Universidad de Costa Rica. Escuela de ingeniería eléctrica.

Proyecto eléctrico.

Jafet Soto Arrieta - B77543. Guía 01. I Semestre, 2024.

Cython.

Esta guía muestra el proceso de instalación y generación de código Cython para utilizar bibliotecas de código Python dentro de C, se encuentra en @GitHub.

Cython es un lenguaje que facilita escribir extensiones en C para Python, optimizando la integración y la velocidad de ejecución al permitir tipado estático opcional y operar con clases de C++. *Versión 3.0.10*

---

1. Instalación:

El código fuente de *Cython* se traduce a código optimizado C/C++ y se compila como módulos de extensión de *Python*, lo que permite una ejecución de programas muy rápida y una integración estrecha con bibliotecas externas en C, manteniendo al mismo tiempo la alta productividad para los programadores que caracteriza a *Python*.

Para instalar Cython 3.0.10:

```
> pip install Cython
```

Nota: Es posible que el directorio de instalación no se almacene en el directorio PATH del sistema, lo que genera un error en la detección del paquete. Para solucionarlo modifique la ruta del directorio de instalación:

```
> export PATH="$HOME/.local/bin:$PATH"
```

Mantener este cambio permanente:

```
>  echo 'export PATH="$HOME/.local/bin:$PATH"' >> ~/.bashrc
```

Actualizar la modificación:

```
>source ~/.bashrc
```

2. Utilizar una función Python en C con Cython:

A continuación se muestra el proceso necesario para generar una función Cython que permite comunicar un script de Python con código en C.

Primero debemos definir un archivo Python, para este ejemplo hemos diseñado un módulo que calcula el valor factorial de un número.

```python
def factorial_recursive(n):
    if n == 0 or n == 1:
        return 1
    else:
        return n * factorial_recursive(n - 1)
```

Puede renombrar este archivo como desee, para el ejemplo el nombre del archivo es *Pure_Python.py*, luego debemos diseñar el archivo de código Cython que permite relacionar los programas. En este archivo se debe importarte la función o funciones que deseamos llamar desde Cython:

```
# Importar la función Python

from Pure_Python import factorial_recursive
```

Para el ejemplo el archivo se llama CYTHON_CODE.pyx, además de importar las funciones Python externas, este archivo contiene la definición de la función en Cython que llama al módulo Python y lo comunica con la ejecución del código en C. El resto del archivo *.pyx* es:

```
cpdef public int cython_module(str name, int a, int b):

   print("Inicio del llamado a la función Cython.")

   print("Nombre:", name)



   # Llamada a la función Python desde Cython

   cdef int result_a = factorial_recursive(a)

   cdef int result_b = factorial_recursive(b)



   print(f"El factorial de {a} es {result_a}")

   print(f"El factorial de {b} es {result_b}")



   return a + b
```

Ahora debemos diseñar un script de configuración en Python, típicamente llamado *setup.py,* que utiliza la librería *setuptool*, diseñada para la creación de paquetes Python. El código del archivo de configuración es:

```
from setuptools import setup

from Cython.Build import cythonize

setup(

   name = "CYTHON_PrintLib",

     ext_modules = cythonize("CYTHON_CODE.pyx", compiler_directives =
{'language_level': "3"})

)
```

Se importa la función **setup**, que define cómo debe ser procesado y generado nuestro paquete; esta función configura metadatos, dependencias y otros detalles de los paquetes. Donde **name** es el nombre de nuestro paquete; para la configuración de los módulos externos utilizamos la función **cythonize()**, que permite convertir archivos *.pyx* a extensiones de archivos C, se indica el nombre del archivo fuente *CYTHON_CODE.pyx* e indicamos la version de Python que estamos utilizando, en este caso Python3.

Para compilar el paquete python que deseamos crear ejecutamos en la terminal:

```
python3 setup.py build_ext --inplace
```

La parte más extensa se diseña en C, donde debemos hacer la carga del módulo externo y completar el código, para combinar el llamado de la función Python con la ejecución del código C en tiempo real; definimos una función que solicita un nombre y 2 números para conocer el resultado factorial de los mismos; puede ver el ejemplo de esta función en el archivo **main.c**; este archivo es el código objetivo donde deseamos incorporar la función de Python.

Debemos modificar el PYTHONPATH para que el directorio actual se incluya y tengamos acceso a la función que contiene *CYTHON_CODE.h;* esto se soluciona con el comando:

```
PYTHONPATH=".:$$PYTHONPATH"
```

Esta configuración también se puede realizar de forma manual en el código C, como se muestra en las líneas 16-20 del código **main.c** en el repositorio.

Dentro del código **main.c** debemos llamas los headers necesarios para la implementación, incluyendo el archivo *CYTHON_CODE.h* que hemos generado con los pasos anteriores:

```
#include "CYTHON_CODE.h"                                    // Generado
por Cython.

#include <stdio.h>

#include <string.h>
```

Definimos la función principal ***main()***, aquí debemos llamar al intérprete de Python, lo que permite ejecutar funciones de la API de Python:

```
Py_Initialize();
```

Después de esto definimos la variables y llamamos a la función que solicita los datos del usuario.

Utilizando la función PyObject para definir un objeto Python desde C, el cual se gestiona por medio de un puntero, este objeto contiene la cadena de texto "CYTHON_CODE":

```
PyObject *pName = PyUnicode_FromString("CYTHON_CODE");
```

Ahora definimos un objeto Python que contenga el módulo externo:

```
PyObject *pModule = PyImport_Import(pName);
```

En este caso *pModule* es el nombre de la variable que almacena el puntero al objeto Python que representa el módulo importado. Si la carga falla, *pModule* apuntará a *Null*.

Decrementampos el contador de referencias que se debe hacer manualmente en C:

```
Py_DECREF(pName);
```

Ahora debemos verificar la carga del módulo y llamar a la función Cython.

```
if (pModule != NULL) {

                PyObject *pFunc = PyObject_GetAttrString(pModule,
"cython_module");
```

La función *PyObject_GetAttrString()* obtienen una referencia a la función *cython_module* del módulo *pModule*. Luego verificamos que exista la función y que se puede llamar:

```
    if (pFunc && PyCallable_Check(pFunc)) {
```

Crearemos una tupla que contenga los 3 datos necesarios para nuestro programa (nombre y los 2 números para conocer el resultado factorial) .

```
                PyObject *pArgs = PyTuple_New(3);
```

Para colocar elementos dentro de la tupla utilizamos la función *PyTuple_SetItem*(), indicando el puntero a la tupla, la posición que deseamos modificar y el contenido que vamos a guardar.

```
                PyTuple_SetItem(pArgs, 0, PyUnicode_FromString(name));

                PyTuple_SetItem(pArgs, 1, PyLong_FromLong(a));

                PyTuple_SetItem(pArgs, 2, PyLong_FromLong(b));
```

Las funciones *PyLong_FromLong()* y *PyUnicode_FromString()*, se utilizan para generar objetos Python de tipo entero y de cadena de texto respectivamente.

Ahora creamos un objeto Python para almacenar el resultado del llamado de la función con los parámetros contenidos en la tupla previamente generada:

```
PyObject *pResult = PyObject_CallObject(pFunc, pArgs);

Py_DECREF(pArgs);
```

Finalmente decrementamos el contador de referencias a *pArgs* para liberar la memoria. Después de esto verificamos si el contenido en *pResult* es diferente de Null, lo que implica que se ejecutó la función y decrementamos el contador de referencias a *pResult*. Si *pResult* es Null imprimimos el mensaje de error. Luego de la verificación de *pResul*t decrementamos el contador de referencias a *pFunc,* si pFunc es Null o no se puede llamar, se imprimen los mensajes de error y decrementamos el contador de referencias de *pModule*. En caso de que el módulo *pModule* no se cargara correctamente, llamamos a la función de impresión de errores de Python y finalizamos el intérprete de Python y el programa termina.