Расширение и встраивание Python

Выпуск 3.8.8

Гвидо ван Россум
и команда разработчиков Python

августа 12, 2021

https://Digitology.tech
Email: tweakit@bk.ru

Оглавление

В этом документе описывается, как писать модули на C или C++, чтобы расширить Python интерпретатор новыми модулями. Эти модули могут определять не только новые функции, но и новые типы объектов и их методы. В документе также описывается встраивание Python интерпретатора в другое приложение для использования в качестве языка расширения. Наконец, в нем показано, как компилировать и связывать модули расширения так, чтобы их можно было динамически загружать (во время выполнения) в интерпретатора, если базовая операционная система поддерживает эту функцию.

В этом документе предполагается наличие базовых знаний о Python. Неформальное введение в язык см. в разделе tutorial-index. reference-index дает более формальное определение языка. library-index документирует существующие типы объектов, функции и модули (встроенные и написанные на Python языке), которые предоставляют языку широкий диапазон приложений.

Подробное описание всего API Python/C см. в отдельном c-api-index.

ГЛАВА 1

Рекомендуемые сторонние инструменты

В данном руководстве рассматриваются только основные средства создания расширений, предоставляемые в рамках данной версии CPython. Сторонние инструменты, такие как Cython, cffi, SWIG и Numba, предлагают более простые и сложные подходы к созданию расширений C и C++ для Python.

См.также:

Руководство пользователя пакетизации Python: двоичные расширения Руководство пользователя по упаковке Python не только охватывает несколько доступных инструментов, которые упрощают создание двоичных расширений, но также обсуждает различные причины, по которым создание модуля расширения может быть желательно в первую очередь.

ГЛАВА 2

Создание расширений без сторонних инструментов

В этом разделе руководства рассматривается создание расширений C и C++ без помощи сторонних инструментов. Он предназначен в первую очередь для создателей этих инструментов, а не является рекомендуемым способом создания собственных расширений C.

2.1 Расширение Python с помощью C или C++

Очень легко добавить в Python новые встроенные модули, если вы умеете программировать на C. Такие *модули расширения* могут делать две вещи, которые невозможно сделать непосредственно в Python: они могут реализовывать новые встроенные типы объектов, и они могут вызывать функции библиотеки C и системные вызовы.

Для поддержки расширений Python API (программный интерфейс приложения) определяет набор функций, макросов и переменных, которые обеспечивают доступ к большинству аспектов Python системы времени выполнения. API Python включается в исходный файл C путем включения `"Python.h"` заголовка.

Компиляция модуля расширения зависит от его предполагаемого использования, а также от настройки системы; подробности приведены в последующих главах.

Примечание: Интерфейс расширения C специфичен для CPython, и модули расширения не работают в других реализациях Python. Во многих случаях можно избежать записи расширений C и сохранить переносимость на другие реализации. Например, если используется вызов функций библиотеки C или системных вызовов, вместо записи пользовательского кода C следует использовать модуль `ctypes` или библиотеку cffi. Эти модули позволяют писать Python код с интерфейсом C кода и является более переносимыми между реализациями Python, чем запись и компиляция модуля расширения C.

2.1.1 Простой пример

Давайте создадим модуль расширения под названием `spam` (любимая еда фанатов Монти Пайтон...) и допустим, мы хотим создать Python интерфейс к функции C библиотеки `system()`[1]. Эта функция принимает в качестве аргумента строки символов завершаемая нулем и возвращает целое число. Мы хотим, чтобы эта функция вызывалась из Python следующим образом:

```
>>> import spam
>>> status = spam.system("ls -l")
```

Начните с создания `spammodule.c` файла. (Исторически, если модуль называется `spam`, файл C, содержащий его реализацию, называется `spammodule.c`; если имя модуля очень длинное, как и `spammify`, имя модуля может быть только `spammify.c`.)

Первые две строки нашего файла могут быть:

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

который получает Python API (вы можете добавить комментарий, описывающий назначение модуля и уведомление об авторских правах, если хотите).

Примечание: Поскольку Python могут определять некоторые предпроцессорные определения, которые влияют на стандартные заголовки в некоторых системах, вы *должны* включить `Python.h` перед включением каких-либо стандартных заголовков.

Рекомендуется всегда определять `PY_SSIZE_T_CLEAN` перед включением `Python.h`. Описание этого макроса см. в разделе *Извлечение параметров в функциях расширения*.

Все видимые пользователем символы, определенные `Python.h`, имеют префикс `Py` или `PY`, за исключением символов, определенных в стандартных файлах заголовков. Для удобства и так как они широко используются Python интерпретатором, `"Python.h"` включает несколько стандартных заголовочных файлов: `<stdio.h>`, `<string.h>`, `<errno.h>` и `<stdlib.h>`. Если последний заголовочный файл не существует в системе, он объявляет функции `malloc()`, `free()` и `realloc()` напрямую.

Следующее, что мы добавим в наш файл модуля, это функция C, которая будет вызвана при вычислении Python `spam.system(string)` выражения (мы скоро посмотрим, как она в конечном итоге будет вызвана):

```
static PyObject *
spam_system(PyObject *self, PyObject *args)
{
    const char *command;
    int sts;

    if (!PyArg_ParseTuple(args, "s", &command))
        return NULL;
    sts = system(command);
    return PyLong_FromLong(sts);
}
```

Существует прямой перевод из списка аргументов в Python (например, `"ls -l"` одного выражения) в аргументы, переданные функции C. Функция C всегда имеет два аргумента, условно именованные *self* и

[1] Интерфейс для этой функции уже существует в стандартном `os` модуле – он был выбран в качестве простого и понятного примера.

args.

Аргумент *self* указывает на объект модуля для функций уровня модуля; для метода он указывает на сущность объекта.

Аргумент *args* будет указателем на объект кортежа Python, содержащий аргументы. Каждый элемент кортежа соответствует аргументу в списке аргументов вызова. Аргументы являются Python объектами — для того, чтобы сделать что-либо с ними в нашей функции C, мы должны преобразовать их в C значения. Функция, `PyArg_ParseTuple()` в API Python, проверяет типы аргументов и преобразует их в C значения. Он использует строку шаблона для определения требуемых типов аргументов, а также типов переменных C, в которых будет храниться преобразованный значения. Подробнее об этом позже.

`PyArg_ParseTuple()` возвращает true (ненулевое значение), если все аргументы имеют правильный тип и их компоненты сохранены в переменных, адреса которых переданы. Если передан недопустимый список аргументов, возвращает false (ноль). В последнем случае также возникает соответствующее исключение, так что вызывающая функция может возвращает `NULL` немедленно (как мы видели в примере).

2.1.2 Интермеццо: ошибки и исключения

Важным соглашением во всем интерпретаторе Python является следующее: при сбое функции она должна задать условие исключения и возвращает значение ошибки (обычно указатель на `NULL`). Исключения хранятся в статической глобальной переменной внутри интерпретатора; если эта переменная не `NULL` исключение. Вторая глобальная переменная хранит «связанный значение» исключения (второй аргумент `raise`). Третья переменная содержит трейсбэк стека в случае возникновения ошибки в Python коде. Эти три переменные - эквиваленты C результату в Python `sys.exc_info()` (см. раздел на модуле `sys` в справочнике библиотеки Python). Важно знать о них, чтобы понять, как передаются ошибки.

API Python определяет ряд функций для установки различных типов исключений.

Самый распространенный - `PyErr_SetString()`. Его аргументы являются объектом исключения и C-строка. Объект исключения обычно является предопределенным объектом типа `PyExc_ZeroDivisionError`. C строка указывает причину ошибки и преобразуется в объект Python строка и сохраняется как «связанный значение» исключения.

Другой полезной функцией является функция `PyErr_SetFromErrno()`, которая принимает только аргумент исключения и создает связанный значение путем проверки `errno` глобальной переменной. Наиболее общей функцией является `PyErr_SetObject()`, которая принимает два аргумента объекта, исключение и связанные с ним значение. Не требуется `Py_INCREF()` объекты, переданные ни одной из этих функций.

Можно проверить без разрушения, была ли установлена особая ситуация с помощью `PyErr_Occurred()`. Она возвращает текущий объект исключения или `NULL`, если исключения не возникло. Как правило, нет необходимости вызывать `PyErr_Occurred()`, чтобы узнать, произошла ли ошибка в вызове функции, так как вы должны быть в состоянии сообщить из возвращаемого значения.

Когда функция *f*, вызывающая другую функцию *g*, обнаруживает, что последняя терпит неудачу, *f* сама должна вернуть значение ошибки (обычно NULL или -1). Она *не* должна вызывать одну из функций `PyErr_*()` — одна уже была вызвана *g*. Тогда предполагается, что вызывающая *f* также должена вернуть ошибку указание на *его* вызывающего, опять же *без* вызова `PyErr_*()` и так далее — самая подробная причина ошибки уже сообщена функцией которая первой его обнаружила. Как только ошибка достигает основного цикла интерпретатора Python, это прерывает выполнение текущего кода Python и пытается найти обработчик исключений, указанный программистом Python.

(Бывают ситуации, когда модуль может фактически выдать более подробное сообщение об ошибке, вызвав другую функцию `PyErr_*()`, и в таких случаях это нормально. Как правило, однако, это не обязательно, и может привести к потере информации о причине ошибки: большинство операций могут завершиться неудачей по самым разным причинам.)

Чтобы игнорировать исключение, установленное неуспешным вызовом функции, условие исключения должно быть удалено явным вызовом `PyErr_Clear()`. Единственный раз, когда C код должен вызвать `PyErr_Clear()`, это если он не хочет передавать ошибку на интерпретатор, но хочет справиться с ней полностью самостоятельно (возможно, попытавшись что-то еще, или притворившись, что ничего не пошло не так).

Каждый неудачный `malloc()` вызов должен быть превращен в исключение, — прямой вызов `malloc()` (или `realloc()`) должен вызвать `PyErr_NoMemory()` и сам вернуть индикатор отказа. Все функции создания объектов (например, `PyLong_FromLong()`) уже выполняют эту функцию, поэтому эта заметка относится только к тем, кто вызывает `malloc()` напрямую.

Также следует отметить, что, за важным исключением `PyArg_ParseTuple()` и друзей, функции, которые возвращает целочисленный статус, обычно возвращает положительное значение или ноль для успеха и `-1` при сбое, как вызовы системы Unix.

Наконец, будьте внимательны к очистке мусора (совершая `Py_XDECREF()` или `Py_DECREF()` вызовы уже созданных объектов) при возвращении индикатора ошибки!

Выбор, какое исключение поднять, полностью зависит от вас. Существуют предварительно объявленные объекты C, соответствующие всем встроенным исключениям Python, например, `PyExc_ZeroDivisionError`, которые можно использовать напрямую. Конечно, вы должны выбирать исключения разумно — не использовать `PyExc_TypeError`, чтобы означать, что файл не может быть открыт (вероятно, это должен быть `PyExc_IOError`). Если со списком аргументов что-то не так, функция `PyArg_ParseTuple()` обычно вызывает `PyExc_TypeError`. Если у вас есть аргумент, значение которого должны находиться в определенном диапазоне или должны удовлетворять другим условиям, `PyExc_ValueError` подходит.

Можно также определить новое исключение, уникальное для модуля. Для этого обычно объявляется статическая переменная объекта в начале файла:

```
static PyObject *SpamError;
```

и инициализировать его в функции инициализации модуля (`PyInit_spam()`) с помощью объекта исключения:

```
PyMODINIT_FUNC
PyInit_spam(void)
{
    PyObject *m;

    m = PyModule_Create(&spammodule);
    if (m == NULL)
        return NULL;

    SpamError = PyErr_NewException("spam.error", NULL, NULL);
    Py_XINCREF(SpamError);
    if (PyModule_AddObject(m, "error", SpamError) < 0) {
        Py_XDECREF(SpamError);
        Py_CLEAR(SpamError);
        Py_DECREF(m);
        return NULL;
    }

    return m;
}
```

Обратите внимание, что имя Python для объекта исключения `spam.error`. Функция

`PyErr_NewException()` может создать класс с `Exception` базовым классом (если не передан другой класс вместо `NULL`), описанный в bltin-exceptions.

Следует также отметить, что `SpamError` переменная сохраняет ссылку на вновь созданный класс исключений; это намеренно! Так как исключение может быть удалено из модуля внешними кодом, необходима собственная ссылка на класс, чтобы гарантировать, что он не будет удален, в результате чего `SpamError` станет висящим указателем. Если он станет висящим указателем, C код, который поднимает исключение, может вызвать дамп ядра или другие непреднамеренные побочные эффекты.

Ниже в этом примере рассматривается использование `PyMODINIT_FUNC` в качестве функции возвращаемого типа.

Исключение `spam.error` исключение может быть вызвано в вашем модуле расширения с помощью вызова `PyErr_SetString()`, как показано ниже:

```
static PyObject *
spam_system(PyObject *self, PyObject *args)
{
    const char *command;
    int sts;

    if (!PyArg_ParseTuple(args, "s", &command))
        return NULL;
    sts = system(command);
    if (sts < 0) {
        PyErr_SetString(SpamError, "System command failed");
        return NULL;
    }
    return PyLong_FromLong(sts);
}
```

2.1.3 Вернемся к примеру

Возвращаясь к нашей функции примера, теперь вы должны уметь понимать эту инструкцию:

```
if (!PyArg_ParseTuple(args, "s", &command))
    return NULL;
```

Он возвращает `NULL` (индикатор ошибки для функций, возвращающих указатели объектов), если ошибка обнаружена в списке аргументов, полагаясь на исключение, установленное `PyArg_ParseTuple()`. В противном случае строка значения аргумента была скопирована в локальную переменную `command`. Это назначение указателя, и вы не должны изменять строку, на которые он указывает (поэтому в стандарте C переменная `command` должна быть правильно объявлена как `const char *command`).

Следующий инструкция - вызов функции Unix `system()`, передавая ей строку, которую мы только что получили от `PyArg_ParseTuple()`:

```
sts = system(command);
```

Наша функция `spam.system()` должна возвращает значение `sts` как объект Python. Это выполняется с помощью функции `PyLong_FromLong()`.:

```
return PyLong_FromLong(sts);
```

В этом случае он будет возвращает целочисленным объектом. (Да, даже целые числа являются объектами в куче в Python!)

Если функция C не возвращает полезного аргумента (функция, возвращающая `void`), соответствующая функция Python должна возвращать `None`. Для этого нужна эта идиома (которая реализуется `Py_RETURN_NONE` макросом):

```
Py_INCREF(Py_None);
return Py_None;
```

`Py_None` - это имя C для специального Python объекта `None`. Это подлинный Python объект, а не указатель `NULL`, что означает «ошибка» в большинстве контекстов, как мы видели.

2.1.4 Таблица методов модуля и функция инициализации

Я обещал показать, как `spam_system()` вызывается из Python программ. Сначала необходимо перечислить его имя и адрес в «таблице методов»:

```
static PyMethodDef SpamMethods[] = {
    ...
    {"system",  spam_system, METH_VARARGS,
     "Execute a shell command."},
    ...
    {NULL, NULL, 0, NULL}        /* Страж */
};
```

Обратите внимание на третью запись (`METH_VARARGS`). Это флаг, указывающий интерпретатору соглашение о вызове, которое должно быть используемо для функции C. Обычно оно всегда должено быть `METH_VARARGS` или `METH_VARARGS | METH_KEYWORDS`; значение `0` означает `PyArg_ParseTuple()` устаревшего варианта использования.

При использовании только `METH_VARARGS` функция должна ожидать, что параметры Python-уровня будут переданы как кортеж, приемлемые для парсинга через `PyArg_ParseTuple()`; более подробная информация об этой функции приводится ниже.

Бит `METH_KEYWORDS` может быть установлен в третьем поле, если ключевой аргументы должны быть переданы функции. В этом случае функция C должна принимать третий параметр `PyObject *`, который будет словарем ключевых слов. Используйте `PyArg_ParseTupleAndKeywords()` для разбора аргументов такой функции.

На таблицу методов необходимо ссылаться в структуре определения модуля:

```
static struct PyModuleDef spammodule = {
    PyModuleDef_HEAD_INIT,
    "spam",   /* имя модуля */
    spam_doc, /* документация модуля, может иметь значение NULL */
    -1,       /* размер состояния модуля для каждого интерпретатора, или -1,
↪если
                 модуль сохраняет состояние в глобальных переменных. */
    SpamMethods
};
```

Эта структура, в свою очередь, должна быть передана интерпретатор в функции инициализации модуля. Функция инициализации должна иметь имя `PyInit_name()`, где *name* - имя модуля, и должна быть единственным элементом не в формате `static`, определенным в файле модуля:

```
PyMODINIT_FUNC
PyInit_spam(void)
```

(продолжение на следующей странице)

(продолжение с предыдущей страницы)

```
{
    return PyModule_Create(&spammodule);
}
```

Обратите внимание, что PyMODINIT_FUNC объявляет функцию как `PyObject *` возвращает type, объявляет любые специальные объявления связей, требуемые платформой, и для C++ объявляет функцию как `extern "C"`.

Когда программа Python впервые импортирует `spam` модуль, вызывается `PyInit_spam()`. (См. ниже комментарии о встраивании Python.) он вызывает `PyModule_Create()`, который возвращает объект модуля, и вставляет объекты встроенных функций в только что созданный модуль на основе таблицы (массива `PyMethodDef` структур), найденной в определении модуля. `PyModule_Create()` возвращает указатель на создаваемый объект модуля. Он может прерваться с неустранимой ошибкой для определенных ошибок или возвращает `NULL`, если модуль не может быть инициализирован удовлетворительно. Функция init должна возвращать объект модуля вызывающему, чтобы затем он был вставлен в `sys.modules`.

При встраивании Python функция `PyInit_spam()` не вызывается автоматически, если в таблице `PyImport_Inittab` нет записи. Чтобы добавить модуль в таблицу инициализации, используйте `PyImport_AppendInittab()`, после чего при необходимости выполните импорт модуля:

```
int
main(int argc, char *argv[])
{
    wchar_t *program = Py_DecodeLocale(argv[0], NULL);
    if (program == NULL) {
        fprintf(stderr, "Fatal error: cannot decode argv[0]\n");
        exit(1);
    }

    /* Добавить встроенный модуль перед Py_Initialize */
    if (PyImport_AppendInittab("spam", PyInit_spam) == -1) {
        fprintf(stderr, "Error: could not extend in-built modules table\n");
        exit(1);
    }

    /* Передайть argv[0] интерпретатору Python */
    Py_SetProgramName(program);

    /* Инициализировать Python интерпретатор. Необходимое.
       Если этот шаг завершится неудачно, это будет фатальная ошибка. */
    Py_Initialize();

    /* При необходимости импортировать модуль; кроме того,
       импорт можно отложить до тех пор, пока его не будет
       импортирован встроенный сценарий. */
    pmodule = PyImport_ImportModule("spam");
    if (!pmodule) {
        PyErr_Print();
        fprintf(stderr, "Error: could not import module 'spam'\n");
    }

    ...
```

(продолжение на следующей странице)

(продолжение с предыдущей страницы)

```
    PyMem_RawFree(program);
    return 0;
}
```

Примечание: Удаление записей из `sys.modules` или импорта скомпилированных модулей в несколько интерпретаторов в рамках процесса (или после `fork()` без промежуточного `exec()`) может создать проблемы для некоторых модулей расширения. Авторам модулей расширения следует проявлять осторожность при инициализации внутренних структур данных.

Более существенный пример модуля включен в дистрибутиве сорцов Python, как `Modules/xxmodule.c`. Этот файл может быть используем как шаблон или просто прочитан как пример.

Примечание: В отличие от нашего `spam` примера, `xxmodule` использует *многофазная инициализация* (новое в Python 3.5), где структура PyModureDef возвращенный от `PyInit_spam`, а создание модуля остается на усмотрение машенерии импорта. Дополнительные сведения о многофазной инициализации см. в разделе **PEP 489**.

2.1.5 Компиляция и линковка

Есть еще две вещи, которые нужно сделать, прежде чем использовать новое расширение: компиляция и линковка его с Python системой. При использовании динамической загрузки подробные данные могут зависеть от стиля динамической загрузки, используемого системой; для получения дополнительной информации см. разделы о компоновке модулей расширения (глава *Сборка C и C++ расширений*) и дополнительную информацию, относящуюся только к компоновке в Windows (глава *Создание расширений C и C++ в Windows*).

Если динамическая загрузка невозможна или требуется сделать модуль постоянной частью Python интерпретатора, необходимо изменить настройку конфигурации и перестроить интерпретатор. К счастью, это очень просто в Unix: просто поместите свой файл (`spammodule.c` например) в каталог `Modules/` распакованного исходного дистрибутива, добавьте строку в файл `Modules/Setup.local` описывающую ваш файл:

```
spam spammodule.o
```

и пересоберите интерпретатор, запустив **`make`** в каталоге верхнего уровня. Вы также можете запустить **`make`** в подкаталоге `Modules/`, но затем вы должны сначала пересобрать `Makefile` там, запустив „**`make`** Makefile“. (Это необходимо при каждом изменении файла `Setup`.)

Если модулю требуются дополнительные библиотеки для связи, они также могут быть перечислены в строке файла конфигурации для сущности:

```
spam spammodule.o -lX11
```

2.1.6 Вызов функций Python из C

Пока мы сконцентрировались на том, чтобы сделать функции C вызываемыми из Python. Также полезен реверс: вызов Python функций из C. Особенно это касается библиотек, поддерживающих так называемые «колбэк» функции. Если интерфейс C использует колбэки, эквивалентный Python часто должен обеспечивать механизм колбэков программисту Python; реализация потребует вызова функций Python колбэк из колбэк C. Другие виды использования также можно себе представить.

К счастью, Python интерпретатор легко вызывается рекурсивно, и существует стандартный интерфейс для вызова функции Python. (Я не буду останавливаться на том, как вызвать парсер Python с конкретной строкой в качестве входных данных — если вам интересно, посмотрите на реализацию параметра командной строки `-c` в `Modules/main.c` из исходного кода Python.)

Вызов функции Python прост. Во-первых, программа Python должна каким- то образом передать объект функции Python. Для этого необходимо предоставить функцию (или какой-либо другой интерфейс). При вызове этой функции сохраните указатель на объект Python функции (будьте внимательны, чтобы это `Py_INCREF()`!) в глобальной переменной — или там, где вы считаете нужным. Например, следующая функция может быть частью определения модуля:

```
static PyObject *my_callback = NULL;

static PyObject *
my_set_callback(PyObject *dummy, PyObject *args)
{
    PyObject *result = NULL;
    PyObject *temp;

    if (PyArg_ParseTuple(args, "O:set_callback", &temp)) {
        if (!PyCallable_Check(temp)) {
            PyErr_SetString(PyExc_TypeError, "parameter must be callable");
            return NULL;
        }
        Py_XINCREF(temp);         /* Добавить ссылку на новый обратный вызов
↪*/
        Py_XDECREF(my_callback);  /* Удалить предыдущий обратный вызов */
        my_callback = temp;       /* Запомнить новый обратный вызов */
        /* Шаблон для возврата "None" */
        Py_INCREF(Py_None);
        result = Py_None;
    }
    return result;
}
```

Эта функция должна быть зарегистрирована в интерпретаторе с использованием флага `METH_VARARGS`; это описано в разделе *Таблица методов модуля и функция инициализации*. Функция `PyArg_ParseTuple()` и ее аргументы описаны в разделе *Извлечение параметров в функциях расширения*.

Макросы `Py_XINCREF()` и `Py_XDECREF()` увеличивают/уменьшают количество ссылок на объект и безопасны в присутствии указателей `NULL` (но обратите внимание, что *temp* не будет `NULL` в этом контексте). Подробнее о них в разделе *Ссылочные счетчики*.

Позднее, когда наступает время вызова функции, вызывается функция C `PyObject_CallObject()`. Эта функция имеет два аргумента, указывающих на произвольные объекты Python: функцию Python и список аргументов. Список аргументов всегда должен быть объектом кортежа, длина которого является числом аргументов. Чтобы вызвать функцию Python без аргументов, передайте `NULL` или пустой кортеж; чтобы назвать его одним аргументом, передайте одноместный кортеж. `Py_BuildValue()` возвращает кортеж, если его строка формата состоит из нулевого или более кодов формата между скобками. Например:

```
int arg;
PyObject *arglist;
PyObject *result;
```

(продолжение на следующей странице)

(продолжение с предыдущей страницы)

```
...
arg = 123;
...
/* Время для вызова обратного вызова */
arglist = Py_BuildValue("(i)", arg);
result = PyObject_CallObject(my_callback, arglist);
Py_DECREF(arglist);
```

`PyObject_CallObject()` возвращает указатель Python объекта: это возвращаемое значение функции Python. `PyObject_CallObject()` является «ссылочно-счётно-нейтральным» по отношению к своим аргументам. В примере был создан новый кортеж, служащий списком аргументов, который является `Py_DECREF()` сразу после вызова `PyObject_CallObject()`.

Возвращаемое значение `PyObject_CallObject()` является «новым»: либо это совершенно новый объект, либо это существующий объект, число ссылок на который было увеличено. Так что, если вы не хотите сохранить его в глобальной переменной, вы должны как- то сохранить `Py_DECREF()` результат, даже (особенно!), если вы не заинтересованы в его значении.

Перед этим, однако, важно проверить, что возвращаемое значение не `NULL`. Если это так, функция Python прерывается, вызывая исключение. Если C код, который вызывает `PyObject_CallObject()` вызывается из Python, он должен теперь возвращает ошибочный признак его вызываемому Python, таким образом, интерпретатор может напечатать трассировку стека, или вызывающий Python код может обрабатывать исключение. Если это невозможно или нежелательно, исключение должно быть устранено вызовом `PyErr_Clear()`. Например:

```
if (result == NULL)
    return NULL; /* Вернуть ошибку */
...use result...
Py_DECREF(result);
```

В зависимости от требуемого интерфейса функции Python колбэк может потребоваться также предоставить список аргументов для `PyObject_CallObject()`. В некоторых случаях список аргументов также предоставляется программой Python через тот же интерфейс, который указывал функцию колбэка. Затем его можно сохранить и использовать таким же образом, как и объект функции. В других случаях для передачи в качестве списка аргументов может потребоваться создать новый кортеж. Самый простой способ сделать это - вызвать `Py_BuildValue()`. Например, если требуется передать код интегрального события, можно использовать следующий код:

```
PyObject *arglist;
...
arglist = Py_BuildValue("(l)", eventcode);
result = PyObject_CallObject(my_callback, arglist);
Py_DECREF(arglist);
if (result == NULL)
    return NULL; /* Вернуть ошибку */
/* Здесь может быть использован результат */
Py_DECREF(result);
```

Обратите внимание на размещение `Py_DECREF(arglist)` непосредственно после вызова, перед проверкой на ошибки! Также отметим, что строго говоря этот код не является полным: `Py_BuildValue()` может закончиться память, и это следует проверить.

Можно также вызвать функцию с ключевыми аргументами, используя `PyObject_Call()`, которая поддерживает аргументы и ключевой аргументы. Как и в приведенном выше примере, мы используем `Py_BuildValue()` для построения словаря.:

```
PyObject *dict;
...
dict = Py_BuildValue("{s:i}", "name", val);
result = PyObject_Call(my_callback, NULL, dict);
Py_DECREF(dict);
if (result == NULL)
    return NULL; /* Вернуть ошибку */
/* Здесь может быть использован результат */
Py_DECREF(result);
```

2.1.7 Извлечение параметров в функциях расширения

Функция `PyArg_ParseTuple()` объявляется следующим образом:

```
int PyArg_ParseTuple(PyObject *arg, const char *format, ...);
```

Аргумент *arg* должен быть объектом кортежа, содержащим список аргументов, передаваемых из функции Python в функцию C. Аргумент *format* должен быть форматной строкой, синтаксис которой объясняется в arg-parsing в справочном руководстве API Python/C. Остальные аргументы должны быть адресами переменных, тип которых определяется строкой формата.

Заметим, что в то время как `PyArg_ParseTuple()` проверяет, что Python аргументы имеют требуемые типы, он не может проверить действительность адресов переменных C, переданных вызову: если вы совершаете там ошибки, ваша код, вероятно, потерпит крах или, по крайней мере, перезапишет случайные биты в памяти. Так что будьте осторожны!

Следует отметить, что любые Python ссылки на объекты, которые предоставляются вызывающему, являются *заимствованными* ссылками; не уменьшая их количество ссылок!

Некоторые примеры вызовов:

```
#define PY_SSIZE_T_CLEAN  /* Сделайте так, чтобы "s#" использовал Py_ssize_t,
↪a не int. */
#include <Python.h>
```

```
int ok;
int i, j;
long k, l;
const char *s;
Py_ssize_t size;

ok = PyArg_ParseTuple(args, ""); /* Без аргументов */
    /* Вызов Python: f() */
```

```
ok = PyArg_ParseTuple(args, "s", &s); /* Строка */
    /* Возможный вызов Python: f('whoops!') */
```

```
ok = PyArg_ParseTuple(args, "lls", &k, &l, &s); /* Две длинные и строка */
    /* Возможный вызов Python: f(1, 2, 'three') */
```

```
ok = PyArg_ParseTuple(args, "(ii)s#", &i, &j, &s, &size);
    /* Пара целых чисел и строка, размер которой также возвращается */
    /* Возможный вызов Python: f((1, 2), 'three') */
```

```
{
    const char *file;
    const char *mode = "r";
    int bufsize = 0;
    ok = PyArg_ParseTuple(args, "s|si", &file, &mode, &bufsize);
    /* Строка и, возможно, другая строка и целое число */
    /* Возможные вызовы Python:
       f('spam')
       f('spam', 'w')
       f('spam', 'wb', 100000) */
}
```

```
{
    int left, top, right, bottom, h, v;
    ok = PyArg_ParseTuple(args, "((ii)(ii))(ii)",
             &left, &top, &right, &bottom, &h, &v);
    /* Прямоугольник и точка */
    /* Возможные вызовы Python:
       f(((0, 0), (400, 300)), (10, 10)) */
}
```

```
{
    Py_complex c;
    ok = PyArg_ParseTuple(args, "D:myfunction", &c);
    /* комплекс, также предоставляющий имя функции для ошибок */
    /* Возможный вызов Python: myfunction(1+2j) */
}
```

2.1.8 Ключевые параметры для функций расширения

Функция `PyArg_ParseTupleAndKeywords()` объявляется следующим образом:

```
int PyArg_ParseTupleAndKeywords(PyObject *arg, PyObject *kwdict,
                                const char *format, char *kwlist[], ...);
```

Параметры *arg* и *format* идентичны параметрам функции `PyArg_ParseTuple()`. Параметр *kwdict* - это словарь ключевых слов, полученных в качестве третьего параметра из среды выполнения Python. Параметр *kwlist* представляет собой `NULL` завершаемых список строк, идентифицирующих параметры; имена сопоставляются с информацией о типе из *format* слева направо. При успехе `PyArg_ParseTupleAndKeywords()` возвращает true, в противном случае возвращает false и вызывает соответствующее исключение.

Примечание: Невозможно выполнить синтаксический анализ вложенных кортежей при использовании ключевой аргументов! Переданные параметры ключевого слова, отсутствующие в *kwlist*, вызовут `TypeError`.

Вот примерный модуль, в котором используются ключевые слова, основанный на примере Джеффа Филбрика (philbrick@hks.com):

```
#define PY_SSIZE_T_CLEAN  /* Сделать "s#" используя Py_ssize_t, а не int. */
#include <Python.h>
```

(продолжение на следующей странице)