

# INTERCONEXIÓN PYTHON Y C/C++: PYPY



*Manuel Sierra Higuera*

*Jesús Medina Taboada*

*Jose Antonio Martín Melguizo*

*Najib Saadouli Arco*

10/12/2020

PROGRAMACIÓN TÉCNICA Y CIENTÍFICA

Profesor: Miguel García Silvente

## INTRODUCCIÓN

A diferencia de otros lenguajes de programación, Python enfatiza ampliamente la legibilidad del código (uso de indentación para definir bloques). Su sintaxis simple y expresiva permite a los desarrolladores agregar funcionalidad a sus aplicaciones sin escribir código largo y complejo. Incluso tienen la opción de elegir entre varias implementaciones de Python. La implementación predeterminada del lenguaje de programación Python es CPython.

A pesar de estar escrito en C, CPython se distribuye como una biblioteca estándar de Python. Compila el código fuente de Python en un código binario intermedio. Ese código de binario intermedio lo ejecuta una máquina virtual CPython. Muchos programadores optan por CPython debido a su compatibilidad con una amplia gama de paquetes, pero la compilación de CPython hace que Python sea más lento que otros lenguaje de programación compilados y requiere recursos adicionales de la máquina/servidor.

Por lo tanto muchos programadores optan por una implementación alternativa de Python para aumentar la velocidad de ejecución del código. Hay varias alternativas: Jython, IronPython, Cython, Pypy o CLPython. Un gran porcentaje de programadores prefiere **Pypy** ya que está escrito en Python y utiliza un intérprete que está escrito en un subconjunto del propio lenguaje: **RPython**.

### ¿Qué es Pypy?

Históricamente Pypy ha tenido 2 significados diferentes:

1. Cadena de herramientas de traducción RPython (o un framework) para implementar intérpretes y máquinas virtuales para lenguajes de programación, especialmente lenguajes dinámicos.
2. Implementación particular de Python producida con él.

Pypy aumenta drásticamente la velocidad de ejecución del código Python a través de la compilación **Just-In-Time** (JIT). Aprovecha los métodos de compilación JIT para mejorar la eficiencia y el rendimiento del sistema de interpretación.

Esta técnica aumenta la velocidad de ejecución de los programas compilando partes de

un programa a código máquina en tiempo de ejecución. Además admite una versión mejorada: Stackless Python (que se comentará más adelante, en el apartado de eficiencia), que ejecuta programas basados en subprocesos de manera más eficiente.

El compilador JIT además hace que PyPy ejecute programas Python cortos y largos mucho más rápido que implementaciones similares. Varios estudios incluso sugieren que Pypy es aproximadamente **4.2 veces más rápido que CPython**.

Cada nueva versión de PyPy viene con un rendimiento mejorado y ejecuta los programas de Python más rápido que su predecesor.

## Compatibilidad

PyPy está disponible para las versiones Python 2 y 3. Ofrece compatibilidad para toda la funcionalidad «core» de Python y la mayoría de los módulos estándar del lenguaje. Aunque, desafortunadamente, no es compatible con todas las librerías.

En cuanto a compatibilidad en las plataformas de ejecución, es posible usar PyPy en la mayoría de los sistemas operativos, incluidos Windows, Linux y Mac.

## INSTALACIÓN

El método más sencillo y seguro de instalación es descargar un ‘pre-built’ de pypy eligiendo tu sistema operativo (Windows, Linux, Mac), la arquitectura de tu computador (32/64-bits) y la versión de python para el que funcionará (3.7/3.6/2.7).

Desde <https://www.pypy.org/download.html> puedes hacerlo.












Nosotros explicaremos la **instalación** tanto en **windows** como en **ubuntu**.

## Instalación en Windows

PyPy v7.3.3

OS	PyPy3.7	PyPy3.6	PyPy2.7	Notes
Linux x86 64 bit	<a href="#">Download</a>	<a href="#">Download</a>	<a href="#">Download</a>	compatible with CentOS6 and later
Windows 32 bit	<a href="#">Download</a>	<a href="#">Download</a>	<a href="#">Download</a>	compatible with any windows, 32- or 64-bit you might need the VC runtime library installer <a href="#">vcredist.x86.exe</a>
MacOS	<a href="#">Download</a>	<a href="#">Download</a>	<a href="#">Download</a>	High Sierra >= 10.13, not for Sierra and below
Linux ARM64	<a href="#">Download</a>	<a href="#">Download</a>	<a href="#">Download</a>	compatible with CentOS6 and later

PyPy estará listo para usarse en cuanto descomprimamos el archivo zip/tar.

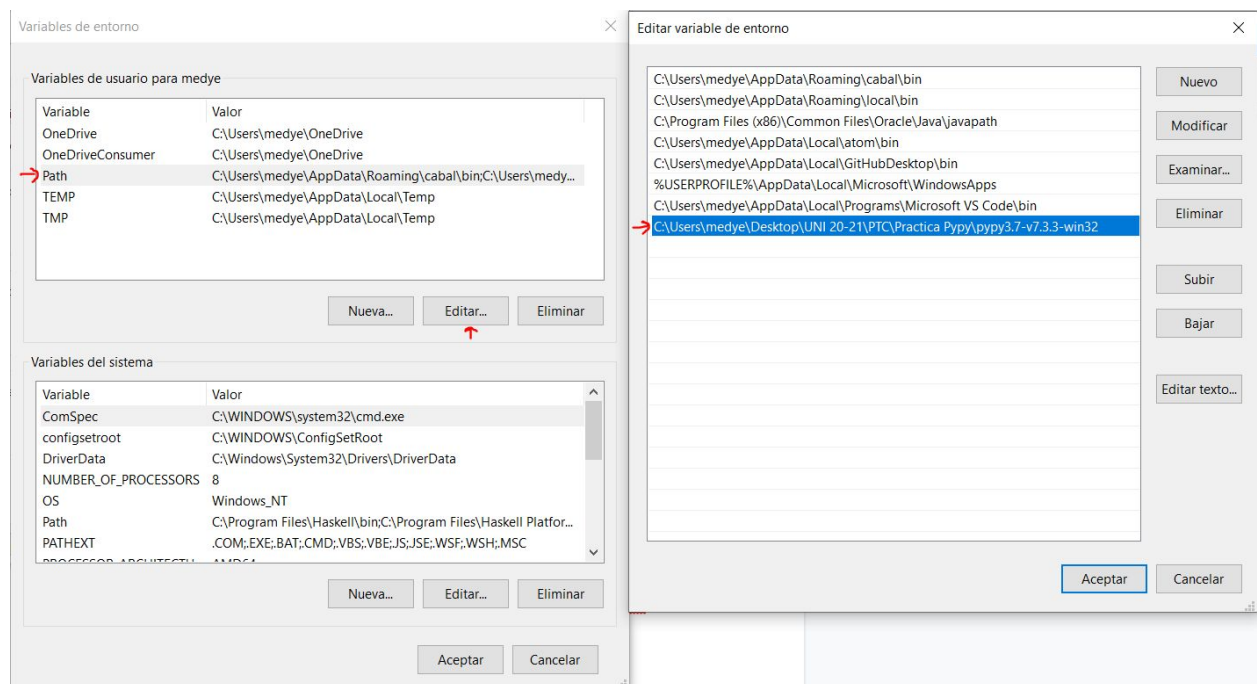
 include	21/11/2020 17:41	Carpeta de archivos	
 lib_pypy	03/12/2020 11:47	Carpeta de archivos	
 lib-python	21/11/2020 17:41	Carpeta de archivos	
 libs	21/11/2020 17:41	Carpeta de archivos	
 site-packages	21/11/2020 17:41	Carpeta de archivos	
 tcl	21/11/2020 17:41	Carpeta de archivos	
 libpypy3-c.dll	18/11/2020 13:45	Extensión de la ap...	35.857 KB
 LICENSE	18/11/2020 13:45	Archivo	14 KB
 pypy3.exe	18/11/2020 13:45	Aplicación	121 KB
 pypy3w.exe	18/11/2020 13:45	Aplicación	121 KB
 README.rst	18/11/2020 13:45	Archivo RST	2 KB

Si clicamos en 'pypy3.exe' se nos abrirá un terminal para trabajar con pypy.

Pero nosotros queremos usarlo para nuestros programas, por lo que necesitamos declarar una variable de entorno en Windows para poder trabajar con ella más tarde.

Nos dirigimos al apartado de **Panel de control**>**Editar las variables del entorno del sistema**

Clicamos **Variables de entorno**



Seleccionamos la variable 'Path' y la editamos. Añadimos al final el path donde hayamos descargado la distribución de pypy.

Reiniciamos el ordenador para que se guarde la variable de entorno.

Podemos verificar que la instalación es correcta desde el prompt ejecutando en la terminal:

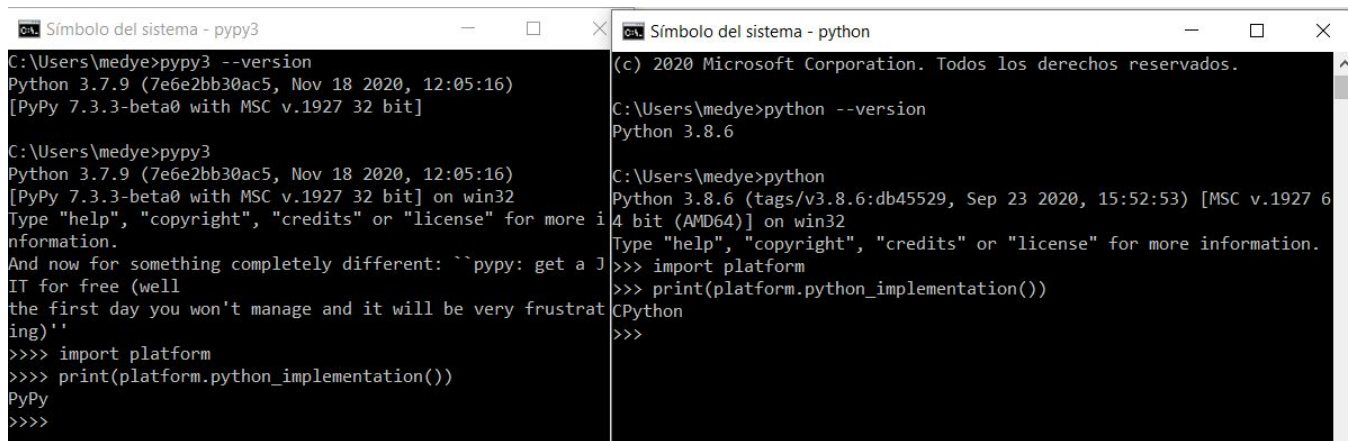
```
>pypy3 --version
```

Vemos como Pypy puede utilizar internamente una versión de python diferente a la que

tenemos instalada.

Vamos a aprovechar para mostrar cómo efectivamente python utiliza CPython por defecto en su implementación.

En la derecha tenemos Python usando como implementación CPython, y ejecutando Pypy(izquierda) la implementación de python cambia a Pypy.



The image shows two side-by-side terminal windows. The left window is titled 'Símbolo del sistema - pypy3' and shows the command 'C:\Users\medye>pypy3 --version' resulting in 'Python 3.7.9 (7e6e2bb30ac5, Nov 18 2020, 12:05:16) [PyPy 7.3.3-beta0 with MSC v.1927 32 bit]'. It then shows 'C:\Users\medye>pypy3' followed by the same version string and 'on win32'. A prompt 'Type "help", "copyright", "credits" or "license" for more information.' is shown. Below that, a message says 'And now for something completely different: ``pypy: get a J IT for free (well the first day you won't manage and it will be very frustrating)''. Then, the command '>>> import platform' is entered, followed by '>>> print(platform.python\_implementation())' which outputs 'PyPy'. The right window is titled 'Símbolo del sistema - python' and shows '(c) 2020 Microsoft Corporation. Todos los derechos reservados.' followed by 'C:\Users\medye>python --version' resulting in 'Python 3.8.6'. Then 'C:\Users\medye>python' is entered, showing 'Python 3.8.6 (tags/v3.8.6:db45529, Sep 23 2020, 15:52:53) [MSC v.1927 64 bit (AMD64)] on win32'. A prompt 'Type "help", "copyright", "credits" or "license" for more information.' is shown. Then, the command '>>> import platform' is entered, followed by '>>> print(platform.python\_implementation())' which outputs 'CPython'.

```
C:\Users\medye>pypy3 --version
Python 3.7.9 (7e6e2bb30ac5, Nov 18 2020, 12:05:16)
[PyPy 7.3.3-beta0 with MSC v.1927 32 bit]

C:\Users\medye>pypy3
Python 3.7.9 (7e6e2bb30ac5, Nov 18 2020, 12:05:16)
[PyPy 7.3.3-beta0 with MSC v.1927 32 bit] on win32
Type "help", "copyright", "credits" or "license" for more i
nformation.
And now for something completely different: ``pypy: get a J
IT for free (well
the first day you won't manage and it will be very frustrat
ing)''
>>> import platform
>>> print(platform.python_implementation())
PyPy
>>>

(c) 2020 Microsoft Corporation. Todos los derechos reservados.

C:\Users\medye>python --version
Python 3.8.6

C:\Users\medye>python
Python 3.8.6 (tags/v3.8.6:db45529, Sep 23 2020, 15:52:53) [MSC v.1927 6
4 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import platform
>>> print(platform.python_implementation())
CPython
>>>
```

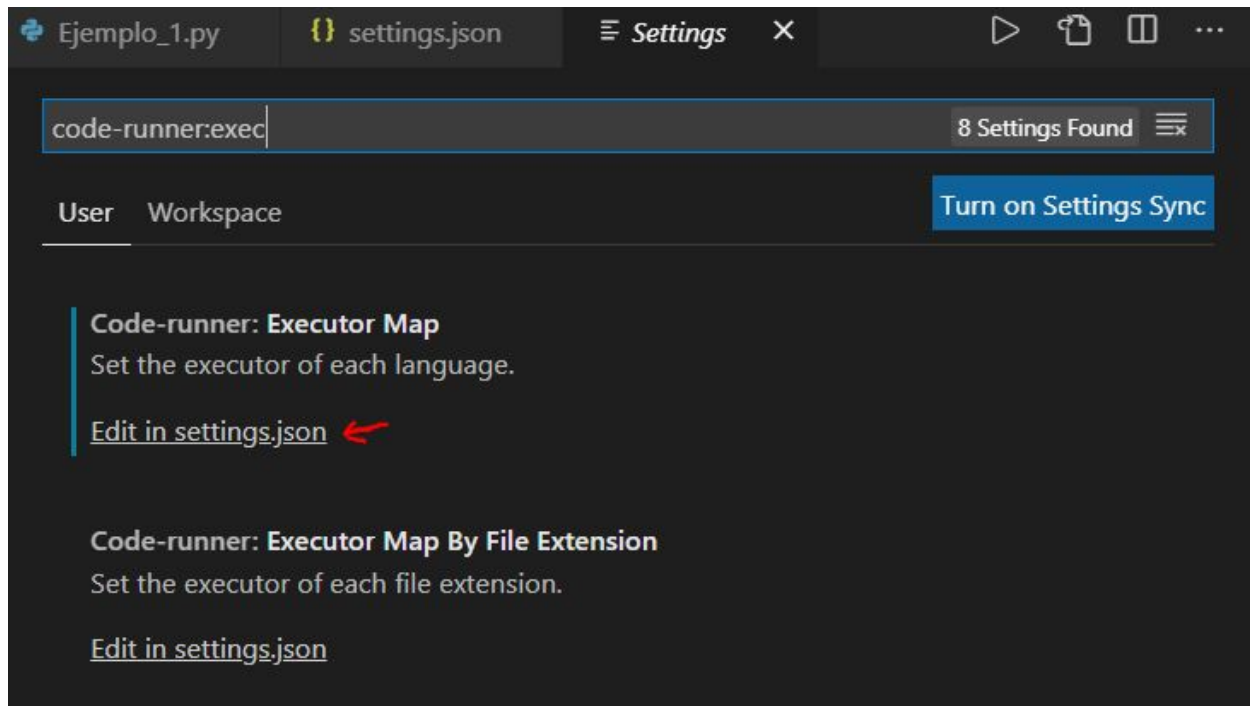
Nosotros usaremos **VSCode** para trabajar con algunos ejemplos:

Necesitaremos tener instalada la **extensión Code Runner**.

Para **cambiar entre el intérprete** por defecto CPython y el que queremos usar **Pypy**, solo tenemos que **modificar la variable 'python'** en las preferencias de CodeRunner.

File > Preferences > Settings

Y buscamos: code-runner:exec y clicamos en 'edit in setting.json'



En este archivo será donde modificaremos la variable cada vez que queramos ejecutar un programa python con diferentes intérpretes:

```
C: > Users > medye > AppData > Roaming > Code > User > {} settings.json > {} code-  
1 {  
2   "window.zoomLevel": 0,  
3   "liveServer.settings.donotShowInfoMsg": true,  
4   "python.pythonPath": "C:\\Users\\medye\\Anaconda3\\envs\\  
5   "code-runner.executorMap": {  
6  
7     "javascript": "node",  
8     "java": "cd $dir && javac $fileName && java $fileNan  
9     "c": "cd $dir && gcc $fileName -o $fileNameWithoutE  
10    "cpp": "cd $dir && g++ $fileName -o $fileNameWithout  
11    "objective-c": "cd $dir && gcc -framework Cocoa $fil  
12    "php": "php",  
13    "python": "pypy3 -u",  
14    "perl": "perl",  
15    "perl6": "perl6",  
16    "ruby": "ruby",  
17    "go": "go run",  
18    "lua": "lua",
```

‘Pypy3 -u’ para usar Pypy y ‘python -u’ para usar Cpython.



## Instalación en Ubuntu

La instalación de Ubuntu es bastante sencilla. Nos basta con abrir una terminal y escribimos las siguientes órdenes:

```
$> sudo add-apt-repository ppa:pypy/ppa
```

```
$> sudo apt update
```

```
$> sudo apt install pypy3
```

Estas líneas nos permitirán incluir el repositorio desde donde descargaremos el intérprete, actualizar la lista de paquetes de nuestros repositorios e instalar Pypy finalmente.

Pypy también tiene a **pip** como **administrador de paquetes** y su uso es el mismo que cuando lo utilizamos Python. No podemos usarlo directamente, tenemos que tener un script adicional para poder utilizarlo. Lo descargamos ejecutando la siguiente línea:

```
$> wget https://bootstrap.pypa.io/get-pip.py
```

Por ejemplo, si quisiéramos instalar el módulo de **numpy**, nos basta con ejecutar la siguiente línea de comandos:

```
$> pypy3 -m pip install numpy
```

Una vez hemos descargado el intérprete de Pypy, podemos **comprobar la versión** de éste escribiendo desde la línea de comandos:

```
(base) jose@linux:~/Escritorio/pypy$ pypy3 --version
Python 3.6.12 (7.3.3+dfsg-1~ppa1~ubuntu20.04, Nov 21 2020, 20:44:03)
[PyPy 7.3.3 with GCC 9.3.0]
(base) jose@linux:~/Escritorio/pypy$
```



## USO

### Por qué utilizar Pypy

Usamos Pypy cuando queremos ejecutar nuestro código Python de manera rápida, sabiendo que esta implementación saca el mayor rendimiento cuando ejecuta código en Python puro, y funciona peor cuando ejecuta librerías implementadas directamente en C, ya que CPython realiza mejor este apartado.

### Cómo usar Pypy

Para usar esta implementación podemos realizar lo que hemos indicado en el apartado de instalación para VisualStudio Code, o simplemente en la terminal, una vez instalado Pypy, utilizar el comando:

```
pypy3 nombre_archivo
```

```
pypy nombre_archivo
```

Teniendo en cuenta que el primero es para Python 3 y el segundo para Python 2.

## EJEMPLOS

### 1. Ejemplo básico con operación sencilla suma 2+3 un millón de veces

```
1  import platform
2  import timeit
3
4  print(platform.python_implementation())
5  print(timeit.timeit('2+3', number=1000000))
```

Ejecutamos con Python normal, es decir, con CPython como ya sabemos, y cambiando el intérprete a Pypy:

```
[Running] python -u "c:\Users\
CPython
0.0080352
```

```
[Running] pypy3 -u "c:\Users\  
PyPy  
0.0015294
```

Como vemos Pypy es mucho más eficiente y mejora considerablemente el tiempo.

Obteniendo una mejora de tiempo de 5.25 veces más rápido para este ejemplo.

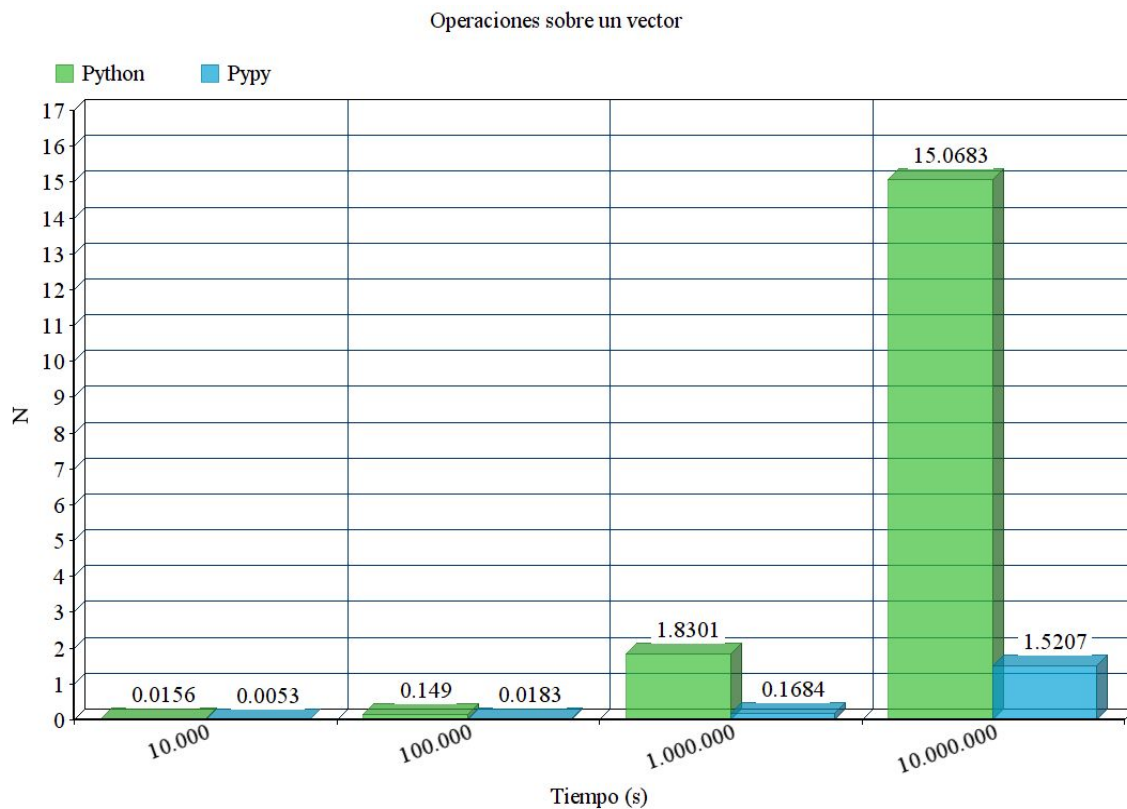
Si incrementamos el número de repeticiones de la operación de 1 millón a 10 millones, encontramos que Pypy trabaja 14.39 veces más rápido.

## 2. Operaciones sobre un vector

A continuación, sobre un vector unidimensional de tamaño 10, vamos a aplicar una serie de operaciones (suma, multiplicación, división) sobre cada uno de los elementos de éste, para un número de iteraciones establecido. De forma que se fomentará el uso de operaciones aritméticas sobre números reales.

```
# Operaciones sobre un vector  
def test1(num_iteraciones):  
    a = [1,2,3,4,5,6,7,8,9,10]  
  
    for _ in range(num_iteraciones):  
        for idx, element in enumerate(a):  
            a[idx] = (element * 0.9) / (element + element/2)  
    return a
```

Probaremos para distinto número de iteraciones: 10.000, 100.000, 1 millón y 10 millones. Si graficamos los resultados obtenidos tanto para el intérprete de python como para pypy, obtenemos la siguiente gráfica de barras:



Podemos observar que para un número de iteraciones algo inferior como es 10.000, la velocidad de pypy es 3 veces más rápido aprox.

A partir de 100.000 iteraciones se nota una clara superioridad (en torno a 10x - 15x superior) de pypy sobre python.

### 3. Operaciones con matrices

A continuación, vamos a mostrar la eficiencia de ambos intérpretes sobre operaciones con matrices cuadradas de dimensión  $N$ , como son **la suma y producto de matrices**.

Para ello vamos a utilizar una **función** que nos ayudará a **generar** la matrices (antes de hacer las operaciones con éstas), de forma que los tiempos medidos serán sobre las propia ejecución de los algoritmos (sin tener en cuenta el tiempo de generación de éstas).

La inicializamos con número enteros comprendidos entre 0 y 100.

```
# Generar matrices cuadradas (m x m)
def crear_matrices(dimension):

    matriz_1 = []
    matriz_2 = []

    for i in range(dimension):
        matriz_1.append([])
        matriz_2.append([])
        for _ in range(dimension):
            matriz_1[i].append(random.randint(0,100))
            matriz_2[i].append(random.randint(0,100))

    return matriz_1, matriz_2
```

### 3.1 Suma de matrices

La función para el segundo test que implementa la suma de 2 matrices cuadradas con la misma dimensionalidad es la siguiente:

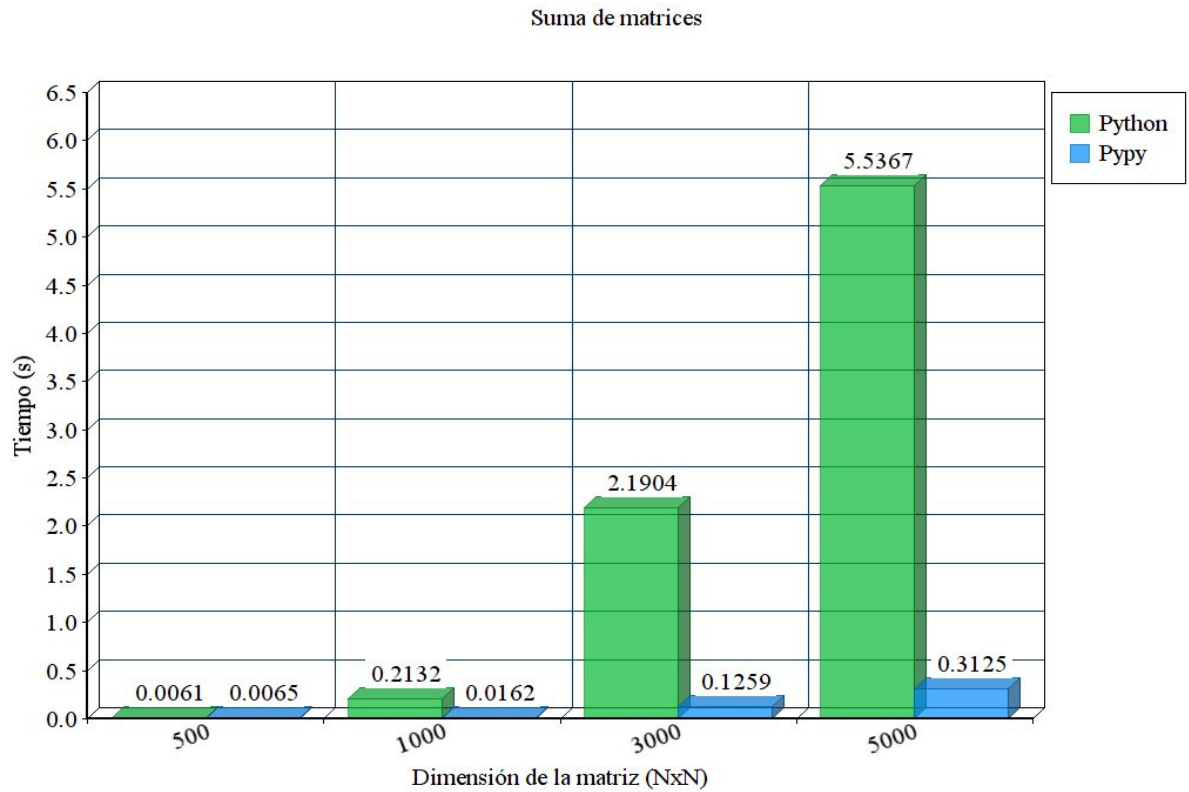
```
# Suma de matrices
def test2(A, B):

    filas = len(A)
    columnas = len(A[0])
    suma = []

    for i in range(filas):
        suma.append([0] * columnas)
        for j in range(columnas):
            suma[i][j] += A[i][j] + B[i][j]

    return suma
```

Si observamos de forma gráfica los resultados obtenidos para este test, obtenemos el siguiente gráfico de barras:



Los resultados obtenidos para una dimensión pequeña como 500 son prácticamente similares, pero volvemos a observar una mejora ascendente de hasta 17.71x veces más rápido para un tamaño de problema 10 veces mayor (5000).

### 3.2 Producto de matrices

La función para el segundo test que implementa el producto de 2 matrices cuadradas con la misma dimensionalidad es la siguiente:

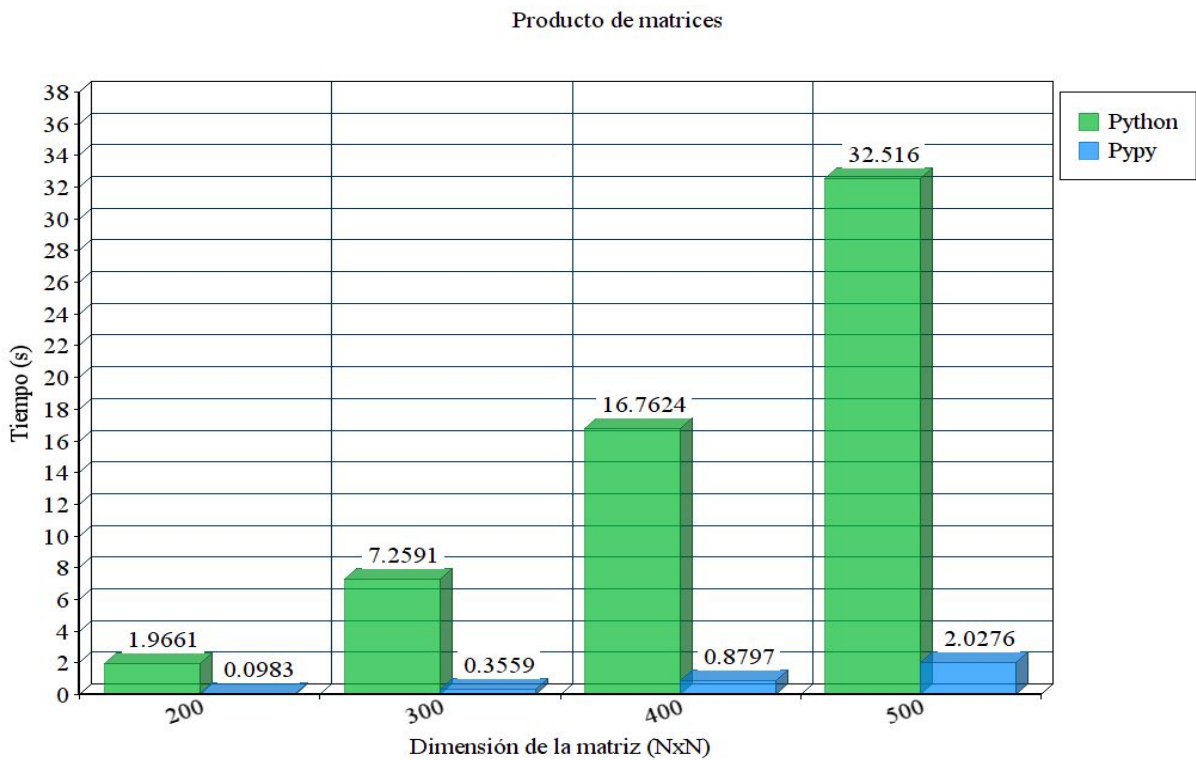
```
# Producto de matrices
def test3(A, B):

    filas = len(A)
    columnas = len(A[0])
    producto = []

    for i in range(filas):
        print(i)
        producto.append([0] * columnas)
        for j in range(columnas):
            for k in range(filas):
                producto[i][j] += A[i][k] * B[k][j]

    return producto
```

Si observamos de forma gráfica los resultados obtenidos para este test, obtenemos



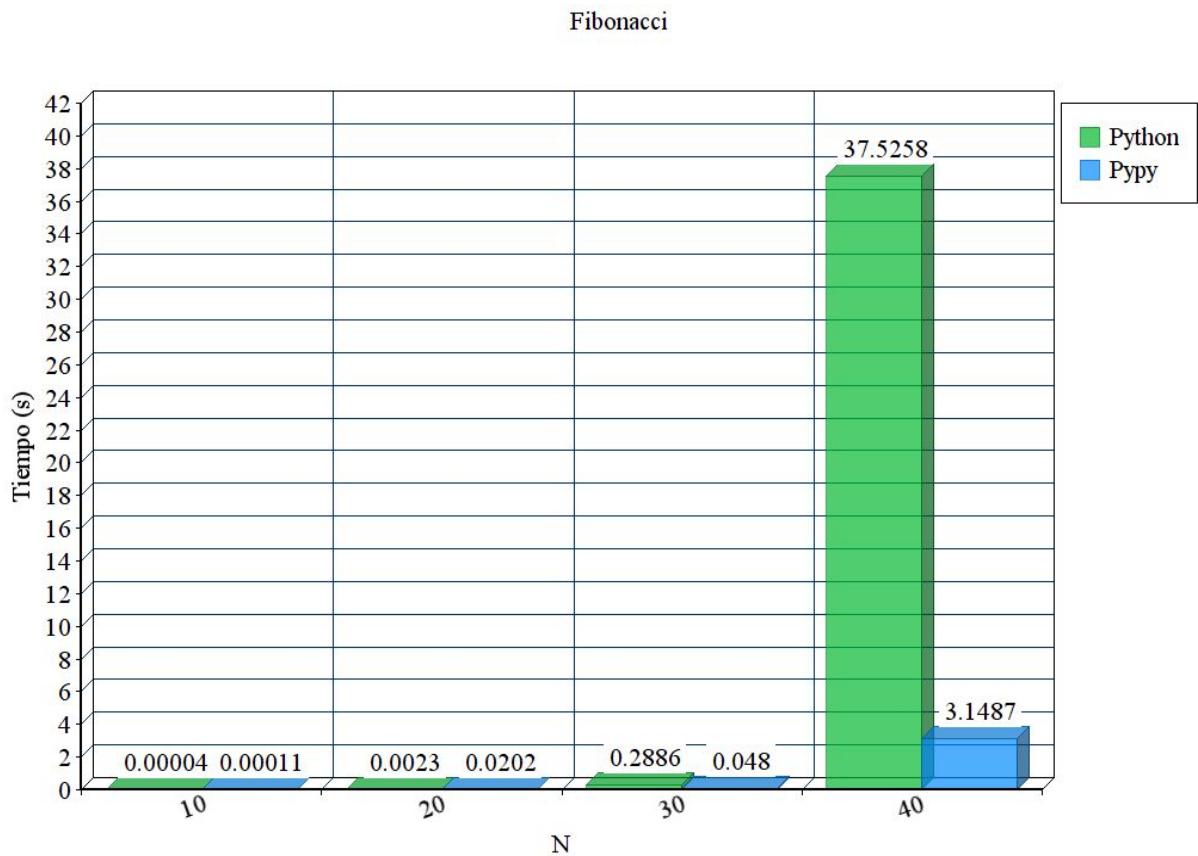
#### 4. Fibonacci

A continuación, vamos a ver qué tal se comportan los intérpretes cuando ejecutamos un algoritmo que es propiamente recursivo.

Definición de la función para el test:

```
# Fibonacci
def test4(N):
    if N < 2:
        return N
    else:
        return test4(N-1) + test4(N-2)
```

Si observamos de forma gráfica los resultados obtenidos para este test, obtenemos el siguiente gráfico de barras:



En este ejemplo, podemos observar que para el **cálculo del Fibonacci de número que son pequeños**, como 10 o 20, obtenemos **mejores resultados en Python** (es 2.75 veces más rápido y 8.78 veces más rápido respectivamente).



Sin embargo, **a partir de 30**, la explosión combinatoria que originan las dos llamadas recursivas cada llamada a la función, originan una **clara desventaja al intérprete de python** (como hemos visto, Pypy puede utilizar más conocimiento que tiene en tiempo de ejecución y podrá simplificar los cálculos, frente a Python que se limita a seguir instrucciones originadas por el intérprete).

Concretamente para el cálculo de Fibonacci de 30 y 40 obtenemos una mejora de Pypy respecto a Python de 6x y 11.92x veces más rápido respectivamente.

## 5. Factorial

Otra interesante prueba puede ser la del cálculo del factorial de un número.

En este caso hay 2 formas de implementar el algoritmo, empleando llamadas recursivas (de forma similar al ejemplo anterior) o haciéndolo de forma iterativa, y acumulando las multiplicaciones sobre una variable.

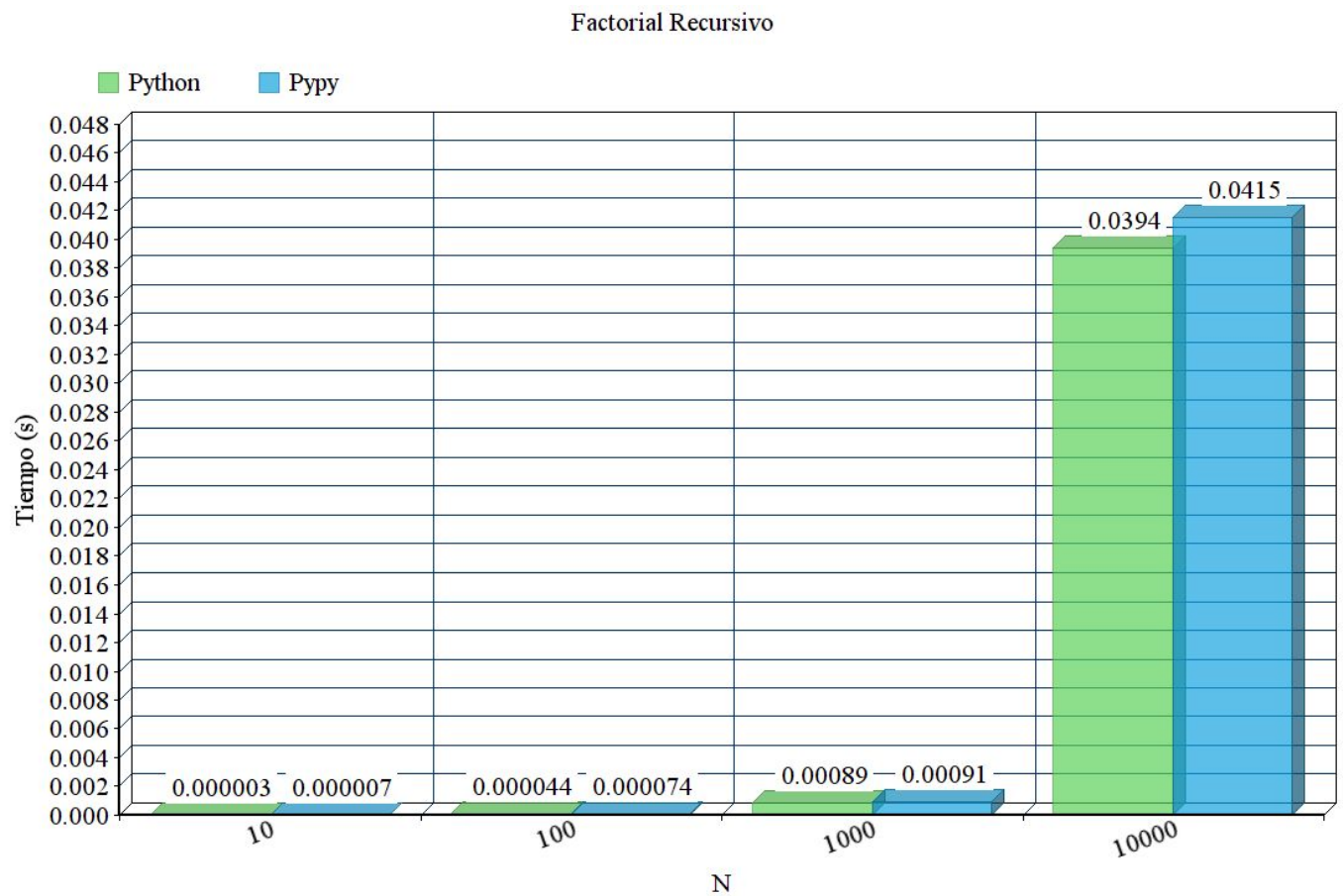
A continuación presentamos ambas opciones:

### 5.1 Factorial usando recursividad

```
# Factorial recursivo
def test5(N):
    if N == 0:
        return 1
    else:
        return N * test5(N-1)
```

Es la implementación más común para la implementación del factorial.

Los resultados obtenidos para esta implementación y su representación gráfica es la siguiente:



Llama la atención que hemos obtenidos **resultados similares** para ambos, en este caso los **tiempos de ejecución** han sido **algo inferiores para el caso de Python**.

Podríamos deducir que cuando tenemos un **número inferior de llamadas recursivas** en nuestra función, ambos intérpretes obtienen unos tiempos de ejecución similares. Cuando **aumenta el número de llamadas recursivas** (como por ejemplo en Fibonacci, que teníamos el doble que en factorial) **Pypy es más eficiente**.

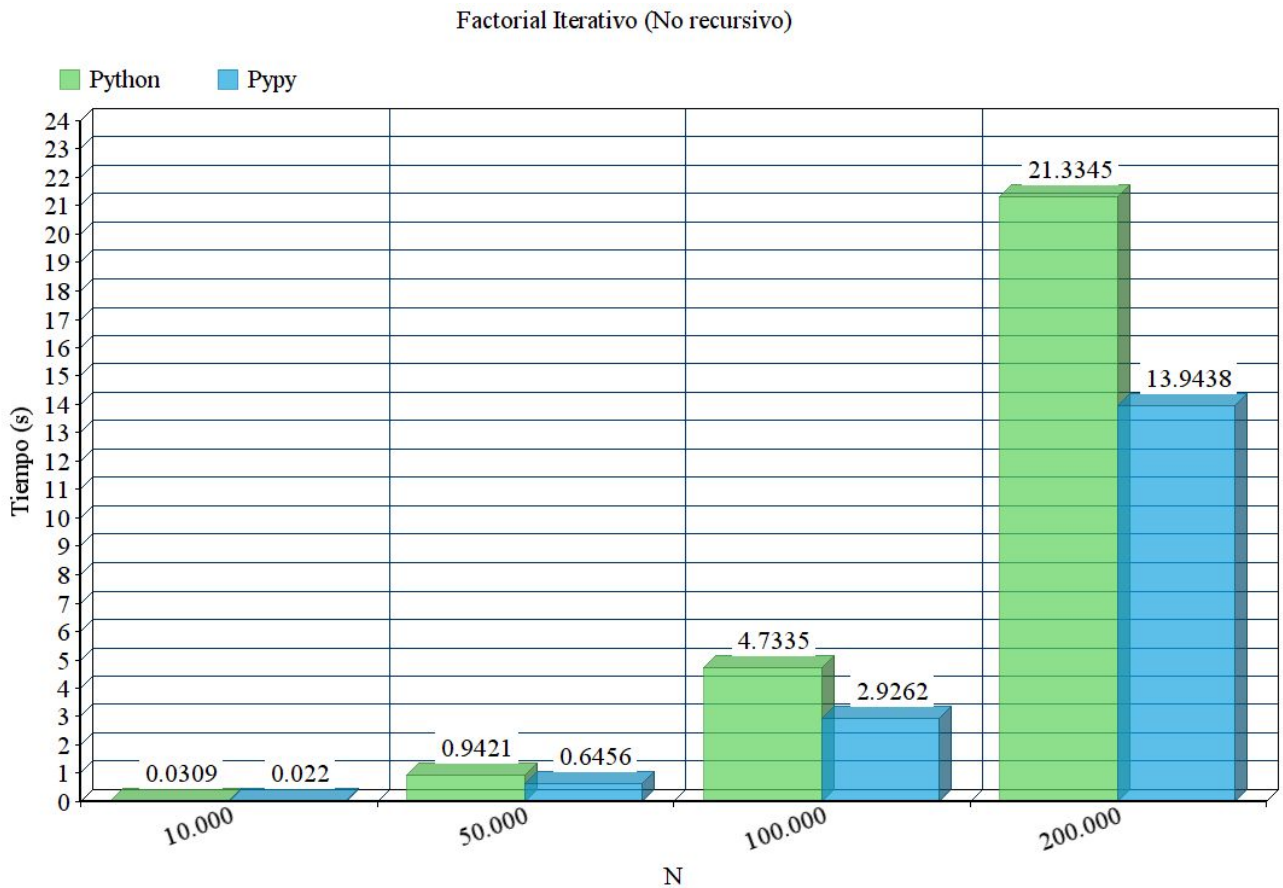
## 5.2 Factorial iterativo (sin recursividad)

```
# Factorial no recursivo
def test5Bis(N):
    resultado = 1
    for i in range(1,N):
        resultado = resultado * i

    return resultado
```

En este caso, en lugar de llamadas recursivas, vamos acumulando el valor sobre una variable.

Los resultados obtenidos para ambas implementaciones y su representación gráfica es la siguiente:



De nuevo se observa una **mejoría**, de Pypy sobre Python **en todos los casos**.

En la primera y segunda ejecución, los tiempos de ejecución son más similares. Para el cálculo del factorial de 100.000 y 200.000 obtenemos una mejora de 1.61x y 1.53x aproximadamente.

Si comparamos los resultados respecto al resto de ejemplos, estos han sido los peores resultados, obteniendo así una **mayor ganancia de velocidad** a la hora de ejecutar **operaciones que involucran estructuras de datos más grandes** (pypy optimiza con más facilidad).

## 6. Algoritmos de ordenación

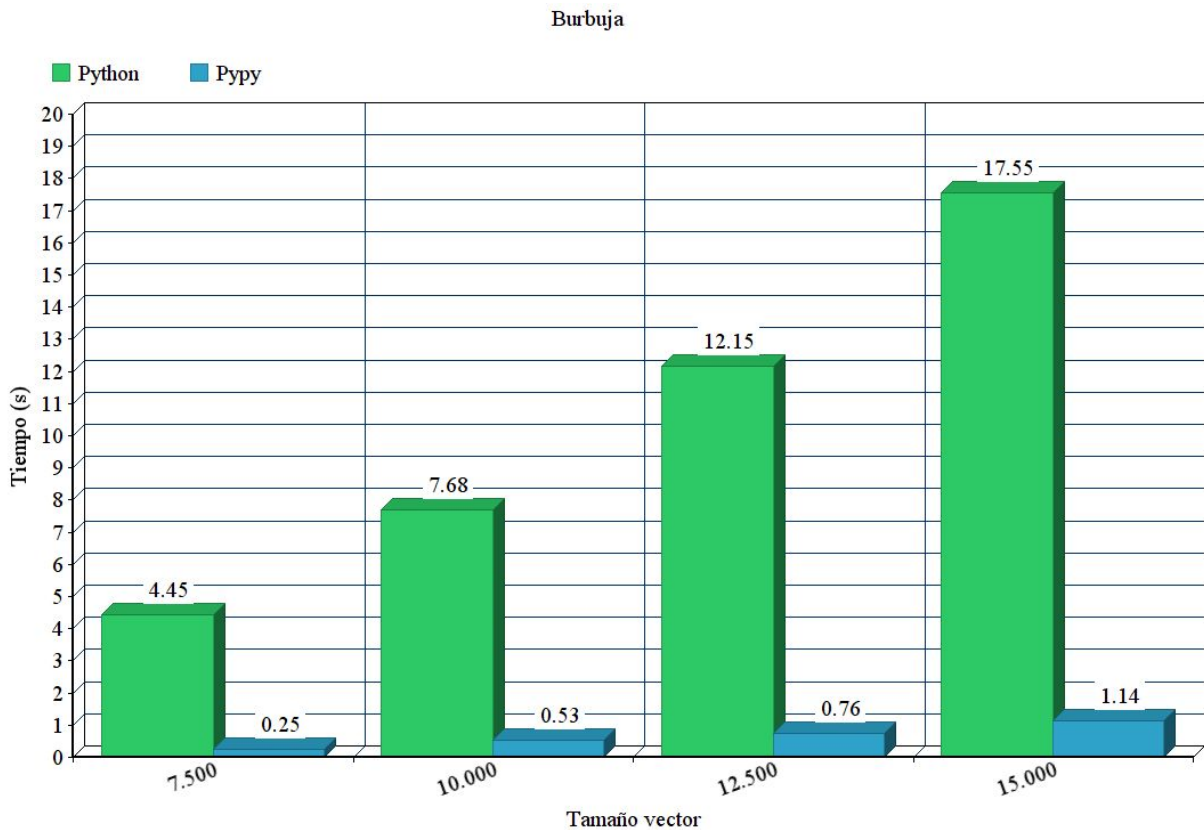
Ahora vamos a probar la eficiencia de Pypy con distintos algoritmos de ordenación, en concreto con el algoritmo burbuja, el de inserción y mergesort.

### 6.1 Algoritmo burbuja

Para esta prueba he utilizado vectores con valores aleatorios, de 4 tamaños diferentes: 7500, 10000, 12500 y 15000.

```
def burbuja (nueva_lista):  
    n = len(nueva_lista)  
    for i in range(1,n):  
        for j in range(n-1):  
            if (nueva_lista[j]>nueva_lista[j+1]):  
                nueva_lista[j],nueva_lista[j+1] = nueva_lista[j+1], nueva_lista[j]
```

En esta gráfica podemos observar los resultados medios de ejecutar este algoritmo sobre los distintos vectores.



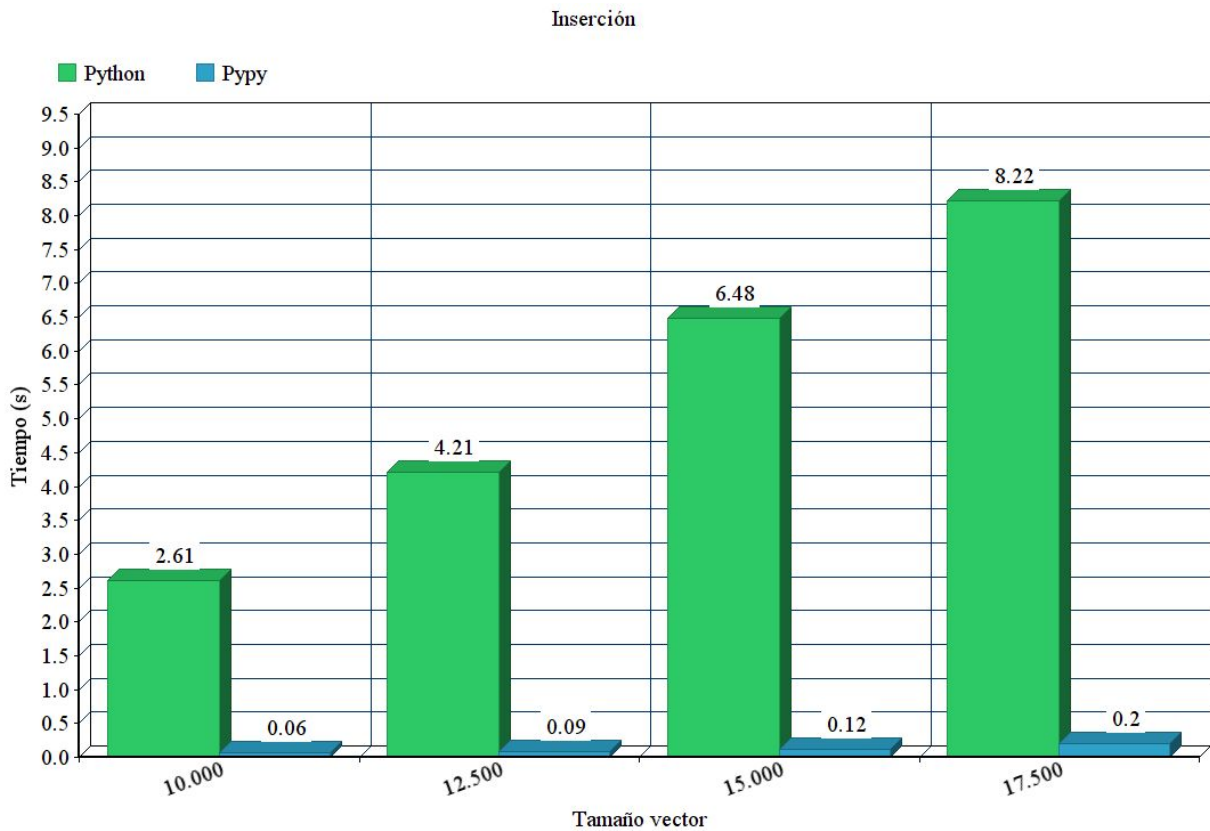
El algoritmo burbuja tiene una complejidad  $O(n^2)$ , y podemos ver cómo, para valores de vector medianamente grandes, ya tarda muchos segundos con la implementación que trae por defecto Python. Estos tiempos son considerablemente más pequeños con Pypy. ¿Por qué sucede esto? Es simple, la compilación JIT hace que el código que más utilizamos en el programa se pase a código binario y por tanto sea más rápido de ejecutar.

## 6.2 Algoritmo de inserción

Como el algoritmo de inserción tiene el mismo orden de complejidad que el de burbuja pero únicamente en el peor de los casos, hemos aumentado el tamaño de los vectores para hacer las pruebas con este algoritmo. Los tamaños serán: 10000, 12500, 15000 y 17500.

```
def insercion (nueva_lista):
    n = len(nueva_lista)
    for i in range(1,n):
        val = nueva_lista[i]
        j = i
        while (j > 0 and nueva_lista[j-1]>val):
            nueva_lista[j] = nueva_lista[j-1]
            j -= 1
        nueva_lista[j] = val
```

Los tiempos obtenidos con Python y Pypy son los siguientes:



Nuevamente, Pypy funciona mucho mejor que CPython. Esta es una de las pruebas en las que más se nota la mejora usando Pypy.

### 6.3 Algoritmo Mergesort

El último algoritmo de ordenamiento que veremos será Mergesort. Este es un algoritmo más complejo y optimizado, que funciona bien y tiene un orden de complejidad  $O(n \log n)$ , mucho más rápido que los otros dos. Es por ello que hemos usado tamaños mucho mayores para medir los tiempos, pues si lo hacíamos con los mismos no se vería el potencial de Pypy. Tamaños elegidos: 500.000, 1.000.000, 1.500.000 y 2.000.000.

```
def mergesort (nueva_lista):
    if (len(nueva_lista)<=1):
        return nueva_lista
    medio = int(len(nueva_lista)/2)
    izquierda = nueva_lista[:medio]
    derecha = nueva_lista[medio:]

    izquierda = mergesort(izquierda)
    derecha = mergesort(derecha)

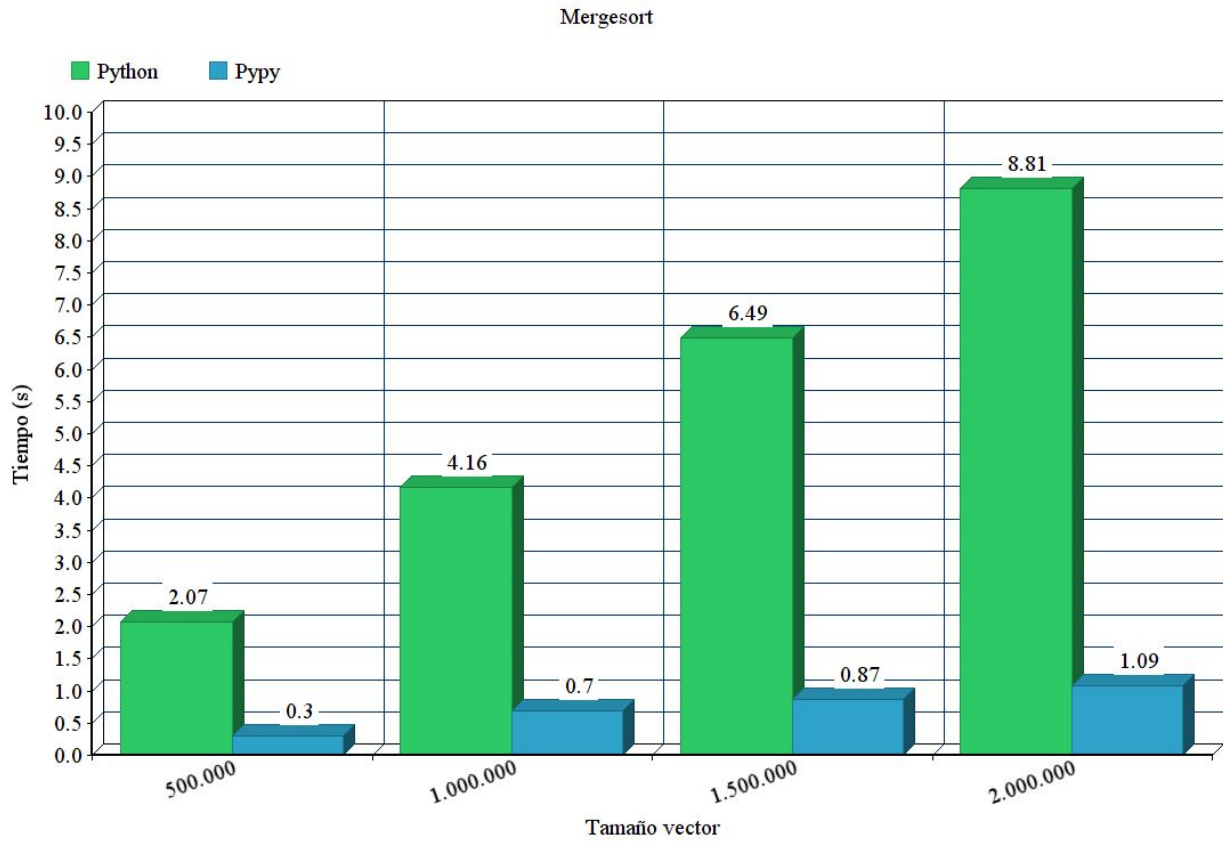
    return merge (izquierda,derecha)

def merge (listaA, listaB):
    lista_nueva = []
    a = 0
    b = 0

    while (a < len(listaA) and b < len(listaB)):
        if (listaA[a]<listaB[b]):
            lista_nueva.append(listaA[a])
            a+=1
        else:
            lista_nueva.append(listaB[b])
            b+=1
    while (a < len(listaA)):
        lista_nueva.append(listaA[a])
        a+=1
    while (b < len(listaB)):
        lista_nueva.append(listaB[b])
        b+=1
    return lista_nueva
```



Como podemos ver, Mergesort utiliza recursividad y muchas llamadas a la función merge. Los resultados obtenidos son los siguientes:



Aquí seguimos viendo que PyPy sigue funcionando mucho mejor que Python, pero la mejora es inferior a los anteriores algoritmos.

## EFICIENCIA

### Mejora aplicaciones con Pypy

La mayoría de las aplicaciones se ejecuta bien sobre PyPy, aunque las mejoras serán percibidas cuando se trata de programas que requieren un tiempo considerable de procesamiento, donde una fracción relevante de ese tiempo se consume en ejecutar código Python real.

En concreto existen dos requisitos para que se obtenga mayor rendimiento:

1. Que **las aplicaciones o procesos sean medianamente complejos**, ya que si se tienen que ejecutar unas pocas decenas de líneas de código el compilador JIT no tendrá el tiempo suficiente para demostrar sus ventajas.

2. Que **la principal parte del código a ejecutar sea realmente código Python** y no librerías run-time, como por ejemplo funciones C, en las PyPy no puede optimizar su ejecución.

PyPy saca su mayor potencial al agregar JIT, por lo que, cuanto más código se tenga que repetir en una ejecución, mayor será la ganancia en eficiencia respecto a CPython.

### Consumo de memoria reducido

Además de disminuir el tiempo de ejecución del código, PyPy también hace que los programas de Python consuman menos memoria que en CPython. Sin embargo, la disminución del uso de memoria puede variar de un programa a otro.

### PyPy en aplicaciones web

Como hemos dicho, PyPy es capaz de obtener mayor rendimiento cuando la complejidad del procesamiento es alta. Pero esta no suele ser una constante en los sitios web, donde la mayoría de las operaciones son de entrada/salida.

Sin embargo, existe toda una comunidad de usuarios de PyPy con proyectos en

funcionamiento en frameworks tan sofisticados como Django. Las mejoras que anuncian son relevantes, obteniendo entre 3 y 4 veces más rendimiento para solicitudes complejas. Pero yendo al detalle, lo cierto es que otras implementaciones como CPython consiguen responder de manera más rápida a una renderización de una página con pocos datos. En estos casos PyPy no podrá ofrecer mucha mejora, pero sí es capaz de optimizar el tiempo de CPU dedicado para las operaciones de su ORM, lo que se notará si es necesario realizar una mediana o gran cantidad de consultas por solicitud.

Otra de las desventajas que podemos encontrar con PyPy es que no se puede usar bien con el servidor Apache. Sin embargo, esto no es un gran problema, ya que funciona de manera estupenda usando un servidor HTTP basado en Python y Nginx como proxy inverso.

### **Compatibilidad con Python Stackless**

PyPy admite además una versión mejorada del lenguaje de programación Python: Stackless Python. Stackless Python se ejecuta en programas basados en subprocesos de manera más eficiente que Python. Incluso ayuda a los programadores a evitar algunos de los problemas de complejidad y rendimiento relacionados con los subprocesos convencionales. Mientras usan PyPy, los programadores pueden incluso acelerar las aplicaciones escribiendo código en estilo concurrente.

## CONCLUSIONES

En general, PyPy es mucho más rápido que otras implementaciones de Python. Como destacan varios estudios, es aproximadamente 4.2 veces más rápido que CPython. Además, cada nueva versión de PyPy viene con un rendimiento mejorado. Pero el tiempo de ejecución puede variar de un programa a otro.

Según estudios específicos, PyPy ejecuta código Python puro mucho más rápido que los programas que llaman a funciones codificadas en C. Por lo tanto, los desarrolladores de Python deben tener en cuenta los pros y los contras de PyPy para optimizar la velocidad de ejecución del código Python.

## REFERENCIAS

1. <https://www.arsys.es/blog/pypy/>
2. <https://www.pypy.org/download.html>
3. <https://doc.pypy.org/en/latest/install.html>
4. [https://www.youtube.com/watch?v=-wSifiOGnNM&ab\\_channel=RishishDixit](https://www.youtube.com/watch?v=-wSifiOGnNM&ab_channel=RishishDixit)
5. <https://www.toptal.com/python/por-que-hay-tantos-pythons>