



Universidad Agraria de La Habana "Fructuoso Rodríguez"

**Monografía: Creación de utilidad en
Python para ayuda en la selección de
código de mejor tiempo de ejecución.**

Ing. Pedro Pablo Delgado Martell
ppmartell@unah.edu.cu

Mayabeque, 2020

Índice

1	Introducción	1
2	Desarrollo	2
2.1	¿Cómo funciona Python?	2
2.1.1	Lenguaje compilado y lenguaje interpretado	2
2.1.2	Python, interpretado	4
2.1.3	Lentitud debido al intérprete	4
2.2	Lo más rápido posible	6
2.2.1	VSCodium, un IDE completamente <i>open source</i>	6
2.2.2	Enfoque teórico de la utilidad	7
2.2.3	Describiendo la utilidad	10
2.2.4	Uso de la librería <i>Matplotlib</i> para visualizar datos	13
2.3	Aplicando la utilidad a un caso de prueba	15
2.3.1	Creando el entorno virtual	16
2.3.2	Descargando el repositorio con <i>git</i>	18
2.3.3	Configurando el código de la utilidad antes de ejecutarla	19
2.3.4	Resultados obtenidos del caso de prueba	21
2.3.5	Afinando los resultados con virtualización	22
2.3.6	Mayor consistencia con diferente orden de ejecución	25
3	Conclusiones	28
4	Bibliografía	29
5	Anexos	A
5.1	Instalar VSCodium en Ubuntu 20.04.1 LTS	A
5.2	Configurar VSCodium para programar en Python	A
5.3	Código de la utilidad	B

1. Introducción

Python es un lenguaje de programación dinámico y funcional que a día de hoy goza de una gran popularidad entre desarrolladores. Este lenguaje de programación, de Guido van Rossum, puede ser usado para la creación de páginas web, aplicaciones para móviles, scripts para *devops*, soluciones informáticas en el campo de la Inteligencia Artificial, entre otras cosas. Aunque muchos aseguran que la curva de aprendizaje de este lenguaje es realmente baja, otros afirman que su gran desventaja es su lentitud. Es un tema¹ que se trata regularmente y existen varios desarrolladores trabajando por mejorar el *performance* de Python constantemente.

La presente monografía tiene como objetivo principal presentar al desarrollador una solución mediante la cual se pueda elegir una porción de código por encima de otra, teniendo en cuenta la rapidez de ejecución del mismo. Aunque existe la forma de hallar la complejidad algorítmica a una porción de código de forma teórica, en este caso, la utilidad que se propone permitirá observar el tiempo que demora la ejecución de varias porciones de código para seleccionar la que más rápido lleva a cabo la tarea. Por supuesto, ambas porciones de código tendrían como objetivo resolver un mismo problema, lo que de diferente forma. Python es un lenguaje de programación que pone en manos del desarrollador muchos recursos para lograr un mismo objetivo.

En la presente monografía se presentarán al lector una serie de herramientas necesarias para llevar a cabo la idea principal. Por ejemplo, resulta muy necesario utilizar la herramienta Oracle VM VirtualBox² a la hora de ejecutar las porciones de código a comparar, gracias a que los recursos del ordenador son «enjaulados» para cada máquina virtual. En caso de no ser así, el tiempo de ejecución aproximado se vería muy afectado por otros procesos del sistema operativo anfitrión que se encuentren consumiendo recursos sustanciales del ordenador. Al final del documento se presentarán al lector los métodos de instalación de las herramientas utilizadas en forma de anexos. Esto es en caso que el lector desee reproducir la idea expuesta en la monografía.

¹<https://www.welcometothejungle.com/en/articles/btc-performance-python>

²<https://www.virtualbox.org/>

2. Desarrollo

Todo desarrollador desea que su aplicación no contenga *bugs*, se ejecute sin errores y que lo haga de la forma más rápida posible. Volviendo al tema de la lentitud de Python, muchos aseguran que al ser un lenguaje de programación compilado-interpretado, el lenguaje no es capaz de competir con otros lenguaje programación en cuanto a rapidez. Para una aplicación relativamente pequeña esto no es ningún inconveniente. Sin embargo, para una aplicación que se usa a gran escala y de forma concurrente, el caso no es el mismo. Aunque el código sea limpio y escalable, siempre se busca mayor rapidez y también optimización. Si cada función o método puede ser modificado para que su tiempo de ejecución disminuya aunque sea unos milisegundos, la suma de todas las modificaciones ayudará a largo plazo al producto final. Esto también será algo bueno para desplegar el programa a gran escala.

2.1. ¿Cómo funciona Python?

Como se indicó con anterioridad en este documento, Python es un lenguaje de programación dinámico y funcional. Adicionalmente, es un lenguaje de alto nivel y de propósito general. Es *dynamically-typed*³ y posee un colector de basura que se encarga de liberar recursos del ordenador ocupados por un programa luego de que este ha sido ejecutado.

Otra característica interesante de Python es que el código escrito por el desarrollador es primeramente compilado y posteriormente interpretado.

2.1.1. Lenguaje compilado y lenguaje interpretado

Cuando se escribe un programa en C/C++, este programa debe ser compilado. La compilación involucra convertir el código escrito en lenguaje humano entendible (inglés en este caso) en un lenguaje que la máquina «entienda», o como es conocido, *Machine code*. En lo adelante este término será referido simplemente como **código máquina**. El código máquina es el lenguaje base mediante el cual las instrucciones pueden ser ejecutadas por la CPU⁴. Una vez la compilación ha sido llevada a cabo de forma exitosa, se genera un fichero ejecutable. Cuando se ejecuta este fichero, las instrucciones en el código inicial son llevadas a cabo paso a paso. Lo que significa que compilar un código es, en esencia, traducir un lenguaje de programación a lenguaje máquina y después ejecutarlo paso a paso.

Por otra parte, interpretar un código escrito en un lenguaje de programación, significa ejecutar el código de forma directa y libre, no existe el proceso de compilar

³Un lenguaje de programación de tipado dinámico es aquel en el cual una variable puede tomar valores de tipos no predefinidos. A diferencia de lenguajes de programación de tipado estático o fuertemente tipados en el cual hay que definir el tipo de la variable cuando esta es declarada, e.g. string, char, int, ...

⁴Central Processing Unit o Unidad Central de Procesamiento.

el programa a código máquina. El intérprete⁵ traduce cada sentencia en una secuencia de una o más subrutinas, y después en otro lenguaje (regularmente código máquina). La figura 1 muestra la diferencia principal entre lenguajes de programación compilados e interpretados.

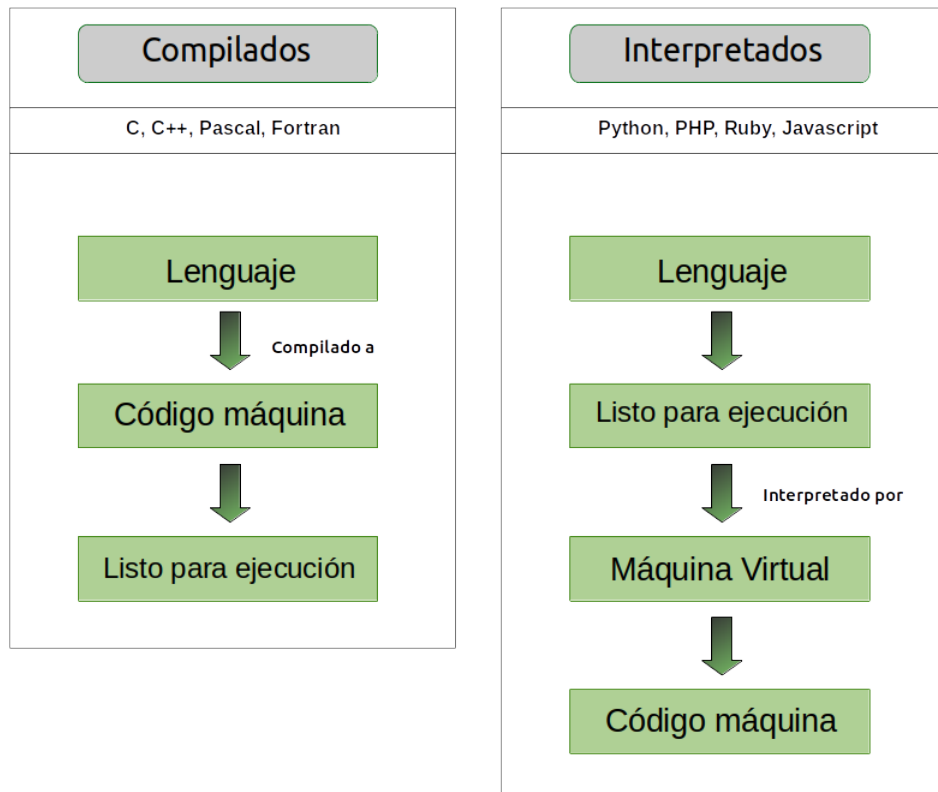


Figura 1: Representación gráfica de la diferencia principal entre un lenguaje de programación compilado y uno interpretado.

En su mayoría, Python es un lenguaje interpretado y no compilado. Aunque la compilación en Python es un paso intermedio. Por ejemplo, el código que el desarrollador escribe en Python se almacena en un fichero con extensión **.py**. Cuando este fichero con el código es ejecutado, entonces este código es primeramente compilado, lo que resulta en la creación de otro fichero con extensión **.pyc** o **pyo**. En este proceso se obtiene lo que se conoce como *bytecode*.

En vez de convertir el código Python a código máquina como sucede en lenguajes como C/C++ o Pascal, lo que sucede es que se traduce el código a *bytecode* (figura 2). Este *bytecode* no es más que un conjunto de instrucciones de bajo nivel que pueden ser ejecutadas por un intérprete. En vez de ejecutar las instrucciones directamente en el CPU, las instrucciones *bytecode* son ejecutadas en una máquina

⁵En Ciencias de la Computación, un intérprete es un programa de computación que ejecuta directamente las instrucciones escritas en un lenguaje de programación sin que este haya sido previamente compilado.

virtual.

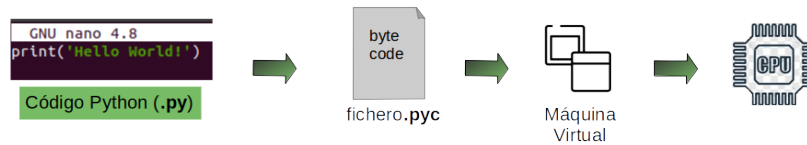


Figura 2: Proceso de compilación del código Python a **.pyc** para luego ser interpretado por la máquina virtual.

2.1.2. Python, interpretado

Una ventaja muy popular de los lenguajes de programación interpretados es que son multiplataformas. Siempre que el *bytecode* de Python y la máquina virtual tengan la misma versión, el *bytecode* de Python puede ser ejecutado en cualquier plataforma (e.g. Linux, MacOS, Windows, etc.).

Otra ventaja es el tipado dinámico, o *dynamically-typed* como se referenciaba anteriormente. En lenguajes de tipado estático como C++, el desarrollador tiene que declarar la variable haciendo referencia al tipo de la misma, y cualquier discrepancia semántica, como adicionar un número con una palabra, será chequeado durante la compilación. En Python, quien se encarga de estas tareas análogas es el intérprete.

2.1.3. Lentitud debido al intérprete

Muchos lenguajes de programación interpretados son considerados lentos. A diferencia de los lenguajes compilados, que no necesitan un intermediario (máquina virtual) para ejecutar las instrucciones escritas en el código. Un lenguaje de programación compilado ejecuta las instrucciones directamente en el procesador, sin embargo, al ser compilado, debe ser desarrollado diferente en cada plataforma. Tengamos en cuenta la siguiente analogía discutida en el famoso sitio de desarrolladores *stackoverflow*⁶:

«Imagina que habla con una persona que domina el mismo idioma que usted. Imagina que después habla con otra persona, pero lo hacen mediante un intérprete profesional ya que ambos hablan idiomas diferentes. Evidentemente la conversación más rápida y fluida será la primera mencionada.» (figura 3)

Esta misma analogía puede ser utilizada en la situación actual. Por ejemplo, el lenguaje de programación C es considerado por muchos como el más rápido que existe y es un lenguaje compilado. Mientras que Python es ampliamente usado, quizás el lenguaje más usado hoy día, pero el simple hecho de ser interpretado hace

⁶<https://stackoverflow.com/questions/1694402/why-are-interpreted-languages-slow>

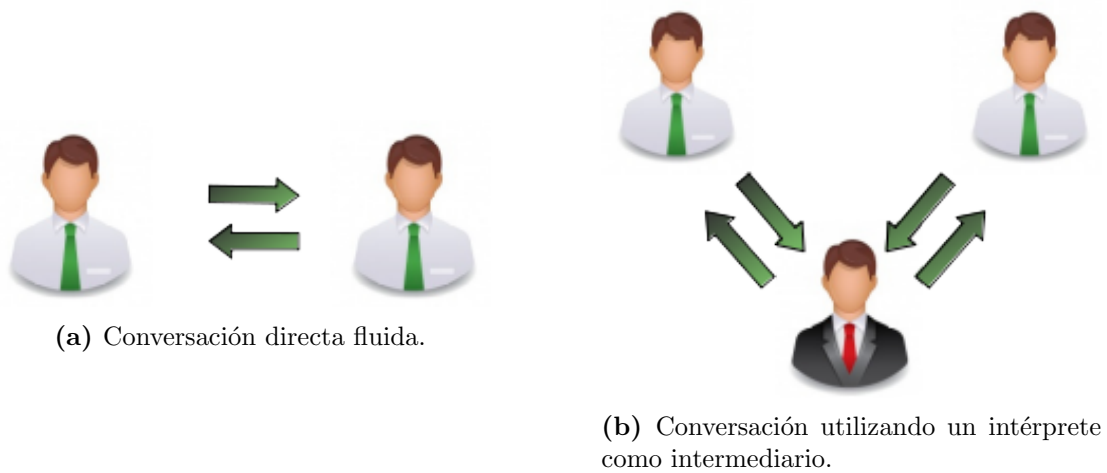


Figura 3: Imagen gráfica de la analogía.

que sufra problemas de rapidez debido a su intérprete. Aunque muchos desarrolladores tratan de solventar esto, siempre será un problema intrínseco de los lenguajes de programación interpretados. Una comparación interesante entre Python y C++ se puede encontrar en el siguiente enlace ⁷.

⁷<https://www.bitdegree.org/tutorials/python-vs-c-plus-plus/>

2.2. Lo más rápido posible

El hecho que Python sea un lenguaje de programación interpretado en su mayoría, hace que sea un poco más lento de lo normal por el proceso de utilizar una máquina virtual para procesar el *bytecode*. Sin embargo, eso no quiere decir que no se pueda escribir código que sea rápido. Uno de los propósitos de esta monografía es describir una utilidad que permita escoger la más rápida entre dos porciones de código.

La idea es lograr crear una utilidad que pueda ejecutar dos porciones de código y registrar el tiempo en que demora cada porción en completar la ejecución. Luego de realizar esto, se le debe mostrar al desarrollador los tiempos de ejecución de una forma detallada y que sea lo más precisa posible. El IDE⁸ elegido para desarrollar la utilidad es VSCodium⁹

2.2.1. VSCodium, un IDE completamente *open source*

VSCodium es un IDE que deriva de Visual Studio Code¹⁰, un entorno desarrollado por Microsoft. El problema con Visual Studio Code es que la versión para descarga disponible en el sitio oficial cuenta con una licencia de software privativa. Al mismo tiempo, el código de este IDE se encuentra en GitHub (VSCo¹¹) disponible para descarga.

A diferencia del sitio de descarga oficial que posee los binarios del entorno listos para instalar, el código en GitHub de VSCo no está compilado, el mismo debe ser descargado y compilado por el usuario para obtener los binarios de instalación. Otra diferencia con el disponible en el sitio oficial es la licencia que se usa en este caso: MIT License¹².

Hoy en día los datos son un activo muy importante, por eso las compañías tratan de sacar el mayor provecho de estos. Muchas veces a través de los softwares que usan los usuarios. Esto se conoce muchas veces como telemetría o uso de datos. La telemetría en productos es muy común, incluso Ubuntu lo hace, pero de una manera más transparente. Algunos usuarios no están de acuerdo con esta práctica muchas veces, razón principal por la cual nace el proyecto VSCodium. O sea que en esencia, VSCodium es prácticamente igual a VSCo (no confundir con los binarios para descarga de Visual Studio Code). La gran diferencia es que VSCodium fue «creado» para no enviar el uso de datos (telemetría) por parte del usuario a la compañía Microsoft. Incluso visualmente, ambos proyectos son prácticamente iguales (figura 4).

A continuación, una tabla comparativa con características principales entre los tres proyectos mencionados:

⁸Integrated Development Environment o Entorno de Desarrollo Integrado, en español.

⁹<https://vscodium.com/>

¹⁰<https://code.visualstudio.com/>

¹¹<https://github.com/microsoft/vscode>

¹²<https://github.com/microsoft/vscode/blob/master/LICENSE.txt>

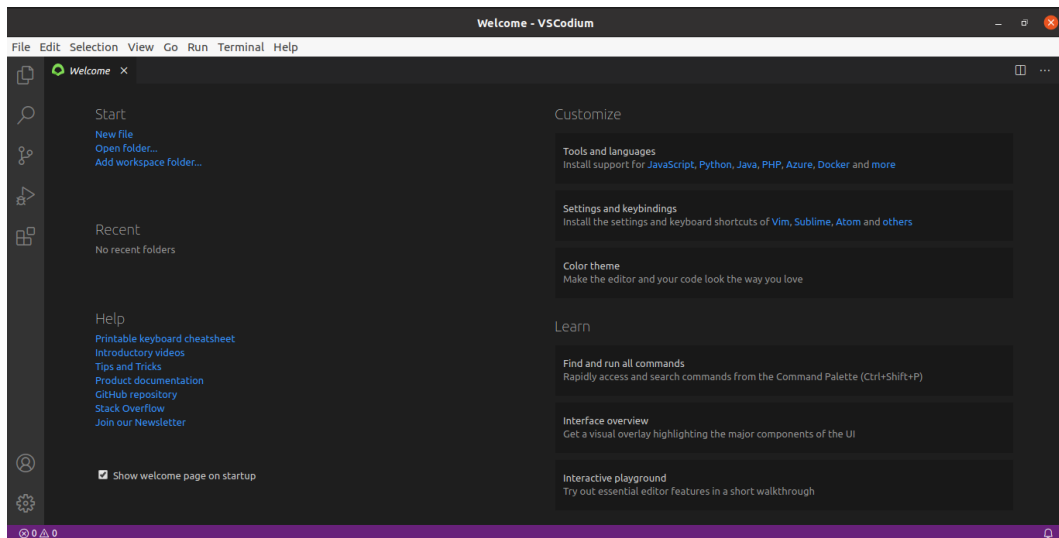


Figura 4: Pantalla de bienvenida de VSCodium.

	Visual Studio Code	VSCode	VSCodium
Licencia	MICROSOFT LICENSE	MIT LICENSE	MIT LICENSE
Telemetría	Activada	Activada	Desactivada
Libre	NO	SÍ	SÍ
Binarios	SÍ	NO	SÍ

Cuadro 1: Tabla comparativa entre los tres IDE.

VSCodium es un Entorno de Desarrollo Integrado multiplataforma, lo que significa que puede instalarse en sistemas operativos tales como GNU/Linux, MacOS, Windows, entre otros. El proceso de instalación es bastante sencillo y se puede encontrar en la subsección 5.1 de la presente monografía, específicamente en la sección de los anexos. Es aconsejable que el lector que desee instalar VSCodium cuente con conexión estable a internet en caso de que sea necesario resolver alguna dependencia.

2.2.2. Enfoque teórico de la utilidad

En Ciencias de la Computación, la Complejidad Computacional¹³ es la cantidad de recursos requeridos para ejecutar un algoritmo. Con especial énfasis en requerimientos de tiempo y memoria. El estudio de la complejidad de ciertos algoritmos se conoce como **Análisis de algoritmos**.

Generalmente es muy complejo computar con precisión la complejidad en el peor de los casos y en el caso promedio de un algoritmo. Por lo que se vuelve un proceso poco práctico. De hecho, el uso de los recursos de un ordenador no se vuelve algo crítico para valores pequeños de un conjunto. A ese conjunto hipotético le llamaremos n .

¹³Término también conocido como Complejidad algorítmica.

Por estas razones, el enfoque generalmente se encuentra en la complejidad de algoritmos para valores grandes de n . Específicamente cuando n tiende a infinito. Por lo tanto, en estos casos la complejidad se expresa generalmente en la notación «*O grande (Big O notation)*»¹⁴.

Por ejemplo, la ejecución del recorrido de un ciclo *for*¹⁵ con una cantidad de valores que tiende al infinito tiene como complejidad algorítmica $O(n)$. Mientras que la complejidad algorítmica de dos ciclos *for* anidados con una cantidad de valores que tiende al infinito es $O(n^2)$. A efectos prácticos esta notación puede que no diga mucho. Pero si el desarrollador pudiese ejecutar dos porciones de código, que realicen las mismas operaciones pero de diferente forma, y aún así pudiese obtener el tiempo de ejecución de cada una para poder realizar una comparación, entonces sería algo diferente. Quizás no de una forma tan científica como con los recursos de la Complejidad Computacional pero igual de útil y quizás más práctica.

Para tener una idea más clara, consideremos la siguiente porción de código escrito en Python:

```
lista = [1, 2, 3, 4, 5, 6]
lista2 = list(map(lambda i: i + 2, lista))
print(lista2)
```

El código anterior es un código de ejemplo que muestra la ejecución de la función *map()*¹⁶.

Tras la ejecución, la salida por consola es la siguiente:

```
[3, 4, 5, 6, 7, 8]
```

En ensencia, lo que hace el código en 2.2.2 es tomar una lista con los elementos [1, 2, 3, 4, 5, 6] y a cada uno sumarle 2. Este proceso es posible y de forma sencilla gracias al uso de la función *lambda*¹⁷ de Python. En este caso es la función (*lambda*) que se pasa como parámetro a la función *map()*.

Ahora consideremos este siguiente código:

¹⁴https://en.wikipedia.org/wiki/Big_O_notation

¹⁵Un ciclo *for* es una estructura de control cíclica utilizada en lenguajes de programación para realizar tareas repetitivas que evitan la reescritura de la misma porción de código una y otra vez.

¹⁶La función *map()* es una función de orden superior, o sea, una función que recibe como parámetro otra función. Esta función viene por defecto con Python

¹⁷Las funciones *lambda* son funciones anónimas las cuales no reciben ningún nombre y de cierta forma permiten ahorrar líneas de código. Generalmente se usan para ejecutar porciones de código sencillas. Las funciones *lambda* puede recibir una cantidad arbitraria de parámetros pero debe tener una sola expresión.

```
lista = [1, 2, 3, 4, 5, 6]
lista3 = [item + 2 for item in lista]
print(lista3)
```

Tras la ejecución, la salida por consola es la siguiente:

```
[3, 4, 5, 6, 7, 8]
```

O sea que estamos ante la misma salida que obtuvimos cuando se ejecutó el código con la función de orden superior que recibía como parámetro una función *lambda*. En este caso el recurso de Python que se usó es llamado *List Comprehension*¹⁸. Claro está que al igual que la porción de código anterior, lo que se realiza en esta ocasión es la suma de cada valor de la lista por dos.

Aunque en ambos casos se usan pocas líneas de código (tres) para realizar la suma, los recursos usados son diferentes. Y si tenemos en cuenta la Complejidad algorítmica mencionada en 2.2.2 tendríamos una complejidad teórica de $O(n)$ en ambos casos. Aunque debemos recordar que la teoría en la Complejidad algorítmica se establece para una cantidad de elementos que tiende a infinito.

Sin embargo, el uso práctico vuelve a ser muy pobre. No obstante, si pudiésemos separar la ejecución de las dos porciones de código en momentos diferentes, marcados por nanosegundos¹⁹, llegaríamos a la conclusión que es prácticamente imposible que ambos se hubiesen ejecutado en la misma cantidad de nanosegundos. Incluso, si por la gran capacidad computacional de los ordenadores de hoy día fuese imposible separar la ejecución de ambos algoritmos en nanosegundos, entonces pudiésemos pensar en el orden de los yoctosegundos²⁰.

En esencia, la idea del párrafo anterior es que si se pudiese identificar qué porción de código se ejecuta en menor cantidad de tiempo, tendríamos la posibilidad de poco a poco ir haciendo nuestro programa más rápido de forma general²¹. Este proceso sería de forma gradual. Una especie de suma de optimizaciones diminutas. Claro está, sería poco lógico tratar de dedicar tiempo a optimizar un proyecto personal pequeño como por ejemplo una aplicación móvil sencilla. Pero para proyectos realmente grandes, tales como sitios institucionales o redes sociales, la optimización de código en cuanto al tiempo podría ser considerada como una buena práctica.

¹⁸*List Comprehension* es un mecanismo poderoso de Python que puede ser usado para generar nuevas listas, casi siempre usando poco código.

¹⁹Un nanosegundo es una unidad de tiempo de la International System of Units (SI) que representa un 10^{-9} segundo.

²⁰Un yoctosegundo es una unidad de tiempo de la International System of Units (SI) que representa un 10^{-24} segundo.

²¹*N. del A.*: La presente monografía se basa en la racionalización del tiempo de ejecución, pero hay otros factores que se deben tener en cuenta a la hora de hacer un software lo más óptimo posible, e.g. consumo de memoria.

2.2.3. Describiendo la utilidad

La utilidad fue diseñada para que el desarrollador o lector que desee replicar los resultados pueda ejecutar las pruebas de velocidad de las funciones que desee. En la descripción siguiente se describe el proceso de ejecución para tres funciones, pero el lector puede modificar el código para agregar cualquier cantidad de funciones. Para evitar problemas a la hora de obtener el código de la utilidad, este propio documento cuenta con todo el código en el anexo 5.3. Para realizar las pruebas se debe copiar y pegar todo el código en VSCode y guardar con el nombre **test_rapidez.py**²². Una vez guardado el fichero con ese nombre, el lector deberá prestar atención a cualquier problema de indentación²³.

Módulos importados a tener en cuenta

Para poder medir y almacenar los tiempos de ejecución de las funciones fue necesario importar el módulo *time*. Mientras que para poder mostrar los mensajes a color por la consola fue necesario importar *Fore* y *Style* del módulo *colorama*²⁴. La librería *Matplotlib* no viene instalada por defecto por lo que hay que instalarla con la instrucción **sudo python3 -m pip install --user matplotlib**.

```
import time
from colorama import Fore, Style
import numpy as np
import matplotlib.pyplot as plt
```

Usando Decoradores

Para poder ejecutar cierta cantidad de veces cada función fue necesario hacer uso de Decoradores²⁵. La definición e implementación de uno de los decoradores se puede observar a continuación:

```
def logger(fn):
    @ciclo
    def interna():
        inicio = time.time()
        fn()
        duracion = time.time() - inicio
        if fn.__name__ not in VALORES:
            VALORES[fn.__name__] = [0, []]
```

²²O simplemente clonar el código de GitHub.

²³Python es un lenguaje de programación que inicia y termina los bloques de código teniendo en cuenta los espacios.

²⁴La versión de Python usada para desarrollar la utilidad fue la **3.8.2**. En esa versión vienen instalados por defecto los módulos *time* y *colorama*.

²⁵En Python un decorador es un recurso que funciona igual que una función de nivel superior.

```

        VALORES[fn.__name__][0] += 1
        VALORES[fn.__name__][1].append(duracion)
    return interna

```

El decorador anterior tiene como nombre *logger* y para ser utilizado, debe ser colocado justamente encima de cada función a la cual se le desea realizar una prueba de velocidad. Por ejemplo, en el desarrollo de la utilidad se implementaron tres funciones extremadamente sencillas para realizar las pruebas. Los nombres de cada una son: **func_uno()**, **func_dos()** y **func_tres()**. Para indicar que se realizaran las pruebas de velocidad de ejecución a las tres, fue necesario declarar el decorador *logger* sobre cada una como se muestra a continuación:

```

@logger
def func_uno():
    print('Esta es la función UNO.')

@logger
def func_dos():
    print('Esta es la función DOS.')

@logger
def func_tres():
    print('Esta es la función TRES.')

```

Declarar sobre una función el decorador *logger* significa que antes que comience la ejecución de la función, se registrará el tiempo de inicio. Cuando se termine la ejecución de la función, se registrará el tiempo también. Al restar el tiempo final menos el tiempo inicial se obtiene el tiempo de ejecución. Este proceso se realiza la cantidad de veces que el usuario defina. Si el usuario quisiera agregar otra función, por ejemplo **func_cuatro()**, deberá hacerlo en el bloque marcado por el comentario *Espacio de configuración*. También deberá registrar la función en el bloque:

```

def funciones_registradas():
    func_uno()
    func_dos()
    func_tres()
    func_cuatro()    ### -----> Función agregada

```

Interacción con el usuario

La utilidad inicia preguntándole al usuario la cantidad de veces que se va a ejecutar cada función, luego le pregunta al usuario si desea que se muestre en una gráfica los valores de tiempos obtenidos tras la ejecución de cada función. Esos resultados son, para cada una de las funciones:

- Tiempo mínimo
- Tiempo máximo
- Tiempo promedio
- Cantidad de veces que se ejecutó cada función

El método que permite obtener el tiempo de ejecución del módulo *time* es *time()*. Luego que se realiza la resta entre el tiempo de inicio y el tiempo de fin, el resultado se muestra en formato decimal con 20 lugares después de la coma. Un poco más por encima de la escala de los nanosegundos. La figura 5 muestra una ejecución de prueba como ejemplo.

```
#####
#                      RESULTADOS DE LA EJECUCIÓN:                      #
#####

La función "func_uno" se ejecutó 5 veces con los resultados:

Tiempo mínimo:      3.314018249511719e-05
Tiempo máximo:      0.0001232624053955078
Tiempo promedio:     5.283355712890625e-05

-----

La función "func_dos" se ejecutó 5 veces con los resultados:

Tiempo mínimo:      3.266334533691406e-05
Tiempo máximo:      4.0531158447265625e-05
Tiempo promedio:     3.5858154296875e-05

-----

La función "func_tres" se ejecutó 5 veces con los resultados:

Tiempo mínimo:      3.24249267578125e-05
Tiempo máximo:      3.6716461181640625e-05
Tiempo promedio:     3.35693359375e-05

-----
```

Figura 5: Ejecución de la utilidad con la opción de graficar deshabilitada.

En la figura anterior se observa la ejecución de las tres funciones que fueron registradas en la porción de código:

```
def funciones_registradas():
    func_uno()
    func_dos()
    func_tres()
```

Esa es la porción de código de la utilidad que es configurable por el desarrollador.

Quizás la salida por consola que se muestra en la figura 5 no sea lo suficientemente explicativa, a fin de cuentas, siempre es mejor utilizar una imagen para exponer de mejor forma los resultados. Esa sería la función en este caso de la librería *Matplotlib* que se importó en 2.2.3.

2.2.4. Uso de la librería *Matplotlib* para visualizar datos

De acuerdo con Wikipedia²⁶, «*Matplotlib es una librería de ploteo para el lenguaje de programación Python y su extensión matemática NumPy*». Esta última también se usa junto con *Matplotlib* en la utilidad que se describe en esta monografía. Al artículo lo acompaña la siguiente figura:

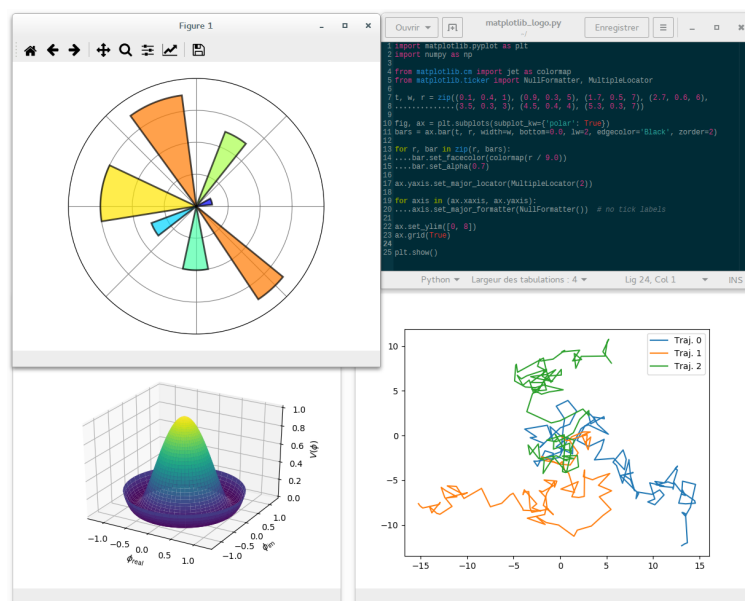


Figura 6: Imagen que muestra los diferentes gráficos que pueden generarse usando *Matplotlib*. Foto tomada de Wikipedia.

En la utilidad desarrollada, para graficar los resultados obtenidos en cuanto al tiempo de ejecución, se le muestra por consola al usuario un mensaje de selección. Esto sucede inmediatamente después que se muestran por consola los números de ejecución, como se muestra en la figura 7.

En caso que el usuario introduzca el caracter "N", la ejecución termina. Si el usuario introduce el caracter "S", entra en ejecución el código referente a la librería *Matplotlib* y se muestra una gráfica de barras con los tiempos obtenidos para cada función. Por ejemplo, para los tiempos de ejecución de las tres funciones de prueba que se han utilizado hasta ahora, luego de 10 iteraciones, se obtienen los resultados de la figura 8. En la imagen se puede observar que para cada función se muestran tres barras de diferentes colores. La barra de color «azul» muestra el tiempo máximo para cada función, la barra de color «verde» muestra el tiempo

²⁶<https://en.wikipedia.org/wiki/Matplotlib>

```
laptop@home: ~/Documents/trapidez_env

La función "func_uno" se ejecutó 10 veces con los resultados:
Tiempo mínimo: 3.314018249511719e-05
Tiempo máximo: 0.00012731552124023438
Tiempo promedio: 4.937648773193359e-05
-----
La función "func_dos" se ejecutó 10 veces con los resultados:
Tiempo mínimo: 4.601478576660156e-05
Tiempo máximo: 5.0067901611328125e-05
Tiempo promedio: 4.699230194091797e-05
-----
La función "func_tres" se ejecutó 10 veces con los resultados:
Tiempo mínimo: 4.673004150390625e-05
Tiempo máximo: 0.00013756752014160156
Tiempo promedio: 5.695819854736328e-05
-----
Desea graficar resultados? (S/N): s
```

Figura 7: Mensaje por consola para selección de graficado.

promedio, mientras que la barra de color «negro» muestra el tiempo mínimo. El eje de coordenadas **X** representa los nombres de cada función y el eje **Y** representa los tiempos de ejecución obtenidos.

Una pregunta lógica que puede surgir luego de analizar bien la figura 8 es la siguiente:

¿Por qué si las tres funciones de prueba ejecutan prácticamente la misma porción de código los tiempos de ejecución son notablemente diferentes?

La razón principal se debe a que la ejecución de la utilidad se realiza utilizando recursos que también están siendo consumidos por otras aplicaciones del sistema. Por lo tanto, en momentos donde la utilidad necesita todos los posibles recursos para ejecutarse de forma uniforme e ininterrumpida, es muy probable que no los tenga.

Sin embargo, la utilidad fue desarrollada para de alguna forma ser capaz de mitigar este problema. Es por eso que lejos de ejecutarse cada función una sola vez, lo hace una cantidad de veces definida por el propio usuario. Debido a esto, debe ser considerado como de más importancia el tiempo promedio y no el tiempo mínimo de ejecución. En la imagen 8 se llevó a cabo la ejecución de cada función 10 veces. El resultado será más parejo a medida que este número de ejecuciones crece.

Por ejemplo, en la figura 9 se muestra la ejecución de la utilidad 1000 veces. Como se puede observar, los tiempos mínimos y promedios apenas pueden distinguirse porque aparte de ser tiempos muy parejos, también son tiempos cercanos a cero. Todo esto ocurre además, gracias a los picos de tiempo máximo alcanzado por las tres funciones, que a su vez ocurren por tener que compartir recursos del ordenador con otras aplicaciones de mayor prioridad.

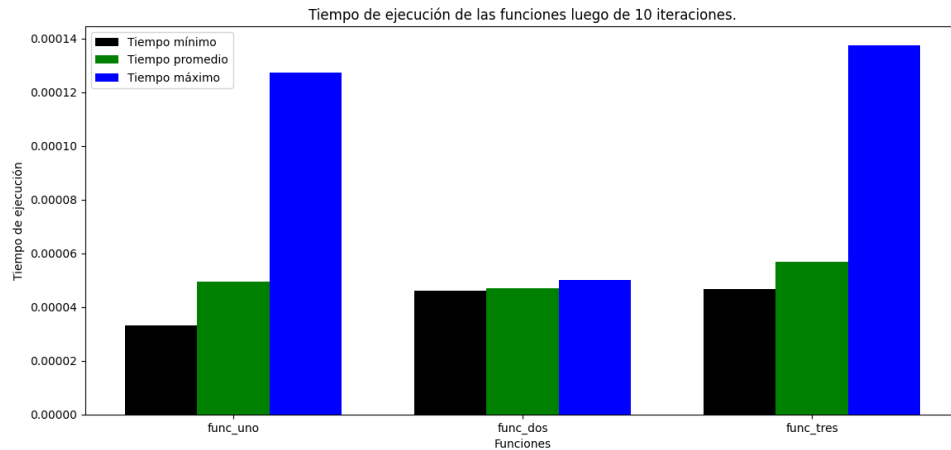


Figura 8: Gráfico de tiempos obtenidos para las tres funciones de prueba.

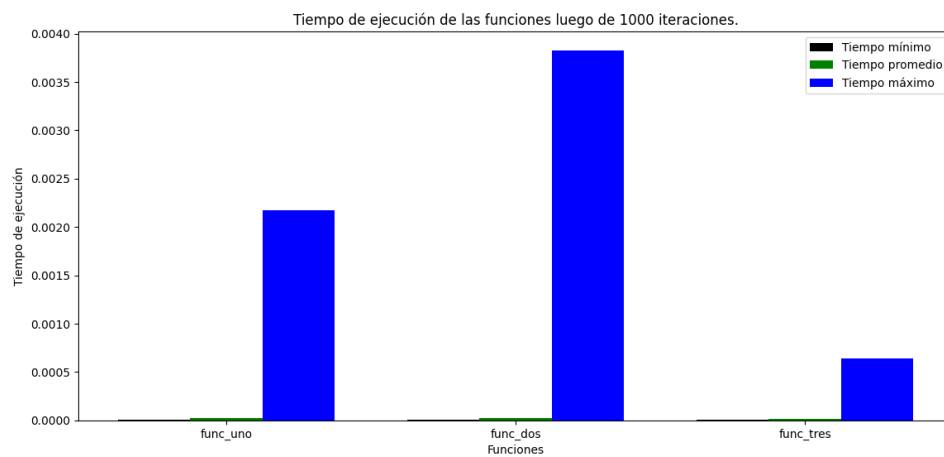


Figura 9: Gráfico de tiempos obtenidos luego de 1000 iteraciones.

2.3. Aplicando la utilidad a un caso de prueba

En el código de la sección 2.2.2 se plantea el problema de incrementar los valores de una misma lista en dos mediante soluciones y mecanismos de Python diferentes. En este caso la lista inicial es `[1, 2, 3, 4, 5, 6]` y la lista resultante es `[3, 4, 5, 6, 7, 8]` para ambos casos. El primer recurso utilizado es la función `map()`, la cual es una función de orden superior. El segundo recurso utilizado es *List Comprehension*, que como se indicaba anteriormente, es un mecanismo para obtener nuevas listas. El objetivo de esta sección es explicar los pasos necesarios para conocer cuál de los dos mecanismos anteriores es más eficiente a nivel de tiempo de ejecución.

2.3.1. Creando el entorno virtual

El lector que desee replicar los resultados mostrados en la monografía debe tener en cuenta que la versión de Python en la cual fue desarrollada la utilidad es la **3.8.2**. Adicionalmente, el sistema operativo que debe ser usado es **Ubuntu 20.04.1 LTS (Focal Fossa)**. Existe la posibilidad que el lector no tenga el sistema operativo antes mencionado, o quizás puede que la versión de Python que esté utilizando sea una un tanto inferior. Todo esto tendría como posible consecuencias que los módulos y librería utilizada no estén disponibles por defecto. Este problema puede ser solucionado usando un **entorno virtual**.

Un entorno virtual es un entorno de Python en el cual el intérprete, las librerías y los *scripts* del propio entorno virtual se encuentran aislados de otros entornos virtuales; y también de las librerías instaladas en el propio Sistema Operativo. Esto permite que la aplicación desarrollada pueda ser ejecutada en otro sistema operativo sin tener en cuenta si ese otro sistema operativo tiene instaladas las librerías necesarias para ejecutar la misma.

El comando necesario para crear un entorno virtual de Python es ***venv*** y supongamos que el nombre que le pondremos será **caso_prueba**. En este caso la línea de comandos necesaria para crear el entorno virtual es:

```
python3 -m venv caso_prueba
```

Si todo salió bien, se debe haber ejecutado el paso sin mensajes en la consola como se muestra en la figura 10.

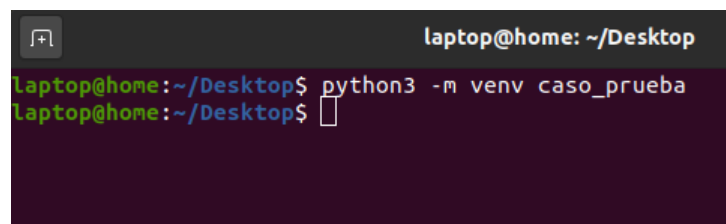
A screenshot of a terminal window with a dark background. The title bar at the top reads 'laptop@home: ~/Desktop'. The terminal shows two lines of text: the first line is 'laptop@home:~/Desktop\$ python3 -m venv caso_prueba' and the second line is 'laptop@home:~/Desktop\$' followed by a cursor. The text is in a light green color.

Figura 10: Comando necesario para crear un entorno virtual.

Como el entorno virtual se creó en la ruta **/home/laptop/Desktop** entonces debe verse una carpeta en esa ruta, precisamente en el escritorio como se observa en la figura 11. Cuando se crea un entorno virtual, se debe proceder a realizar la activación del mismo. Esto se logra con la línea de comandos:

```
source caso_prueba/bin/activate
```

La no existencia de errores en la activación del entorno virtual se evidencia con la aparición del prefijo (**caso_prueba**) en cada línea nueva en la consola. Esto

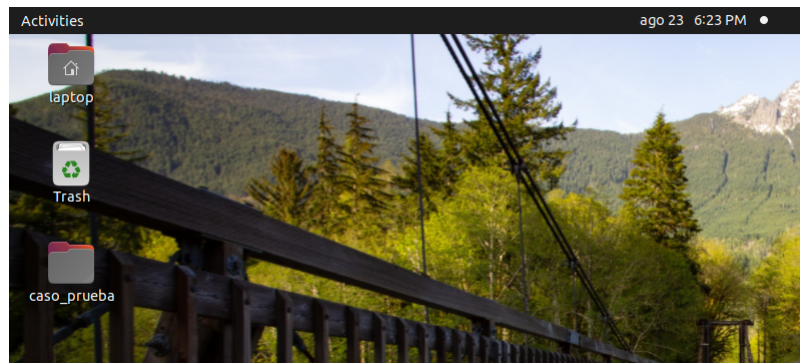


Figura 11: La carpeta con el nombre del entorno virtual se crea en la ruta indicada.

significa que a partir de esa línea, toda nueva línea de comandos que se ejecute debe afectar el entorno virtual. (figura 12)

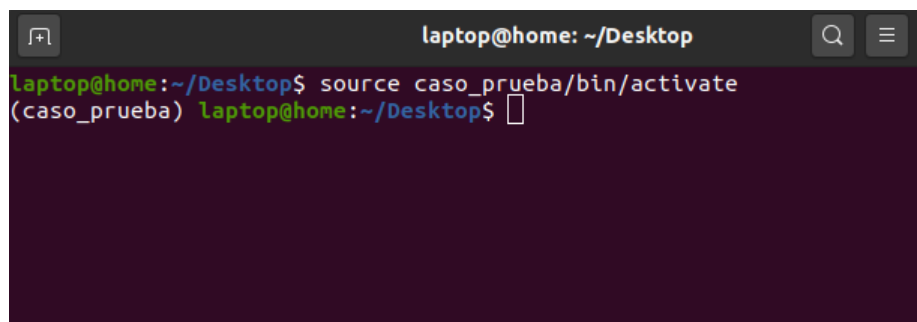


Figura 12: Activación del entorno virtual.

Luego de crear y activar el entorno virtual, lo recomendado es instalar las librerías necesarias para ejecutar la utilidad. Como se mencionaba en 2.2.3, el módulo *time* viene por defecto en Python. Pero *colorama* y *Matplotlib* no vienen por defecto y deben ser instalados para poder ejecutar la utilidad y mostrar el gráfico de barras con los tiempos de ejecución. Esto se logra ejecutando la línea:

pip install matplotlib && pip install colorama

Estas librerías son instaladas por Python desde internet por lo que es recomendable contar con conexión de buena calidad para una descarga sin problemas. Posteriormente a la instalación de la librería *matplotlib* y el módulo *colorama* siempre es una buena idea chequear que la descarga se realizó sin problemas. Esto se realiza mediante la línea de comandos siguiente:

pip list

```
(caso_prueba) laptop@home:~/Desktop/caso_prueba$ pip list
Package            Version
-----
certifi            2020.6.20
colorama           0.4.3
cyclor             0.10.0
kiwisolver         1.2.0
matplotlib         3.3.1
numpy              1.19.1
Pillow             7.2.0
pip                20.0.2
pkg-resources      0.0.0
pyparsing          2.4.7
python-dateutil    2.8.1
setuptools         44.0.0
six                1.15.0
```

Figura 13: Lista de librerías instaladas en el entorno virtual.

Como se puede ver en la figura 13 la librería *Matplotlib* se encuentra instalada y disponible para ser usada. Lo mismo sucede con el módulo *numpy*, que aunque no fue instalado mediante comandos, viene incluido dentro de la librería *Matplotlib*.

2.3.2. Descargando el repositorio con *git*

Todo el código fuente de la utilidad se encuentra alojado en el repositorio público del autor en GitHub²⁷ bajo la licencia MIT LICENSE. Esto significa que el mismo se encuentra disponible para ser descargado siempre que el lector lo desee. Existen varias vías para descargar el repositorio de la utilidad, sin embargo, el que se utilizará en esta monografía será la vía mediante la utilización por consola de la herramienta *git*²⁸. Para instalar *git* se debe ejecutar en consola el comando **sudo apt install git** como se muestra en la imagen 14.

```
laptop@home: ~/Desktop
(caso_prueba) laptop@home:~/Desktop$ sudo apt install git
[sudo] password for laptop:
```

Figura 14: Comando para instalar la herramienta de consola *git*.

²⁷https://github.com/ppdmartell/test_rapidez
²⁸*git* es un sistema distribuido de control de versiones que permite estar al tanto de cambios del código fuente de una aplicación en el área del desarrollo del software.Fuente: <https://en.wikipedia.org/wiki/Git>

Una vez instalada la herramienta *git* solo debemos clonar el repositorio del autor y ejecutar la utilidad desde la consola (ubicados dentro del entorno virtual). Para clonar el repositorio debemos introducir la siguiente línea de comandos:

```
git clone https://github.com/ppdmartell/test_rapidez
```

El comando anterior clona la utilidad directamente desde GitHub y al hacerlo crea una carpeta con nombre **test_rapidez** dentro del entorno virtual en cuestión de nombre **caso_prueba**. (figura 15)

Name	Size	Modified
bin	14 items	6:56 PM
include	0 items	6:14 PM
lib	1 item	6:14 PM
lib64	1 item	6:14 PM
share	1 item	6:14 PM
test_rapidez	4 items	8:06 PM
pyvenv.cfg	69 bytes	6:14 PM

Figura 15: La carpeta con el repositorio se crea dentro del entorno virtual.

2.3.3. Configurando el código de la utilidad antes de ejecutarla

El paso anterior permitió descargar el repositorio de la utilidad desde GitHub utilizando la herramienta *git*. Sin embargo, para poder comparar el los tiempos de ejecución del caso de prueba, debemos modificar el código de la propia utilidad. Esto se debe a que el código original fue escrito para comprobar los tiempos de ejecución de las funciones:

- `func_uno()`
- `func_dos()`
- `func_tres()`

Mientras que ahora las funciones que van a ser analizadas son:

- `func_clist()`
- `func_ordensup()`

Eso significa que hay que eliminar las funciones originales del fichero de la utilidad (**test_rapidez.py**) y en cambio agregar las dos nuevas. La porción de código de la utilidad destinada a ser modificada por el lector es la siguiente:

```
##### Espacio de configuración#####
```

```
@logger
def func_uno():
    print('Esta es la función UNO.')

@logger
def func_dos():
    print('Esta es la función DOS.')

@logger
def func_tres():
    print('Esta es la función TRES.')

def funciones_registradas():
    func_uno()
    func_dos()
    func_tres()
```

```
#####
```

El código anterior viene por defecto en la utilidad. A su vez, y para estar en el contexto actual, debe ser modificado de la siguiente forma:

```
##### Espacio de configuración#####
```

```
@logger
def func_clist():
    lista = [i for i in range(1000001)]
    lista2 = list(map(lambda i: i + 2, lista))
    print(lista2)

@logger
def func_ordensup():
    lista = [i for i in range(1000001)]
    lista3 = [item + 2 for item in lista]
    print(lista3)

def funciones_registradas():
    func_clist()
    func_ordensup()
```

```
#####
```

Como se puede apreciar, fue removido el código por defecto de las tres funciones

y en su lugar se agregó el código de las dos funciones encargadas de sumar 2 a cada elemento de la lista. Una vía usando *List Comprehension* y otra usando la función *map()* que es una función de orden superior.

Si prestamos atención al código en 2.3.3 podemos notar algo diferente y es que en vez de utilizar la lista [1, 2, 3, 4, 5, 6] se ha agregado una porción de código (utilizando *List Comprehension*) que genera una lista con números desde 0 hasta 1 000 000. La teoría de Complejidad Computacional establece que la complejidad algorítmica se debe aplicar con una cantidad de elementos que tiendan a infinito²⁹.

2.3.4. Resultados obtenidos del caso de prueba

Como se muestra³⁰ en los gráficos de la figura 16, la segunda función, en este caso la de orden superior, fue la que obtuvo mejores resultados en cuanto a menor tiempo de ejecución. Esto no es una prueba definitiva para afirmar que la función de orden superior tiene un mejor *performance* que *List Comprehension*. El hardware utilizado para realizar las pruebas fue el de una laptop convencional de bajas prestaciones. A lo anterior habría que agregarle que la utilidad se ejecutó de forma concurrente con otras aplicaciones propias del sistema operativo que tienen un consumo de recursos mayor. Sin embargo, sería interesante eliminar estas limitantes hasta donde se pueda y volver a comparar los resultados que se obtengan.

No.	func_	T. mínimo (seg)	T. máximo (seg)	T. promedio (seg)
1	clist ordensup	3.7509214878082275 3.5764341354370117	4.01102614402771 4.2225165367126465	3.8509971618652346 3.8683979749679565
2	clist ordensup	3.966761589050293 3.6924571990966797	4.211878061294556 4.051555633544922	4.101371574401855 3.8603779554367064
3	clist ordensup	3.811283588409424 3.705578327178955	4.1150734424591064 4.024996280670166	3.9174386978149416 3.8813881874084473
4	clist ordensup	3.8141355514526367 3.6887686252593994	4.134859800338745 3.9996981620788574	4.027432537078857 3.8301402807235716

Cuadro 2: Tabla de los tiempos que se obtuvieron al ejecutar la utilidad 4 veces con 10 iteraciones en cada una.

En la tabla 2, **señalados en verde**, se encuentran los tiempos de ejecución más óptimos de cada serie (4 en total) tras 10 ejecuciones. Se nota que los mejores

²⁹Tanto la monografía como la utilidad fue desarrollada en una laptop con prestaciones modestas. El millón de elementos fue fijado para simular una cantidad suficientemente superior a los 6 elementos iniciales de la lista.

³⁰Para poder apreciar correctamente los cuatro gráficos, aumente el *zoom* de su lector PDF a un 300 %

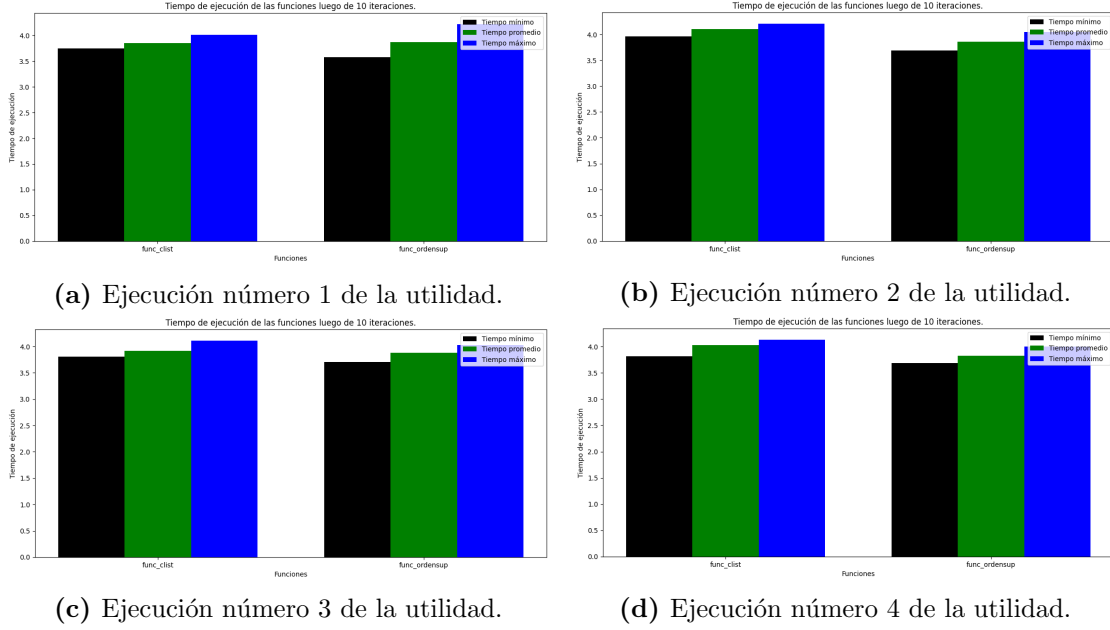


Figura 16: Gráficos de barra resultantes de la ejecución de la utilidad 4 veces con listas de 1 000 000 de elementos.

tiempos pertenecen a la función que usa la función `map()` de orden superior. Sin embargo, y al igual que lo que sucede con la figura 16, no se puede afirmar de forma definitiva que los resultados son concluyentes.

2.3.5. Afinando los resultados con virtualización

Luego de ejecutar la utilidad varias ocasiones y mostrar los resultados en la monografía, se ha podido confirmar no son concluyentes. Esto ha sido mencionado anteriormente y en varias ocasiones en este documento. La causa principal es que durante su ejecución, la utilidad comparte el consumo de recurso con otras aplicaciones, e.g. procesos principales del sistema, entorno de escritorio, procesos en segundo plano, etc.

O sea que este problema es inherente a la ejecución de la aplicación. Esa puede que sea una de las causas principales a los tiempos máximos registrados en algunos resultados, los cuales se alejan demasiado de los tiempos mínimos y promedios (figura 9). Siempre será común que al ejecutar la utilidad o cualquier software, existan otros programas consumiendo de los mismos recursos del ordenador. Este efecto se nota más en este caso, debido a que las pruebas fueron llevadas a cabo en una laptop con modestas prestaciones.

Una posible vía de afinar los resultados sería el uso de la virtualización. La cual es el proceso de ejecutar una instancia virtual de una computadora en una capa de abstracción del hardware real. Comúnmente se conoce como a ejecutar varios sistemas operativos en un mismo ordenador y de forma simultánea si así lo desea el usuario. Las aplicaciones que se ejecutan en la máquina virtual son

propias del sistema operativo que se está virtualizando, mientras que por otro lado, el usuario tiene las aplicaciones del sistema operativo anfitrión.³¹ Una de las principales ventajas que pueden ser aprovechadas, en este caso, es la asignación estáticas de recursos. Por ejemplo, si un usuario crea una máquina virtual con propiedades tales como 2GB de RAM y 2 hilos de procesamiento, entonces estos recursos van a ser bloqueados para el uso exclusivo de la máquina virtual. Esto permite que la ejecución de las aplicaciones que pertenecen a la máquina virtual no se vean afectadas en cuanto al consumo de recursos del ordenador por otras aplicaciones con mayor prioridad.

Los archivos y herramientas utilizadas fueron las siguientes:

Característica	Descripción
Sistema Operativo anfitrión	Ubuntu 20.04.1 LTS (Focal Fossa)
Sistema Operativo <i>guest</i>	Ubuntu-Server 20.04.1 LTS (Focal Fossa)
Archivo .ISO	ubuntu-20.04.1-live-server-amd64.iso
Versión del VirtualBox (fig. 17)	VirtualBox 6.1.10-Ubuntu r138449
Memoria RAM asignada	4GB
Disco duro	16GB

Cuadro 3: Tabla de características y herramientas utilizadas.



Figura 17: Versión de VirtualBox utilizada.

³¹<https://opensource.com/resources/virtualization>

Resultados obtenidos mediante la virtualización

Como se mencionaba anteriormente, el objetivo de utilizar la virtualización es el de afinar los resultados, evitando obtener picos de tiempo de ejecución máximos tan elevados por causa de compartir recursos con otros procesos de mayor demanda. Luego de llevar a cabo la ejecución de la utilidad 4 veces con 10 iteraciones cada vez, se obtuvieron los resultados que se muestran en la tabla 4. Como se puede apreciar, los valores de tiempo máximos continúan siendo un problema, pero existe consistencia en cuanto a la mayor rapidez de ejecución de la función de orden superior sobre la función *List Comprehension*. Este patrón se observó también en la figura 16 que representa los números obtenidos en la tabla 2. La representación de estos resultados mediante gráfico de barras se puede observar en la figura 18.

No.	func_	T. mínimo (seg)	T. máximo (seg)	T. promedio (seg)
1	clist ordensup	3.700131416320801 3.7106707096099854	4.071504354476929 3.9892477989196777	3.8887980699539186 3.8201268672943116
2	clist ordensup	3.831714630126953 3.663576602935791	4.240365743637085 4.046422004699707	4.088127183914184 3.79477915763855
3	clist ordensup	3.8226566314697266 3.780625343322754	4.281528472900391 4.234777450561523	4.0547373533248905 4.002378153800964
4	clist ordensup	3.870190382003784 3.7251315116882324	4.232051610946655 3.9919981956481934	3.9929197072982787 3.849376749992371

Cuadro 4: Tabla de los tiempos que se obtuvieron al ejecutar la utilidad la misma cantidad de veces pero usando una máquina virtual.

Dado que el sistema operativo *guest* usado para crear la máquina virtual fue Ubuntu Server³², hubo la necesidad de guardar los resultados obtenidos por consola y modificar el código de la utilidad para generar los cuatro gráficos de barras pertenecientes a la virtualización. Sin embargo, como los valores son los mismos de la consola, el proceso posterior de generación de los gráficos ocurrió como si hubiese sido una ejecución continua. Lo anterior significa que estos datos tienen la misma veracidad que unos resultados obtenidos de forma continua en el sistema operativo anfitrión.

³²Ubuntu Server es un sistema operativo basado en consola, sin interfaz gráfica, lo que significa que no se pudo ejecutar la librería *Matplotlib* para generar los gráficos.

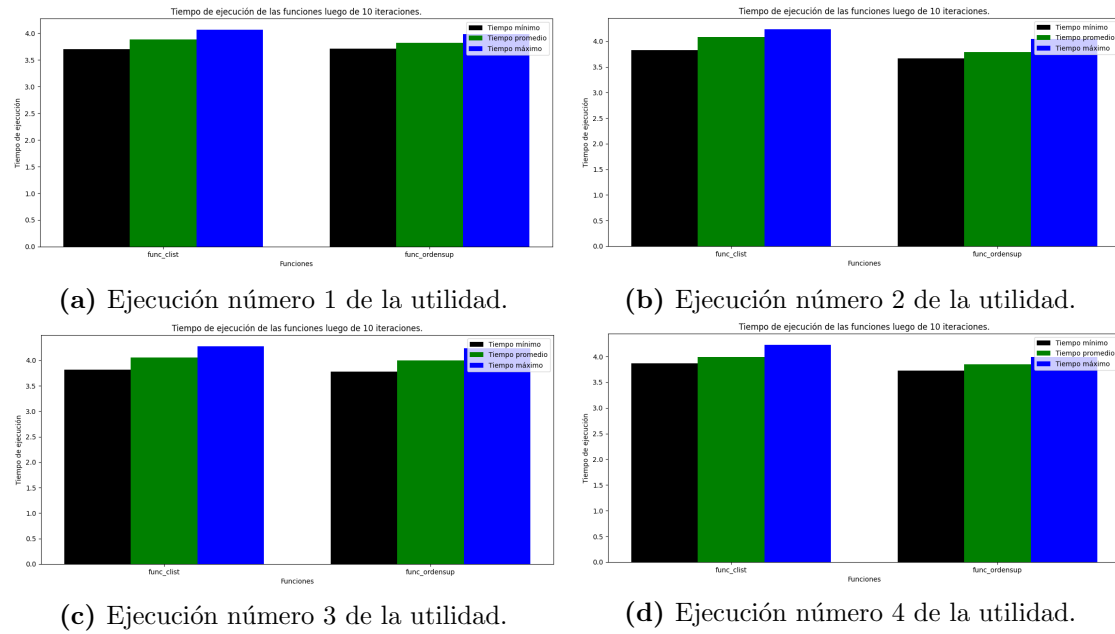


Figura 18: Gráficos de barra resultantes de la ejecución en una máquina virtual de la utilidad 4 veces con listas de 1 000 000 de elementos.

2.3.6. Mayor consistencia con diferente orden de ejecución

El orden de ejecución de las funciones *func_clist()* y *func_ordensup()* sucede de la siguiente forma:

Espacio de configuración#####

```
@logger
def func_clist():
    lista = [i for i in range(1000001)]
    lista2 = list(map(lambda i: i + 2, lista))
    print(lista2)

@logger
def func_ordensup():
    lista = [i for i in range(1000001)]
    lista3 = [item + 2 for item in lista]
    print(lista3)

def funciones_registradas():
    func_clist()
    func_ordensup()
```

```
#####
```

Esto significa que primero se ejecuta la función *func_clist()* y luego *func_ordensup()*. Para quitar de la ecuación el factor *idle*³³ del procesador, como un supuesto alterador de resultados de tiempos máximos, se ejecutó la utilidad con el orden de ejecución de las dos funciones invertido.

El código de la utilidad fue modificado nuevamente, quedando de la siguiente forma:

```
##### Espacio de configuración#####
```

```
@logger
def func_ordensup():
    lista = [i for i in range(1000000)]
    lista3 = [item + 2 for item in lista]
    print(lista3)

@logger
def func_clist():
    lista = [i for i in range(1000000)]
    lista2 = list(map(lambda i: i + 2, lista))
    print(lista2)

def funciones_registradas():
    func_ordensup()
    func_clist()
```

```
#####
```

El cambio anterior ocasiona que primero se ejecute la función *func_ordensup()* y luego *func_clist()*. Lo interesante de la figura 19 es que viene a reforzar la idea que, para una cantidad de elementos grande en una lista, la función de orden superior se ejecuta más rápido que la función que implementa el recurso *List Comprehension*. En las 4 ocasiones que la utilidad se ejecutó con el orden de las funciones intercambiadas, los valores de tiempos de la función de orden superior fueron mejores que la otra en cuestión.

Es necesario tener en cuenta aún que las pruebas de la obtención de tiempos de la utilidad se realizó en una laptop, no un ordenador científico, algo que sería lo ideal. Tampoco se realizó la ejecución de la utilidad con el orden de las funciones intercambiados en una máquina virtual. Hecho que teóricamente no debería cambiar

³³*idle* es el término que se da cuando algo está sin hacer nada a través del tiempo. En el caso del CPU se dice que está ocioso.

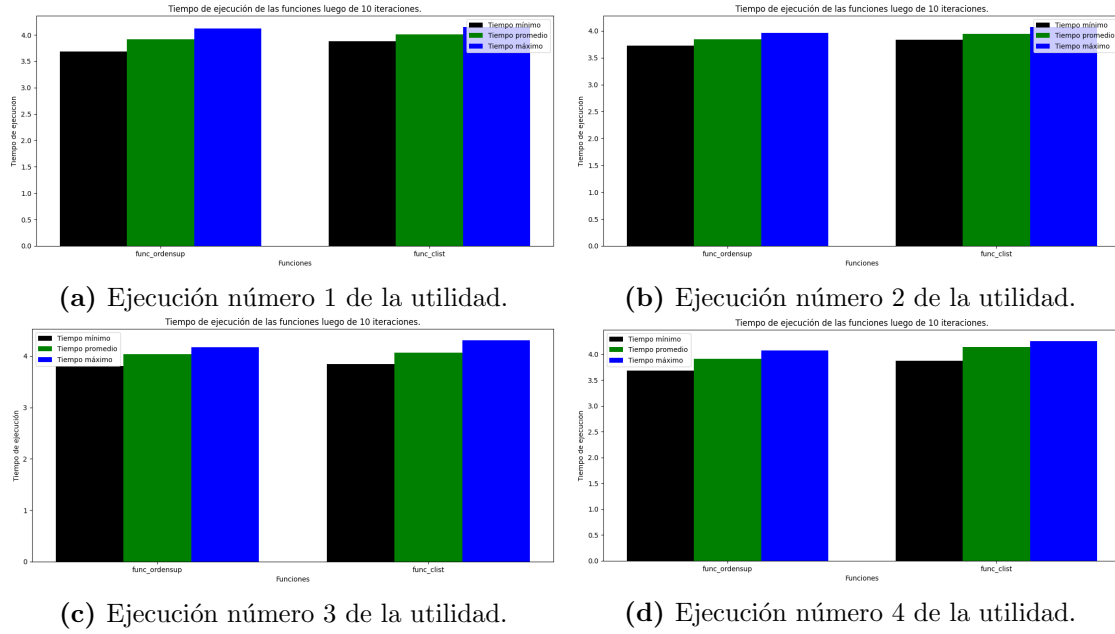


Figura 19: Gráficos de barra resultantes con el orden de ejecución de las funciones intercambiados.

mucho los resultados, pero sería una confirmación más de la idea que la función de orden superior obtiene mejores tiempos que la que usa *List Comprehension*. Sería una idea interesante realizar una utilidad que registre varios parámetros aparte del tiempo de ejecución y realizar comparaciones nuevas. Si se usa un ordenador con prestaciones muy superiores, entonces mejor.

3. Conclusiones

Python es un lenguaje de programación que ofrece muchas comodidades al desarrollador así como muchas herramientas. En este caso, fue imprescindible el uso del módulo *time* para registrar los tiempos de ejecución de las funciones utilizadas. Los tiempos de ejecución obtenidos por estas funciones fueron graficados y en varias partes de la monografía se evidenció que, a niveles superiores a los nanosegundos, existen funciones que realizan tareas de una forma más rápida que otras. Siempre y cuando la tarea sea la misma en ambos casos.

Las prestaciones del ordenador utilizado para llevar a cabo las ejecuciones eran bajas, por lo que no se puede establecer este hecho de forma concluyente y haría falta quizás más investigación. Incluso podría realizarse un estudio un tanto más extenso y profundo que el que suele presentarse en las monografías. Aún así, la utilidad que se presenta en esta monografía puede ser utilizada por los desarrolladores noveles para comprobar y conocer acerca de funciones y recursos de Python que se ejecutan más rápido que otros. El código de la utilidad estará siempre disponible para descarga mientras lo está el sitio online que lo contiene, en este caso, GitHub. Se presenta como una idea la extensión de la utilidad y como futuras características que se le podrían añadir figura el registro de otros recursos que también se consumen durante la ejecución de código. Estos recursos son la memoria y el nivel del uso del CPU.

4. Bibliografía

- <https://www.welcometothejungle.com/en/articles/btc-performance-python>
- <https://stackoverflow.com/questions/1694402/why-are-interpreted-languages-slow>
- <https://www.bitdegree.org/tutorials/python-vs-c-plus-plus/>
- <https://github.com/microsoft/vscode>
- https://en.wikipedia.org/wiki/Big_O_notation
- <https://www.khanacademy.org/computing/computer-science/algorithms/asymptotic-notation/a/big-o-notation>
- <https://en.wikipedia.org/wiki/Matplotlib>
- <https://try.github.io/>
- <https://opensource.com/resources/virtualization>

5. Anexos

5.1. Instalar VSCodium en Ubuntu 20.04.1 LTS

VSCodium se puede instalar de varias formas en Ubuntu 20.04.1 LTS. En este caso se describirá el proceso relacionado con la descarga del fichero **.deb** de instalación. A continuación los pasos necesarios a seguir:

1. Descargar el fichero de instalación desde el enlace el proyecto en GitHub
2. Una vez completada la descarga, se debe abrir la terminal (Ctrl + Alt + t) y posicionarse en la carpeta donde se almacenó el fichero de instalación descargado. En este caso usaremos como ejemplo la ruta **/nombre-pc/usuario/Downloads**
3. Ejecutar el comando **sudo dpkg -i codium_1.48.0-1597345748_amd64.deb**³⁴

```
■ laptop@home:~/Downloads$ sudo dpkg -i codium_1.48.0-1597345748_amd64.deb
```

4. Ejecutar el comando **sudo apt-get install -f** para resolver las dependencias en caso de que haga falta instalar alguna necesaria para el VSCodium

```
■ laptop@home:~/Downloads$ sudo apt-get install -f
```

5. Buscar VSCodium en las aplicaciones instaladas. Debe ejecutarse sin problemas como se muestra en la figura 4

5.2. Configurar VSCodium para programar en Python

VSCodium permite al desarrollador descargar los plugins desde el mismo IDE con una conexión a internet estable y sin restricciones de puerto. Sin embargo, en este caso se presentará la forma de configuración del plugin Python de forma manual. Los pasos son los siguientes:

1. Descargar el plugin Python con la extensión **.vsix** desde la página *marketplace*³⁵ seleccionando la opción **Download extension**
2. Una vez descargada la extensión con un nombre muy parecido a **ms-python.python-2020.8.101144.vsix**³⁶ se debe abrir la paleta de extensiones (Ctrl + Shift + X) y seleccionar la opción **Install from VSIX...**
3. Luego de instalar la extensión de forma satisfactoria solo queda crear un fichero con extensión **.py** y empezar a programar

³⁴El nombre del fichero `codium_1.48.0-1597345748_amd64.deb` está dado por la versión del instalador. Este nombre cambia a medida que se publica una nueva actualización del VSCodium.

³⁵<https://marketplace.visualstudio.com/items?itemName=ms-python.python>

³⁶El nombre de la extensión depende de la versión.

5.3. Código de la utilidad

Esta sección fue creada para que el lector no tenga que buscar en todo el documento para encontrar la dirección del repositorio de GitHub.

`https://github.com/ppdmartell/test_rapidez`