *member,
                char *valfmt,    ... /* cval(s) */);
```

```
/********************************************************************************
 * misc. stuff
 ********************************************************************************/


extern void
KEL_save_object(void *pobj);        // 'O' format: inc ref count


extern void
KEL_drop_object(void *pobj);        // 'O' format: dec ref count manually


extern int
KEL_run_command_result(char *expression, char *resfmt,  void *cresult);


extern int
KEL_run_command_line(char *prompt);      // goto interactive loop


extern int
KEL_unpack_object(void *pyobj, char *cnvfmt, ... /* &cval(s) */);


extern void *
KEL_pack_object(char *cnvfmt, ... /* cval(s) */);


extern void *
KEL_fetch_None();                  // get 'None': python's void
```

Appendix B: KaPRE's account-key rule system ('flexkey')

Here's an example of one way KEL is currently being used in KaPRE's financial systems. Roughly, our general-ledger system allows users/sites to specify rules about account key validity, along with sets of legal values for a key's components. For instance,

> **"Given an account key 'type:name:center':**
> **If this is an 'asset' type account, and the name is 'Guido',**
> **Then the center must be in ['spam', 'eggs', 'hash']."**

These rules are associated with nodes in an account-hierarchy model, and applied whenever a new account is created (statically or dynamically). Originally, this system used a proprietary language parser/interpreter. By moving to embedded Python code, we were able to provide a much richer configuration language, achieve a radical speedup, and eliminate yet another parser module. For instance, rules can pop-up Tkinter dialogs.

This is fairly representative of our current KEL applications:

- Code is stored as multi-line C++ character strings on persistent (database) objects
- Global (module) variables are set by C++ as input to the embedded Python code
- Global variables are set in Python, and fetched by C++ as the code's result.

The enclosing C++ program calls: KEL_make_dummy_module(), KEL_set_global(), KEL_run_statements(), KEL_get_global(). This KEL application does not (yet) make use of the 'generic-access' interface for using passed-in C++ instances in Python. Embedded flexkey code just uses a passed-in string, and returns an integer result.

A support file, 'flexkey.py', is automatically pre-imported into the flexkey module/namespace by the C++ system [it runs a "from flexkey import *" string in the created module where embedded strings are run]. This file, and an example of flexkey code appear below.

```
######################################################################
# 1) Support file 'flexkey.py' (user-configurable):
######################################################################

def validateSegment( segment, validValues ):        # simple membership
    return segment in validValues:

def validateSegment2( segment, validValues ):       # allow ranges: tuples
    for test in validValues:
        if type(test) == type(""):
            if segment != test:
                return 0
        else:
            if not ( test[0] < segment < test[1] ):
                return 0
    return 1
```

```
################################################################
# 2) embedded code attached to a database object as a character string;
# - 'segment' is preset by C++ to pass-in the key string
# - 'segmentValue' is set here to send the result back to C++
################################################################

def trace(text):
    import fkstuff                          # a local module
    if fkstuff.debug == 'all': print text   # or send a message, gui,...


# legal values
fqaOrgs1 = [ "999" ]
fqaOrgs2 = [ "111", "213" ]
fqaOrgs3 = [ "122", "224" ]

fqaTbs1 = [ "1111", "1112", "1120", "1130", "1210", "1220", "2110",
    "2120", "2210", "3100", "3200", "3300" ]

fqaTbs2 = [ "4100", "4500", "5100", "5200", "5310", "5320", "5330",
    "5340", "6100", (6500, 7000), 7510]

fqaTbs3 = [ "4100", "4500", "5100", "5200", "5310", "5320", "5330" ]


org = segment[ :3 ]          # split the key string
tb   = segment[ 3: ]


# some inter-key constraints...
segmentValid = 1

if validateSegment( org, fqaOrgs1 ):
    if not validateSegment( tb, fqaTbs1 ):
        trace("Trial balance %s is not valid for organization %s" % (tb, org))
        segmentValid = 0

elif validateSegment( org, fqaOrgs2 ):
    if not validateSegment( tb, fqaTbs1 + fqaTbs3):
        trace("Trial balance %s is not valid for organization %s" % (tb, org))
        segmentValid = 0

elif validateSegment( org, fqaOrgs3 ):
    if not validateSegment2( tb, fqaTbs2 ):
        trace("Trial balance %s is not valid for organization %s" % (tb, org))
        segmentValid = 0

else:
    trace("%s is not a valid organization" % org)
    segmentValid = 0
```

