# Python - Outras Extensões

Qualquer código escrito usando qualquer linguagem compilada como C, C++ ou Java pode ser integrado ou importado para outro script Python. Este código é considerado uma "extensão".

Um módulo de extensão Python nada mais é do que uma biblioteca C normal. Em máquinas Unix, essas bibliotecas geralmente terminam em **.so** (para objetos compartilhados). Em máquinas Windows, você normalmente vê **.dll** (para biblioteca vinculada dinamicamente).

## Pré-requisitos para escrever extensões

Para começar a escrever sua extensão, você precisará dos arquivos de cabeçalho do Python.

- Em máquinas Unix, isso geralmente requer a instalação de um pacote específico do desenvolvedor.
- Os usuários do Windows obtêm esses cabeçalhos como parte do pacote quando usam o instalador binário do Python.

Além disso, presume-se que você tenha um bom conhecimento de C ou C++ para escrever qualquer extensão Python usando programação C.

## Primeiro, dê uma olhada em uma extensão Python

Para dar uma primeira olhada em um módulo de extensão Python, você precisa agrupar seu código em quatro partes -

- O arquivo de cabeçalho Python.h .
- As funções C que você deseja expor como interface do seu módulo.
- Uma tabela mapeando os nomes de suas funções conforme os desenvolvedores Python as veem como funções C dentro do módulo de extensão.
- Uma função de inicialização.

## O arquivo de cabeçalho Python.h

Você precisa incluir o arquivo de cabeçalho Python.h em seu arquivo de origem C, o que lhe dá acesso à API Python interna usada para conectar seu módulo ao interpretador.

Certifique-se de incluir Python.h antes de qualquer outro cabeçalho necessário. Você precisa seguir as inclusões com as funções que deseja chamar do Python.

## As funções C

As assinaturas da implementação C de suas funções sempre assumem uma das três formas a seguir -

```
static PyObject *MyFunction(PyObject *self, PyObject *args);
static PyObject *MyFunctionWithKeywords(PyObject *self,
   PyObject *args,
   PyObject *kw);
static PyObject *MyFunctionWithNoArgs(PyObject *self);
```

Cada uma das declarações anteriores retorna um objeto Python. Não existe função void em Python como existe em C. Se você não deseja que suas funções retornem um valor, retorne o equivalente em C do valor **None** do Python . Os cabeçalhos do Python definem uma macro, Py_RETURN_NONE, que faz isso para nós.

Os nomes das suas funções C podem ser o que você quiser, pois nunca são vistos fora do módulo de extensão. Eles são definidos como funções estáticas .

Suas funções C geralmente são nomeadas combinando o módulo Python e os nomes das funções, conforme mostrado aqui -

```
static PyObject *module_func(PyObject *self, PyObject *args) {
   /* Do your stuff here. */
   Py_RETURN_NONE;
}
```

Esta é uma função Python chamada func dentro do módulo module. Você colocará ponteiros para suas funções C na tabela de métodos do módulo que geralmente vem a seguir em seu código-fonte.



## A tabela de mapeamento de métodos

Esta tabela de métodos é uma matriz simples de estruturas PyMethodDef. Essa estrutura se parece com isto -

```
struct PyMethodDef {
   char *ml_name;
   PyCFunction ml_meth;
   int ml_flags;
```

```
    char *ml_doc;
};
```

Aqui está a descrição dos membros desta estrutura -

- **ml_name** - Este é o nome da função que o interpretador Python apresenta quando é usado em programas Python.
- **ml_meth** - Este é o endereço de uma função que possui qualquer uma das assinaturas descritas na seção anterior.
- **ml_flags** - Isso informa ao intérprete qual das três assinaturas ml_meth está usando.
    - Esse sinalizador geralmente tem um valor METH_VARARGS.
    - Este sinalizador pode receber OR bit a bit com METH_KEYWORDS se você quiser permitir argumentos de palavras-chave em sua função.
    - Isso também pode ter um valor METH_NOARGS que indica que você não deseja aceitar nenhum argumento.
- **mml_doc** - Esta é a docstring da função, que pode ser NULL se você não quiser escrever uma.

Esta tabela precisa ser encerrada com um sentinela que consiste em valores NULL e 0 para os membros apropriados.

## Exemplo

Para a função definida acima, temos a seguinte tabela de mapeamento de métodos -

```
static PyMethodDef module_methods[] = {
    { "func", (PyCFunction)module_func, METH_NOARGS, NULL },
    { NULL, NULL, 0, NULL }
};
```



## A função de inicialização

A última parte do seu módulo de extensão é a função de inicialização. Esta função é chamada pelo interpretador Python quando o módulo é carregado. É necessário que a função seja nomeada **initModule** , onde Module é o nome do módulo.

A função de inicialização precisa ser exportada da biblioteca que você irá construir. Os cabeçalhos Python definem PyMODINIT_FUNC para incluir os encantamentos apropriados para que isso aconteça no ambiente específico em que estamos compilando. Tudo que você precisa fazer é usá-lo ao definir a função.

Sua função de inicialização C geralmente tem a seguinte estrutura geral -

```
PyMODINIT_FUNC initModule() {
   Py_InitModule3(func, module_methods, "docstring...");
}
```

Aqui está a descrição da função Py_InitModule3 -

- **func** - Esta é a função a ser exportada.
- **module_methods** - Este é o nome da tabela de mapeamento definido acima.
- **docstring** - Este é o comentário que você deseja fazer em sua extensão.

Juntando tudo isso, fica assim -

```
#include <Python.h>
static PyObject *module_func(PyObject *self, PyObject *args) {
   /* Do your stuff here. */
   Py_RETURN_NONE;
}
static PyMethodDef module_methods[] = {
   { "func", (PyCFunction)module_func, METH_NOARGS, NULL },
   { NULL, NULL, 0, NULL }
};
PyMODINIT_FUNC initModule() {
   Py_InitModule3(func, module_methods, "docstring...");
}
```

## Exemplo

Um exemplo simples que faz uso de todos os conceitos acima -

```
#include <Python.h>
static PyObject* helloworld(PyObject* self)
{
   return Py_BuildValue("s", "Hello, Python extensions!!");
}
static char helloworld_docs[] =
   "helloworld( ): Any message you want to put here!!\n";
static PyMethodDef helloworld_funcs[] = {
   {"helloworld", (PyCFunction)helloworld,
```

```
   METH_NOARGS, helloworld_docs},
   {NULL}
};
void inithelloworld(void)
{
   Py_InitModule3("helloworld", helloworld_funcs,
      "Extension module example!");
}
```

Aqui, a função Py_BuildValue é usada para construir um valor Python. Salve o código acima no arquivo hello.c. Veríamos como compilar e instalar este módulo para ser chamado a partir do script Python.



## Construindo e instalando extensões

O pacote distutils torna muito fácil distribuir módulos Python, tanto Python puro quanto módulos de extensão, de maneira padrão. Os módulos são distribuídos no formato fonte, construídos e instalados por meio de um script de configuração geralmente chamado setup.pyas .

Para o módulo acima, você precisa preparar o seguinte script setup.py -

```
from distutils.core import setup, Extension
setup(name='helloworld', version='1.0', \
   ext_modules=[Extension('helloworld', ['hello.c'])])
```

Agora, use o seguinte comando, que executaria todas as etapas necessárias de compilação e vinculação, com os comandos e sinalizadores corretos do compilador e do vinculador, e copiaria a biblioteca dinâmica resultante em um diretório apropriado -

```
$ python setup.py install
```

Em sistemas baseados em Unix, você provavelmente precisará executar este comando como root para ter permissões para gravar no diretório site-packages. Isso geralmente não é um problema no Windows.

## Importando extensões

Depois de instalar suas extensões, você poderá importar e chamar essa extensão em seu script Python da seguinte maneira -

```
import helloworld
print helloworld.helloworld()
```

Isso produziria a seguinte **saída** -

```
Hello, Python extensions!!
```

## Passando parâmetros de função

Como você provavelmente desejará definir funções que aceitem argumentos, você pode usar uma das outras assinaturas para suas funções C. Por exemplo, a função a seguir, que aceita um certo número de parâmetros, seria definida assim -

```
static PyObject *module_func(PyObject *self, PyObject *args) {
   /* Parse args and do something interesting here. */
   Py_RETURN_NONE;
}
```

A tabela de métodos contendo uma entrada para a nova função ficaria assim -

```
static PyMethodDef module_methods[] = {
   { "func", (PyCFunction)module_func, METH_NOARGS, NULL },
   { "func", module_func, METH_VARARGS, NULL },
   { NULL, NULL, 0, NULL }
};
```

Você pode usar a função API PyArg_ParseTuple para extrair os argumentos de um ponteiro PyObject passado para sua função C.

O primeiro argumento para PyArg_ParseTuple é o argumento args. Este é o objeto que você analisará. O segundo argumento é uma string de formato que descreve os argumentos como você espera que eles apareçam. Cada argumento é representado por um ou mais caracteres na string de formato como segue.

```
static PyObject *module_func(PyObject *self, PyObject *args) {
   int i;
   double d;
   char *s;
   if (!PyArg_ParseTuple(args, "ids", &i, &d, &s)) {
      return NULL;
   }

   /* Do something interesting here. */
   Py_RETURN_NONE;
}
```

Compilar a nova versão do seu módulo e importá-la permite invocar a nova função com qualquer número de argumentos de qualquer tipo -

```
module.func(1, s="three", d=2.0)
module.func(i=1, d=2.0, s="three")
module.func(s="three", d=2.0, i=1)
```

Você provavelmente poderá criar ainda mais variações.

## A função PyArg_ParseTuple

re é a assinatura padrão para a função **PyArg_ParseTuple** -

```
int PyArg_ParseTuple(PyObject* tuple,char* format,...)
```

Esta função retorna 0 para erros e um valor diferente de 0 para sucesso. Tupla é o PyObject* que foi o segundo argumento da função C. Aqui, o formato é uma string C que descreve argumentos obrigatórios e opcionais.

Aqui está uma lista de códigos de formato para a função **PyArg_ParseTuple** -

| Código | Tipo C | Significado |
|---|---|---|
| c | Caracteres | Uma string Python de comprimento 1 torna-se um caractere C. |
| d | dobro | Um float Python se torna um duplo C. |
| f | flutuador | Um float Python se torna um float C. |
| eu | interno | Um int Python se torna um int C. |
| eu | longo | Um int Python se torna um C longo. |
| eu | longo longo | Um int do Python se torna um C longo. |
| Ó | PyObject* | Obtém referência emprestada não NULL ao argumento Python. |
| S | Caracteres* | String Python sem nulos incorporados em C char*. |
| e# | caractere*+int | Qualquer string Python para endereço e comprimento C. |
| t# | caractere*+int | Buffer de segmento único somente leitura para endereço e comprimento C. |
| você | Py_UNICODE* | Python Unicode sem nulos incorporados em C. |

| | | |
|---|---|---|
| você# | Py_UNICODE*+int | Qualquer endereço e comprimento Python Unicode C. |
| c# | caractere*+int | Buffer de leitura/gravação de segmento único para endereço e comprimento C. |
| z | Caracteres* | Assim como s, também aceita None (define C char* como NULL). |
| z# | caractere*+int | Assim como s#, também aceita None (define C char* como NULL). |
| (...) | conforme ... | Uma sequência Python é tratada como um argumento por item. |
| \| | | Os seguintes argumentos são opcionais. |
| : | | Fim do formato, seguido do nome da função para mensagens de erro. |
| ; | | Fim do formato, seguido por todo o texto da mensagem de erro. |

## Retornando Valores

Py_BuildValue aceita uma string de formato muito parecida com PyArg_ParseTuple . Em vez de passar os endereços dos valores que você está construindo, você passa os valores reais. Aqui está um exemplo que mostra como implementar uma função add.

```
static PyObject *foo_add(PyObject *self, PyObject *args) {
   int a;
   int b;
   if (!PyArg_ParseTuple(args, "ii", &a, &b)) {
      return NULL;
   }
   return Py_BuildValue("i", a + b);
}
```

Seria assim se fosse implementado em Python -

```
def add(a, b):
   return (a + b)
```

Você pode retornar dois valores da sua função da seguinte maneira. Isso seria capturado usando uma lista em Python.

```
static PyObject *foo_add_subtract(PyObject *self, PyObject *args) {
   int a;
   int b;
   if (!PyArg_ParseTuple(args, "ii", &a, &b)) {
      return NULL;
   }
   return Py_BuildValue("ii", a + b, a - b);
}
```

Seria assim se fosse implementado em Python -

```
def add_subtract(a, b):
   return (a + b, a - b)
```

## A função Py_BuildValue

Aqui está a assinatura padrão para a função **Py_BuildValue** -

```
PyObject* Py_BuildValue(char* format,...)
```

Aqui, format é uma string C que descreve o objeto Python a ser construído. Os seguintes argumentos de Py_BuildValue são valores C a partir dos quais o resultado é construído. O resultado PyObject* é uma nova referência.

A tabela a seguir lista as cadeias de caracteres de código comumente usadas, das quais zero ou mais são unidas em um formato de cadeia de caracteres.

| Código | Tipo C | Significado |
|---|---|---|
| c | Caracteres | AC char se torna uma string Python de comprimento 1. |
| d | dobro | AC double se torna um float Python. |
| f | flutuador | O flutuador AC se torna um flutuador Python. |
| eu | interno | C int se torna um int Python |
| eu | longo | AC long se torna um Python int |
| N | PyObject* | Passa um objeto Python e rouba uma referência. |
| Ó | PyObject* | Passa um objeto Python e o INCREF normalmente. |
| O& | converter+anular* | Conversão arbitrária |
| é | Caracteres* | Char* terminado em 0 em C para string Python ou NULL para None. |

| | | |
|---|---|---|
| e# | caractere*+int | C char* e comprimento para string Python ou NULL para None. |
| você | Py_UNICODE* | String terminada em nulo em todo o C para Python Unicode ou NULL para None. |
| você# | Py_UNICODE*+int | String e comprimento em todo C para Python Unicode ou NULL para None. |
| c# | caractere*+int | Buffer de leitura/gravação de segmento único para endereço e comprimento C. |
| z | Caracteres* | Assim como s, também aceita None (define C char* como NULL). |
| z# | caractere*+int | Assim como s#, também aceita None (define C char* como NULL). |
| (...) | conforme ... | Constrói tupla Python a partir de valores C. |
| [...] | conforme ... | Constrói lista Python a partir de valores C. |
| {...} | conforme ... | Constrói dicionário Python a partir de valores C, alternando chaves e valores. |

O código {...} constrói dicionários a partir de um número par de valores C, alternadamente chaves e valores. Por exemplo, Py_BuildValue("{issi}",23,"zig","zag",42) retorna um dicionário como o {23:'zig','zag':42} do Python