## línea horizontal



Cython

Manuel Alejandro Sánchez Molina

Miguel Ángel Hinojosa Castro

Carlos Núñez Molina

Jose Manuel Pérez Lendínez

Índice

Introducción 2

Proceso de instalación 2

Tutorial de uso 2

Proceso de compilación 5

Proceso de compilación: *setup.py* 6

Proceso de compilación: manual 8

Gráficas comparativas 8

Introducción

Cython es un lenguaje de programación que hace más fácil escribir código de C/C++, que se usará como módulo de python. Concretamente a partir de código python genera código C/C++ , que al compilarlo es capaz de ser llamado por este, de tal manera que ganamos la eficiencia que caracteriza a los lenguajes compilados.

La gran ventaja de cython es que no necesita interfaces y mantiene una característica positiva de python, la simplicidad del lenguaje( es igual pero añadiendo algunos tipos estáticos y algunas adaptaciones que se explicarán ).

# Proceso de instalación

El proceso de instalación de cython es bastante simple, se puede hacer de varias maneras:

$ apt-get install cython cython3

$ pip install cython

$ conda install cython

De tal manera que teniendo anaconda en windows , o en cualquier terminal de ubuntu se puede obtener rápidamente.

Esto no es lo único, ya que es necesario tener un compilador de C, por el simple de hecho de que es necesario para el código C generado en cython

# Tutorial de uso

Para el uso de esta herramienta es necesario tener un main.py que es el que llamara a los módulos.

Se tendrá un archivo módulo.pyx, que es el que contendrá el código python con ciertas modificaciones para ser traducido posteriormente a código C/C++.

Modificaciones:

-las variables que se usen en las funciones que van a ser modularizadas, deben de estar declaradas como en C, iniciando la línea de declaración con la palabra cdef, por ejemplo para un entero : **cdef int variable** . Para struct es similar, pero el cdef solo lo necesita el propio struct, por ejemplo:

**cdef struct persona:**

**int años**

Una facilidad que ofrece el cdef es que se pueden agrupar en un bloque tal que asi:

**cdef:**

**int var**

**float var2**

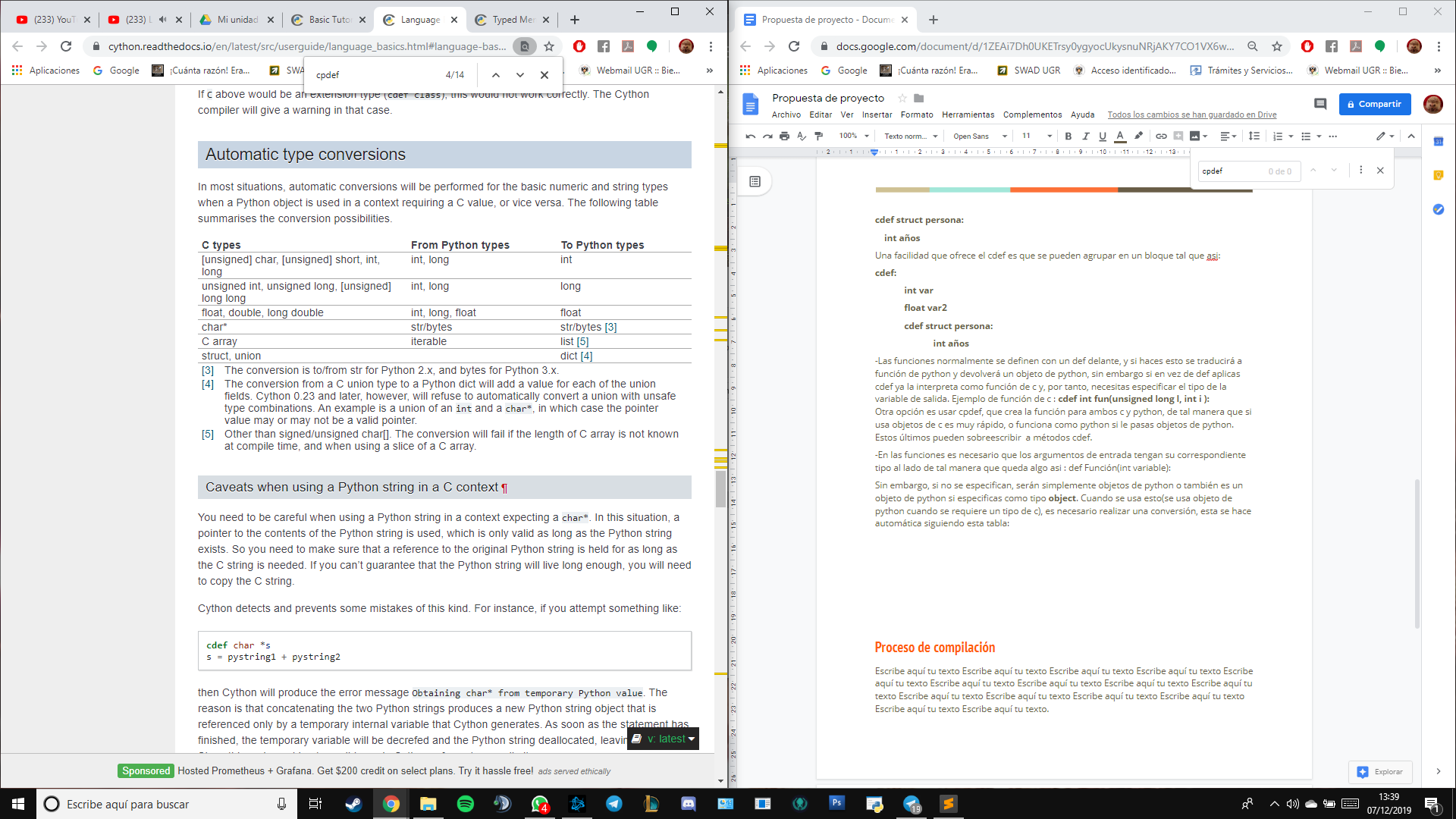
**cdef struct persona:**

**int años**

-Las funciones normalmente se definen con un def delante, y si haces esto se traducirá a función de python y devolverá un objeto de python, sin embargo si en vez de def aplicas cdef ya la interpreta como función de c y, por tanto, necesitas especificar el tipo de la variable de salida. Ejemplo de función de c : **cdef int fun(unsigned long l, int i ):** Otra opción es usar cpdef, que crea la función para ambos c y python, de tal manera que si usa objetos de c es muy rápido, o funciona como python si le pasas objetos de python. Estos últimos pueden sobreescribir a métodos cdef.

-En las funciones es necesario que los argumentos de entrada tengan su correspondiente tipo al lado de tal manera que queda algo asi : def Función(int variable):

Sin embargo, si no se especifican, serán simplemente objetos de python o también es un objeto de python si especificas como tipo **object**. Cuando se usa esto(se usa objeto de python cuando se requiere un tipo de c), es necesario realizar una conversión, esta se hace automática siguiendo esta tabla:



-En los casting, cython usa <> cuando c usa (), por ejemplo, para transformar un número de python a un entero de C : **<int>num\_python**

-En las expresiones hay ciertas diferencias:

->no existe el operador -> , se usa el . , asi que p->x se escribiría p.x

->no existe el operador unario de dirección \*

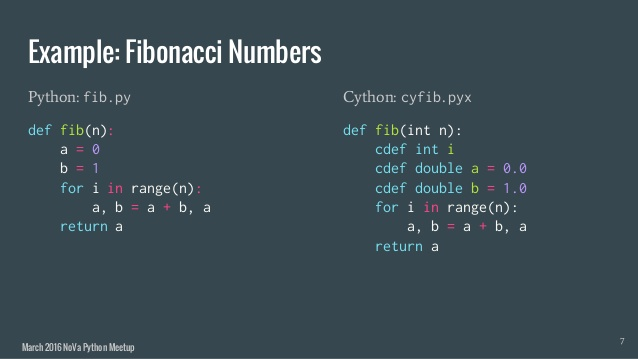
->el puntero a nulo de C se usa con la palabra reservada NULL

-En los bucles típicos de python, la variable que se itera, por ejemplo i, (for i in range ...) , si esa i está definida como cdef, el bucle se optimizará mejor.

-Para importar las librerías de C, por ejemplo, para importar la función sin de math de c: **from libc.math cimport sin** .

-Todo lo dicho es para C, para usar C++ necesitas poner en la primera línea la directriz **# distutils: language=c++** y para importar también se usa cimport, pero ahora la librería se llama libcpp, para importar la clase vector por ejemplo: **from libcpp.vector cimport vector**.

Ejemplo sencillo de la diferencia entre dos código uno en python y otro en cython:



# Proceso de compilación

A diferencia de un programa creado únicamente con Python, si utilizamos código de Cython debe ser compilado para poder ejecutarse. Este proceso de compilación ocurre en dos etapas:

* Un archivo con extensión *.pyx* es compilado a un archivo con extensión *.c* por Cython. En este archivo, contiene todo el programa de Python.
* El archivo *.c* que hemos obtenido en el proceso anterior de compilación, se vuelve a compilar por nuestro compilador de C. Este proceso da como resultado un nuevo archivo con extensión *.so* o *.py*, dependiendo de nuestro sistema operativo. Este nuevo archivo puede ser incluido en nuestro programa Python mediante *import*.

Hay varias formas de compilar código Cython:

* Actualmente, la forma más común es creando un archivo denominado *setup.py* utilizando para ello *distutils.*
* Usando directamente el comando *cython*  sobre nuestro archivo *.pyx* desde la línea de comandos para obtener el archivo *.c* y posteriormente compilarlo manualmente usando nuestro compilador C. Esta forma es muy interesante ya que nos permite introducir las optimizaciones de código del compilador.
* Otra forma muy similar a la primera, es utilizando directamente el comando *cythonize* en la consola para compilar el archivo *.pyx*.

A parte de estas tres formas, existen herramientas en línea que permiten experimentar y trabajar con Cython sin tener que preocuparse por los detalles de compilación.

Proceso de compilación: *setup.py*

En este apartado vamos a hacer un pequeño ejemplo para mostrar cómo podemos compilar nuestro código Cython usando un archivo auxiliar: *setup.py*.

Nuestro ejemplo estará formado por los siguientes archivos:

* *helloworld.pyx*. Un script básico en Python.

**def** say\_hello():

    print("Hello World")

* *setup.py*. El script que nos permitirá compilar el archivo creado anteriormente en C.

from distutils.core import setup

from Cython.Build import cythonize

setup(

    ext\_modules = cythonize("helloworld.pyx")

)

* *main.py.* Un script que ejecuta la función del primer archivo.

import helloworld

helloworld.say\_hello()

Para compilar nuestro archivo .pxy debemos de utilizar el siguiente comando en la línea de comandos:

$ python setup.py build\_ext --inplace

Cuando ejecutemos el comando anterior, en el directorio se habrá creado un nuevo directorio llamado *build* y dos nuevos archivos denominados *helloworld.c* y *helloworld.so*. Como hemos comentado en el apartado anterior, el archivo *.c* contiene la lógica de nuestro programa en C y el archivo *.so* es el ejecutable del programa.

Ya podemos ejecutar nuestro programa *main.py* y comprobar que funciona correctamente.

Hay varias modificaciones que hacen este método el más versátil. Un ejemplo concreto, se puede ver a continuación, donde en el archivo *setup.py* indicamos que se compilen todos los archivos con extensión *.pyx* del directorio de trabajo *ejemplo/*. Con la opción *name*  indicamos que todos los archivos pertenecen a la misma aplicación.

from distutils.core import setup

from Cython.Build import cythonize

setup( name = 'Hello world app',

    ext\_modules = cythonize("ejemplo/\*.pyx")

)

Hay otra forma más robusta de hacer el ejemplo anterior: mediante el uso del paquete *Extension*. Este paquete nos permite controlar mejor los archivos que van a ser compilados y, además, permite especificar opciones del compilador, las bibliotecas que queremos añadir y otras opciones del vinculador.

from distutils.core import setup

from distutils.extension import Extension

from Cython.Build import cythonize

extensions=[

    Extension("hello",       ["helloworld.pyx"]),

    Extension("bye",         ["bye.pyx"]),

]

setup(

    name = 'Hello world app',

    ext\_modules=cythonize(extensions),

)

Aunque el paquete *Extension* permite añadir las opciones del compilador como hemos comentado, la forma más extendida para hacer esto es especificar cada opción directamente en el archivo *.pyx* a través de un comentario.

*#helloworld.pyx*

*# distutils: language = c++*

*# distutils: libraries = spam eggs*

*# distutils: include\_dirs = /opt/food/include*

Como alternativa a los pasos anteriores, podemos usar el comando cythonize -a -i helloworld.pyx en la línea de comandos. En este caso el archivo *setup.py* no será necesario. Este comando crea también el archivo *.so* y el archivo *.s* en el directorio actual (con la opción *–i*). Con la opción *–a* generamos un archivo html con el código fuente.

Proceso de compilación: manual

Para compilar nuestro programa creado en el apartado anterior de forma manual, debemos de usar el comando *cython* sobre el archivo *.pyx*. Este comando es una herramienta de línea de comandos más simple que *cythonize* y nos permite invocar el traductor de código fuente, generando el archivo *.c*.

$ cython helloworld.pyx

Con el archivo *.c* generado, debemos de invocar nuestro compilador. Nosotros vamos a usar gcc. Para que funcione la compilación, es necesario que ejecutemos la siguiente orden completa:

$ gcc -shared -pthread -fPIC -fwrapv -O2 -Wall -fno-strict-aliasing -I/usr/include/python3.6

-o helloworld.so helloworld.c

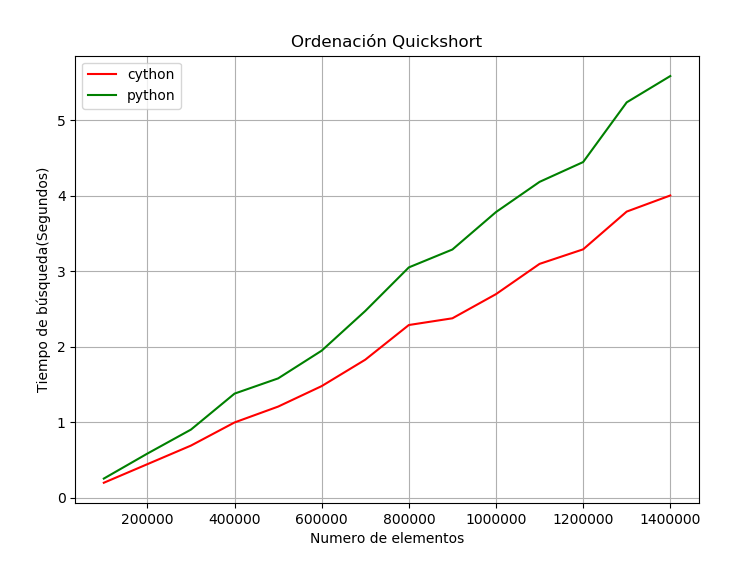
Debemos de especificar la biblioteca donde se encuentra python, cambiando la ruta del comando anterior si no tenemos la versión 3.6 de python.

Con este modo de compilación hay que tener en cuenta una dificultad añadida, ya que la compilación depende del sistema operativo y del compilador que estamos usando. Sin embargo, puede ser un método muy útil para experimentar con las diferentes optimizaciones del código que nos permite añadir el compilador.

# Gráficas comparativas

Para comparar la mejora en tiempo de un algoritmos con Cython y Python vamos a realizar dos ejemplos sencillos que ilustraran estas diferencias.

El primer ejemplo que realizaremos será el algoritmo de búsqueda Quicksort.

En este caso se ve como tenemos una mejora sustancial en el caso de la ejecución en Cython, llegando a mejorar el tiempo aproximadamente entre un 30% y 35% el tiempo que se consigue con Python.

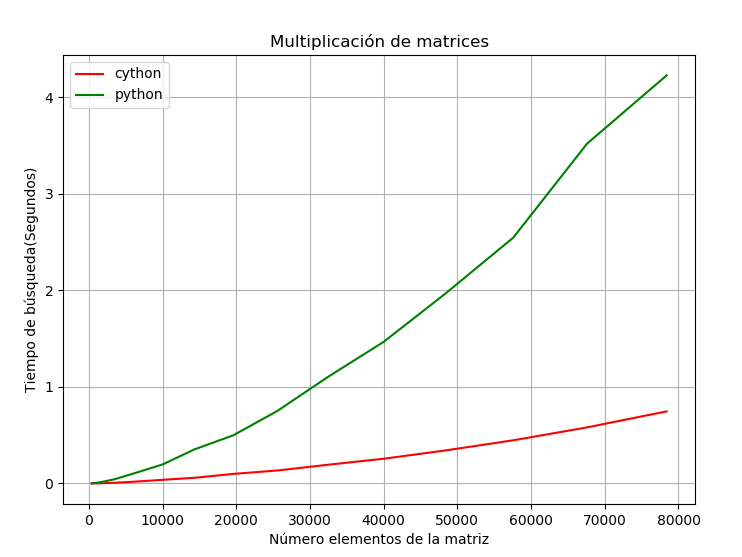
Para probar la mejora que se consigue en los bucles con Cython vamos a realizar otro ejemplo en el que se llevara acabo la multiplicación de una matriz cuadrada. En este ejemplo tendremos 3 bucles for que hará que mejoremos mucho en Cython.

for i in range(numElementos)

for j in range(numElementos)

for k in range(numElementos)

resultado[i][j]+=m1[i][k]\*matriz[k][j]



En este caso se ve como Cython es muy superior a Python, esto se da porque Python es muy lento trabajando con bucles al ser interpretado. En cambio a la hora de compilar esa parte del código y no utilizar Python se ve una mejor mucho mayor. Se ve como ya con una matriz con 8000 elementos en Python tardamos más de 4 segundos cuando en Cython no alcanzamos ni 1 segundo en ejecutarlo.