Extending Python

Presented by Alexandre Vassalotti

Email: alexandre@peadrop.com
Blog: http://peadrop.com/blog/

Presentation Outline

1. When extending Python is a good idea?
2. Overview the techniques available.
3. Introduction to the C API of Python.
4. Q&A Session.

Things to keep in mind

- When I say Python, I mean CPython. Not Jython, IronPython or PyPy.
- Extending is not the same thing as embedding.
- Examples will use the C API Python 2.x, not of Python 3.x (the differences are minor).

What is an extension?

- A piece of code loaded dynamically into the interpreter.
- Visible as a module by the import system.
- Contains functions or new object types.

What is an extension?

- A good extension is indistinguishable from a normal Python module.
- Python's standard libary is full of them.
- Think about `datetime`, `time`, `socket`, `signal`, `bz2`, `re`, `hashlib`, `ssl`, `sqlite3`, `heapq`, `math`, etc

What is an extension?

- All the built-in functions and types are created using the same API as extensions.
- So it is possible to make extension types that feels as well integrated as lists, tuples, dictionaries, and sets.

Major use cases

- To give Python programs access to a C or C++ library.
- Providing performance critical components of an application.
- Rapid prototyping and debugging.

Techniques for extending the Python interpreter

Python C API

- The standard way to create extension modules.
- It is a well designed and fully documented API.

C API: Major advantages

- Full control over Python object model.
- Best performance
- Compatible with C++

C API: Major disavantages

- Time-consuming to develop and debug.
- It is easy to get reference counting wrong.
- A binary must be created for every combination of supported operating system, machine architecture and Python version.

C API: Example

```c
#include <Python.h>
#include <sys/time.h>
static PyObject *
gettime(PyObject *self, PyObject *unused)
{
    struct timeval tv;
    if (gettimeofday(&tv, NULL) < 0) {
        PyErr_SetString(PyExc_OSError, "gettimeofday failed");
        return NULL;
    }
    return Py_BuildValue("ll", tv.tv_sec, tv.tv_usec);
}

static PyMethodDef methods[] = {
    {"gettime", gettime, METH_NOARGS, "Get the current time."},
    {0, 0, 0, 0}   /* sentinel */
};

PyMODINIT_FUNC
initexample(void)
{
    Py_InitModule("example", methods);
}
```

The ctypes module

- Easy to use Python module for calling C functions in shared libraries.
- Included in the standard library since Python 2.5.

ctypes: Advantages

- Do not need to compile anything.
- Easy to write and maintain.
- Good Windows support.

ctypes: Disadvantages

- Limited to shared library.
- No support for C++ name mangling.
- Significant overhead.
- Can handle opaque structures only through void pointers.

ctypes: Example

```python
import ctypes
from ctypes.util import find_library

class timeval(ctypes.Structure):
    _fields_ = (('tv_sec', ctypes.c_long),
                ('tv_usec', ctypes.c_long))

libc = ctypes.CDLL(find_library("c"))

def gettime():
    tv = timeval()
    if libc.gettimeofday(ctypes.byref(tv), None) < 0:
        raise OSError("gettimeofday failed")
    return (tv.tv_sec, tv.tv_usec)
```

ctypes: Example 2

```
import ctypes
import ctypes.wintypes
import time

class LastInputInfo(ctypes.Structure):
    _fields_ = [("cbSize", ctypes.wintypes.UINT),
                ("dwTime", ctypes.wintypes.DWORD)]

GetTickCount = ctypes.windll.kernel32.GetTickCount
GetLastInputInfo = ctypes.windll.user32.GetLastInputInfo

def get_idle_seconds():
    inputinfo = LastInputInfo()
    inputinfo.cbSize = ctypes.sizeof(inputinfo)
    if not GetLastInputInfo(ctypes.byref(inputinfo)):
        raise OSError("GetLastInputInfo failed")
    return (GetTickCount() - inputinfo.dwTime) / 1000
```

Cython

- "Python with C data types"
- It is a domain-specific language for generating Python extensions.

Cython: Advantages

- Can handle complex structures.
- Cython extensions are generally concise.
- The generated code is fairly good.
- Some support for C++.

Cython: Disavantages

- The hybrid C and Python syntax can be daunting to learn and remember.
- You still have to compile the extension.
- Not as mature as the alternatives (yet).

Cython: Example

```
cdef extern from "sys/time.h":
    ctypedef long time_t
    struct timeval:
        time_t tv_sec
        time_t tv_usec
    struct timezone:
        pass
    int gettimeofday(timeval *tv, timezone *tz)

def gettime():
    cdef timeval tv
    if gettimeofday(&tv, NULL) < 0:
        raise OSError("gettimeofday failed")
    return (tv.tv_sec, tv.tv_usec)
```

SWIG

- A C and C++ compiler that generates extensions for managed programming languages.
- It supports over 18 target languages, including Python, Ruby, Perl, Tcl, Lua, C#, Java and PHP.

SWIG: Advantages

- Suitable for large codebase where writing wrappers by hand would be cumbersome.
- Useful if you want to support more than one language.
- Easy to use. *

SWIG: Disadvantages

- Step learning curve.
- Require you to get unconfortably intimate with its internals for anything morderately complex.
- Encourage leaky extensions.
- Generated extensions are "unpythonic" unless heavily customized.
- Each targeted language must be customized seperately.

SWIG: Example

```
%module timeswig
%include cpointer.i
%pointer_functions(struct timeval, timeval);
%inline %{
    long timeval_get_sec(struct timeval *tv) {
        return tv->tv_sec;
    }
    long timeval_get_usec(struct timeval *tv) {
        return tv->tv_usec;
    }
%}

%{
#include <sys/time.h>
%}
extern int gettimeofday(struct timeval *tv, struct timezone *tz);
```

SWIG: Example

```python
import timeswig

def gettime():
    tv = timeswig.new_timeval()
    if timeswig.gettimeofday(tv, None) < 0:
        raise OSError("gettimeofday failed")
    result = (timeswig.timeval_get_sec(tv),
              timeswig.timeval_get_usec(tv))
    timeswig.delete_timeval(tv)
    return result
```

SWIG: Example

```
import timeswig

def gettime():
    tv = timeswig.new_timeval()
    if timeswig.gettimeofday(tv, None) < 0:
        raise OSError("gettimeofday failed")
    result = (timeswig.timeval_get_sec(tv),
              timeswig.timeval_get_usec(tv))
    timeswig.delete_timeval(tv)
    return result
```

Oops, there is leak...

SWIG: Example

```
import timeswig

class timeval:
    def __init__(self):
        self.value = timeswig.new_timeval()
    def __del__(self):
        timeswig.delete_timeval(self.value)

def gettime():
    tv = timeval()
    if timeswig.gettimeofday(tv.value, None) < 1:
        raise OSError("gettimeofday failed")
    result = (timeswig.timeval_get_sec(tv.value),
              timeswig.timeval_get_usec(tv.value))
    return result
```

Boost.Python

- C++ template library for writing Python extensions.

Boost.Python: Advantages

- Make Python's C API more friendly to C++ programmers.
- Concise code.

Boost.Python: Disadvantages

- Poor documentation.
- Complicated build system.
- Requires a runtime library.
- Encourage C++ semantics in Python code.

Boost.Python: Example

```
#include <boost/python.hpp>
#include <sys/time.h>
using namespace boost::python;

object gettime() {
    struct timeval tv;
    if (gettimeofday(&tv, NULL) < 0) {
        PyErr_SetString(PyExc_OSError,
                        "gettimeofday failed");
        throw_error_already_set();
    }
    return make_tuple(tv.tv_sec, tv.tv_usec);
}
BOOST_PYTHON_MODULE(btime_ext)
{
    def("gettime", gettime);
}
```

This was just the tip of the iceberg.

- Jython and IronPython allows extensions to be written using Java and C#.
- With the Pyd library, CPython can be extended using the D programming language.
- Even PyPy can be extended!

Introduction to the C API of Python

Building a Python extension

Step #1: Create a silly Python extension.

```
#include <Python.h>

PyMODINIT_FUNC
initspam(void)
{
    Py_InitModule("spam", NULL);
    printf("Nobody expect the Spanish Inquisition!\n");
}
```

Building a Python extension

Step #2: Write the setup.py build script.

```
from distutils.core import setup, Extension

setup(name="spam",
      ext_modules = [
          Extension("spam", ["spam.c"])])
```

Building a Python extension

Step #3: Build it.

```
$ python setup.py build_ext --in-place
running build_ext
building 'spam' extension
gcc -pthread -fno-strict-aliasing -DNDEBUG -g -fwrapv -O2
-Wall -Wstrict-prototypes -fPIC -I/usr/include/python2.6 -c
05spam.c -o build/temp.linux-x86_64-2.6/05spam.o
gcc -pthread -shared -Wl,-O1 -Wl,-Bsymbolic-functions build/
temp.linux-x86_64-2.6/05spam.o -o spam.so
```

Building a Python extension

Step #4: Run it!

```
$ python
Python 2.6.2 (release26-maint, Apr 19 2009, 01:58:18)
[GCC 4.3.3] on linux2
Type "help", "copyright", "credits" or "license" for
more information.
>>> import spam
Nobody expect the Spanish Inquisition!
```

Declaring Functions

```
static PyObject *
spam_system(PyObject *self, PyObject *obj)
{
    char *command;

    if (!PyString_Check(obj)) {
        PyErr_SetString(PyExc_TypeError,
                        "argument must be a string");
        return NULL;
    }
    command = PyString_AS_STRING(obj);
    return PyInt_FromLong(system(command));
}
```

Declaring Functions

```
static PyMethodDef methods[] = {
    {"system", spam_system, METH_O, "Execute a command."},
    {0, 0, 0, 0}
};
```

Possible values:
METH_NOARGS — No argument.
METH_O — One argument.
METH_VARARG — Many arguments.
METH_KEYWORDS

```
PyMODINIT_FUNC
initspam(void)
{
    Py_InitModule("spam", methods);
}
```

Parsing Arguments

- If you use METH_VARARGS or METH_KEYWORDS, you need to parse (or unpack) the arguments.

```
static PyObject *
deque_rotate(dequeobject *deque, PyObject *args)
{
    Py_ssize_t n=1;

    if (!PyArg_ParseTuple(args, "|n:rotate", &n))
        return NULL;
    if (_deque_rotate(deque, n) == 0)
        Py_RETURN_NONE;
    return NULL;
}
```

Parsing Keyword Arguments

```
static int
stringio_init(stringio *self, PyObject *args, PyObject *kwds)
{
    char *kwlist[] = {"initial_value", "newline", NULL};
    PyObject *value = NULL;
    char *newline = "\n";

    if (!PyArg_ParseTupleAndKeywords(args, kwds, "|Oz:__init__"
        kwlist, &value, &newline))
        return -1;
    ...
```

The Abstract Interface

- Almost 1:1 mapping between built-in operators and functions.
- It is (almost) like writing Python code!

The Abstract Interface

Python	C API Equivalent
`len(x)`	`PyObject_Length(obj)`
`x[k]`	`PyObject_GetItem(obj, key)`
`x[k] = v`	`PyObject_SetItem(obj, key, value)`
`x > y`	`PyObject_RichCompare(x, y, Py_GT)`
`x.attr`	`PyObject_GetAttrString(x, "attr");`

Built-in Types

Python Name	C Prefix
str	PyString
unicode	PyUnicode
list	PyList
tuple	PyTuple
dict	PyDict

Type-specific Interfaces

- Each built–in types has is own interface.
- These are analogous to the methods of the type.
- Usually available in 2 flavours:
  - Fast inline macros without runtime type checking
  - Safe functions with type checking.

PyString

- Not available in Python 3.x. Use the PyBytes interface, instead.
- Thin box around a bytes array (`char *`).

PyString

PyString_FromString(char *s)

Create a new string object from a NUL-terminated byte array.

PyString_FromSizeAndSize(char *s, int len)

Create a new string object from a length delimited byte array.

PyString

char *PyString_AsString(PyObject *s)

Return a pointer to the internal byte array of the given string object. Macro version: PyString_AS_STRING

int PyString_Size(PyObject *s)

Get the size of the string. Macro version: PyString_GET_SIZE

PyUnicode

- This is the new str type in Python 3.x.
- Stores characters internally as UTF–16 or UTF–32, depending of the build configuration choosen.
- Supports all encoding forms of Unicode: UTF–8, UTF–7, UTF–16 and UTF–32.
- Other encodings supported via the **codecs** module.

PyUnicode

PyUnicode_Decode(char *s, int len,
char *encoding, char *error)

Create a new unicode object by decoding the given byte array.

PyUnicode_AsEncodedString(
PyObject *u, char *encoding, char *error)

Return a PyString object containing the encoded form of the given PyUnicode object.

PyTuple

- Used throughout the interpreter for containing the arguments of functions.
- Mutable when brand-new. Immutable otherwise.
- Represented by a simple array of PyObject pointers.

PyTuple

PyTuple_New(int size)

Create a new empty tuple the given size.

PyTuple_Pack(int n, ...)

Create a new tuple with n items packed in it.

PyTyple

PyTuple_SetItem(PyObject *obj,
int index, PyObject* item)

Put an item in the tuple object without incrementing the reference count of item! Macro version: PyTuple_SET_ITEM

PyTuple_GetItem(PyObject *obj, int index)

Get an item from a tuple without incrementing the count of the item! Macro version: PyTuple_GET_ITEM

PyList

- Interface almost identical to the one of PyTuple.
- Always mutable.

PyList

- PyList_Append(
  - PyObject *obj, PyObject *item)
    - Append the given item with incremented reference count.

PyDict

- Used everywhere: stores instance attributes, class methods, local and global variables, keyword arguments, etc.
- Probably the most optimized data-structure in the interpreter.
- It is a hash table using open addressing with quadratic probing.
- There is no macro version for the interface.

PyDict

PyDict_New()

Create a new empty dictionary.

PyDict_GetItem(PyObject *obj, PyObject *k)

Get an item from the dictionary. Suppresses all errors that may occur.

PyDict_SetItem(

PyObject *obj, PyObject *k, PyObject *v)

Put an item in the dictionary.

PyDict

PyDict_Update(

PyObject *self, PyObject *other)

Merge the other dictionary into self.

PyDict_Next(*see below*)

Iterate over a dictionary. Use like so:

```
PyObject *key, *value;
Py_ssize_t i = 0;
while (PyDict_Next(dict, &i, &key, &value)) {
      Refer to borrowed references in key and value.
}
```

PyInt/PyLong

- I almost forgot!
- In Python 3.x, PyInt was removed in favour of PyLong.

PyLong

PyLong_FromLong(long v)

Create a new PyLong with the value v.

long PyLong_AsLong(PyObject *)

Pack a PyLong into a long integer if possible. Return -1 on error.

Reference Counting

- The most annoying part of the C API.
- It is easy to forget to decref temporary objects.
- Important concepts:
    - Ownership
    - Borrowing — Think about the PyDict API
    - Stealing — Think about the PyTuple and PyList API

Reference Counting

- Every Python functions and methods own their arguments.
- The return value of every Python functions and methods must have an incremented reference count.

Reference Counting

Py_INCREF(obj)

Increment the reference count of the object.

Py_DECEF(obj)

Decrement the reference count of the object. Free the object when the reference count becomes zero.

Reference Counting

Py_XINCREF(obj)

- Same as Py_INCREF, except it checks if obj is NULL.

Py_XDECREF(obj)

- Same as Py_DECREF, except it checks if obj is NULL.

Py_CLEAR(obj)

- Safe way to nullify an object.
- Py_XDECREF(obj); obj = NULL;

Handling Errors

- If the Python or the C API function that you called failed, just propagate the error.
- But don't forget to clean yourself up before!
- If you want to set a new exception, use the PyErr API.

PyErr

PyErr_SetString(PyObject *exc, char *msg)

Set an exception with the given message.

PyErr_Format(PyObject *exc, char *fmt, ...)

Set an exception with a formatted message.

```
if (!PyString_Check(items)) {
   PyErr_Format(PyExc_TypeError,
      "argument should be a str, not %.200s",
      Py_TYPE(items)->tp_name);
   return NULL;
}
```

Exception types

- They are all prefixed with PyExc_
- PyExc_TypeError, PyExc_ValueError, PyExc_RuntimeError, etc

Catching Exceptions

- How would you imitate the following code using the C API?

```
try:
    module_name = obj.__module__
except AttributeError:
    pass
```

Catching Exceptions

```
module_name = PyObject_GetAttrString(obj, "__module__");
if (module_name == NULL &&
    PyErr_ExceptionMatches(PyExc_AttributeError))
    PyErr_Clear();
else
    return NULL;
```

Using Python code from C

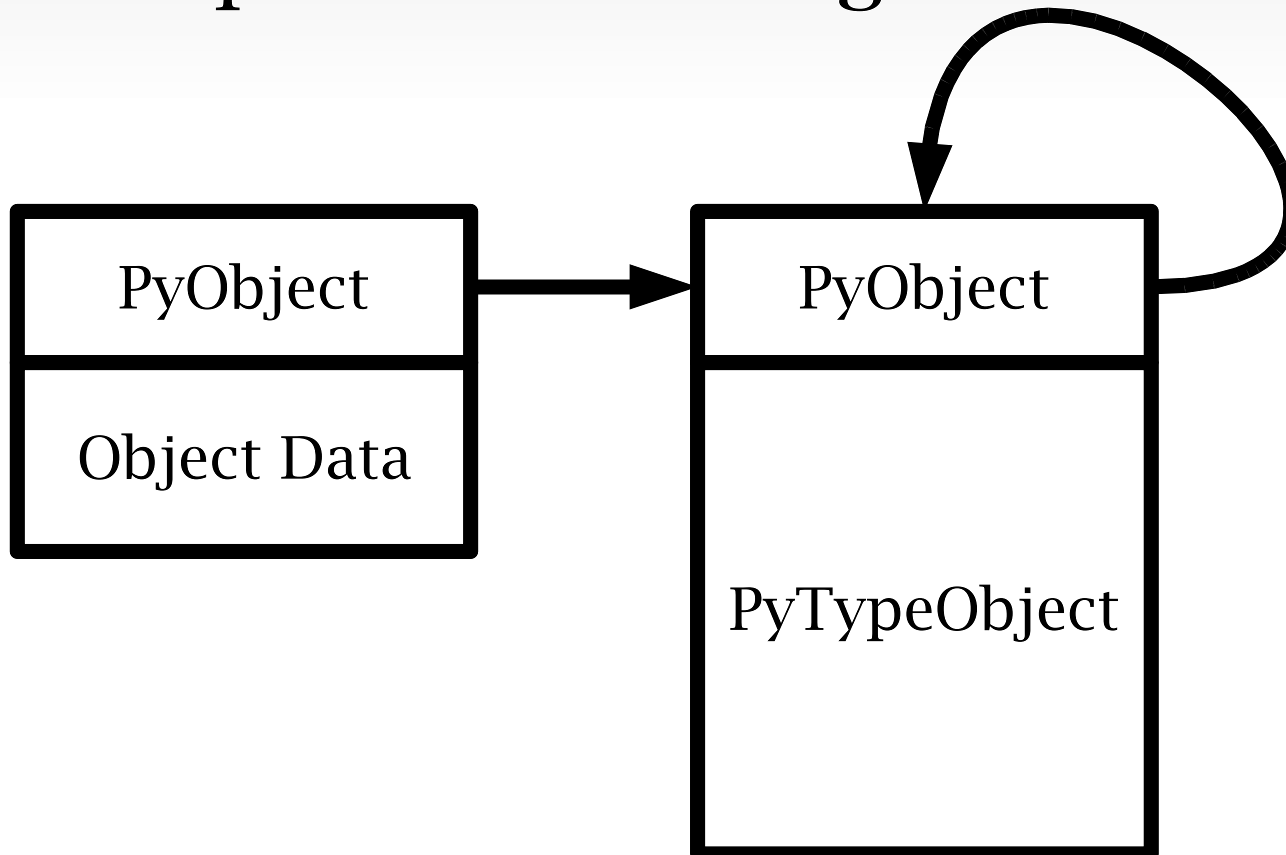
```
/* Extract the symbolic value for errno and use it if it
   is available. Ex: use 'ENOTDIR' instead of 20 */
if (self->myerrno) {
    errnomod = PyImport_ImportModule("errno");
    if (errnomod == NULL)
        goto error;

    errorcode_dict = PyObject_GetAttrString(
                        errnomod, "errorcode");
    if (errorcode_dict == NULL)
        goto error;

    errno_str = PyDict_GetItem(errorcode_dict,
                               self->myerrno);
    Py_DECREF(errorcode_dict);
    Py_DECREF(errnomod);
    if (errno_str == NULL) {
    ...
```

Extension Types

- Not covered today.
- Mainly involves copying a 50-lines type description and filling the blanks.



Putting it all together

- Everything is an object; every function is a method.
- The abstract interface provides an easy to use and type-agnostic API.
- Each built-in type has its own interface.
- If it can be done in Python, it can also be done in C.

Further Information

- C API Reference Manual:

  http://docs.python.org/c-api/

- Extending and Embeding Python:

  http://docs.python.org/extending/

Questions?