13. Лекция: Интеграция Python с другими языками программирования

C API

Доступные из языка Python модули расширяются за счет **модулей расширения (extension modules)**. Модули расширения можно писать на языке C или C++ и вызывать из программ на Python. В этой лекции речь пойдет о реализации Python, называемой CPython(Jython, реализация Python на платформе Java не будет рассматриваться).

Сама необходимость использования языка C может возникнуть, если реализуемый алгоритм, будучи запрограммирован на Python, работает медленно. Например, высокопроизводительные операции с массивами модуля Numeric (о котором говорилось в одной из предыдущих лекций) написаны на языке C. Модули расширения позволяют объединить эффективность порождаемого компилятором C/C++ кода с удобством и гибкостью интерпретатора Python. Необходимые сведения для создания модулей расширения для Python даны в исчерпывающем объеме в стандартной документации, а именно в документе "Python/C API Reference Manual" (справочное руководство по "Python/C API"). Здесь будут рассмотрены лишь основные принципы построения модуля расширения, без детальных подробностей об API. Стоит заметить, что возможности Python равно доступны и в C++, просто они выражены в C-декларациях, которые можно использовать в C++.

Все необходимые для модуля расширения определения находятся в заголовочном файле Python.h, который должен находится где-то на пути заголовочных файлов компилятора C/C++. Следует пользоваться теми же версиями библиотек, с которыми был откомпилирован Python. Желательно, и той же маркой компилятора C/C++.

Связь с интерпретатором Python из кода на C осуществляется путем вызова функций, определенных в интерпретаторе Python. Все функции начинаются на `Py` или `_Py`, потому во избежание конфликтов в модулях расширения не следует определять функций с подобными именами.

Через C API доступны все встроенные возможности языка Python (при необходимости, детальнее изучить этот вопрос можно по документации):

1. высокоуровневый интерфейс интерпретатора (функции и макросы `Py_Main()`, `PyRun_String()`, `PyRun_File()`, `Py_CompileString()`, `PyCompilerFlags()` и т.п.),
2. функции для работы со встроенным интерпретатором и потоками (`Py_Initialize()`, `Py_Finalize()`, `Py_NewInterpreter()`, `Py_EndInterpreter()`, `Py_SetProgramName()` и другие),
3. управление подсчетом ссылок (макросы `Py_INCREF()`, `Py_DECREF()`, `Py_XINCREF()`, `Py_XDECREF()`, `Py_CLEAR()`). Требуется при создании или удалении Python-объектов в C/C++-коде.
4. обработка исключений (`PyErr*` -функции и `PyExc_*` -константы, например, `PyErr_NoMemory()` и `PyExc_IOError`)
5. управление процессом и сервисы операционной системы (`Py_FatalError()`, `Py_Exit()`, `Py_AtExit()`, `PyOS_CheckStack()`, и другие функции/макросы `PyOS*`),
6. импорт модулей (`PyImport_Import()` и другие),

7. поддержка сериализации объектов (`PyMarshal_WriteObjectToFile()`, `PyMarshal_ReadObjectFromFile()` и т.п.)
8. поддержка анализа строки аргументов (`PyArg_ParseTuple()`, `PyArg_VaParse()`, `PyArg_ParseTupleAndKeywords()`, `PyArg_VaParseTupleAndKeywords()`, `PyArg_UnpackTuple()` и `Py_BuildValue()`). С помощью этих функций облегчается задача получения в коде на C параметров, заданных при вызове функции из Python. Функции `PyArg_Parse*` принимают в качестве аргумента строку формата полученных аргументов,
9. поддержка протоколов абстрактных объектов: + Протокол объекта (`PyObject_Print()`, `PyObject_HasAttrString()`, `PyObject_GetAttrString()`, `PyObject_HasAttr()`, `PyObject_GetAttr()`, `PyObject_RichCompare()`, ..., `PyObject_IsInstance()`, `PyCallable_Check()`, `PyObject_Call()`, `PyObject_Dir()` и другие). То, что должен уметь делать любой объект Python + Протокол числа (`PyNumber_Check()`, `PyNumber_Add()`, ..., `PyNumber_And()`, ..., `PyNumber_InPlaceAdd()`, ..., `PyNumber_Coerce()`, `PyNumber_Int()`, ...). То, что должен делать любой объект, представляющий число + Протокол последовательности (`PySequence_Check()`, `PySequence_Size()`, `PySequence_Concat()`, `PySequence_Repeat()`, `PySequence_InPlaceConcat()`, ..., `PySequence_GetItem()`, ..., `PySequence_GetSlice()`, `PySequence_Tuple()`, `PySequence_Count()`, ...) + Протокол отображения (например, словарь является отображением) (функции: `PyMapping_Check()`, `PyMapping_Length()`, `PyMapping_HasKey()`, `PyMapping_Keys()`, ..., `PyMapping_SetItemString()`, `PyMapping_GetItemString()` и др.) + Протокол итератора (`PyIter_Check()`, `PyIter_Next()`) + Протокол буфера (`PyObject_AsCharBuffer()`, `PyObject_AsReadBuffer()`, `PyObject_AsWriteBuffer()`, `PyObject_CheckReadBuffer()`)
10. поддержка встроенных типов данных. Аналогично описанному в предыдущем пункте, но уже для конкретных встроенных типов данных. Например: + Булевский объект (`PyBool_Check()` - проверка принадлежности типу `PyBool_Type`, `Py_False` - объект `False`, `Py_True` - объект `True`,
11. управление памятью (то есть кучей интерпретатора Python) (функции `PyMem_Malloc()`, `PyMem_Realloc()`, `PyMem_Free()`, `PyMem_New()`, `PyMem_Resize()`, `PyMem_Del()`). Разумеется, можно применять и средства выделения памяти C/C++, однако, в этом случае не будут использоваться преимущества управления памятью интерпретатора Python (сборка мусора и т.п.). Кроме того, освобождение памяти нужно производить тем же способом, что и ее выделение. Еще раз стоит напомнить, что повторное освобождение одной и той же области памяти (а равно использование области памяти после ее освобождения) чревато серьезными ошибками, которые компилятор C не имеет возможности распознать.
12. структуры для определения объектов встроенных типов (`PyObject`, `PyVarObject` и много других)

Примечание

Под **протоколом** здесь понимается набор методов, которые должен поддерживать тот или иной класс для организации операций со своими экземплярами. Эти методы доступны не только из Python (например, `len(a)` дает длину последовательности), но и из кода на C (`PySequence_Length()`).

Написание модуля расширения

Если необходимость встроить Python в программу возникает нечасто, то его расширение путем написания модулей на C/C++ - довольно распространенная практика. Изначально Python был нацелен на возможность расширения, поэтому в настоящий момент очень многие C/C++-библиотеки имеют привязки к Python.

Привязка к Python, хотя и может быть несколько автоматизирована, все же это процесс творческий. Дело в том, что если предполагается интенсивно использовать библиотеку в Python, ее привязку желательно сделать как можно более тщательно. Возможно, в ходе привязки будет сделана объектно-ориентированная надстройка или другие архитектурные изменения, которые позволят упростить использование библиотеки.

В качестве примера можно привести выдержку из исходного кода модуля `md5`, который реализует функцию для получения md5-дайджеста. Модуль приводится в целях иллюстрации (то есть, с сокращениями). Модуль вводит собственный тип данных, `MD5Type`, поэтому можно увидеть не только реализацию функций, но и способ описания встроенного типа. В рамках этого курса не изучить все тонкости программирования модулей расширения, главное понять дух этого занятия. На комментарии автора курса лекций указывает двойной слэш `//`:

```
// заголовочные файлы
            #include "Python.h"
            #include "md5.h"

            // В частности, в заголовочном файле md5.h есть следующие определения:
            // typedef unsigned char *POINTER;
            // typedef unsigned int UINT4;

            // typedef struct {
            //  UINT4 state[4];         /* state (ABCD) */
            //  UINT4 count[2];         /* number of bits, modulo 2^64 (lsb first) */
            //  unsigned char buffer[64];  /* input buffer */
            // } MD5_CTX;

            // Структура объекта MD5type
            typedef struct {
                    PyObject_HEAD
                    MD5_CTX     md5;            /* the context holder */
            } md5object;

            // Определение типа объекта MD5type
            static PyTypeObject MD5type;

            // Макрос проверки типа MD5type
            #define is_md5object(v)             ((v)->ob_type == &MD5type)

            // Порождение объекта типа MD5type
            static md5object *
            newmd5object(void)
            {
                    md5object *md5p;
                    md5p = PyObject_New(md5object, &MD5type);
                    if (md5p == NULL)
                            return NULL;        // не хватило памяти
                    MD5Init(&md5p->md5);        // инициализация
                    return md5p;
            }

            // Определения методов

            // Освобождение памяти из-под объекта
            static void
            md5_dealloc(md5object *md5p) { PyObject_Del(md5p); }

            static PyObject *
            md5_update(md5object *self, PyObject *args)
            {
                    unsigned char *cp;
                    int len;

                    // разбор строки аргументов. Формат указывает следующее:
                    // s# - один параметр, строка (заданная указателем и длиной)
```

```
        // : - разделитель
        // update - название метода
        if (!PyArg_ParseTuple(args, "s#:update", &cp, &len))
                return NULL;

        MD5Update(&self->md5, cp, len);

        // Даже возврат None требует увеличения счетчика ссылок
        Py_INCREF(Py_None);
        return Py_None;
}

// Строка документации метода update
PyDoc_STRVAR(update_doc,
"update (arg)\n\
\n\
Update the md5 object with the string arg. Repeated calls are\n\
equivalent to a single call with the concatenation of all the\n\
arguments.");

// Метод digest
static PyObject *
md5_digest(md5object *self)
{
        MD5_CTX mdContext;
        unsigned char aDigest[16];

        /* make a temporary copy, and perform the final */
        mdContext = self->md5;
        MD5Final(aDigest, &mdContext);

        // результат возвращается в виде строки
        return PyString_FromStringAndSize((char *)aDigest, 16);
}

// и строка документации
PyDoc_STRVAR(digest_doc,  "digest() -> string\n\ ...");


static PyObject *
md5_hexdigest(md5object *self)
{
    // Реализация метода на C
}

PyDoc_STRVAR(hexdigest_doc,  "hexdigest() -> string\n...");


// Здесь было определение метода copy()

// Методы объекта в сборе.
// Для каждого метода указывается название, имя метода на C
// (с приведением к типу PyCFunction), способ передачи аргументов:
// METH_VARARGS (переменное кол-во) или METH_NOARGS (нет аргументов)
// В конце массива - метка окончания списка аргументов.
static PyMethodDef md5_methods[] = {
{"update",    (PyCFunction)md5_update,    METH_VARARGS, update_doc},
{"digest",    (PyCFunction)md5_digest,    METH_NOARGS,  digest_doc},
{"hexdigest", (PyCFunction)md5_hexdigest, METH_NOARGS,  hexdigest_doc},
{"copy",      (PyCFunction)md5_copy,      METH_NOARGS,  copy_doc},
{NULL, NULL}                             /* sentinel */
};

// Атрибуты md5-объекта обслуживает эта функция, реализуя метод
// getattr.
static PyObject *
md5_getattr(md5object *self, char *name)
{
        // атрибут-данное digest_size
        if (strcmp(name, "digest_size") == 0) {
                return PyInt_FromLong(16);
```

```
        }
        // поиск атрибута-метода ведется в списке
        return Py_FindMethod(md5_methods, (PyObject *)self, name);
}

// Строка документации к модулю md5
PyDoc_STRVAR(module_doc, "This module implements ...");

// Строка документации к классу md5
PyDoc_STRVAR(md5type_doc, "An md5 represents the object...");

// Структура для объекта MD5type с описаниями для интерпретатора
static PyTypeObject MD5type = {
        PyObject_HEAD_INIT(NULL)
        0,                      /*ob_size*/
        "md5.md5",              /*tp_name*/
        sizeof(md5object),      /*tp_size*/
        0,                      /*tp_itemsize*/
        /* methods */
        (destructor)md5_dealloc,  /*tp_dealloc*/
        0,                      /*tp_print*/
        (getattrfunc)md5_getattr, /*tp_getattr*/
        0,                      /*tp_setattr*/
        0,                      /*tp_compare*/
        0,                      /*tp_repr*/
        0,                      /*tp_as_number*/
        0,                          /*tp_as_sequence*/
        0,                      /*tp_as_mapping*/
        0,                      /*tp_hash*/
        0,                      /*tp_call*/
        0,                      /*tp_str*/
        0,                      /*tp_getattro*/
        0,                      /*tp_setattro*/
        0,                      /*tp_as_buffer*/
        0,                      /*tp_xxx4*/
        md5type_doc,                    /*tp_doc*/
};


// Функции модуля md5:

// Функция new() для получения нового объекта типа md5type
static PyObject *
MD5_new(PyObject *self, PyObject *args)
{
        md5object *md5p;
        unsigned char *cp = NULL;
        int len = 0;

        // Разбор параметров. Здесь вертикальная черта
        // в строке формата означает окончание
        // списка обязательных параметров.
        // Остальное - как и выше: s# - строка, после : - имя
        if (!PyArg_ParseTuple(args, "|s#:new", &cp, &len))
                return NULL;

        if ((md5p = newmd5object()) == NULL)
                return NULL;

        // Если был задан параметр cp:
        if (cp)
                MD5Update(&md5p->md5, cp, len);

        return (PyObject *)md5p;
}

// Строка документации для new()
PyDoc_STRVAR(new_doc, "new([arg]) -> md5 object ...");

// Список функций, которые данный модуль экспортирует
static PyMethodDef md5_functions[] = {
```

```
                {"new",             (PyCFunction)MD5_new, METH_VARARGS, new_doc},
                {"md5",             (PyCFunction)MD5_new, METH_VARARGS, new_doc},
                {NULL,              NULL}    /* Sentinel */
        };
        // Следует заметить, что md5 - то же самое, что new. Эта функция оставлена
для
        // обратной совместимости со старым модулем md5

        // Инициализация модуля
        PyMODINIT_FUNC
        initmd5(void)
        {
                PyObject *m, *d;

                MD5type.ob_type = &PyType_Type;
                // Инициализируется модуль
                m = Py_InitModule3("md5", md5_functions, module_doc);
                // Получается словарь с именами модуля
                d = PyModule_GetDict(m);
                // Добавляется атрибут MD5Type (тип md5-объекта) к словарю
                PyDict_SetItemString(d, "MD5Type", (PyObject *)&MD5type);
                // Добавляется целая константа digest_size к модулю
                PyModule_AddIntConstant(m, "digest_size", 16);
        }
```

На основе этого примера можно строить собственные модули расширения, ознакомившись с документацией по C/API и документом "Extending and Embedding" ("Расширение и встраивание") из стандартной поставки Python. Перед тем, как приступать к созданию своего модуля, следует убедиться, что это целесообразно: подходящего модуля еще не создано и реализация в виде чистого Python неэффективна. Если создан действительно полезный модуль, его можно предложить для включения в поставку Python. Для этого нужно просто связаться с кем-нибудь из разработчиков по электронной почте или предложить модуль в виде "патча" через http://sourceforge.net.

Пример встраивания интерпретатора в программу на C

Интерпретатор Python может быть встроен в программу на C с использованием C API. Это лучше всего демонстрирует уже работающий пример:

```
/* File : demo.c */
        /* Пример встраивания интерпретатора Python в другую программу */
        #include "Python.h"

        main(int argc, char **argv)
        {
            /* Передает argv[0] интерпретатору Python */
            Py_SetProgramName(argv[0]);

            /* Инициализация интерпретатора */
            Py_Initialize();

            /* ... */

            /* Выполнение операторов Python (как бы модуль __main__) */
            PyRun_SimpleString("import time\n");
            PyRun_SimpleString("print time.localtime(time.time())\n");

            /* ... */

            /* Завершение работы интерпретатора */
            Py_Finalize();
        }
```

Компиляция этого примера с помощью компилятора gcc может быть выполнена, например, так:

```
ver="2.3"
        gcc -fpic demo.c -DHAVE_CONFIG_H -lm -lpython${ver} \
```

```
-lpthread -lutil -ldl \
-I/usr/local/include/python${ver} \
-L/usr/local/lib/python${ver}/config \
-Wl,-E \
-o demo
```

Здесь следует отметить следующие моменты:

- программу необходимо компилировать вместе с библиотекой libpython соответствующей версии (для этого используется опция `-l`, за которой следует имя библиотеки) и еще с библиотеками, которые требуются для Python: `libpthread`, `libm`, `libutil` и т.п.)
- опция `pic` порождает код, не зависящий от позиции, что позволяет в дальнейшем динамически компоновать код
- обычно требуется явно указать каталог, в котором лежит заголовочный файл Python.h (в gcc это делается опцией `-I`)
- чтобы получившийся исполняемый файл мог корректно предоставлять имена для динамически загружаемых модулей, требуется передать компоновщику опцию `-E`: это можно сделать из `gcc` с помощью опции `-Wl`, `-E`. (В противном случае, модуль `time`, а это модуль расширения в виде динамически загружаемого модуля, не будет работать из-за того, что не увидит имен, определенных в libpython)

Здесь же следует сделать еще одно замечание: программа, встраивающая Python, не должна много раз выполнять `Py_Initialize()` и `Py_Finalize()`, так как это может приводить к утечке памяти. Сам же интерпретатор Python очень стабилен и в большинстве случаев не дает утечек памяти.

Использование SWIG

SWIG (Simplified Wrapper and Interface Generator, упрощенный упаковщик и генератор интерфейсов) - это программное средства, сильно упрощающее (во многих случаях - автоматизирующее) использование библиотек, написанных на C и C++, а также на других языках программирования, в том числе (не в последнюю очередь!) на Python. Нужно отметить, что SWIG обеспечивает достаточно полную поддержку практически всех возможностей C++, включая предобработку, классы, указатели, наследование и даже шаблоны C++. Последнее очень важно, если необходимо создать интерфейс к библиотеке шаблонов.

Пользоваться SWIG достаточно просто, если уметь применять компилятор и компоновщик (что в любом случае требуется при программировании на C/C++).

Простой пример использования SWIG

Предположим, что есть программа на C, реализующая некоторую функцию (пусть это будет вычисление частоты появления различных символов в строке):

```
/* File : freq.c */
                #include <stdlib.h>

                int * frequency(char s[]) {
                  int *freq;
                  char *ptr;
                  freq = (int*)(calloc(256, sizeof(int)));
                  if (freq != NULL)
                    for (ptr = s; *ptr; ptr++)
                      freq[*ptr] += 1;
                  return freq;
                }
```

Для того чтобы можно было воспользоваться этой функцией из Python, нужно написать интерфейсный файл (расширение .i) примерно следующего содержания:

```
/* File : freq.i */
                %module freq

                %typemap(out) int * {
                  int i;
                  $result = PyTuple_New(256);
                  for(i=0; i<256; i++)
                    PyTuple_SetItem($result, i, PyLong_FromLong($1[i]));
                  free($1);
                }

                extern int * frequency(char s[]);
```

Интерфейсные файлы содержат инструкции самого SWIG и фрагменты C/C++-кода, возможно, с макровключениями (в примере выше: `$result`, `$1`). Следует заметить, что для преобразования массива целых чисел в кортеж элементов типа `long`, необходимо освободить память из-под исходного массива, в котором подсчитывались частоты.

Теперь (подразумевая, что используется компилятор gcc), создание модуля расширения может быть выполнено примерно так:

```
swig -python freq.i
               gcc -c -fpic freq_wrap.c freq.c  -DHAVE_CONFIG_H
                  -I/usr/local/include/python2.3 -I/usr/local/lib/python2.3/config
               gcc -shared freq.o freq_wrap.o -o _freq.so
```

После этого в рабочем каталоге появляется файлы `_freq.so` и `freq.py`, которые вместе и дают доступ к требуемой функции:

```
>>> import freq
                >>> freq.frequency("ABCDEF")[60:75]
                (0L, 0L, 0L, 0L, 0L, 1L, 1L, 1L, 1L, 1L, 1L, 0L, 0L, 0L, 0L)
```

Помимо этого, можно посмотреть на содержимое файла freq_wrap.c, который был порожден SWIG: в нем, среди прочих вспомогательных определений, нужных самому SWIG, можно увидеть что-то подобное проиллюстрированному выше примеру модуля `md5`. Вот фрагмент этого файла с определением обертки для функции `frequency()`:

```
extern int *frequency(char []);
               static PyObject *_wrap_frequency(PyObject *self, PyObject *args) {
                   PyObject *resultobj;
                   char *arg1 ;
                   int *result;

                   if(!PyArg_ParseTuple(args,(char *)"s:frequency",&arg1)) goto fail;
                   result = (int *)frequency(arg1);

                   {
                       int i;
                       resultobj = PyTuple_New(256);
                       for(i=0; i<256; i++)
                       PyTuple_SetItem(resultobj, i, PyLong_FromLong(result[i]));
                       free(result);
                   }
                   return resultobj;
                   fail:
                   return NULL;
               }
```

В качестве упражнения, предлагается сопоставить это определение с файлом freq.i и понять, что происходит внутри функции `_wrap_frequency()`. Подсказка: можно посмотреть еще раз комментарии к C-коду модуля `md5`.

Стоит еще раз напомнить, что в отличие от Python, в языке C/C++ управление памятью должно происходить в явном виде. Именно поэтому добавлена функция `free()` при преобразовании типа. Если этого не сделать, возникнут **утечки памяти**. Эти утечки можно обнаружить, при многократном выполнении функции:

```
>>> import freq
                >>> for i in xrange(1000000):
                ...   dummy =  freq.frequency("ABCDEF")
                >>>
```

Если функция `freq.frequency()` имеет утечки памяти, выполняемый процесс очень быстро займет всю имеющуюся память.

Интеграция Python и других систем программирования

Язык программирования Python является сценарным языком, а значит его основное назначение - интеграция в единую систему разнородных программных компонентов. Выше рассматривалась (низкоуровневая) интеграция с C/C++-приложениями. Нужно заметить, что в большинстве случаев достаточно интеграции с использованием протокола. Например, интегрируемые приложения могут общаться через XML-RPC, SOAP, CORBA, COM, .NET и т.п. В случаях, когда приложения имеют интерфейс командной строки, их можно вызывать из Python и управлять стандартным вводом-выводом, переменными окружения. Однако есть и более интересные варианты интеграции.

Современное состояние дел по излагаемому вопросу можно узнать по адресу: http://www.python.org/moin/IntegratingPythonWithOtherLanguages

Java

Документация по Jython (это реализация Python на Java-платформе) отмечает, что Jython обладает следующими неоспоримыми преимуществами над другими языками, использующими Java-байт-код:

- Jython-код динамически компилирует байт-коды Java, хотя возможна и статическая компиляция, что позволяет писать апплеты, сервлеты и т.п.;
- Поддерживает объектно-ориентированную модель Java, в том числе, возможность наследовать от абстрактных Java-классов;
- Jython является реализацией Python - языка с практичным синтаксисом, обладающего большой выразительностью, что позволяет сократить сроки разработки приложений в разы.

Правда, имеются и некоторые ограничения по сравнению с "обычным" Python. Например, Java не поддерживает множественного наследования, поэтому в некоторых версиях Jython нельзя наследовать классы от нескольких Java-классов (в то же время, множественное наследование поддерживается для Python-классов).

Следующий пример (файл lines.py) показывает полную интеграцию Java-классов с интерпретатором Python:

```
# Импортируются модули из Java
                from java.lang import System
                from java.awt import *
                # А это модуль из Jython
                import random

                # Класс для рисования линий на рисунке
                class Lines(Canvas):
                  # Реализация метода paint()
                  def paint(self, g):
                    X, Y = self.getSize().width, self.getSize().height
                    label.setText("%s x %s" % (X, Y))
                    for i in range(100):
                      x1, y1 = random.randint(1, X), random.randint(1, Y)
                      x2, y2 = random.randint(1, X), random.randint(1, Y)
                      g.drawLine(x1, y1, x2, y2)

                # Метки, кнопки и т.п.
                panel = Panel(layout=BorderLayout())
                label = Label("Size", Label.RIGHT)
                panel.add(label, "North")
                button = Button("QUIT", actionPerformed=lambda e: System.exit(0))
                panel.add(button, "South")
                lines = Lines()
                panel.add(lines, 'Center')

                # Запуск панели в окне
                import pawt
                pawt.test(panel, size=(240, 240))
```

Программы на Jython можно компилировать в Java и собирать в jar-архивы. Для создания jar-архива на основе модуля (или пакета) можно применить команду `jythonc`, которая входит в комплект Jython. Из командной строки это можно сделать примерно так:

```
jythonc -d -c -j lns.jar lines.py
```

Для запуска приложения достаточно запустить `lines` из командной строки:

```
java -classpath "$CLASSPATH" lines
```

В переменной `$CLASSPATH` должны быть пути к архивам `lns.jar` и `jython.jar`.

Prolog

Для тех, кто хочет использовать Prolog из Python, существует несколько возможностей:

- Версия GNU Prolog (сайт: http://gprolog.sourceforge.net) интегрируется с Python посредством пакета bedevere (сайт: http://bedevere.sourceforge.net)
- Имеется пакет PyLog (http://www.gocept.com/angebot/opensource/Pylog) для работы с SWI-Prolog (http://www.swi-prolog.org) из Python
- Можно использовать пакет `pylog` (доступен с сайта: http://christophe.delord.free.fr/en/pylog/), который добавляет основные возможности Prolog в Python

Эти три варианта реализуют различные способы интеграции возможностей Prolog в Python. Первый вариант использует SWIG, второй организует общение с Prolog-системой через конвейер, а третий является специализированной реализацией Prolog.

Следующий пример показывает использование модуля `pylog`:

```
from pylog import *

                exec(compile(r"""
                man('Socrates').
                man('Democritus').
                mortal(X) :- man(X).
                """))

                WHO = Var()
                queries = [mortal('Socrates'),
                           man(WHO),
                           mortal(WHO)]
                for query in queries:
                  print "?", query
                  for _ in query():
                    print "    yes:", query
```

Что выдает результат:

```
? mortal(Socrates)
                    yes: mortal(Socrates)
                ? man(_)
                    yes: man(Socrates)
                    yes: man(Democritus)
                ? mortal(_)
                    yes: mortal(Socrates)
                    yes: mortal(Democritus)
```

Разумеется, это не "настоящий" Prolog, но с помощью модуля `pylog` любой, кому требуются логические возможности Prolog в Python, может написать программу с использованием Prolog-синтаксиса.

OCaml

Язык программирования OCaml - это язык функционального программирования (семейства ML, что означает Meta Language), созданный в институте INRIA, Франция. Важной особенностью OCaml является то, что его компилятор порождает исполняемый код, по быстродействию сравнимый с C, родной для платформ, на которых OCaml реализован. В то же время, будучи функциональным по своей природе, он приближается к Python по степени выразительности. Именно поэтому для OCaml была создана **библиотека Pycaml**, фактически реализующая аналог C API для OCaml. Таким образом, в программах на OCaml могут использоваться модули языка Python, в них даже может быть встроен интерпретатор Python. Для Python имеется большое множество адаптированных C-библиотек, это дает возможность пользователям OCaml применять в разработке комбинированное преимущество Python и OCaml. Минусом является только необходимость знать функции Python/C API, имена которого использованы для связи OCaml и Python.

Следующий пример (из Pycaml) показывает программу для OCaml, которая определяет модуль для Python на OCaml и вызывает встроенный интерпретатор Python:

```
let foo_bar_print = pywrap_closure
                  (fun x -> pytuple_fromarray (pytuple_toarray x)) ;;
                let sd = pyimport_getmoduledict () ;;
                let mx = pymodule_new "CamlModule" ;;
                let cd = pydict_new () ;;
                let cx = pyclass_new (pynull (), cd, pystring_fromstring "CamlClass") ;;
                let cmx = pymethod_new (foo_bar_print,(pynull ()),cx) ;;
                let _ = pydict_setitemstring (cd, "CamlMethod", cmx) ;;
                let _ = pydict_setitemstring (pymodule_getdict mx, "CamlClass", cx) ;;
                let _ = pydict_setitemstring (sd, "CamlModule", mx) ;;
                let _ = pyrun_simplestring
                  ("from CamlModule import CamlClass\n" ^
                   "x = CamlClass()\n" ^
                   "for i in range(100000):\n" ^
                   "  x.CamlMethod(1,2,3,4)\n" ^
                   "print 'Done'\n")
```

Pyrex

Для написания модулей расширения можно использовать специальный язык - Pyrex - который совмещает синтаксис Python и типы данных C. Компилятор Pyrex написан на Python и превращает исходный файл (например, primes.pyx) в файл на C - готовый для компиляции модуль расширения. Язык Pyrex заботится об управлении памятью, удаляя после себя ставшие ненужными объекты. Пример файла из документации к Pyrex (для вычисления простых чисел):

```
def 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
                        k = k + 1
                        result.append(n)
                      n = n + 1
                    return result
```

В результате применения компилятора Pyrex, нехитрой компиляции и компоновки (с помощью GCC):

```
pyrexc primes.pyx
                gcc primes.c -c -fPIC -I /usr/local/include/python2.3
                gcc -shared primes.o -o primes.so
```

Получается модуль расширения с функцией `primes()`:

```
>>> import primes
                >>> primes.primes(25)
                [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61,
                67, 71, 73, 79, 83, 89, 97]
```

Разумеется, в Pyrex можно использовать C-библиотеки, именно поэтому он, как и SWIG, может служить для построения оберток C-библиотек для Python.

Следует отметить, что для простых операций Pyrex применяет C, а для обращения к объектам Python - вызовы Python/C API. Таким образом, объединяется выразительность Python и эффективность C. Конечно, некоторые вещи в Pyrex не доступны, например, генераторы, списковые включения и Unicode, однако, цель Pyrex - создание быстродействующих модулей расширения, и для этого он превосходно подходит. Ознакомится с Pyrex можно по документации (которая, к сожалению, есть пока только на английском языке).

Заключение

В этой лекции кратко рассматривались основные возможности интеграции интерпретатора Python и других систем программирования. Базовая реализация языка Python написана на C, поэтому Python имеет программный интерфейс Python/C API, который позволяет программам на C/C++ обращаться к интерпретатору Python, отдельным объектам, модулям и типам данных. Состав Python/C API достаточно обширен, поэтому речь шла лишь о некоторых основных его элементах.

Был рассмотрен процесс написания модуля расширения на C как напрямую, так и с использованием генератора интерфейсов SWIG. Также кратко говорилось о возможности встраивания интерпретатора Python в программу на C или OCaml.

Язык Python (с помощью специальной его реализации - Jython) прозрачно интегрируется с языком Java: в Python-программе, выполняемой под Jython в Java-апплете или Java-приложении, можно использовать практически любые Java-классы.

На примере языка Prolog были показаны различные подходы к добавлению возможностей логического вывода в Python-программы: независимая реализация Prolog-машины, связь с Prolog-интерпретатором через конвейер, связь через Python/C API.

Интересный гибрид C и Python представляет из себя язык Pyrex. Этот язык создан с целью упростить написание модулей расширения для Python на C, и использует структуры данных C и подобный Python синтаксис. Несмотря на некоторые смысловые и синтаксические отличия как от C, так и от Python, язык Pyrex помогает существенно сократить время разработки модулей расширения, сохранив эффективность компилятора C и знакомый синтаксис Python.

В данной лекции не были представлены другие возможности интеграции, например библиотека шаблонов C++ Boost Python, которая позволяет интегрировать Python и C++. Кроме того, из Python можно использовать библиотеки, написанные на Фортране (проект F2PY).

Развитые и гибкие интеграционные возможности Python являются его основным преимуществом в качестве языка для интеграции приложений. Из лекции нетрудно заключить, что Python легко взаимодействует с другими системами.