Embedded Python: Using C API

개괄

파이썬의 C API를 직접적으로 이용하여 파이썬 스크립트를 호출, 활용합니다. 파이썬의 C API는 C/C++를 이용해 파이썬 삽입/확장을 위해 제공하는 API입니다. 파이썬의 소스가 C로 되어 있는데, 그 안에서도 이 API를 사용하고 있습니다. 파이썬 자신이 우선 C 기반의 API를 선언, 구현한 후, 이것을 외부에도 사용할 수 있도록 공개해 둔 것입니다.

차후에 설명할 boost::python등의 라이브러리가 실제 프로그래밍에 있어서는 보다 생산적입니다만, 결국은 모두 파이썬 C API에서 출발합니다. 그러므로 C API를 먼저 이해하는 것이 중요합니다.

본 문서에서는 파이썬 삽입에 대한 일반적인 시나리오를 하나 가정하고, 그것을 달성하기 위해 파이썬 객체에 접근하는 여러 방법에 대해 간략하게 서술합니다. 이 문서는 개인적인 어떤 프로젝트를 하나 수행하고, 그 결과를 정리해 두기 위해 작성된 것입니다. 부분에 따라서는 정확하지 않은 부분도 있을 수 있으니 항상 파이썬 레퍼런스를 확인하시기 바랍니다.

시작 및 종료

파이썬 API를 사용하기 위해서는 반드시 전역 초기화를 수행하고, 사용을 종료한 후에는 전역 마무리를 해야 합니다.

초기화를 수행하는 함수는 Py_Initialize()입니다. 모듈을 생성, 로딩하고 경로를 수집하는 등의 전반적인 초기화를 수행합니다. 이 함수를 시작하지 않고서는 프로그램 진행이 제대로 되지 않습니다. 일반적으로 초기화는 프로그램 전역에 걸쳐 한 번만 실행되어야 합니다. 혹시 전역 초기화가 이루어졌는지 확인하기 위해서는 Py_IsInitialized() 함수를 호출하여 판단하면 됩니다.

또한 프로그램이 종료되기 전에 파이썬 마무리 함수가 반드시 한 번 불려져야 합니다. Py_Finalize() 함수를 불러주세요. 그렇다면 파이썬을 이용해 “Hello, World!”를 출력하는 간단한 예제부터 시작해 보도록 하지요.

hello_python.c

```c
#include <Python.h>

int main(int argc, char** argv)
{
  Py_Initialize();

  if (Py_IsInitialized()) {
    PyRun_SimpleString("print \'Hello, World!\'\n");
    Py_Finalize();
  }
```

```
    return 0;
}
```

경로 및 인자 입력

파이썬이 초기화되기 전에 Py_SetProgramName() 함수를 호출할 수 있습니다. 기본값은 'python'입니다. (이것을 변경한다고 해서 크게 의미는 없습니다.)

PySys_SetArgv() 함수로 파이썬에게 인자를 전달할 수 있습니다. PySys_SetArgv() 함수는 PySys_SetArgvEx() 함수의 축약형인데, 마지막 updatepath의 인자를 1로 전달합니다. 이 값이 0이 아니면 다음과 같은 작업이 일어납니다.

- 첫번째 인자(argv[0])의 절대 경로가 sys.path의 목록 가장 처음에 삽입(prepend)됩니다.
- 만약 첫번째 인자의 경로가 정확하지 않거나 argc가 0일 경우에는 프로그램이 실행되는 경로 상대값이(".") sys.path의 목록 처음에 삽입됩니다.

파이썬 경로

파이썬 스크립트가 어떤 모듈을 가져오려면(import) 모듈의 경로를 미리 알고 있어야 합니다. 이 경로는 보통 sys.path에 기록됩니다. PySys_SetArgv() 함수를 지정해 준다면 현재 스크립트가 실행되는 경로를 기준으로 sys.path에 기록이 됩니다. 그러나 다른 경로에 있는 모듈을 파이썬 스크립트에서 사용하고 싶다면 반드시 이 경로를 입력해 주어야 합니다.

```
PyRun_SimpleString("import sys\n"
           "sys.path.append(<path>)\n");
```

위와 같은 코드를 사용하면 간단하게 되겠죠.

간단한 예제

단순 문자열 출력만으로는 우리가 원하는 형태의 임베딩이 되지 않습니다. 조건에 따라 함수도 호출해야 합니다. 그 함수에는 인자도 자유롭게 입력할 수 있어야 하고 결과도 받아올 수 있어야 합니다. 그러면 또 하나의 소스 코드 예제를 통해 이러한 작업을 어떻게 할지 설명해 보도록 하겠습니다.

소스 코드: 파이썬

이제 파이썬 스크립트 자체를 불러와서 실행하는 예를 기록해 보고자 합니다. 우선 삽입(embedding)될 파이썬 스크립트입니다.

my_def.py

```
def PrintMyDef():
    print "Hello, MyDef!"

def Multiply(x, y):
    return x * y
```

간단하게 아무 인자 없이 “Hello, MyDef!”를 출력하는 함수 하나와, 두 인자를 받아 그 합을 출력하는 함수 하나를 정의했습니다.

소스 코드: C

아래와 같이 파이썬 스크립트를 호출하는 C 코드를 작성해 보았습니다.

my_def.c

```
#include <Python.h>

void PrintMyDef()
{
  PyObject* mydef = PyImport_ImportModule("scripts.my_def");

  if(mydef) {
    PyObject* printHello = PyObject_GetAttrString(mydef, "PrintMyDef");

    if(printHello) {
      PyObject *r = PyObject_CallFunction(printHello, NULL);

      if (r == Py_None) {
        printf("None is returned.\n");
        Py_XDECREF(r);
      }

      Py_XDECREF(printHello);
    }
    Py_XDECREF(mydef);
  }
}

int Multiply(int x, int y)
{
  PyObject* mydef = PyImport_ImportModule("scripts.my_def");
  int result;

  if(mydef) {
    PyObject* multiply = PyObject_GetAttrString(mydef, "Multiply");

    if(multiply) {
```

```
      PyObject *r = PyObject_CallFunction(multiply, "ii", x, y);

      if(r) {
        result = (int)PyInt_AS_LONG(r);
        Py_XDECREF(r);
      }
      Py_XDECREF(multiply);
    }
    Py_XDECREF(mydef);
  }

  return result;
}

void SysPath()
{
  printf("sys.path:");
  PyRun_SimpleString("import sys\nprint sys.path\nprint\n");
}

int main(int argc, char** argv)
{
  Py_SetProgramName(argv[0]);
  Py_Initialize();
  printf("GetProgramName: %s\n\n", Py_GetProgramName());

  if (Py_IsInitialized()) {
    PySys_SetArg(argc, argv);
    printf("GetPath: %s\n\n", Py_GetPath());
    SysPath();
    PrintMyDef(); // Calling python functions
    printf("%d * %d = %d\n", 10, 15, Multiply(10, 15));
    Py_Finalize();
  }
  return 0;
}
```

소스 코드의 경로

C 실행 파일이 생성되는 곳에 “scripts”라는 디렉토리를 하나 만들고, 그곳에 my_def.py 파일을 저장합니다. 그리고 scripts 디렉토리에는 __init__.py라는 빈 파일을 하나 생성하세요. 언더바('_')는 init 앞뒤 각각 2개입니다. 대략 아래처럼 구성되겠죠.

```
./src
     my_def.c                  # C source code
./bin
     my_def                    # binary
```

```
    ./scripts/
              __init__.py      # for import
              my_def.py        # python source code
```

결과 설명

실행 결과는 다음과 유사합니다.

```
GetProgramName: ./my_def

... (생략) ...

Hello MyDef!
None is returned.
10 * 15 = 150
```

물론 'GetPath: ...' 등등의 설명을 위한 코드가 삽입되었다고는 하나, 파이썬으로는 사실 엄청나게 짧은 분량으로 가능할만한 코드가 C에서는 상당한 분량으로 늘어납니다. 게다가 프로그램이 에러 없이 잘 돌아갈 것이라는 확신 하에 에러 처리도 대충대충한 코드입니다. 그것까지 꼼꼼하게 한다면 분량은 더욱 늘어나겠죠.

간략하게 코드 설명을 하자면, 먼저 `Py_GetPath()` 함수를 호출하여 파이썬이 어떤 패스를 참조하고 있는지 출력해봅니다. 이런 부분에서 시스템에 따라 값이 달라질 수 있습니다. 참고로 제가 이 코드를 실행했을 때, 윈도우에서는 이 부분에서도 실행 파일이 있는 디렉토리까지의 절대 경로가 출력되는 반면, 리눅스에서는 그렇지 않았습니다. 그러므로 그런 차이를 `PySys_SetArgv()` 함수 콜을 통해 메꾸어 줍니다. 이제 `SysPath()` C 함수는 간단한 문자열로 파이썬 코드를 실행합니다. 파이썬의 `sys.path`를 호출해 보아서 경로가 제대로 나오는지 확인합니다.

이제 파이썬 mydef 모듈의 두 함수를 호출하는 코드를 수행합니다. 두 함수의 패턴은 비슷합니다.

1. 먼저 모듈을 import 합니다.
2. 모듈이 임포트 되어서 모듈을 참조하는 오브젝트를 얻으면, 각 함수를 참조하는 오브젝트를 얻어냅니다.
3. 함수의 오브젝트도 얻어지면, 그 함수를 불러냅니다.
   - "Hello MyDef!"를 출력하는 `PrintMyDef()` 함수도 실은 `None` 오브젝트를 리턴합니다. 이것은 C의 `NULL` 포인터와는 다릅니다. 확실히 메모리를 점유하고 있는 객체입니다.
   - `Multiply` 함수에는 인자 2개를 집어넣고, 결과를 받아옵니다.
4. 함수가 불러지고 나면 생성된 오브젝트들을 각각 정리(`Py_XDECREF`)합니다.

일반적인 시나리오

파이썬 스크립트가 어떤 동작을 하더라도 결국은 전산작업의 일반적인 흐름인 "(입력) → (작업) → (결과)"라는 도식 하에서 벗어날 일이 없을 것입니다. 다시 말해, C와 파이썬의 작업 관계에서 '입력' 단계와 '결과' 단계에서 C 코드가 능동적인 주체가 될 것이고, 파이썬 코드가 그것을 받아들이는 시스템이 될 겁니다. 입력값은 C가 처리해서 넘기고, 결과는 파이썬이 가져와 주되, 이것을 C가 읽을 수 있도록 해 주는 부가적인 작업이 포함됩니다. '작업' 단계 자체는 파이썬 스크립트가 주로 맡는 부분이 되지요.

이러한 패턴을 생각해 볼 때, C에서는 거의 다음과 같은 작업들을 정형적으로 수행합니다.

- **모듈 로딩**
  PyRun_SimpleString() 과 같은 짜여진 스크립트를 단순 실행하는 작업을 전적으로 할 수도 있지만, 아마 대개는 스크립트의 클래스, 변수, 함수들을 마치 파이썬을 이용하듯 자유자재로 이용하려고 하겠죠. 그렇다면 우선 파이썬 모듈을 임포트하는 과정이 반드시 포함되겠죠.

- **인스턴스화(선언) 및 오브젝트 참조**
  모듈에는 클래스, 변수, 그리고 함수들이 선언되어 있을 것입니다. 미리 선언된 변수라면 그것을 참조해야 하고, 함수라면 매개변수를 집어 넣어 호출을 해야 하고, 클래스라면 인스턴스화 하여 클래스 변수/클래스 함수에 접근해야 합니다.

- **리턴 값 추출**
  어떤 함수를 호출했으면 응답을 받아야 합니다. 파이썬이 생성해낸 값을 가져올 수 있어야 최종적인 목적이 달성될 것입니다.

- **모듈 및 파이썬 객체 제어**
  C 프로그램 철학 자체가 "프로그래머를 믿으라"는 것이니 모든 제어는 프로그래머가 해야 합니다. 파이썬의 메모리, 모듈, 객체 - 초기화, 마무리, 실행 중 관리 모두를 알아서 처리해야 합니다.

PyObject

일반적인 시나리오에서 말한 작업들을 알아보기 전에 우선 "PyObject"객체에 대해 잡깐 설명하고자 합니다.

파이썬의 자료는 C처럼 타입이 명시되어 있지 않죠. 타입이 없는 건 아닌데, 실행시 타입이 동적으로 변합니다. 그래서 한 변수가 문자열도, 정수형도, 실수형도, 때로는 클래스와 같은 오브젝트들도 다 참조 가능합니다.

이러한 데이터를 C API에서는 PyObject 구조체로 해석합니다. 이 구조체 안에 있는 데이터는 거의 직접적으로 접근하지 않습니다. 그 데이터를 처리하는 API를 통해 입출력을 수행합니다. 또한 오브젝트는 거의 스택에서 선언되는 일이 없습니다. 주로 힙에 선언이 되죠.

타입 판별

파이썬에서는 다음과 같은 코드가 가능합니다.

```
var = [1, 2, 3, 4]
var = {'wiki': 'doku', 'cms': 'wordpress', }
var = 3
var = "Foo"
var = 68.23
```

'var'라는 변수 안에 리스트, 딕셔너리, 정수, 문자열, 실수까지 맘대로 대입 가능합니다. 그러므로 실행 시점에서 어떤 변수는 어떤 타입을 가지는지는 직접 그 타입이 어떤지 체크를 해 보아야 합니다.

PyObject가 어떤 타입인지 체크를 하기 위해 Py*_Check() 함수들이 있습니다. '*' 안에는 해당 타입이 들어갑니다. 예를 들어 PyObject 포인터 타입인 p가 리스트인지 확인하려면,

```
if(PyList_Check(p)) {
```

```
    .....
```

이렇게 되겠죠.

레퍼런스 카운팅

파이썬에서 객체는 파이썬이 알아서 관리합니다. 선언된 객체는 boost::shared_ptr와 비슷한 동작을 합니다. 객체 내부에 카운터가 있어서 대입되면 카운터가 늘어나고, 대입된 변수의 범위를 벗어나면 카운터가 줄어듭니다. 만약 카운터가 0이 되면 메모리는 알아서 해제됩니다. 그런데 C API를 쓰기 시작한 이상, 이 편한 기능은 포기해야 합니다. 나중에 boost::python 같은 C++ 클래스를 도입하면 조금 숨통이 트입니다만, C API만으로 프로그램을 짜려면 좀 까다롭습니다.

레퍼런스 카운팅은 프로그래머가 알아서 대입이 되는 순간 객체의 카운터를 늘려주고, 범위를 벗어날 때 알아서 카운터를 줄여 주어야 합니다. 예를 들면 이렇게 합니다.

```
PyObject *p = Py.....(); # 객체를 최초 할당받았습니다.
.....
  { // 지역 스코프 시작
  PyObject *r = p; # 객체가 한 번 더 참조되었습니다.
  Py_INCREF(r);    # 여기서 p를 넣든, q를 넣든 큰 차이는 없습니다.
  ....
  } // 지역 스코프 종료
Py_DECREF(p);      # 변수 r의 범위를 범위를 벗어났으므로 p의 레퍼런스 카운터를 감소시킵니다.
                   # 만약 p의 카운터가 0이되면 객체는 알아서 해제됩니다.
.....
```

INCREF

카운트 증가는 Py_INCREF() 매크로를 쓰면 됩니다. 비슷한 것으로 Py_XINCREF()가 있는데 이것은 NULL 포인터에 대해서는 아무런 동작을 하지 않아 보다 안전합니다. 파이썬 소스의 Py_INCREF 매크로는 다음처럼 되어 있습니다.

```
#define Py_INCREF(op) (                         \
    _Py_INC_REFTOTAL  _Py_REF_DEBUG_COMMA       \
    ((PyObject*)(op))->ob_refcnt++)
```

DECREF

카운트 감소는 Py_DECREF() 매크로와 Py_XDECREF() 매크로가 담당하고 있습니다. 'X'가 붙은쪽이 NULL 포인터에 대해서도 안전합니다. Py_DECREF 매크로는 다음처럼 선언되어 있습니다. _Py_CHECK_REFCNT() 매크로는 레퍼런스 카운팅이 음수로 떨어졌을 때 시스템에게 경고를 띄우고 abort()를 실행하도록 짜여 있습니다.

```
#define Py_DECREF(op)                                   \
```

```
    do {                                                    \
        if (_Py_DEC_REFTOTAL  _Py_REF_DEBUG_COMMA           \
        --((PyObject*)(op))->ob_refcnt != 0)                \
            _Py_CHECK_REFCNT(op)                            \
        else                                                \
        _Py_Dealloc((PyObject *)(op));                      \
    } while (0)
```

CLEAR

그리고 마지막으로 Py_CLEAR는 레퍼런스 카운트를 줄이되, 인자로 넣은 오브젝트를 안전하게 NULL로 만들어줍니다. 즉 이런 코드를 대신하는 것과 유사합니다.

```
Py_XDECREF(op);
op = NULL;
```

그러나 이 코드는 위험성을 가질 수 있습니다. op의 레퍼런스 카운트가 0이 되고 나서는 소멸자가 불리게 되는데, 이때 __del__과 같은 소멸자가 호출될 수도 있습니다. 문제는 이 소멸자는 같은 스레드가 아닌, 다른 스레드에서 실행될 가능성이 있다는 것입니다. op = NULL은 그것대로 실행되었고, 다른 스레드에서는 Py_DECREF(op)가 따로 실행됩니다. Py_DECREF() 매크로는 해제를 할 때 PyObject 자신을 참조해야 할 수가 있습니다. 그러나 이미 op는 NULL이 되어 버리니, segmentation fault 같은 문제를 일으킬 소지가 발생합니다. 그러므로 이를 막기 위해 Py_CLEAR() 매크로를 씁니다. 아래는 Py_CLEAR()의 구현입니다.

```
#define Py_CLEAR(op)                                    \
    do {                                                \
        if (op) {                                       \
            PyObject *_py_tmp = (PyObject *)(op);               \
            (op) = NULL;                                \
            Py_DECREF(_py_tmp);                         \
        }                                               \
    } while (0)
```

New, Borrow, Steal

어떤 API 함수는 PyObject를 새로 생성합니다. 예를 들자면 PyList_New() 함수입니다. 이 함수는 리스트를 새로 생성합니다. 당연히 리턴된 PyObject 객체는 프로그래머가 미리 선언한 포인터로 레퍼런스 하고 있어야 합니다. 프로그래머가 직접 레퍼런스 카운팅을 담당하여 자원 관리를 해야 합니다. 일종의 hard pointer입니다.

한편 어떤 API 함수는 객체의 소유권을 빌려(borrow)옵니다. 예를 들어 리스트의 객체 중 특정 위치의 객체를 가져오는 PyList_GetItem() 같은 함수가 그 좋은 예입니다. 이것은 PyList_New()처럼 PyObject에 대한 포인터를 리턴하지만, '빌려'온 것이므로 이 PyObject 포인터는 객체에 대한 소유권을 가지지 않습니다. 소유권은 리스트에 여전히 있다는 거죠. 그러므로 이렇게 가져온 객체에 대해서는 레퍼런스 카운터를 늘이거나 줄여서는 안 됩니다. 일종의 weak pointer네요.

다음은 PyTuple_GetItem() 함수의 구현 부분입니다.

```
PyObject *
PyTuple_GetItem(register PyObject *op, register Py_ssize_t i)
{
    if (!PyTuple_Check(op)) {
        PyErr_BadInternalCall();
        return NULL;
    }
    if (i < 0 || i >= Py_SIZE(op)) {
        PyErr_SetString(PyExc_IndexError, "tuple index out of range");
        return NULL;
    }
    return ((PyTupleObject *)op) -> ob_item[i];
}
```

튜플이 가지고 있는 객체의 포인터를 INC/DEC 처리 없이 그냥 리턴합니다. 이렇게 리턴한 객체에 대해 우리가 함부로 INC/DEC를 시행하면 나중에 문제가 발생하겠죠.

또 어떤 API 함수는 객체의 소유권을 강탈(steal)합니다. PyList_SetItem()이나 PyTuple_SetItme()과 같은 함수들이 이런 케이스입니다. 즉, 컨테이너인 자기 자신 안에 객체를 담아 두면서 소유권을 원래의 함수 인자로부터 빼앗아 옵니다. 다시 말해서 튜플이나 리스트 내의 PyObject 포인터가 객체를 레퍼런스 하면서 레퍼런스의 카운터를 증가시키지 않습니다. 만약 여기서 우리가 임의로 강탈된 포인터에 대해 DECREF를 해 버리면 객체는 해제되어 버리고 튜플이나 리스트 안의 포인터는 '댕글링(dangling)' 포인터가 됩니다.

다음은 Tuple의 SetItem 함수 구현 소스 코드입니다.

```
/* tupleobject.c */
int
PyTuple_SetItem(register PyObject *op, register Py_ssize_t i, PyObject *
newitem)
{
    register PyObject *olditem;
    register PyObject **p;
    if (!PyTuple_Check(op) || op->ob_refcnt != 1) {
        Py_XDECREF(newitem);
        PyErr_BadInternalCall();
        return -1;
    }
    if (i < 0 || i >= Py_SIZE(op)) {
        Py_XDECREF(newitem);
        PyErr_SetString(PyExc_IndexError,
                        "tuple assignment index out of range");
        return -1;
    }
    p = ((PyTupleObject *)op) -> ob_item + i;
    olditem = *p;
    *p = newitem;
    Py_XDECREF(olditem);
    return 0;
}
```

두 개의 if 구문은 타입 체크, 경계 체크를 위함입니다. i번째 PyObject 포인터를 가져와 newItem으로 대체할 때

Py_INCREF()를 호출하지 않죠. 반면 대치되는 olditem에는 Py_XDECREF()를 호출하지요. 즉 SetItem을 호출한 다음 객체는 튜플의 것이 됩니다. 혹시라도 튜플 밖에서도 계속 객체의 소유를 유지하고 싶다면 미리 SetItem을 호출하기 전에 레퍼런스를 증가시켜 놓아야 합니다.

어떤 함수가 객체의 소유권을 새로 만드는지, 빌려주는지, 강탈하는지는 레퍼런스에 잘 나와 있습니다. 객체의 '소유권'이 어떤 곳에 있느냐는 것은 꽤 중요한 문제입니다. 프로그래머가 직접 자원을 할당하고 해제하기 때문에 어떤 순간에 어떤 레퍼런스가 객체를 소유하고 있는지 명확하게 알아야 합니다. 보통 객체의 소유권을 가진 레퍼런스가 Py_DEFREF(), Py_INCREF()를 실행할 책임이 있고, 만약 이것이 틀리게 되면 레퍼런스 카운팅을 할 때 문제가 될 수 있습니다.

모듈 로딩

자세한 사항은 Importing Modules의 문서를 참고합니다.

파이썬의 import 명령어에 대한 C API입니다. 한 예는 PyImport_Import()입니다. 함수는 한 개의 인자를 받는데 C-스타일의 문자열이 아니라 PyObject 객체의 문자열을 넘깁니다. 해당 모듈을 레퍼런스하는 PyObject 객체를 리턴합니다.

PyImport_Import()와 비슷한 함수로 PyImport_ImportModule() 함수가 있습니다. 똑같이 한 개의 인자를 넘길 수 있는데, 여기는 C의 char 타입을 넣을 수 있습니다.

```
PyObject *pModuleName = PyString_FromString("sys");    # 모듈명 sys
PyObject *pSysModule  = PyImport_Import(pModuleName);  # 모듈 sys 임포트
```

인스턴스화

기본 자료형들

기본자료형들은 Py<X>_From<Y> 타입의 함수로부터 인스턴스화 할 수 있습니다. X는 파이썬 오브젝트가 내부적으로 가지는 파이썬의 자료형들이고, Y는 C 자료형들입니다.

```
PyObject *pi = PyInt_FromString("1000", NULL, 10); # 정수형 10진수 1000을 문자열로부터.
PyObject *pf = PyFloat_FromDouble(3.14);           # 실수 3.14로부터.
PyObject *ps = PyString_FromString("code");        # 문자열로부터.
```

클래스

다음과 같은 파이썬 스크립트가 있습니다.

my_def.py

```
def PrintMyDef():
    print "Hello, MyDef!"

def Multiply(x, y):
    return x * y

class MyClass:
    def __init__(self):
        print "MyClass instantiated"

    def __del__(self):
        print "MyClass deleted"

    def Test(self):
        print "Ok, tested!"
```

파이썬 인터프리터에서 다음과 같은 파이썬 코드를 통해 클래스를 인스턴스화 할 수 있습니다.

```
>>> import my_def
>>> my_def
<module 'my_def' from 'my_def.pyc'>
>>> hasattr(my_def, "MyClass")
True
>>> klass = my_def.'MyClass'
>>> klass
<class my_def.MyClass at 0x01FA7A40>
>>> k = klass()
MyClass instantiated
>>> k
<my_def.MyClass instance at 0x02437698>
>>> del k
MyClass deleted
```

이 파이썬 구문은 C API에서 어떻게 모듈에서 클래스 객체를 불러와야 할지 보여주는 청사진과 같습니다.

```
PyObject *my_def = PyImport_ImportModule("my_def");             # import
my_def
PyObject_HasAttrString(my_def, "MyClass");                      #
hasattr(my_def, "MyClass")
PyObject *klass = PyObject_GetAttrString(my_def, "MyClass");    # klass =
my_def.'MyClass'
PyObject *k = PyObject_CallObject(klass, NULL);                 # k = klass()
Py_XDECREF(k);                                                  # del k
```

함수 호출

모듈 함수

모듈의 함수는 어떤 모듈에서 바로 호출가능하므로 PyObject_CallMethod()를 이용할 수 있습니다.

```
/* import scripts.mydef */
myDef = PyImport_ImportModule("scripts.my_def");

/* res = my_def.Multiply(3, 2) */
PyObject *res = PyObject_CallMethod(myDef, "Multiply", "ii", 3, 2);
if(res) {
  /* print res */
  printf("%d\n", (int)PyInt_AsLong(res));
  Py_XDECREF(res);
}
```

또는 함수의 객체를 끝까지 레퍼런스한 다음 CallFunction() 함수를 부를 수도 있습니다.

```
/* Multiply = my_def.Multiply */
PyObject *multiply = PyObject_GetAttrString(myDef, "Multiply");
if(multiply && PyCallable_Check(multiply)) {
  /* res = Multiply(4, 5) */
  PyObject *res = PyObject_CallFunction(multiply, "ii", 4, 5);
  if(res) {
    /* print res */
    printf("%d\n", (int)PyInt_AsLong(res));
    Py_XDECREF(res);
  }
  Py_XDECREF(multiply);
}
```

클래스 함수

모듈의 내 함수나 클래스의 함수나 같은 관점에서 바라볼 수 있습니다. PyObject_CallMethod() 함수를 이용하여 클래스의 함수를 호출하면 됩니다.

```
if (myDef) {
  /* MyClass = my_def.MyClass */
  PyObject *myClass = PyObject_GetAttrString(myDef, "MyClass");

  if (myClass) {
    /* instance = MyClass() */
    PyObject *instance = PyObject_CallObject(myClass, NULL);

    if (instance) {
      /* noneResult = instance.Test() */
      PyObject *noneResult = PyObject_CallMethod(instance, "Test", NULL);

      if (noneResult == Py_None) {
```

```
        printf("None returned\n");
      }
      Py_XDECREF(instance);
    }
    Py_XDECREF(myClass);
  }
  Py_XDECREF(myDef);
}
```

그런데 이렇게 모듈을 끄집어 내고, 클래스를 끄집어 내고, 함수를 끄집어내고... 끄집어내고 끄집어내다 보면 if 문의 홍수가 나죠. 레퍼런스 관리는 철저하게 해야 하는데, 그렇다고 에러 체크를 안 할 수도 없고... 결국 goto 명령을 쓰게 되더라구요.

리턴 값 추출

함수 호출을 통해 PyObject 객체의 레퍼런스를 받았다면, 이 객체의 값을 C의 자료구조로 해석해 내야 합니다.

기본 자료형

기본 자료형들은 매우 간결하게 Py<X>_As<Y>() 함수 타입을 이용하면 됩니다. 'X'에는 파이썬 객체의 타입, Y는 C 타입의 자료형을 입력하면 됩니다.

튜플/리스트 등의 자료형

이외의 자료형들은 레퍼런스를 참고하라는 마법의 단어로 생략하도록 하겠습니다. 여기까지 읽고 이해했다면 아마 레퍼런스를 보고 어떻게 값을 가져올지 감이 잡히셨으리라 생각합니다. 레퍼런스의 'Abstract Objects Layer'와 'Concrete Objects Layer' 부분을 읽어 보면 각각의 자료형에서 데이터를 어떻게 꺼내올지, 또 반대로 어떻게 집어넣을지에 대해 자세하게 설명되어 있습니다.

파이썬 확장에 대해서는 그나마 몇몇 문서들을 만나긴 했지만, 파이썬 삽입에 대해서는 그다지 문서가 많지 않더군요. 처음에는 갈팡질팡했습니다. 그렇지만 문서를 작성하는 과정을 통해 C API를 레퍼런스를 보다 정독해보고 나니 그렇게까지 어려운 일은 아니라는 결론을 내렸습니다. 결국 파이썬 객체를 다루는 것이니, 파이썬이 동작하는 방식을 유추해보면 거의 흐름이 그려지게 되더군요.

예제: URL의 콘텐츠 가져오기

시나리오에 따라 기본적인 설명은 얼마만큼 작성된 것 나머지 더욱 상세한 사항들은 레퍼런스를 참고하기 바랍니다. 나머지는 예제를 통해 설명하는 것이 더 나을 것 같습니다.

아래 예제는 인자로 받은 URL로부터 약간의 HTML을 분석하는 코드입니다. 모든 a 태그 중 href 속성에서 'http(s)'로 시작하는 외부 링크를 받아 출력합니다. 또한 img 태그에서 jpg, gif, png 그림 파일은 'img'라는 디렉

토리를 만들어 그 곳에 저장하도록 합니다.

제가 C 프로그램에 아주아주 정통한 건 아닌지라 확실하게 단언할 수는 없지만, 일반적으로 별도의 라이브러리 없이 프로그래머가 소켓을 통해 생짜로 C에서 이렇게 HTTP 서버에 접속해서 HTML을 받아오고, 정규 표현식을 쓰거나 문자열 처리를 하기가 그다지 쉽지 않은 편입니다. 사실 작업 자체가 어렵지 않으니 C로 조금만 신경쓰면 곧 짤 수 있을 수도 있긴 하고, 애초에 이런 파이썬-C의 바인딩 따위는 하지 않아도 되지만...

C와 파이썬 사이에는 다음과 같은 입출력을 주고받기로 맞추었습니다.

- C → Python: target URL
- Python → C: List of URLs.
- 이미지 저장은 모두 파이썬 측에서 처리합니다.

finder 파이썬 모듈

finder.py

```
# -*- coding: utf-8 -*-

import urllib
import urlparse
import os
import re
import sys

class url_finder:

    def __init__(self, url):
        html = self.get_html(sys.argv[1])

        # url trimming
        url_parsed = urlparse.urlparse(url)
        self.url = '%s://%s' % (url_parsed[0], url_parsed[1])

        # exclude file name
        spl = url_parsed[2].split('/')
        if len(spl) > 1:
            self.url += '/'.join(spl[0:-1])

        # links, and images
        self.links = self.get_all_external_links(html)
        self.save_all_images(html)

    def get_all_external_links(self, html):
        found = re.findall(r'<a.*href=\"(https?.+?)\"', html)
        return found

    def save_all_images(self, html):
        if os.path.exists('./img') == False:
```

```
            os.mkdir('./img')

        for s in self.get_all_image_links(html):
            url = self.trim_image_url(s)
            filename = './img/' + s.split('/')[-1].strip()
            content = self.get_html(url)
            with open(filename, 'wb') as f:
                print "Saving [%s] to [%s]..." % (url, filename)
                f.write(content)

    def get_all_image_links(self, html):
        found = re.findall(r'<img.*src=\"(.+\.(gif|png|jpg))\"', html)
        return set(x[0] for x in found)

    def trim_image_url(self, src_url):
        if src_url[:4] == 'http':
            return src_url
        elif src_url[0] == '/':
            return "%s%s" % (self.url, src_url)
        else:
            return "%s/%s" % (self.url, src_url)

    def get_html(self, url):
        u = urllib.urlopen(url)
        html = u.read()
        u.close()
        return html

if __name__ == "__main__" :
    finder = url_finder("http://www.google.com")
    for x in finder.links:
        print x
```

finder C 코드

에러 처리가 불완전하지만 보여주는 것에 초점을 맞춰 간결하게 작성했습니다.

finder.c

```
#include <Python.h>

int main(int argc, char** argv)
{
  Py_SetProgramName(argv[0]);
  Py_Initialize();

  if (Py_IsInitialized()) {
```

```
    /* module, class, instance */
    PyObject *mod = NULL, *cls = NULL, *ins = NULL;

    /* argument, return */
    PyObject *url = NULL, *ret = NULL;

    PySys_SetArgv(argc, argv);

    mod = PyImport_ImportModule("scripts.finder");   // module
    cls = PyObject_GetAttrString(mod, "url_finder"); // class

    /* argument must be a tuple */
    url = PyTuple_New(1);
    PyTuple_SetItem(url, 0, PyString_FromString(argv[1])); // (url)

    ins = PyObject_CallObject(cls, url);        // instance
    ret = PyObject_GetAttrString(ins, "links"); // instance.links

    if (PyList_Check(ret)) {
      PyObject* iter = PyObject_GetIter(ret);
      PyObject* item;

      while(item = PyIter_Next(iter)) {
        if (PyString_Check(item)) {
          printf("%s\n", PyString_AsString(item));
        }
        Py_XDECREF(item);
      }
      Py_XDECREF(iter);
    }
    Py_XDECREF(ret);
    Py_XDECREF(ins);
    Py_XDECREF(url);
    Py_XDECREF(cls);
    Py_XDECREF(mod);
    Py_Finalize();
  }

  return 0;
}
```

파이썬 스크립트를 C에서 동작시키는 것에 불과하니, 서로 동작 결과는 동일합니다.

마치며

파이썬에는 여러 라이브러리들이 잘 갖춰져 있고, 또 쓰기 매우 쉽게 되어 있습니다. 기존의 어떤 작업을 C/C++, 혹은 다른 언어 기반으로 하고 있다가 전혀 다른 아이디어를 떠올리게 되었습니다. 사용 중인 언어 환경에서는 외부

라이브러리를 가져다가 빌드도 하고 세팅도 해야하고 ... 이것저것 밑작업도 많이 해야 하고 본 코드를 작성하는데도 조금 시간이 걸릴 수 있습니다.

이럴 때 이 새로운 부분만을 파이썬 스크립트로 프로토타이핑을 할 수 있습니다. 그런데 양 프로그램간 데이터 전송, 좀 더 거창하게는 프로세스 통신이 이뤄지면 보다 더 좋겠죠. 이럴 때 파이썬 삽입이 적합한 것 같습니다.

저도 이 문서를 제작하게 된 계기가 된 것이, 어떤 프로그램을 제작하는데, 네트워킹 기능과 텍스트 처리가 다소 필요한 부분이었습니다. C++로 이것을 시작하려 하니 한숨만 나오더군요. 그래서 우선 파이썬을 통해 빨리빨리 아이디어 스케치를 했습니다. 기능 구현은 빠르게 되었고, 이렇게 구성을 하면 좋겠다는 확신이 들었습니다. 또한 같은 객체 설계를 하더라도 스크립트로는 후딱후딱 만들고 바꾸기도 쉬우니 편리했습니다. 여기서 단순하게 기능이 동작된다는 것도 확인하였지만, 여기서 빠르게 프로토타이핑을 한 것이 설계적인 측면에서도 나름 이득이 되더군요.