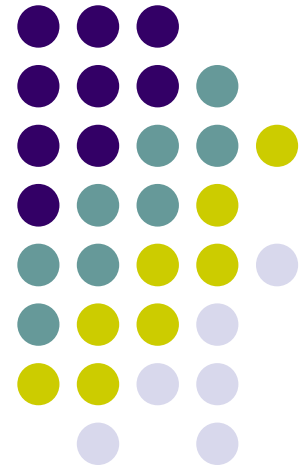


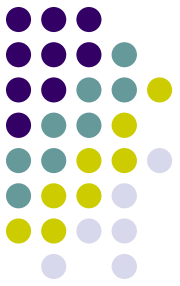
# Python ++ C

Юрий Бабуров

Омск, 2007

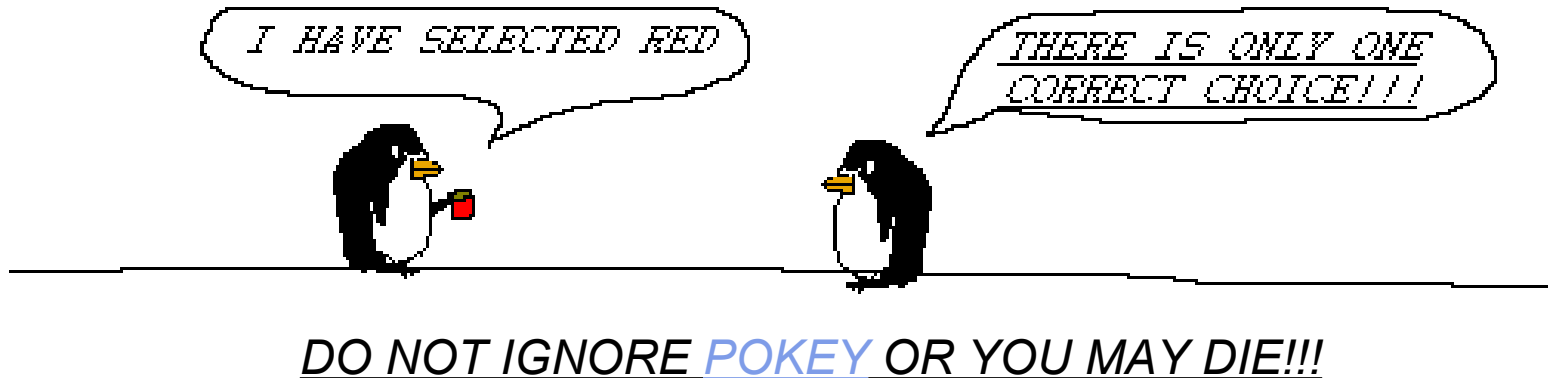
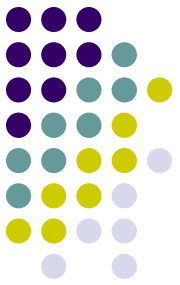


# Extending & embedding Расширение & встраивание



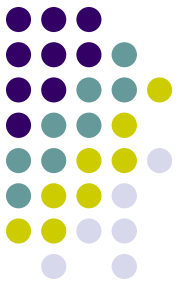
- Extending Python
- Вызов C из Python
  - SWIG
  - Boost.python
  - distutils
  - PyInline
  - Pyrex
  - Ctypes
  - Weave
  - Instant
- Embedding Python
- Вызов Python из C
  - вручную
  - Pyrex
  - Boost

# Extending || embedding Расширение || встраивание

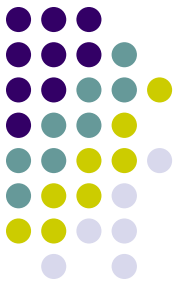


- Используйте встраивание только если:
  - Вам нужен ограниченный скриптовый язык в программах на C или C++
  - Вам нужно, чтобы ваша программа была показана в процессах как «тургога», а не как «python»
- Во всех остальных случаях расширять питон лучше
- Мантра питониста: **extend, don't embed**

# Расширяем!



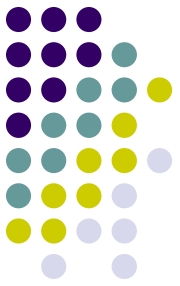
- С помощью чего?
  - SWIG
  - Boost.python
  - distutils
  - PyInline
  - Pyrex
  - ctypes
  - Weave
  - Instant
- Зачем?
  - Новая функциональность
  - Чужие API
  - Поддержка уже написанного кода (legacy code)
  - Скорость
  - Требуемая память



# Точки расширения в python

- Модули
  - Python libraries (.pyd)
  - ctypes
- Типы
- Классы
- Инлайны (inlines)
  - PyInline
  - Weave
  - Instant
- Много точек расширения:
  - SWIG
  - Pyrex
  - вручную

# PyInline



```
import PyInline, __main__
m = PyInline.build(code="""
    double my_add(double a, double b) {
        return a + b;
    }""",
    targetmodule=__main__, language="C")

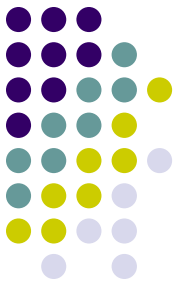
print my_add(4.5, 5.5) # Should print out
    "10.0"
```

# Instant

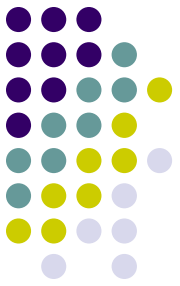
```
from instant import inline
add_func = inline("""
    double add(double a, double b){
        return a+b;
    }""")
print "The sum of 3 and 4.5 is ", add_func(3,
    4.5)
```

```
from instant import inline_with_numpy
c_code = """
double sum (int n1, double* array1){
    double tmp = 0.0;
    for (int i=0; i < n1; i++) {
        tmp += array1[i];
    }
    return tmp;
}
"""

sum_func = inline_with_numpy(
    c_code, arrays = [['n1', 'array1']])
a = numpy.arange(1000000)
a = numpy.sin(a)
sum1 = sum_func(a)
```



# Weave



```
import inline_tools
```

```
def c_sort(adict):
    code = """
        py::list keys = adict.keys();
        py::list items(keys.length());
        keys.sort();
        PyObject* item = NULL;
        int N = keys.length();
        for (int i = 0; i < N; i++)
        {
            item = PyList_GetItem(keys, i);
            item = PyDict_GetItem(adict, item);
            Py_XINCREF(item);
            PyList_SetItem(items, i, item);
        }
        return_val = items;
    """
    return inline_tools.inline( code, ['adict'])
```

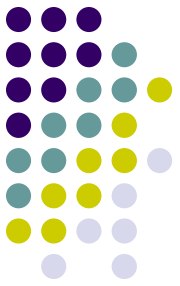
```
def sortedDictValues1(adict):
    items = adict.items()
    items.sort()
    return [value for key, value in items]
```

```
def c_sort2(adict):
    code = """
        py::list keys = adict.keys();
        py::list items(keys.len());
        keys.sort();
        int N = keys.length();
        for (int i = 0; i < N; i++)
        {
            items[i] = adict[int( keys[i] )];
        }
        return_val = items;
    """
    return inline_tools.inline(code, ['adict'], verbose=1)
```

```
def sortedDictValues2(adict):
    keys = adict.keys()
    keys.sort()
    return [adict[key] for key in keys]
```

```
def sortedDictValues3(adict):
    keys = adict.keys()
    keys.sort()
    return map(adict.get, keys)
```

# Как бы выглядел Python-код на C



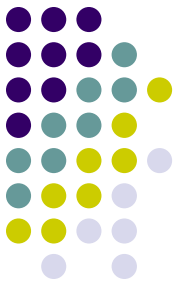
- Пусть на Python у нас есть следующий фрагмент:

```
modname = "test"
mod = __import__(modname)
rslt = mod.doit("this is a test")
```

- Тогда аналогичный код на C будет выглядеть примерно следующим образом:

```
#include "Python.h"
int main(int argc, char* argv[])
{
    long answer;
    PyObject *modname, *mod, *mdict, *func, *stringarg,
        *args, *rslt;
    Py_Initialize();
    modname = PyString_FromString("test");
    mod = PyImport_Import(modname);
```

```
    if (mod) {
        mdict = PyModule_GetDict(mod);
        func = PyDict_GetItemString(mdict, "doit");
        /* borrowed reference */
        if (func) {
            if (PyCallable_Check(func)) {
                stringarg = PyString_FromString("this is a test");
                args = PyTuple_New(1);
                PyTuple_SetItem(args, 0, stringarg);
                rslt = PyObject_CallObject(func, args);
                if (rslt) {
                    answer = PyInt_AsLong(rslt);
                    Py_XDECREF(rslt);
                }
                Py_XDECREF(stringarg);
                Py_XDECREF(args);
            }
            Py_XDECREF(mod);
        }
        Py_XDECREF(modname);
        Py_Finalize();
        return 0;
    }
}
```



# Pyrex

- Pyrex – язык с синтаксисом, похожим на python, который компилируется в код на C.

```
# simple pyrex sample
```

```
import string
```

```
def formatString(object s1, object s2):
```

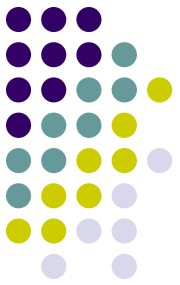
```
    s1 = string.strip(s1)
```

```
    s2 = string.strip(s2)
```

```
    s3 = '<<%s||%s>>' % (s1, s2)
```

```
    s4 = s3 * 4
```

```
return s4
```



# Еще один пример pyrex

```
cdef primes(int kmax):
```

```
    cdef int n, k, i
```

```
    cdef int p[1000]
```

```
    result = []
```

```
    if kmax > 1000:
```

```
        kmax = 1000
```

```
    k = 0
```

```
    n = 2
```

```
    while k < kmax:
```

```
        i = 0
```

```
        while i < k and n % p[i] <> 0:
```

```
            i = i + 1
```

```
        if i == k:
```

```
            p[k] = n
```

```
# line 14
```

```
            k = k + 1
```

```
# line 15
```

```
            result.append(n)
```

```
# line 16
```

```
            n = n + 1
```

```
    return result
```

```
def showPrimes(int kmax):
```

```
    plist = primes(kmax)
```

```
    for p in plist:
```

```
        print 'prime: %d' % p
```

- Фрагмент кода на C

```
...
```

```
...
```

```
/* "ProjectsA:Python:Pyrex:Demos:primes.pyx":14 */
```

```
(__pyx_v_p[__pyx_v_k]) = __pyx_v_n;
```

```
/* "ProjectsA:Python:Pyrex:Demos:primes.pyx":15 */
```

```
__pyx_v_k = (__pyx_v_k + 1);
```

```
/* "ProjectsA:Python:Pyrex:Demos:primes.pyx":16 */
```

```
__pyx_1 = PyObject_GetAttrString(__pyx_v_result, "append");
```

```
if (!__pyx_1) goto __pyx_L1;
```

```
__pyx_5 = PyInt_FromLong(__pyx_v_n);
```

```
if (!__pyx_5) goto __pyx_L1;
```

```
__pyx_6 = PyTuple_New(1);
```

```
if (!__pyx_6) goto __pyx_L1;
```

```
PyTuple_SET_ITEM(__pyx_6, 0, __pyx_5);
```

```
__pyx_5 = 0;
```

```
__pyx_5 = PyObject_CallObject(__pyx_1, __pyx_6);
```

```
if (!__pyx_5) goto __pyx_L1;
```

```
Py_DECREF(__pyx_6); __pyx_6 = 0;
```

```
Py_DECREF(__pyx_5); __pyx_5 = 0;
```

```
goto __pyx_L7;
```

```
}
```

```
__pyx_L7::
```

```
...
```

```
...
```

# Pyrex для Win32API



```
ctypedef unsigned long DWORD
ctypedef unsigned int UINT
ctypedef void *HANDLE
ctypedef HANDLE HWND
cdef extern from "memory.h":
    void *memcpy(void *, void *, DWORD)
cdef extern from "windows.h":
    int OpenClipboard(HWND hWndNewOwner)
    int EmptyClipboard()
    HANDLE GetClipboardData(UINT uFormat)
    HANDLE SetClipboardData(UINT uFormat, HANDLE hMem)
    int CloseClipboard()
    void *GlobalLock(HANDLE hMem)
    HANDLE GlobalAlloc(UINT uFlags, DWORD dwBytes)
    int GlobalUnlock(HANDLE hMem)
```

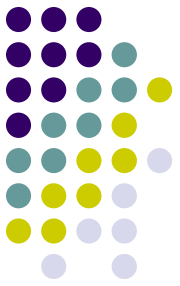
```
from win32con import CF_TEXT, GHND
```

```
def SetClipboardText(text):
    return _SetClipboardText(text, len(text))
```

```
cdef _SetClipboardText(char *text, int textlen):
    cdef HANDLE hGlobalMem
    cdef void *lpGlobalMem
    hGlobalMem = GlobalAlloc(GHND, textlen+1)
    lpGlobalMem = GlobalLock(hGlobalMem)
    memcpy(lpGlobalMem, text, textlen+1)
    GlobalUnlock(hGlobalMem)
    if OpenClipboard(<HWND>0):
        EmptyClipboard()
        SetClipboardData
        CloseClipboard()
```

```
def GetClipboardText():
    cdef HANDLE hClipMem
    cdef char *p
    text = ""
    if OpenClipboard(<HWND>0):
        hClipMem = GetClipboardData(CF_TEXT)
        p = <char *>GlobalLock(hClipMem)
        if p:
            text = p
        GlobalUnlock(hClipMem)
        CloseClipboard()
    return text
```

# ctypes для Win32API



```
from ctypes import *
from win32con import CF_TEXT, GHND
OpenClipboard = windll.user32.OpenClipboard
EmptyClipboard = windll.user32.EmptyClipboard
GetClipboardData = windll.user32.GetClipboardData
SetClipboardData = windll.user32.SetClipboardData
CloseClipboard = windll.user32.CloseClipboard
GlobalLock = windll.kernel32.GlobalLock
GlobalAlloc = windll.kernel32.GlobalAlloc
GlobalUnlock = windll.kernel32.GlobalUnlock
memcpy = cdll.msvcrt.memcpy

def GetClipboardText():
    text = ""
    if OpenClipboard(c_int(0)):
        hClipMem = GetClipboardData(c_int(CF_TEXT))
        GlobalLock.restype = c_char_p
        text = GlobalLock(c_int(hClipMem))
        GlobalUnlock(c_int(hClipMem))
        CloseClipboard()
    return text
```

```
def SetClipboardText(text):
    buffer = c_buffer(text)
    bufferSize = sizeof(buffer)
    hGlobalMem = GlobalAlloc(c_int(GHND),
                             c_int(bufferSize))
    GlobalLock.restype = c_void_p
    lpGlobalMem = GlobalLock(c_int(hGlobalMem))
    memcpy(lpGlobalMem, addressof(buffer),
           c_int(bufferSize))
    GlobalUnlock(c_int(hGlobalMem))
    if OpenClipboard(0):
        EmptyClipboard()
        SetClipboardData(
            c_int(CF_TEXT),
            c_int(hGlobalMem))
        CloseClipboard()

if __name__ == '__main__':
    # display last text clipped
    print GetClipboardText()
    # replace it
    SetClipboardText("[Clipboard text replaced]")
    print GetClipboardText()
    # display new clipboard
```

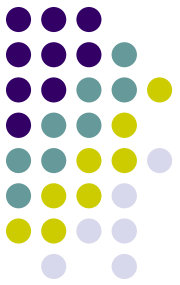
# Boost.python

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

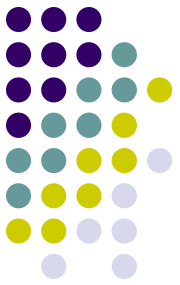
```
char const* greet()
{
    return "hello, world";
}
```

```
BOOST_PYTHON_MODULE(hello)
{
    def("greet", greet);
}
```

```
>>> import hello
>>> print hello.greet()
hello, world
```



# SWIG



```
%module example
```

```
{
```

```
/* Includes the header in the wrapper code */
```

```
#include "header.h"
```

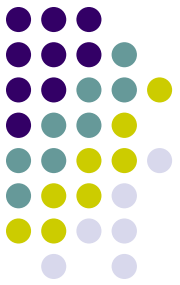
```
}
```

```
/* Parse the header file to generate wrappers */
```

```
%include "header.h "
```

- Как вы поняли, SWIG – средство для самых безнадежных лентяев.

# SWIG поддерживает даже шаблоны C++!

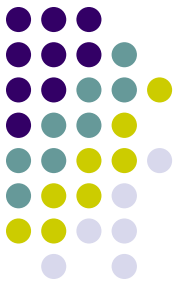


```
// pair.i - SWIG interface
%module pair
%{
#include "pair.h"
%}

// Ignore the default constructor
%ignore std::pair::pair();

// Parse the original header file
#include "pair.h"

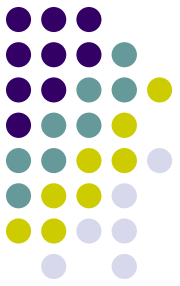
// Instantiate some templates
%template (pairii) std::pair<int,int>;
%template (pairdi) std::pair<double,int>;
```



# Distutils build script

```
# Build using the command:
#  setup.py build_ext --inplace
#
from distutils.core import setup
from distutils.extension import Extension
from Pyrex.Distutils import build_ext
setup(
    name = 'clipboard',
    ext_modules=[
        Extension("clipboard",
            ["clipboard.pyx"], libraries=["USER32"]),
    ],
    cmdclass = {'build_ext': build_ext}
)
```

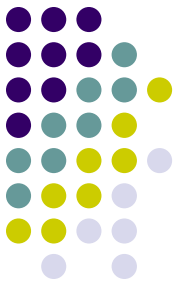
# Вызов Python из C



```
#include <Python.h>
```

```
int main(int argc, char *argv[])  
{  
    Py_Initialize();  
    PyRun_SimpleString(  
        "from time import time,ctime\n"  
        "print 'Today is',ctime(time())\n");  
    Py_Finalize();  
    return 0;  
}
```

# Pure embedding



```
#include <Python.h>
```

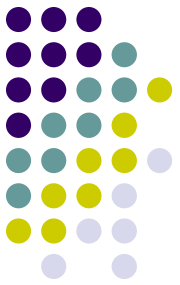
```
int main(int argc, char *argv[])
```

```
{
    int i;
    if (argc < 3) {
        fprintf(stderr, "Usage: call pythonfile funcname\n");
        return 1;
    }
    Py_Initialize();
    pName = PyString_FromString(argv[1]);
    /* Error checking of pName left out */

    pModule = PyImport_Import(pName);
    Py_DECREF(pName);

    if (pModule != NULL) {
        pFunc = PyObject_GetAttrString(pModule, argv[2]);
        if (pFunc && PyCallable_Check(pFunc)) {
            pArgs = PyTuple_New(0);
            pValue = PyObject_CallObject(pFunc, pArgs);
            Py_DECREF(pArgs);
            if (pValue != NULL) {
                printf("Result of call: %ld\n",
                    PyInt_AsLong(pValue));
                Py_DECREF(pValue);
            }
        }
    }
}
```

```
    else {
        Py_DECREF(pFunc);
        Py_DECREF(pModule);
        PyErr_Print();
        fprintf(stderr, "Call failed\n");
        return 1;
    }
}
else {
    if (PyErr_Occurred())
        PyErr_Print();
    fprintf(stderr, "Cannot find function \"%s\"\n", argv[2]);
}
Py_XDECREF(pFunc);
Py_DECREF(pModule);
}
else {
    PyErr_Print();
    fprintf(stderr, "Failed to load \"%s\"\n", argv[1]);
    return 1;
}
Py_Finalize();
return 0;
}
```



# Numeric python

- Что быстрее, (1), (2), или (3)?

(1) `a = [[0 for i in xrange(10)] for j in xrange(10)]`  
#10000 loops, best of 3: **35.6 usec** per loop

(2) `a = [[0]*10 for j in xrange(10)]`  
#100000 loops, best of 3: **13.2 usec** per loop

(3) `import Numeric as N`  
`a = N.zeros((10, 10), N.Int32)`  
#100000 loops, best of 3: **4.27 usec** per loop

(4) `a = [[0]*10]*10` #fast, but wrong variant  
#100000 loops, best of 3: **2.02 usec** per loop

(В варианте (4) создается 2 одномерных массива, а не 11!)

Вывод:

Если хочешь добиться высокой скорости – не пиши циклы сам, а используй по-максимуму функции, уже написанные на C.

Потому что **Nothing can beat a loop written in C** :)



**Спасибо за внимание**

**Вопросы сюда: [yuri@buriy.com](mailto:yuri@buriy.com)**