Python en Ámbitos Científicos

Facundo Batista & Manuel Carlevaro

15 de mayo de 2024

VERSIÓN PRELIMINAR

Título: Python en Ámbitos Científicos
Autores: Facundo Batista & Manuel Carlevaro
ISBN-13 (versión electrónica): ???-?-????-????-?

Primera Edición (versión preliminar)
Escrito con XeLaTeX.

Lugar: Olivos y La Plata, Buenos Aires, Argentina
Año: 2024
Web: https://github.com/facundobatista/libro-pyciencia

10 9 8 7 6 5 4 3 2 1

VERSIÓN PRELIMINAR

Prefacio

Cuando la Comunidad Científica notó que las computadoras podían facilitar su trabajo, sus integrantes se convirtieron posiblemente en los adoptantes tempranos más entusiastas de la tecnología. Aunque el uso y la potencia de las computadoras crece continuamente en el ámbito científico, poco se ha avanzado en la formación y desarrollo de habilidades en Desarrollo de Software.

El cálculo científico requiere realizar combinaciones de múltiples tareas de diversa clase. Por ejemplo, es necesario registrar automáticamente datos de un experimento y visualizarlos, realizar cálculos numéricos o simbólicos, ordenar, clasificar, simular, etc. Muchas veces podemos utilizar paquetes de software que realizan esas tareas por nosotros, pero muchas otras ocurre que nadie ha implementado un determinado cómputo en la forma que necesitamos, o simplemente queremos probar ideas nuevas. Cualquiera sea la naturaleza de nuestra actividad en la ciencia o en la tecnología, no resulta infrecuente la necesidad de interactuar con computadoras a través de programas propios.

El uso de Python en aplicaciones científicas ha aumentado sostenidamente en los últimos años, sin embargo es difícil encontrar libros o manuales en castellano de Python que no estén pensados para programadores.

Este libro nace con la idea primaria de acercar Python al mundo científico, en un libro pensado para científiques, a partir de nuestra experiencia en el dictado del curso "Herramientas Computacionales para Científicos"que ofrecemos en la Universidad Nacional de La Plata y la Universidad Tecnológica Nacional, desde 2007. De la misma manera, la elección del castellano como idioma de escritura es un factor crítico, porque aunque sabemos que el inglés es una herramienta fundamental tanto para programar como para hacer ciencia, estamos convencidos que no debería ser una barrera de entrada.

Más allá de las secciones básicas de un libro (índice, bibliografía, etc.), el libro tiene dos grandes partes. La primera habla de Python, algunas bibliotecas importantes y otros temas que son fundamentales. En la segunda, mostramos cómo abordar temas científicos básicos utilizando Python, de forma teórica y práctica.

En ambos casos este libro esquiva la pretensión de ser una referencia absoluta, sino que tiene el propósito de allanar el camino de les científiques para dar los primeros pasos en el lenguaje y solucionar los problemas básicos (pero no por eso menos importantes) de la ciencia y la tecnología.

Olivos y La Plata, Buenos Aires, Argentina,

Facundo Batista & Manuel Carlevaro

Índice general

Parte I
Herramientas fundamentales

Los capítulos siguientes desarrollan un conjunto de conocimientos y técnicas fundamentales que serán utilizados en la Parte II al abordar temas de aplicaciones específicas. Se sugiere, a los lectores que no tienen experiencia previa en el uso de Python, un recorrido secuencial a través de los capítulos que componen esta Parte.

1 | Sobre la velocidad de procesamiento

Los programas pueden (suelen) ser lentos. Muchas veces esto no es un problema, pero frecuentemente sucede que queremos o necesitamos que un determinado programa (o parte de código dentro de un programa) sea más rápido de lo que es.

Cuando nos enfrentamos a esta situación y decidimos optimizar el programa debemos primero recordar dos reglas de oro:

- Siempre tener el programa terminado y que funcione correctamente: no tiene sentido preocuparse por si va a ser lento cuando esté terminado si todavía falta para ello (no optimizar prematuramente) y confirmar (idealmente con pruebas de unidad como ya hablamos en ??) su correcta operatoria, ya que es más facil optimizar un código que funciona bien, que corregir ese código luego de que fue optimizado.
- Antes de realizar cualquier cambio en el código debemos descubrir en qué partes del mismo se pierde tiempo, porque no tiene sentido optimizar cualquier parte del programa. Por ejemplo, es mucho más útil mejorar en 100 milisegundos una función que se llama mil veces, que mejorar en un segundo una función que se ejecuta una sola vez.

Módulo	Versión
Cython	0.29.35
cffi	1.15.1
line-profiler	4.1.1
mypy	1.5.1
numba	0.57.0
pyinstrument	4.6.0
quantiphy	2.19
scalene	1.5.31.1
snakeviz	2.2.0

Código disponible

En general es muy difícil predecir en qué partes de un código vale la pena invertir tiempo y esfuerzo para optimizar, por lo que es imprescindible medir la ejecución del programa para detectar esas zonas.

En este capítulo veremos cómo realizar esas mediciones en tiempo de ejecución, ya sea de programas enteros (*profiling*) como de pequeños pedacitos de código, haremos foco en algunos casos patológicos que son facilmente optimizables, y finalmente veremos distintas técnicas a la hora de ir más allá de Python como lenguaje (si es realmente necesario).

1.1. Profiling

El perfilamiento (término que casi no se utiliza porque acostumbramos a mencionarlo en inglés, *profiling*) se basa en medir la ejecución de aplicaciones o programas para entender su comportamiento dinámico.

En otras palabras, no es información que se pueda obtener "inspeccionando" el código, sino que debemos ejecutarlo luego de haberlo instrumentado, obtener las distintas mediciones necesarias, y analizarlas para entender cuanto tiempo lleva la ejecución de cada parte.

Por supuesto, también podemos encontrar por inspección problemas de performance en el código (encontrando algoritmos con mal "O grande" [1] o descubriendo algunos de los casos patológicos que mencionaremos más adelante) pero revisar todo el programa en detalle lleva mucho tiempo y no es garantía de encontrar el problema. Con *profiling* podemos encontrar la zona de código culpable del comportamiento lento de nuestro programa y hacer foco ahí para realizar las optimizaciones necesarias.

A lo largo de esta sección veremos distintas herramientas para hacer *profiling*, basándonos en un programa ejemplo. Este programa fue realizado tratando de evitar cualquier optimización que salga naturalmente; casi podemos decir que a propósito cometimos varios errores inocentes, para luego poder mostrar que aunque hay varias partes del código que intuitivamente veríamos de optimizar, en realidad siempre debemos medir para luego saber donde enfocarnos (justamente, hacer *profiling*).

La idea del programa es analizar un texto grande y mostrar las 10 palabras que aparecen con mayor frecuencia. Como texto fuente usamos la obra del Don Quijote, que podemos descargar de aquí.

El programa está estructurado en una función principal y tres auxiliares.

```
51 def word_count():
52     with open("quijote.txt", "rt", encoding="utf8") as fh:
53         text = fh.read()
54     all_lines = _filter_lines(text)
55     count_words, count_values = _count_words(all_lines)
56     _show_top10(count_words, count_values)
```

Esta función lee todo el texto desde el archivo y luego llama a cada una de las funciones auxiliares para filtrar las líneas útiles, contar las palabras y mostrar las 10 principales.

```
43 def _filter_lines(text):
44     all_lines = text.split("\n")
45     start_pos = all_lines.index("El ingenioso hidalgo don Quijote de la Mancha")
46     end_pos = all_lines.index("Fin")
47     all_lines = all_lines[start_pos:end_pos + 1]
48     return all_lines
```

Es necesario filtrar el contenido porque el archivo que bajamos tiene toda una sección al principio y al final que no es el texto puro del libro, que es lo que queremos procesar. Por eso la función, luego de separar el texto en líneas, toma sólo aquellas que van desde el título del libro hasta la marca de "fin".

```
21 def _count_words(all_lines):
22     count_words = []
23     count_values = []
24     for line in all_lines:
25         for word in line.split():
26 
27             for sign in SIGNS:
28                 word = word.replace(sign, "")
29             word = word.lower()
30 
31             try:
```

[1] la notación O grande, o *Big O* en inglés, es una herramienta para describir la complejidad en tiempo de un algoritmo, mostrando cómo se comporta ese algoritmo cuando crecen los datos

```
32                word_position = count_words.index(word)
33            except ValueError:
34                count_words.append(word)
35                count_values.append(1)
36            else:
37                old_value = count_values[word_position]
38                count_values[word_position] = old_value + 1
39
40    return count_words, count_values
```

Para contar las palabras armamos dos listas para esas palabras que vamos encontrando y sus ocurrencias. Luego recorremos las líneas, separando cada una en palabras (limpiando cada una de signos de puntuación y poniéndola en minúsculas) y las contamos. El algoritmo para contar es el siguiente: buscamos la palabra en la lista de palabras que ya tenemos, si está presente (la llamada a `index` nos devuelve la posición) obtenemos el número en la posición correspondiente de la otra lista y le sumamos 1, pero si no está presente (se generó una excepción de tipo `ValueError`) la agregamos al final de la lista de palabras y agregamos el número 1 en la lista de valores de conteo.

```
4  def _show_top10(count_words, count_values):
5      print("Top 10:")
6      for selection_round in range(1, 11):
7          maxvalue = 0
8          maxpos = None
9          for pos in range(len(count_values)):
10             value = count_values[pos]
11             if value > maxvalue:
12                 maxvalue = value
13                 maxpos = pos
14
15         word = count_words[maxpos]
16         print(f"{selection_round:2d}. {maxvalue:5d} {word}")
17         del count_words[maxpos]
18         del count_values[maxpos]
```

Esta función recibe directamente las dos listas que se generaron en el proceso de conteo. Para mostrar las diez principales va a realizar el mismo procedimiento diez veces, en cada caso encuentra el valor máximo de la lista de conteos, y luego informa ese valor y la palabra de la misma posición; antes de la próxima iteración retira de la lista el valor y palabras informados, para así encontrar el próximo par.

Volvemos a recalcar lo absurdo de estos algoritmos, por favor no hagan esto en sus casas.

Veamos entonces cómo se comporta, ejecutando nuestro programa con Python, y a la vez ejecutando todo con `time`, una herramienta que luego de la ejecución del programa nos informará cuanto tiempo pasó (nos da tres valores, de los cuales nos interesa el `real`, que es el tiempo que usó el proceso):

```
$ time python3 quijote-1.py
Top 10:
 1. 20626 que
 2. 18214 de
 3. 18188 y
 4. 10363 la
 5.  9823 a
 6.  8241 en
 7.  8210 el
 8.  6335 no
 9.  4748 los
```


VERSIÓN PRELIMINAR

```
10.  4690 se

real    0m5.871s
user    0m5.854s
sys     0m0,015s
```

Sospechamos fuertemente que el programa podría ser más rápido.

1.1.1. Encontrando dónde un programa es lento

Veamos entonces cómo podemos encontrar en qué zonas del programa debemos trabajar para mejorar su rendimiento. En esta subsección mostraremos tanto herramientas que vienen en la biblioteca estándar como algunas que hay que instalar previamente; excepto algunos casos puntuales donde mencionaremos una alternativa, la forma de instalación será usando `pip` (ya que están publicadas en PyPI) idealmente en un entorno virtual **??**.

La biblioteca estándar de Python trae dos herramientas para hacer *profiling*, cProfile y profile. ¿Por qué dos? En realidad el que normalmente vamos a utilizar es `cProfile` ya que es el que menos carga extra le va a poner a nuestra ejecución cuando mide, mientras que `profile` se utiliza en casos donde se quiera especializar y cambiar el comportamiento de la herramienta en sí.

Es momento de comentar la diferencia entre perfiladores determinísticos y estadísticos. Un perfilador determinístico va a medir todos los pasos de ejecución del programa, generando resultados completos, pero con la desventaja de que la medición en sí afecta el tiempo total de ejecución, lo que en algunos casos es inviable. Por eso nos interesa que el perfilador en sí esté lo mas optimizado posible, y generalmente usamos `cProfile` en lugar de `profile` (ambos determinísticos). Por otro lado, un perfilador estadístico realiza mediciones sólo periódicamente, haciendo un muestreo estadístico para obtener los resultados, que en este caso pueden no ser los más precisos pero suficientes para acotar el problema; la ventaja de este tipo de perfiladores es que alteran menos el tiempo total de ejecución.

Volvamos al `cProfile`. La forma más sencilla de correr este perfilador es invocarlo como módulo en la línea de comandos indicándole el programa a ejecutar:

```
$ python3 -m cProfile quijote-1.py
Top 10:
 1. 20626 que
 2. 18214 de
 3. 18188 y
 4. 10363 la
 5.  9823 a
 6.  8241 en
 7.  8210 el
 8.  6335 no
 9.  4748 los
10.  4690 se
         7326537 function calls in 6.884 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.000    0.000 __init__.py:43(normalize_encoding)
        1    0.000    0.000    0.000    0.000 __init__.py:71(search_function)
        1    0.000    0.000    0.000    0.000 codecs.py:260(__init__)
        1    0.000    0.000    0.000    0.000 codecs.py:309(__init__)
        1    0.000    0.000    0.003    0.003 codecs.py:319(decode)
        1    0.000    0.000    0.000    0.000 codecs.py:94(__new__)
        1    0.001    0.001    6.884    6.884 quijote-1.py:1(<module>)
        1    0.957    0.957    6.858    6.858 quijote-1.py:22(_count_words)
```


VERSIÓN PRELIMINAR

```
      1    0.000    0.000    0.005    0.005 quijote-1.py:44(_filter_lines)
      1    0.012    0.012    0.012    0.012 quijote-1.py:5(_show_top10)
      1    0.000    0.000    6.883    6.883 quijote-1.py:52(word_count)
      1    0.000    0.000    0.000    0.000 utf_8.py:33(getregentry)
      1    0.000    0.000    0.000    0.000 {built-in method __new__ of type object at 0x556ac58c99a0}
      1    0.002    0.002    0.002    0.002 {built-in method _codecs.utf_8_decode}
      1    0.000    0.000    0.000    0.000 {built-in method builtins.__import__}
      1    0.000    0.000    6.884    6.884 {built-in method builtins.exec}
      2    0.000    0.000    0.000    0.000 {built-in method builtins.isinstance}
     10    0.000    0.000    0.000    0.000 {built-in method builtins.len}
     11    0.000    0.000    0.000    0.000 {built-in method builtins.print}
      1    0.000    0.000    0.000    0.000 {built-in method io.open}
      1    0.000    0.000    0.000    0.000 {method '__exit__' of '_io._IOBase' objects}
  45904    0.003    0.000    0.003    0.000 {method 'append' of 'list' objects}
      1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
      2    0.000    0.000    0.000    0.000 {method 'get' of 'dict' objects}
 381219    5.445    0.000    5.445    0.000 {method 'index' of 'list' objects}
      4    0.000    0.000    0.000    0.000 {method 'isalnum' of 'str' objects}
      4    0.000    0.000    0.000    0.000 {method 'isascii' of 'str' objects}
      1    0.000    0.000    0.000    0.000 {method 'join' of 'str' objects}
 381217    0.031    0.000    0.031    0.000 {method 'lower' of 'str' objects}
      1    0.004    0.004    0.007    0.007 {method 'read' of '_io.TextIOWrapper' objects}
6480689    0.401    0.000    0.401    0.000 {method 'replace' of 'str' objects}
  37454    0.024    0.000    0.024    0.000 {method 'split' of 'str' objects}
```

Al ejecutarlo encontramos que primero se muestra la salida normal que ya habíamos visto arriba, y luego todo un reporte del `cProfile`: nos indica que se llamaron funciones más de siete millones de veces en casi siete segundos. Cabe destacar que el tiempo total de ejecución subió un 17 % entre la ejecución inicial y la realizada al obtener mediciones, lo cual se explica por la carga extra que mencionamos antes.

Luego tenemos una serie de columnas:

- `ncalls`: la cantidad de veces que se llamó a esa función.
- `tottime`: el tiempo total usado por esa función, sin contar el tiempo usado por llamadas a sub-funciones.
- `percall`: el tiempo por llamada (o sea, `tottime` dividido `ncalls`).
- `cumtime`: el tiempo usado por esa función ahora sí incluyendo las llamadas a sub-funciones (este número es preciso incluso para funciones recursivas)
- `percall`: `cumtime` dividido `ncalls`
- `filename:lineno(function)`: de qué función está hablando (las filas están ordenadas por esta columna)

A primera vista es difícil entender toda esta información. Mejoremos el listado ordenándolo por el tiempo acumulado de la función (y manualmente sacando aquellos con ese valor en cero, por legibilidad en el libro, así como también dejamos de mostrar el resultado del programa en sí):

```
$ python3 -m cProfile -s cumtime quijote-1.py
(...)
         7326537 function calls in 6.905 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    6.905    6.905 {built-in method builtins.exec}
```

```
        1    0.001    0.001    6.905    6.905 quijote-1.py:1(<module>)
        1    0.000    0.000    6.904    6.904 quijote-1.py:52(word_count)
        1    0.956    0.956    6.879    6.879 quijote-1.py:22(_count_words)
   381219    5.466    0.000    5.466    0.000 {method 'index' of 'list' objects}
  6480689    0.403    0.000    0.403    0.000 {method 'replace' of 'str' objects}
   381217    0.031    0.000    0.031    0.000 {method 'lower' of 'str' objects}
    37454    0.024    0.000    0.024    0.000 {method 'split' of 'str' objects}
        1    0.012    0.012    0.012    0.012 quijote-1.py:5(_show_top10)
        1    0.004    0.004    0.007    0.007 {method 'read' of '_io.TextIOWrapper' objects}
        1    0.000    0.000    0.005    0.005 quijote-1.py:44(_filter_lines)
    45904    0.003    0.000    0.003    0.000 {method 'append' of 'list' objects}
        1    0.000    0.000    0.002    0.002 codecs.py:319(decode)
        1    0.002    0.002    0.002    0.002 {built-in method _codecs.utf_8_decode}
(...)
```

De arriba para abajo, vemos que el total se lo lleva el `exec` (que usa `cPython` mismo para ejecutar todo el script) y el módulo en sí al ser ejecutado. Luego ya vemos las funciones que tenemos en el código: `word_count` se lleva el tiempo total, y justo después tenemos la primer sorpresa con `_count_words`, donde se pierden 6.879 segundos, el 99.6 % del tiempo total.

Esta información es justamente la que vuelve valiosa al perfilamiento: nos podemos olvidar de las otras funciones y enfocarnos en `_count_words`.

Habiendo dicho eso, llama la atención una línea por su gran cantidad de llamadas, la correspondiente al método `replace` de las cadenas. Vemos que el único `replace` que usamos es (¡en `_count_words`!) para sacar los signos de puntuación de cada palabra. Nos damos cuenta que en lugar de llamar a ese replace por cada palabra, podríamos sacar los signos directamente de toda la línea… o incluso de todo el texto a la vez. Claro, no sabemos si será eficiente hacer un reemplazo de todo el texto en un sólo paso, pero esa es la idea: hacemos el cambio que nos parece correcto, y volvemos a ejecutar el perfilamiento para entender si mejoró o empeoró.

Movemos entonces estos reemplazos (y el convertirlo a minúscula) a la función `word_count`, obteniendo la segunda versión de este programa (mostrando aquí sólo las dos funciones afectadas):

```
21 def _count_words(all_lines):
22     count_words = []
23     count_values = []
24     for line in all_lines:
25         for word in line.split():
26             try:
27                 word_position = count_words.index(word)
28             except ValueError:
29                 count_words.append(word)
30                 count_values.append(1)
31             else:
32                 old_value = count_values[word_position]
33                 count_values[word_position] = old_value + 1
34
35     return count_words, count_values
```

```
46 def word_count():
47     with open("quijote.txt", "rt", encoding="utf8") as fh:
48         text = fh.read()
49
50     for sign in SIGNS:
51         text = text.replace(sign, "")
52     text = text.lower()
53
```

```
    all_lines = _filter_lines(text)
    count_words, count_values = _count_words(all_lines)
    _show_top10(count_words, count_values)
```

Volvemos a perfilar:

```
$ python3 -m cProfile -s cumtime quijote-2.py
(...)
         464645 function calls in 5.641 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    5.641    5.641 {built-in method builtins.exec}
        1    0.001    0.001    5.641    5.641 quijote-2.py:1(<module>)
        1    0.000    0.000    5.640    5.640 quijote-2.py:47(word_count)
        1    0.115    0.115    5.588    5.588 quijote-2.py:22(_count_words)
   381217    5.452    0.000    5.452    0.000 {method 'index' of 'list' objects}
    37454    0.022    0.000    0.022    0.000 {method 'split' of 'str' objects}
       17    0.021    0.001    0.021    0.001 {method 'replace' of 'str' objects}
        1    0.012    0.012    0.012    0.012 quijote-2.py:5(_show_top10)
        1    0.005    0.005    0.008    0.008 {method 'read' of '_io.TextIOWrapper' objects}
        1    0.007    0.007    0.007    0.007 {method 'lower' of 'str' objects}
        1    0.000    0.000    0.005    0.005 quijote-2.py:39(_filter_lines)
        1    0.000    0.000    0.003    0.003 codecs.py:319(decode)
        1    0.003    0.003    0.003    0.003 {built-in method _codecs.utf_8_decode}
    45902    0.003    0.000    0.003    0.000 {method 'append' of 'list' objects}
(...)
```

Ahora vemos que `replace` y `lower` todavía están, pero se ejecutan muchísimo menos que en la versión anterior y ya no aportan al tiempo total. Encontramos que `_count_words` sigue siendo muy cara, y la explicación está en la gran cantidad de llamadas a `index` (el 96 % del tiempo total), que se usa en esa función. La razón subyacente es que estamos usando `index` para el conteo de palabras, y buscar en una lista es potencialmente muy costoso (porque va comparando el objeto buscado posición por posición). Pensemos una mejor forma para guardar la cantidad de veces que encontremos cada palabra: un diccionario, donde podemos tener la palabra como clave y la cantidad como valor, teniendo en cuenta que el acceso por clave a un diccionario es extremadamente rápido.

Mejoremos el código, obteniendo la versión 3.

```
def _show_top10(count):
    print("Top 10:")
    for selection_round in range(1, 11):
        maxvalue = 0
        maxword = None
        for word, value in count.items():
            if value > maxvalue:
                maxvalue = value
                maxword = word

        print(f"{selection_round:2d}. {maxvalue:5d} {maxword}")
        del count[maxword]


def _count_words(all_lines):
    count = {}
    for line in all_lines:
        for word in line.split():
```

```
            try:
                value = count[word]
            except KeyError:
                value = 0
            count[word] = value + 1

    return count
```

No sólo modificamos el conteo en sí, que ahora busca cada palabra y guarda su cantidad en un diccionario, sino que este diccionario es el que recibe `_show_top10`, donde mantuvimos el algoritmo "absurdo" original pero ahora trabajando sobre esta nueva estructura.

Veamos cómo se comporta esta nueva versión:

```
(...)
$ python3 -m cProfile -s cumtime quijote-3.py
         37532 function calls in 0.124 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.124    0.124 {built-in method builtins.exec}
        1    0.001    0.001    0.124    0.124 quijote-3.py:1(<module>)
        1    0.000    0.000    0.123    0.123 quijote-3.py:40(word_count)
        1    0.059    0.059    0.075    0.075 quijote-3.py:19(_count_words)
       17    0.021    0.001    0.021    0.001 {method 'replace' of 'str' objects}
    37454    0.020    0.000    0.020    0.000 {method 'split' of 'str' objects}
        1    0.008    0.008    0.008    0.008 quijote-3.py:5(_show_top10)
        1    0.005    0.005    0.008    0.008 {method 'read' of '_io.TextIOWrapper' objects}
        1    0.007    0.007    0.007    0.007 {method 'lower' of 'str' objects}
        1    0.000    0.000    0.005    0.005 quijote-3.py:32(_filter_lines)
        1    0.000    0.000    0.003    0.003 codecs.py:319(decode)
        1    0.003    0.003    0.003    0.003 {built-in method _codecs.utf_8_decode}
(...)
```

¿Podemos seguir mejorando el código? Seguramente. ¿Vale la pena? Quizás no, ya que el tiempo total de ejecución es de alrededor de 100 milisegundos.

Habiendo dicho eso, este código puntual sí merece una mejora, que es la de reemplazar la función de búsqueda propia y el algoritmo para mostrar los primeros 10, utilizando para ello el objeto `Counter` del módulo `collections`, no sólo porque es apenas más rápido, sino porque estamos reemplazando bastante código hecho por nosotros para nuestro programa por funcionalidad que ya viene en la biblioteca estándar, que está mucho más probada y optimizada. Menos código nuestro significa menos fallas ahora y menos mantenimiento en el futuro. Les dejamos como desafío luego de la lectura el adaptar la versión 3 de esta manera (pero si quieren espiar el resultado, busquen la versión 4 del código).

Volviendo al perfilamiento, el ejercicio que hicimos en el libro es válido pero sólo un ejemplo sencillo. Si el programa fuese más complejo y largo, el reporte total puede volverse bastante confuso.

Una forma de mejorar esto es, si ya tenemos detectada la función que se lleva la mayor parte del tiempo, realizar el *profiling* solo allí. Adaptemos de esta manera la versión 1 original del código, dando lugar a una versión 5; mostramos aquí sólo el cambio, que es en la función principal:

```
def word_count():
    with open("quijote.txt", "rt", encoding="utf8") as fh:
        text = fh.read()
    all_lines = _filter_lines(text)
```


VERSIÓN PRELIMINAR

```
58
59      profiler = cProfile.Profile()
60      count_words, count_values = profiler.runcall(_count_words, all_lines)
61      profiler.print_stats(sort=pstats.SortKey.CUMULATIVE)
62
63      _show_top10(count_words, count_values)
```

Aquí vemos que en lugar de llamar directamente a `_count_words` pasándole `all_lines`, le indicamos a un *profiler* que realice esa ejecución con ese argumento; a este *profiler* lo creamos en la línea anterior, y luego también lo usamos para mostrar los resultados, indicándole que ordene de la mejor manera que nos parezca.

Probamos esta versión, entonces:

```
$ python3 quijote-5.py
         7326478 function calls in 6.841 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.950    0.950    6.841    6.841 quijote-5.py:24(_count_words)
   381217    5.437    0.000    5.437    0.000 {method 'index' of 'list' objects}
  6480689    0.400    0.000    0.400    0.000 {method 'replace' of 'str' objects}
   381217    0.030    0.000    0.030    0.000 {method 'lower' of 'str' objects}
    37453    0.020    0.000    0.020    0.000 {method 'split' of 'str' objects}
    45900    0.003    0.000    0.003    0.000 {method 'append' of 'list' objects}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
(...)
```

Para entender resultados complejos también podemos visualizarlos de una forma más gráfica. Los perfiladores de la biblioteca estándar permiten exportar los resultados para luego procesarlos en diferido con el módulo `pstats` (también de la biblioteca estándar) o levantarlos con otras herramientas. Las dos más conocidas para esto son SnakeViz (específica para los resultados del perfilador de Python) y KCachegrind (visualizador genérico, originalmente pensado para los resultados de la herramienta Valgrind).

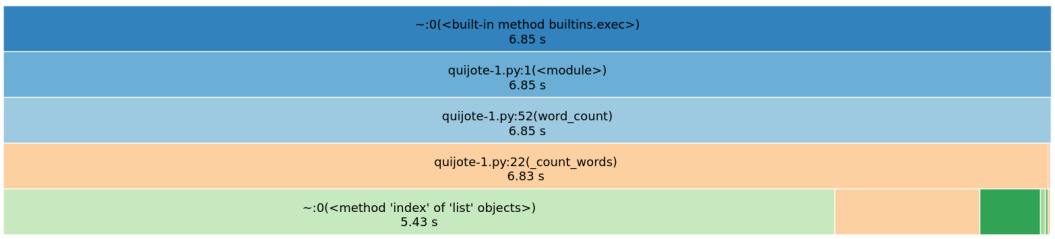
Entonces el primer paso es correr el perfilador pero guardar los resultados en lugar de que los muestre por pantalla (para simular una operatoria normal volvemos a la versión original de nuestro programa):

```
$ python3 -m cProfile -o quijote.prof quijote-1.py
(...)
```

Le indicamos a SnakeViz que procese directamente ese archivo que generamos:

```
(env) $ snakeviz quijote.prof
```

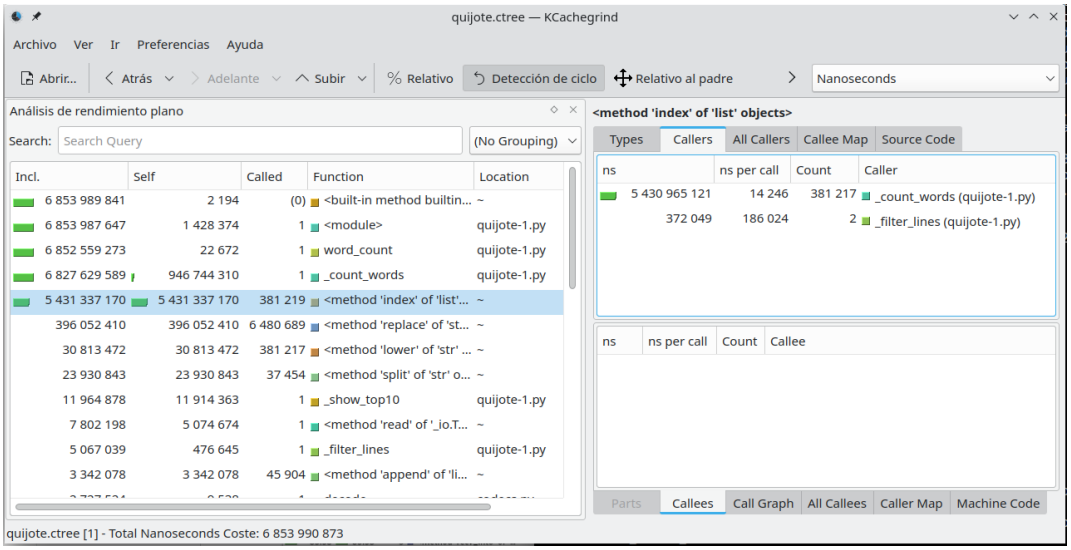
SnakeViz abrirá una página local en nuestro navegador mostrando un gráfico bastante útil porque nos permite ver donde se pierde tiempo en cada nivel de la pila de llamadas (y además nos permite controlar un poco qué vemos y cómo). Para el caso nuestro el gráfico es:



Por otro lado, para KCachegrind primero tenemos que convertir el formato de los datos fuente, lo cual hacemos con otra herramienta, `pyprof2calltree`. En Debian/Ubuntu ambas se instalan con `apt`.

```
$ pyprof2calltree -i quijote.prof -o quijote.ctree
$ kcachegrind quijote.ctree
```

En este caso se nos abrirá un programa de escritorio donde podemos interactuar bastante con las distintas mediciones y filtrarlas. Por ejemplo si a la izquierda elegimos la línea correspondiente a la función `_count_words` a la derecha veremos la información reducida para esa función (lo mismo que vimos en el caso de arriba donde sólo perfilamos dicha función), o si elegimos las llamadas a `index` veremos en dónde se realizaron esas llamadas, como se ve en la siguiente captura de pantalla:



Una forma alternativa de ver los resultados es línea por línea del código, y ese es el formato en que muestra los resultados un perfilador alternativo (si lo usamos de la forma correcta): el `line_profiler`. La utilidad que usamos es `kernprof`, pero antes de ejecutarla debemos decorar la función que queremos perfilar, agregándole un `@profile` antes de su definición (al cual no hay que importar, se encarga el perfilador mismo de proveerlo). Entonces:

```
(env) $ kernprof -lv quijote-1.py
(...)
Wrote profile results to quijote-1.py.lprof
Timer unit: 1e-06 s

Total time: 8.36627 s
File: quijote-1.py
Function: _count_words at line 22

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    22                                           @profile
    23                                           def _count_words(all_lines):
    24         1          1.1      1.1      0.0      count_words = []
    25         1          0.3      0.3      0.0      count_values = []
    26     37454       5414.8      0.1      0.1      for line in all_lines:
    27    418670      99311.0      0.2      1.2          for word in line.split():
    28
    29   6861906     963742.4      0.1     11.5              for sign in SIGNS:
    30   6480689    1498922.1      0.2     17.9                  word = word.replace(sign, "")
    31    381217      87598.5      0.2      1.0              word = word.lower()
```

```
32
33    381217     43142.2      0.1      0.5            try:
34    381217   5497893.0     14.4     65.7                word_position = count_words.index(word)
35     22950      6517.9      0.3      0.1            except ValueError:
36     22950      6013.4      0.3      0.1                count_words.append(word)
37     22950      7600.2      0.3      0.1                count_values.append(1)
38                                                   else:
39    358267     67836.1      0.2      0.8                old_value = count_values[word_position]
40    358267     82275.2      0.2      1.0                count_values[word_position] = old_value + 1
41
42         1         0.7      0.7      0.0        return count_words, count_values
```

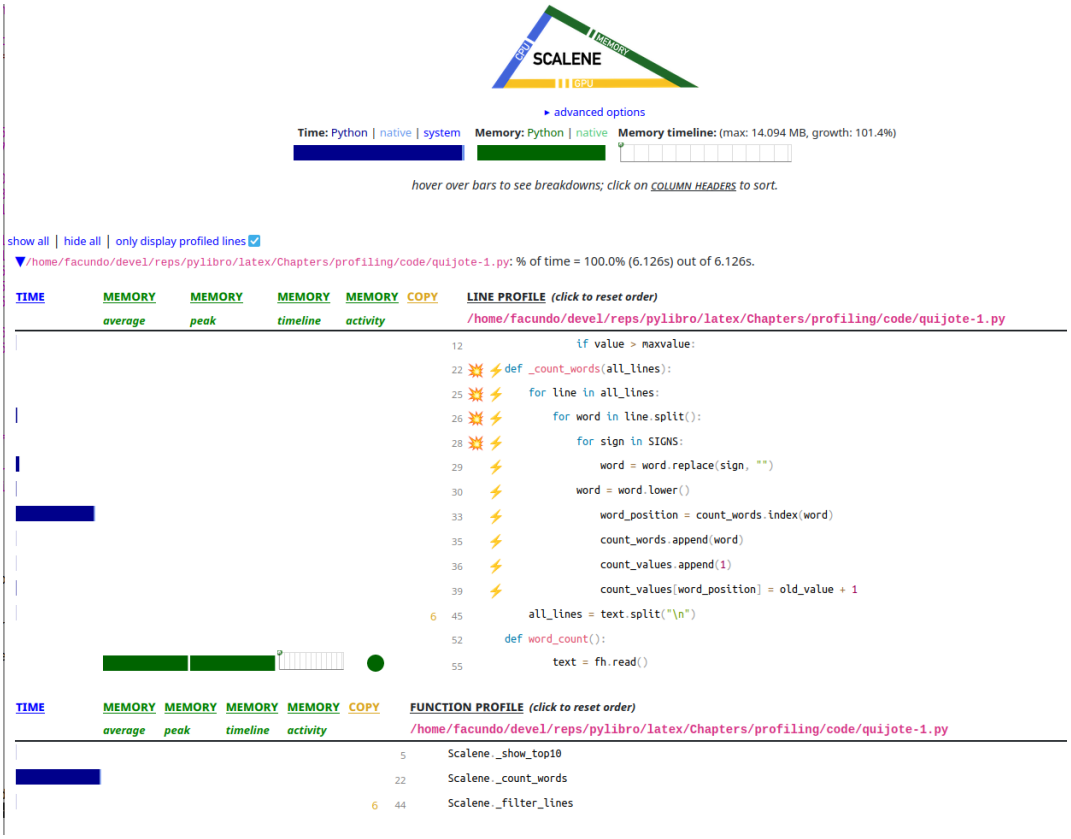
Vemos en este formato de salida que es fácilmente detectable en qué línea(s) de nuestro programa tenemos que poner atención para mejorar el rendimiento.

Otro perfilador muy útil es Scalene. En este caso, antes de instalar la herramienta con `pip` debemos asegurarnos que el sistema esté configurado correctamente (lo que cambia de plataforma a plataforma, ver detalle). El diferencial de este perfilador es que no sólo analiza el uso del procesador general, sino también del consumo de memoria e incluso de la GPU si nuestro programa corre en parte en una placa de video.

La utilización es sencilla:

```
(env) $ scalene quijote-1.py
```

Al terminar abrirá una página web local mostrando los resultados que nos permitirá interactuar de diversas maneras. Para nuestro ejemplo tenemos:



Finalmente, veamos un perfilador que trabaja de forma estadística (lo cual no es necesario para nuestro ejemplo simple pero es vital si el tiempo de ejecución de aquello que queremos perfilar es muy alto): `pyinstrument`.

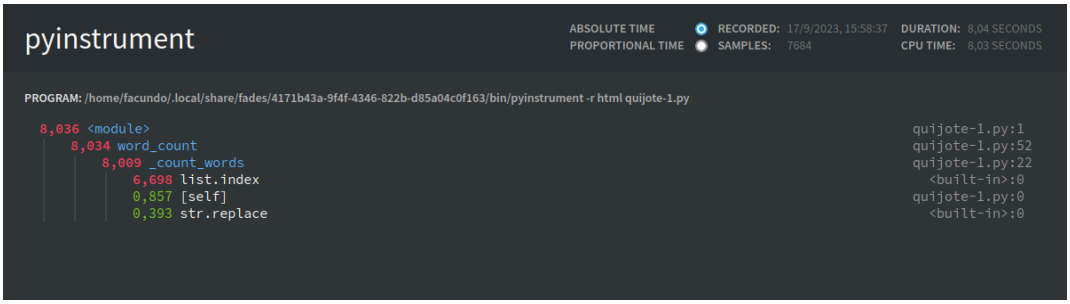
VERSIÓN PRELIMINAR

```
$ pyinstrument quijote-1.py
(...)
  _     ._   __/__   _ _  _  _ _/_   Recorded: 20:54:28  Samples:  8048
 /_//_/// /_\ / //_// / //_'/ //     Duration: 8.479     CPU time: 8.477
/   _/                      v4.5.3

Program: pyinstrument quijote-1.py

8.479 <module>  quijote-1.py:1
└─ 8.478 word_count  quijote-1.py:52
   └─ 8.452 _count_words  quijote-1.py:22
      ├─ 7.217 list.index  <built-in>
      ├─ 0.816 [self]  quijote-1.py
      └─ 0.366 str.replace  <built-in>
```

El resultado por terminal es muy interesante porque de forma concisa nos está indicando los puntos donde debemos hacer foco. Alternativamente podemos pedirle que nos entregue el reporte en una página web:

```
$ pyinstrument -r html quijote-1.py
```

Para nuestro ejemplo, entonces:



1.1.2. Midiendo pequeñas partes de código

Cuando estamos optimizando, muchas veces sucede que encontramos que el punto a optimizar son determinadas líneas que se ejecutan muy frecuentemente, cómo usamos alguna estructura de datos, etc.

Podríamos encarar esa situación cambiando el código o la estructura y volviendo a evaluar si el programa continúa lento, con *profiling* incluido si es el caso, hasta entender si los cambios que realizamos surtieron efecto o no, de formar similar como fuimos haciendo a través de los ejemplos en el comienzo de la sección.

Pero hay una mejor manera de realizar ese proceso, y es medir directamente si el cambio que tenemos pensado efectivamente mejoraría la situación. El inconveniente es que cuanto más pequeña hagamos la medición, más se afectará por el contexto que no controlamos. En otras palabras, si evaluamos la ejecución de nuestro programa que tarda algunos minutos podemos encontrar los puntos calientes sin que variaciones temporales del sistema influyan, pero si vamos a comparar una u otra forma de escribir un algoritmo que en su totalidad tarda algunos milisegundos, cualquier imprevisto del sistema donde estamos midiendo nos puede afectar el resultado.

Para evitar caer en trampas conocidas como carga del sistema, manejo de memoria, otros procesos concurrentes, etc, Python trae una utilidad muy práctica que se ocupa de minimizar todas esas variaciones: `timeit`. Con esta herramienta podemos medir pequeños pedazos de código muy puntuales, entendiendo en qué caso tal o cual forma de escribir un algoritmo es más rápido.

VERSIÓN PRELIMINAR

Por supuesto, es importante entender que estas mediciones donde ganamos milisegundos de un caso al otro sólo tienen sentido si van a ser utilizadas intensamente en el programa real. Si no, realmente será una mejora que no nos afectará el comportamiento general.

Por ejemplo, en nuestro ejemplo antes vimos que era costoso buscar en una lista, y queremos trabajar en ese punto. Se nos ocurren varias ideas, y queremos ver si es más rápido buscar un elemento que no existe en una lista, una tupla, o un diccionario de 100 elementos...

```
>>> import timeit
>>> timeit.timeit("555 in vals", setup="vals = list(range(100))")
0.4607653420243878
>>> timeit.timeit("555 in vals", setup="vals = tuple(range(100))")
0.4076314979849849
>>> timeit.timeit("555 in vals", setup="vals = dict.fromkeys(range(100))")
0.05606273500598036
```

Varios detalles a tener en cuenta en la secuencia mostrada. Por un lado el pedacito de código (que en ingles llamamos *snippet*) va entre comillas, para que no se ejecute antes de la llamada sino cuando esté siendo medido. Como nos interesa medir solamente la búsqueda, no creamos la lista, tupla o diccionario en el primer pedacito de código, sino que lo hacemos en otro código de `setup`. Alternativamente podríamos haberlo creado antes de llamar a `timeit` pero tendríamos que haberle pasado esos objetos en un diccionario con el parámetro `globals` para que lo encuentre como variable. Tengamos en cuenta que el código de `setup` se ejecuta una sola vez para toda la medición, mientras que el que se mide se ejecuta muchísimas veces; esto puede ser importante en la medición si el código medido modifican las estructuras armadas en `setup`. Dejamos para la subsección siguiente el porqué de la diferencia de tiempos observada.

El módulo `timeit` nos da la facilidad de usarlo directamente desde la línea de comandos. Lo anterior lo podríamos reescribir como:

```
$ python3 -m timeit -s "vals = list(range(100))" "555 in vals"
500000 loops, best of 5: 428 nsec per loop
$ python3 -m timeit -s "vals = tuple(range(100))" "555 in vals"
1000000 loops, best of 5: 391 nsec per loop
$ python3 -m timeit -s "vals = dict.fromkeys(range(100))" "555 in vals"
20000000 loops, best of 5: 18.7 nsec per loop
```

Esta forma de ejecución puede ser un poco más trabajosa porque hay que definir siempre todo el contexto (en la cadena que pasamos con `-s` para que no entre en la medición) pero tiene la ventaja que es seguro que no depende de otro contexto que podemos usar accidentalmente.

Los Jupyter Notebook nos dan la facilidad de invocar a `timeit` de forma sencilla, con el detalle que **no** tenemos que poner lo que queremos medir entre comillas sino directamente:

Cell 01

```
vals = list(range(100))
vals_t = tuple(vals)
vals_d = dict.fromkeys(vals)

%timeit 555 in vals
%timeit 555 in vals_t
%timeit 555 in vals_d
```

```
439 ns ± 1.59 ns per loop (mean ± std. dev. of 7 runs, 1,000,000 loops each)
383 ns ± 0.971 ns per loop (mean ± std. dev. of 7 runs, 1,000,000 loops each)
37.2 ns ± 0.0651 ns per loop (mean ± std. dev. of 7 runs, 10,000,000 loops each)
```

Más allá de la forma de invocar al módulo, lo importante es entender que no importa la medición en sí, sino que es un accesorio que nos permite entender qué está sucediendo de forma subyacente. Lo ideal sería armar un modelo mental de cómo se está ejecutando el programa con las distintas estructuras de datos y luego confirmar o refutar el modelo que armamos utilizando la medición.

1.1.3. Usando Python de forma más eficiente

Hay distintas situaciones donde el código es lento por cómo usamos alguna estructura de datos o armamos el programa en sí, como el caso que mencionamos al final de la subsección anterior de buscar en una lista, cuya duración depende del largo de esa lista.

Es útil conocer los casos más evidentes para evitar caer en esas pequeñas trampas, entendiendo que si en nuestro código hacemos algo de esto una, dos o diez veces no es problema, pero si vamos a tener bucles grandes sí podemos tener algo para corregir.

En esta subsección presentamos varios casos, y aunque no son todos los casos posibles, sí son los más conocidos.

En lugar de usar el `%timeit` de Jupyter Notebook armamos una función para medir, principalmente para poder capturar los valores y compararlos, y de yapa nos da la ventaja de ser explícitos con respecto al contexto.

Cell 02

```
import timeit
from quantiphy import Quantity

Quantity.set_prefs(prec=2)


def measure(case1, case2, globals=None):
    delays = []
    for case in (case1, case2):
        t = timeit.Timer(case, globals=globals)
        number, _ = t.autorange()
        total_delay = t.timeit(number=number)
        delays.append(Quantity(total_delay / number, 's'))

    d1, d2 = delays
    print(f"Primero:  {d1}")
    print(f"Segundo:  {d2}")
    speedup = d1 / d2
    print(f"Mejora:   {speedup:.2f} X")
```

La función que armamos recibe los dos casos a comparar y opcionalmente un espacio de nombre con los objetos que necesitan esos casos. Luego, para cada uno, instancia un objeto `Timer` con el caso indicado, le pide evaluar un número de iteraciones que no sean ni pocos (bajo valor estadístico) ni muchos (tardaría demasiado) y luego realiza las mediciones, obteniendo la demora total. La demora para un caso único es ese valor dividido la cantidad de intentos, y la guardamos usando `Quantity` para que al mostrarse automáticamente esté escalado y con una unidad que tenga sentido leer. Luego mostramos los dos valores y la mejora (en "veces") del primero al segundo.

Entonces, el primer caso que evaluamos es el que mencionábamos en la subsección anterior. Aquí comparamos buscando casi el último elemento en una estructura de un millón de largo, con lo cual la mejora es notable:

CELL 03

```
src_list = list(range(1_000_000))
src_set = set(src_list)

bad = "999999 in src_list"
good = "999999 in src_set"

measure(bad, good, globals={"src_set": src_set, "src_list": src_list})
```

```
Primero:  4.8 ms
Segundo:  31.4 ns
Mejora:   152841.03 X
```

La lista y la tupla, más allá de diferentes formas de recorrerlas, van comparando uno a uno los elementos para encontrar si el objeto está incluido, entonces si lo encuentra en la primer posición va a tardar casi nada pero si tiene que recorrer toda la estructura va a tardar en función de cuan larga sea la misma. El diccionario (como el conjunto) calcula el hash del objeto y lo busca directamente; tiene el costo de calcular el hash pero después es un acceso independiente del largo de la estructura. Y esto es muy importante porque si hubiésemos medido el buscar el número `1` en lugar de `555`, o buscarlo en una lista o tupla pequeñas, la medición sería totalmente inútil.

Dejamos como ejercicio para luego de la lectura el jugar buscando elementos que estén al principio o usar estructuras más pequeñas.

Como contra-regla al caso anterior, sólo para advertir no caer en esa trampa, resaltamos que aunque buscar en un conjunto es mucho más rápido que buscar en una lista, no tiene sentido convertir esa lista a conjunto *sólo para la búsqueda*. Esto es porque la conversión a conjunto, además de calcular el hash de todos los objetos y armar la estructura necesaria, obviamente recorre toda la lista de principio a fin, que es un poco lo que queríamos evitar. Por eso vemos que el segundo caso es más lento:

CELL 04

```
src_list = list(range(1_000_000))

bad = "999999 in src_list"
worse = "999999 in set(src_list)"

measure(bad, worse, globals={"src_list": src_list})
```

```
Primero:  4.84 ms
Segundo:  25.6 ms
Mejora:   0.19 X
```

Siguiendo el mismo carril de evitar construir objetos que no son estrictamente necesarios, tenemos que destacar que cuando sacamos una "rebanada" de una lista estamos construyendo toda una nueva estructura. Si vamos a necesitar esa lista como tal, está bien, pero si sólo queremos consumirla de alguna manera podemos usar `itertools.islice` que nos va a permitir iterar sobre esos objetos sin el costo intermedio:

CELL 05

```
import itertools

src = list(range(1_000))

bad = "src[1:]"
good = "itertools.islice(src, 1, None)"

measure(bad, good, globals={"itertools": itertools, "src": src})
```

```
Primero:  1.13 us
Segundo:  118 ns
Mejora:   9.62 X
```

El siguiente caso nos muestra que deberíamos tratar de evitar llamar a funciones en ciclos grandes si la llamada es trivial, porque el costo de preparar y ejecutar la función en si resulta relevante. Lo mostramos comparando cómo elevar al cuadrado una lista de números, primero usando `map` que aplica la función a cada elemento, luego usando una comprensión de listas que hace el cálculo directamente:

CELL 06

```
src = list(range(1_000_000))

def cuad(n):
    return n ** 2

bad = "list(map(cuad, src))"
good = "[x ** 2 for x in src]"
measure(bad, good, globals={"cuad": cuad, "src": src})
```

```
Primero:  221 ms
Segundo:  194 ms
Mejora:   1.14 X
```

Otra estructura que es costosa de preparar es la excepción. Cuando tenemos un bloque `try / except` el hecho de supervisar el código es extremadamente barato, pero si se sucede la excepción, ahí se incurre en un costo relevante. Es por esto que si utilizamos esa estructura dentro de un bucle debemos ver si podemos reemplazar esa funcionalidad de otra manera (validando las variables para no entrar en la situación de excepción, por ejemplo).

CELL 07

```
def bad():
    try:
        1 / 0
    except ZeroDivisionError:
        pass

def good():
    try:
        1 / 1
    except ZeroDivisionError:
        pass

measure(bad, good)
```

```
Primero:  197 ns
Segundo:  37.3 ns
Mejora:   5.27 X
```

Uno de los tipos de datos más utilizados en Python es `list`, por su utilidad y versatilidad. Tenemos que tener en cuenta sin embargo que está implementado de manera que si queremos agregar valores o sacarlos del final es muy barato, pero para insertar valores al principio o en algún punto intermedio de la estructura incurre en el costo de re-acomodarla. Por eso si vamos a estar sacando o insertando valores al principio es muy recomendable usar otra estructura que tenemos en la biblioteca estándar, el `deque`:

CELL 08

```
from collections import deque

src_list = list(range(1000))
src_deque = deque(range(1000))

bad = "src_list.insert(0, 123)"
good = "src_deque.appendleft(123)"

measure(bad, good, globals={"src_deque": src_deque, "src_list": src_list})
```

```
Primero:  17.5 us
Segundo:  34.3 ns
Mejora:   509.80 X
```

Justamente las listas están, como decíamos, pensadas para agregarles elementos y esto está bien optimizado, mientras que en las cadenas no tanto. Si vamos a estar armando una cadena muy larga de a pedacitos, es mejor ir agregando esas partes en una lista y luego construir la cadena al final a partir de esta:

CELL 09

```
def bad():
    final = ""
    for _ in range(10000):
        final += "x"
    return final

def good():
    final = []
    for _ in range(10000):
        final.append("x")
    return "".join(final)

measure(bad, good)
```

```
Primero:  616 us
Segundo:  397 us
Mejora:   1.55 X
```

Ya vimos en el capítulo sobre NumPy **??** que si podemos delegar un procesamiento a una operación interna de esa biblioteca (en vez de iterar y realizar los cálculos de a uno en nuestro código) vamos a tener un resultado más rápido. Lo mismo sucede incluso fuera de esa biblioteca, por eso es siempre útil conocer qué herramientas tenemos en Python tanto integradas en el lenguaje como en la biblioteca estándar. Por ejemplo, veamos el caso de llamar directamente a `sum` en lugar de realizar las sumas a mano:

Cell 10

```
def bad():
    total = 0
    for n in range(10000):
        total += n
    return total

def good():
    return sum(range(10000))

measure(bad, good)
```

```
Primero:  339 us
Segundo:  94 us
Mejora:   3.61 X
```

Debido a la naturaleza dinámica de Python, cada vez que queremos acceder a un atributo o nombre, hay ciertas reglas que se cumplen y pasos que se recorren. Por ejemplo, si accedemos a un nombre, `valor * 3`, Python lo buscará primero en el espacio de nombres local, y si no lo encuentra luego lo buscará en el espacio de nombres global, etc. O si usamos una función de un módulo, por ejemplo `datetime.now()`, Python resolverá primero la búsqueda de `datetime` de la forma que recién vimos, y luego tratará de encontrar `now` allí dentro. Si todas estas operaciones están en un punto álgido de nuestro código, podemos mejorar trayendo todos los nombres al espacio de nombre local para que Python los encuentre en el primer lugar que los busca.

En el siguiente código armamos una estructura que muestra varios de estos casos. Tenemos un factor a nivel global, y una clase que al inicializar la instancia crea una lista de datos de un millón de largo, y guarda un factor de corrección, y tiene dos métodos, el primero escrito todo directamente y el segundo con las mejoras

VERSIÓN PRELIMINAR

CELL 11

```
import math

FACTOR = 0.555

class Foo:

    def __init__(self, correction):
        self.data = list(range(1_000_000))
        self.correction = correction

    def bad(self):
        total = 0
        for number in self.data:
            res = math.sin(number * self.correction * FACTOR)
            total += res
        return total

    def good(self):
        total = 0
        correction = self.correction
        sin = math.sin
        factor = FACTOR
        for number in self.data:
            res = sin(number * correction * factor)
            total += res
        return total

foo = Foo(1.234)
measure(foo.bad, foo.good)
```

```
Primero:  151 ms
Segundo:  84.8 ms
Mejora:   1.78 X
```

El primer método recorre la estructura de datos y va sumando en un total el resultado del seno de la multiplicación de tres números: el obtenido de los datos, ajustado por la corrección y el factor global.

A la hora de aplicar las mejoras mencionadas tenemos que entender en qué caso es útil. A `self.data` lo accedemos una sola vez, así que eso está bien; el punto donde hacer foco es dentro del bucle, que se repite mucho. En la versión "rápida" del código entonces tenemos que dentro del `for` accedemos a `sin`, `correction` y `factor` siendo esos tres nombres locales que nos ocupamos de crear afuera del bucle.

Finalmente, debemos resaltar que siempre tenemos que considerar actualizar la versión misma de Python, ya que la performance en general del lenguaje es algo que siempre se intenta mejorar en cada liberación, haciéndose bastante foco en eso desde 3.9 en adelante. En particular también puede suceder que el lenguaje optimice específicamente algunos de los casos descriptos anteriormente y no sean más un punto a prestar atención, de forma similar a como ha mejorado la concatenación de cadenas.

1.2. OK, sigue siendo lento, ¿entonces?

Cuando hemos optimizado Python al límite y sigue siendo el cuello de botella, es hora de dar el siguiente paso: empezar a trabajar con código compilado. En esta sección veremos distintas técnicas para compilar en mayor o menor medida nuestro código Python para acelerar las porciones de código que sean críticas.

Se puede acelerar mucho ir del mundo Python a código compilado porque justamente vamos perdiendo las ventajas de Python en el camino: su dinamismo, no tener que declarar tipos estrictos, el manejo automático de memoria, etc. El lado positivo es que como ya tenemos código funcionando y pensado bastante, en general no vamos a estar explorando alternativas con ese código, por lo que el precio de hacerlo más rígido y restrictivo no es alto. Pero es parte de la complejidad que vamos a enfrentar en este viaje.

No vamos a explicar las distintas herramientas en detalle porque se escapa del alcance de este libro, pero sí queremos hacer un pequeño recorrido de cada una así ustedes pueden evaluar con cual trabajar y permitirles arrancar con ejemplos sencillos, para luego profundizar según sea necesario.

1.2.1. Un ejemplo usando Python

Armemos un ejemplo que nos va a servir para poder tener un código susceptible de mejoras e ir viendo los cambios. Simulemos que realizamos unas mediciones en una máquina y eso nos deja 16 mil archivos en una estructura de directorios (no todos en el mismo, por eficiencia del sistema de archivos), cada archivo con el estado del sistema en un momento dado siendo ese estado la posición en `(x, y)` de cada partícula y su radio [2]. La tarea de nuestro programa ejemplo será analizar archivo por archivo, contar cuantas partículas están dentro de un determinado círculo o tocándolo, y escribir un archivo de resultado que tenga una línea por archivo fuente: su nombre y la cantidad de partículas filtradas.

Arrancamos con la versión "inocente" del código, `particulas-1.py`. Más allá de las líneas específicas para manejar los parámetros que se le pasan al programa (que omitiremos copiar aquí, por brevedad, pero que pueden ir a investigar al archivo del programa en sí) tenemos una función principal y una específica para procesar cada archivo.

```
16 def main(basedir, outpath, center_x, center_y, radius):
17     outfh = open(outpath, "wt")
18     for dirpath, dirnames, filenames in os.walk(basedir):
19         for filename in filenames:
20             count = proc_file(os.path.join(dirpath, filename), center_x, center_y, radius)
21             outfh.write(f"{filename},{count}\n")
22     outfh.close()
```

Esta función principal recorre toda la estructura de directorios y para cada archivo encontrado llama a la función que lo procesa y devuelve el conteo, que procede a guardar (junto con el nombre del archivo procesado) en el archivo de salida indicado.

```
5  def proc_file(filepath, center_x, center_y, radius):
6      count = 0
7      with open(filepath, "rt") as fh:
8          for line in fh:
9              particle_x, particle_y, particle_radius = map(float, line.strip().split(","))
10             dist_centers = (particle_x - center_x) ** 2 + (particle_y - center_y) ** 2
11             if dist_centers <= (radius + particle_radius) ** 2:
12                 count += 1
13     return count
```

La función que procesa el archivo lo abre y lee línea por línea, separando los tres valores y convirtiéndolos a `float` para poder operar. Luego calcula la distancia entre el centro de la

[2] Para simular estos datos de entrada pueden ejecutar el programa `crear-data-particulas.py` incluido en el código del libro

partícula y el centro de la zona a buscar; si esta distancia es menor que la suma de los radios de la partícula y la zona a buscar significa que la partícula toca esa zona, e incrementa el contador (en realidad la distancia sería la raíz cuadrada de lo calculado, pero es más eficiente elevar al cuadrado la suma de los radios, total para la comparación sirve). Al final devuelve el valor acumulado y listo.

El correr este programa sobre toda la estructura de directorios generada tarda 3 minutos 16 segundos.

```
$ time python3 particulas-1.py superdir/ results.txt 2000 2000 100

real    3m16,482s
user    3m07,887s
sys     0m3,053s
```

Para tener una referencia de todo lo rápido que podría ser hicimos un programa equivalente en C++ [3], y vimos que tarda sólo 2 minutos 11 segundos en realizar el mismo procesamiento.

```
$ time ./particulas superdir/ results.txt 2000 2000 100

real    2m11,621s
user    2m05,795s
sys     0m2,827s
```

Como corresponde, y de la manera antes explicada, realizamos un *profiling* de ese código, encontrando que la mayor parte del tiempo pasa en `proc_file`, particularmente procesando las cadenas del archivo para poder hacer luego los cálculos. Eso nos llevó a dos pequeñas optimizaciones: leer el archivo como bytes (para no convertir a Unicode antes de convertir a número) y ahorrarnos el `strip` ya que el `float` soporta que la cadena tenga un *newline* al final.

También, principalmente por motivos pedagógicos y relacionados con presentar la información en el libro, decidimos separar esa función en otro módulo, ya que es la que vamos a ir trabajando en el resto de las subecciones. Entonces para las próximas iteraciones vamos a tener un sólo programa principal (`particulas-x.py`) al que le indicamos qué número de módulo usar, y los distintos módulos numerados; para el caso de las pequeñas optimizaciones que mencionábamos recién tendríamos el `particulas_mod_2.py`.

```
1 def proc_file(filepath, center_x, center_y, radius):
2     count = 0
3     with open(filepath, "rb") as fh:
4         for line in fh:
5             particle_x, particle_y, particle_radius = map(float, line.split(b","))
6             dist_centers = (particle_x - center_x) ** 2 + (particle_y - center_y) ** 2
7             if dist_centers <= (radius + particle_radius) ** 2:
8                 count += 1
9     return count
```

Con estos cambios obtuvimos una mejora del 4 %, claramente no es suficiente. El próximo cambio fue más disruptivo: en lugar de hacer los cálculos línea por línea, procedimos a cargar todo el archivo con Pandas (de la forma que ya explicamos **??**) y hacer los cálculos a través de esa biblioteca:

```
1 import pandas as pd
2
```

[3] Es el `particulas.cpp`, que compilamos con `g++ -O2 -o particulas particulas.cpp`

```
3
4 def proc_file(filepath, center_x, center_y, radius):
5     df = pd.read_csv(filepath, header=None, names=["x", "y", "radius"])
6     df["included"] = (df.x - center_x) ** 2 + (df.y - center_y) ** 2 <= (df.radius + radius) ** 2
7     return df["included"].value_counts()[True]
```

```
$ time python3 particulas-x.py 3 superdir results.txt 2000 2000 100

real    1m24,140s
user    1m09,423s
sys     0m15,846s
```

Encontramos que esta versión es muchísimo más rápida, lo que nos lleva a una conclusión que también habíamos sacado en el capítulo sobre NumPy **??**: si encontramos que nuestro problema ya está resuelto por una biblioteca, siempre es mejor usarla... va a estar más probada y estabilizada en el tiempo que nuestro propio código, y si abajo utiliza código compilado y optimizado será incluso más rápida que nuestro propio código en C o C++.

Programa	Tiempo	Mejora	Contra C++
C++ puro	2m11	NA	NA
Py 1: Primer intento	3m 16s	NA	+50 %
Py 2: Optimizado	3m 09s	-4 %	+44 %
Py 3: Pandas	1m 22s	-58 %	-37 %

Pero volvamos a código nuestro, porque en definitiva estamos aprendiendo cómo podemos optimizar código para el que no encontraremos bibliotecas que nos resuelven el problema.

En la secuencia que vimos llegamos al límite donde ateniéndonos puramente a Python fuimos mejorando la performance del código pero nos quedamos lejos de los tiempos de la versión en C++.

Es hora de ir un paso más allá.

1.2.2. Usando Mypyc

La herramienta Mypyc compila módulos de Python a extensiones en C.

El lenguaje compilado es una variante de Python en mayor o menor medida tipado, con lo cual gana en performance al restringir el uso de características dinámicas de Python (en la mayoría de los casos sin perder compatibilidad con el Python estándar).

Para definir el tipo de las distintas estructuras usa las sugerencias de tipado estándar de Python. La herramienta `mypyc` es parte de la biblioteca `mypy`, con lo cual aprovecha todo su poder de inferencia de tipos.

Por ejemplo, en nuestro código sólo hace falta algunas modificaciones mínimas:

```
1 def proc_file(filepath: str, center_x: float, center_y: float, radius: float) -> int:
2     count = 0
3     with open(filepath, "rb") as fh:
4         for line in fh:
5             particle_x, particle_y, particle_radius = map(float, line.split(b","))
6             dist_centers = (particle_x - center_x) ** 2 + (particle_y - center_y) ** 2
7             if dist_centers <= (radius + particle_radius) ** 2:
8                 count += 1
9     return count
```


VERSIÓN PRELIMINAR

Vemos que sólo cambiamos la signatura de la función, indicando que el *path* del archivo es una cadena de texto y el resto de los parámetros punto flotante binario, mientras que el resultado es un entero. La biblioteca infiere que los valores leídos del archivo también son punto flotante binario (porque son el resultado de llamar a `float`), que el conteo es un entero (porque lo inicializamos en cero y siempre le sumamos un entero), etc.

Es muy sencillo correr la herramienta sobre nuestro código: `mypyc particulas_mod_4.py`; vemos que mas allá de algunos mensajes de compilación el resultado es un archivo con una estructura específica: `particulas_mod_4.cpython-310-x86_64-linux-gnu.so`.

Este archivo es un módulo compilado que Python puede utilizar directamente, y la estructura de su nombre indica la posibilidad de su utilización (está compilado para CPython 3.10, en Linux, etc.). Este archivo puede convivir sin problemas con el `particulas_mod_4.py` original, Python utilizará automáticamente esta versión compilada si está presente.

```
$ time python3 particulas-x.py 4 superdir results.txt 2000 2000 100

real    2m59,327s
user    2m50,776s
sys     0m5,150s
```

Hay una ganancia marginal de tiempo, esta versión armada con `mypyc` tarda un 5 % menos que el código optimizado anterior. Podrá parecer poco, pero tengamos en cuenta que el costo de ir a esta solución es bajo, o incluso casi nulo si ya teníamos nuestro código con las sugerencias de tipado agregadas previamente.

Volviendo a la instrumentación del código, justamente, es importante entender si Mypyc tiene toda la información necesaria para optimizar el funcionamiento del mismo. Nuestro ejemplo es sencillo, pero así y todo hay varios objetos cuyo tipo debe inferirse, ya sea a partir de devolución de funciones o resultado de cálculos.

Mypyc provee una función que podemos utilizar para entender cómo está trabajando en ese sentido, `reveal_type`, que no tenemos que importar sino directamente usarla. Por ejemplo, si en la línea 9 de `particulas_mod_4.py` agregamos…

```
1            reveal_type(dist_centers)
```

… al correr mypyc vamos a ver esta nueva información como parte de la salida del programa:

```
particulas_mod_4.py:9: note: Revealed type is "builtins.float"
```

Alternativamente con el parámetro `--html-report DIR` podemos generar un reporte que nos indica qué porcentaje de nuestro código tiene el control de tipos; para nuestro ejemplo está cubierto al 100 %.

Para profundizar en esta herramienta les recomendamos no sólo ver su documentación sino también la del módulo sobre tipado de la biblioteca estándar.

1.2.3. Trabajando con Cython

Dando un paso más allá en el proceso de compilar parte del código nos encontramos con Cython.

Cython es un compilador tanto para Python en sí como para un lenguaje de programación propio de Cython que es parecido a Python (derivado de un viejo proyecto llamado Pyrex, de ahí la extensión que vamos a tener en nuestro archivo).

Con Cython entonces podemos ir ajustando código Python con definiciones de tipado estático que le sirven a Cython para lograr código compilado (de forma similar a lo que vimos con Mypyc) y también escribir código en Python que corra código C o C++ nativo, o integrarlo a bibliotecas y aplicaciones ya compiladas.

Esta transición resulta porque Cython es un lenguaje alrededor de Python mismo, que además soporta llamar a funciones en C y declarar tipos de C en variables y atributos de clase. Pero, al mismo tiempo, es código que no es perfectamente compatible con Python, por eso utilizamos la extensión `.pyx`.

Veamos cómo queda nuestro ejemplo:

```
def proc_file(str filepath, double center_x, double center_y, double radius):
    cdef int count
    cdef double particle_x, particle_y, particle_radius

    count = 0
    with open(filepath, "rb") as fh:
        for line in fh:
            particle_x, particle_y, particle_radius = map(float, line.split(b","))
            dist_centers = (particle_x - center_x) ** 2 + (particle_y - center_y) ** 2
            if dist_centers <= (radius + particle_radius) ** 2:
                count += 1
    return count
```

Vemos que agregamos tipos tanto en la signatura de la función como en las primeras dos líneas de la misma. Cuidado que esas dos primeras líneas no pueden estar en otro lado, tienen que figurar al principio de la función.

La forma más fácil de compilar Cython es utilizando un archivo de *setup*, veamos el `particulas-5-setup.py`:

```
from setuptools import setup
from Cython.Build import cythonize

setup(
    name="Ejemplo particulas",
    ext_modules=cythonize("particulas_mod_5.pyx", language_level="3", annotate=True),
)
```

La clave en esas líneas es llamar a `cythonize` como módulo externo, indicándole cual es el archivo fuente en cuestión, que es Python 3, y pidiendo con el `annotate` que genere un reporte (que veremos luego).

Le indicamos a este script que nos compile las extensiones (`python particulas-5-setup.py build_ext --inplace`) y eso nos deja un archivo muy similar al que vimos en la subsección anterior: `particulas_mod_5.cpython-310-x86_64-linux-gnu.so`.

Veamos cómo se comporta:

```
$ time python3 particulas-x.py 5 superdir results.txt 2000 2000 100

real    2m36,131s
user    2m32,264s
sys     0m2,806s
```

En este caso tenemos una mejora de performance del 17 % (comparado con el código optimizado original), pero todavía estamos lejos del código en C++ puro.

A diferencia de Mypyc donde los agregados al código igual nos dejaban algo que Python podía entender, con Cython no sólo salimos de esa situación sino que el código empieza a ser más verborrágico y menos legible, con lo cual lo ideal sería no declarar los tipos de las variables a menos que tengamos una buena razón. Al mismo tiempo, no obtenemos de Cython ningún aviso para aquellas variables que podrían haber sido tipadas y que le permite salir de la naturaleza tan dinámica de Python y generar código en C más simple y rápido.

La buena noticia es tenemos una herramienta que nos puede dar información para entender cómo Cython está traduciendo nuestro código a otro de más bajo nivel; el `annotate` que mencionamos arriba, cuando se compila el código, nos dejará un archivo reporte, un `HTML` que abrimos localmente y que nos mostrará el código original pintado y con un pequeño signo más al costado de cada línea.

El color utilizado en cada línea nos indica cuanto código en C se utilizó para reemplazar esa línea en Python; cuanto más fuerte sea el amarillo más líneas en C se utilizaron, perdiendo simpleza en el reemplazo de un lenguaje por el otro. El signo más nos permite visualizar justamente esas líneas.

En la siguiente captura de pantalla vemos el resultado para nuestro ejemplo. Noten que la línea 11 está "expandida", y se muestra abajo el código correspondiente en C (que es trivial, por eso esa línea directamente está con fondo blanco).

```
+01: def proc_file(str filepath, double center_x, double center_y, double radius):
 02:     cdef int count
 03:     cdef double particle_x, particle_y, particle_radius
 04:
+05:     count = 0
+06:     with open(filepath, "rb") as fh:
+07:         for line in fh:
+08:             particle_x, particle_y, particle_radius = map(float, line.split(b","))
+09:             dist_centers = (particle_x - center_x) ** 2 + (particle_y - center_y) ** 2
+10:             if dist_centers <= (radius + particle_radius) ** 2:
+11:                 count += 1
                __pyx_v_count = (__pyx_v_count + 1);
+12:     return count
```

Para terminar les recomendamos explorar la documentación de Cython para profundizar en su potencial.

1.2.4. Explorando Numba

Numba traduce funciones hechas en Python a código de máquina optimizado durante la ejecución misma.

No se necesita un paso extra de compilación ni tener un compilador de C o C++ instalado, sólo tenemos que aplicar un decorador en nuestra función y Numba hará el resto.

Numba está pensado para optimizar algoritmos numéricos, dónde puede terminar alcanzando velocidades de C o Fortran. Nuestro ejemplo se escapa un poco de ese área, y veremos que los resultados no son los mejores, pero decidimos contarles sobre esta herramienta por completitud.

La primera modificación al código optimizado original es importar `njit` (el compilador justo-a-tiempo de Numba) y aplicarlo como decorador en la función `proc_file`. Pero a diferencia de lo prometido por la documentación encontramos que teníamos que realizar cambios más profundos. Por un lado el `with` (administrador de contexto) no es soportado, ni tampoco si abrimos el archivo directamente con un `open`. Tampoco soportaba llamar a `map` ni a `float`.

En consideración de esas restricciones decidimos separar la función en dos partes, una que abre el archivo, lo lee y guarda en una lista todos los datos para operar luego, y otra función que lo único que hace es trabajar sobre esos números, a la que justamente decoramos con `njit`:

```python
from numba import njit
from numba.typed import List


@njit
def _proc(data, center_x, center_y, radius):
    count = 0
    for particle_x, particle_y, particle_radius in data:
        dist_centers = (particle_x - center_x) ** 2 + (particle_y - center_y) ** 2
        if dist_centers <= (radius + particle_radius) ** 2:
            count += 1
    return count


def proc_file(filepath, center_x, center_y, radius):
    with open(filepath, "rb") as fh:
        data = List()
        for line in fh:
            particle_x, particle_y, particle_radius = map(float, line.split(b","))
            data.append((particle_x, particle_y, particle_radius))
        return _proc(data, center_x, center_y, radius)
```

Incluso habiendo ajustado el área de trabajo de Numba a la parte de nuestro código que hace exclusivamente procesamiento numérico, no tuvimos buenos resultados, encontrando que tarda el doble que nuestro código Python puro.

Evidentemente no estamos en el caso ideal de uso de Numba, pero les sugerimos leer su documentación para que exploren otras funcionalidades que no se aplican en nuestro caso.

1.2.5. Conclusiones de los distintos procesos

Luego de recorrer el uso de las distintas herramientas podemos ver que hay distintos mecanismos para optimizar código Python compilándolo todo o en parte, alejándose en mayor o menor medida del código Python original.

El resumen de los cambios de velocidad obtenidos es:

Programa	Tiempo	Mejora	Contra C++
C++ puro	2m11	NA	NA
Py 2: Optimizado	3m 09s	NA	+44 %
Py 4: Mypyc	2m 59s	-5 %	+37 %
Py 5: Cython	2m 37s	-17 %	+20 %
Py 6: Numba	6m 25s	+102 %	+193 %

Que haya diversas formas de acercarse a este tipo de solución significa que no hay una "bala de plata" que cubra todos los casos de la mejor manera posible, quedará para el público lector probar y utilizar la mejor herramienta según el proyecto que encare, algunas de las exploradas hasta ahora o incluso usando completamente código compilado (que veremos en la próxima sección).

Es de vital importancia entender que compilar el código no es una solución mágica. En general funciona porque minimizamos el dinamismo intrínseco de Python, llevando parte del procesamiento a la capa en C sin pasar por estructuras de Python más complejas, y esto muchas veces implica un recorte de la funcionalidad. Por ejemplo, cuando en Cython declaramos que nuestra variable `count` es ahora de tipo `cdef int`, estamos abandonando los enteros sin límites de Python y yendo a 32 o 64 bits dependiendo de la plataforma.

VERSIÓN PRELIMINAR

Debemos resaltar el mensaje original de hacer `profiling` antes de encarar cualquier optimización: tampoco vale la pena compilar código que no es parte de la "zona caliente" del programa.

Para cerrar esta parte, algunas palabras sobre Nuitka, un compilador de Python escrito en Python que crea ejecutables que luego podrán correr sin necesitar un instalador por separado. Está más pensado para empaquetar el código junto a sus datos y el intérprete de Python en un sólo ejecutable que para acelerar la ejecución propiamente dicha. Para el caso puntual del ejemplo que manejamos en esta sección no mostró ninguna mejora en los tiempos.

1.3. Usando directamente código compilado

Hay veces que toda optimización es corta, siempre necesitamos mejor performance. O puede suceder que ya disponemos de una determinada librería escrita en C++, C, Fortran, etc, y necesitamos utilizarla pero al mismo tiempo no queremos armar *todo* nuestro programa en ese lenguaje.

En otras palabras, queremos hacer el 90 % del programa en Python aprovechando su alto nivel y facilidad de desarrollo pero también usar esa librería que ya tenemos hecha en otro lenguaje súper optimizado para la parte computacional crítica de nuestro sistema.

Es hora de conectar Python con C.

1.3.1. Extendiendo Python con C o C++

El intérprete de Python permite conectar el código hecho en ese lenguaje con código en C que maneja los objetos de Python mismo. Justamente el intérprete mismo del lenguaje hecho en C (que denominamos CPython) trabaja de esa manera.

El truco es que se expone una interfaz bien definida y muy estable en el tiempo para que se pueda trabajar desde y con otros programas hecho en estos otros lenguajes, un conjunto de funciones, macros y variables que dan acceso a la mayoría de los aspectos de Python en tiempo de ejecución. Este factor es parte del éxito de Python: históricamente siempre fue fácil utilizar desde Python otras librerías construidas con anterioridad, lo que fue clave para construir su extensa red de módulos disponibles (tanto dentro como fuera de la biblioteca estándar).

La API de Python entonces les da acceso a los programadores en C y C++ a una variedad de niveles del intérprete de Python. En nuestro caso usaremos sólo un pequeño subconjunto de toda esta funcionalidad, pero recomendamos involucrarse en su documentación para profundizar.

Aunque esta API es igualmente utilizable desde C++ por brevedad se la refiere generalmente como la Python/C API.

Hay dos formas de trabajar una unión entre Python y otros programas hechos en C.

Una es "extender" un programa hecho en Python con código en este otro lenguaje, o en otras palabras tener todo el programa en Python y utilizar alguna función o estructura hecha en C, generalmente como módulo o llamada a una biblioteca ya previamente compilada.

El otro caso es "integrar" Python en un programa o sistema más grande hecho en C. Es el caso común de programas que permiten ser utilizados programáticamente desde Python, tanto para automatizar tareas como para construir estructuras más o menos repetitivas que serían mucho trabajo de hacer a mano. Ejemplos de estos programas o sistemas son Blender, una suite de creación en tres dimensiones, Autodesk Maya, un software de modelado, animación y renderización

orientado a los efectos visuales, o Unreal Engine, utilizado para crear videojuegos y aplicaciones interactivas (en cada caso el enlace apunta justamente a la API de Python de cada programa).

En nuestro caso usaremos el mecanismo de "extensión". Para profundizar en ambas funcionalidades el mejor punto de entrada es la documentación oficial.

Volviendo a nuestro ejemplo, entonces, armaremos todo el módulo directamente en C++. Para ello reutilizaremos la función `proc_file` del programa que hicimos antes para tener una base de comparación de los tiempos. El único cambio que le haremos a esa función es renombrarla, le agregamos un guión bajo al final, ya que no la podemos usar directamente desde Python (y mantendremos el nombre original para aquella que sí).

Entonces copiamos esa función y las cabeceras correspondientes a un nuevo archivo `particulas-mod.cpp`. Pero antes de las cabeceras que copiamos, tenemos que poner:

```
#define PY_SSIZE_T_CLEAN
#include <Python.h>
```

El `Python.h` trae todos los símbolos necesarios, siempre con los prefijos `Py` o `PY`, y es importante que esté junto con esa definición bien al principio del archivo porque afectan el resto de los `include`).

No tiene sentido entrar en el detalle de la función que trajimos del otro código pero veamos su signatura:

```
int _proc_file(const std::string& filepath, double center_x, double center_y, double radius) {
    int count = 0;
    (...)
    return count;
}
```

Vemos que renombramos la función y debemos prestar atención a los tipos de datos de los argumentos que recibe y lo que devuelve, para cuando escribamos la función que la envuelve:

```
55 static PyObject *
56 proc_file(PyObject *self, PyObject *args, PyObject *kwargs)
57 {
58     const char *filepath;
59     double center_x, center_y, radius;
60     int count;
61
62     const char *kwlist[] = {"filepath", "center_x", "center_y", "radius", NULL};
63
64     if (!PyArg_ParseTupleAndKeywords(
65             args, kwargs, "sddd", const_cast<char **>(kwlist),
66             &filepath, &center_x, &center_y, &radius))
67         return NULL;
68
69     count = _proc_file(filepath, center_x, center_y, radius);
70
71     return PyLong_FromLong(count);
72 }
```

Allí tenemos la definición de `proc_file` que indica que devuelve "un objeto de Python" y recibe "argumentos" de forma genérica (tanto posicionales como nombrados… como si hiciésemos `*args, **kwargs` en código Python). Luego tenemos definiciones de variables y la lista de nombres de los argumentos a recibir (porque pueden ser nombrados, justamente).

Todo el procesamiento de los argumentos recibidos lo hace `PyArg_ParseTupleAndKeywords`, a la que le pasamos `args` y `kwargs` (los argumentos que efectivamente recibe la función), qué esperamos de ellos como forma de una cadena de texto ("s" para una cadena, tres "d" para tres *doubles*), la lista de nombres de los argumentos, y los punteros a las variables que definimos arriba y donde efectivamente quedarán guardados los valores recibidos para poder usarlos.

Si `PyArg_ParseTupleAndKeywords` termina correctamente tendremos efectivamente en esas variables los valores que se le pasaron a la función, y lo único que hacemos es llamar a la `_proc_file` que hace el procesamiento real. Esta función devuelve el conteo que es un `int` a nivel de C, debemos construir un objeto entero a nivel de Python (llamado PyLong por razones históricas) para devolverlo a esa capa.

Por otro lado, si `PyArg_ParseTupleAndKeywords` devuelve cero es que hubo un problema en la interpretación de los argumentos recibidos. En ese caso sólo debemos devolver `NULL` porque todos los objetos correspondientes a "generar una excepción" a nivel de Python ya están en su lugar.

A esta altura ya tenemos una función que podemos usar directamente desde afuera sin cambiar ese código externo. Pero todavía no tenemos un módulo propiamente dicho.

Para convertir nuestro código en C++ en un módulo utilizable por Python el punto de entrada es una función de inicialización que va a crear justamente el módulo como objeto de Python:

```
90 PyMODINIT_FUNC
91 PyInit_particulas_mod_7(void)
92 {
93     return PyModule_Create(&particulasmodule);
94 }
```

Vemos que es un paso necesario pero trivial: toda la definición del módulo está en esa estructura `particulasmodule`:

```
81 static struct PyModuleDef particulasmodule = {
82     PyModuleDef_HEAD_INIT,
83     "particulas_mod_7",
84     NULL,
85     -1,
86     ProcFileMethods
87 };
```

Esta estructura puede tener más valores pero sólo estos cinco son requeridos: el primero con un valor fijo `PyModuleDef_HEAD_INIT`, luego el nombre del módulo, el *docstring* del módulo (o `NULL` si no queremos indicar ninguno), y un valor entero que indica la memoria necesaria para contener el estado del módulo en cada subintérprete (para nuestro ejemplo alcanza con poner `-1` para indicar que tiene un estado global, pero no podrá ser usado en diferentes subintérpretes), y finalmente una lista de los métodos/funciones que tenemos en el módulo, `ProcFileMethods`, que también definimos nosotros:

```
75 static PyMethodDef ProcFileMethods[] = {
76     {"proc_file", (PyCFunction)proc_file, METH_VARARGS | METH_KEYWORDS, "Process a file."},
77     {NULL, NULL, 0, NULL}
78 };
```

Aquí tenemos una lista de todas las funciones definidas en el módulo (terminadas por una guarda, al final). En nuestro caso sólo tenemos una función, y para ella indicamos el nombre con

que estará expuesta la función, el puntero a la implementación, que tipo de argumentos reciben (construido como mapa de bits, `METH_VARARGS` para argumentos posicionales y `METH_KEYWORDS` para argumentos nombrados), y luego el *docstring* de esa función.

No necesitamos definir nada más para nuestro ejemplo. Obviamente para utilizar ese código debemos compilarlo. La forma más fácil es utilizar `setuptools` (de manera muy similar a lo que hicimos con Cython).

```
from setuptools import Extension, setup

setup(
    ext_modules=[
        Extension(
            name="particulas_mod_7",
            sources=["particulas-mod.cpp"],
        ),
    ]
)
```

Luego lo ejecutamos igual que lo que ya habíamos visto, `python particulas-7-setup.py build_ext --inplace`, lo cual nos dejará un archivo con la estructura acostumbrada: `particulas_mod_7.cpython-310-x86_64-linu`

Como respetamos las interfaces con las que veníamos trabajando, podemos usarlo directamente:

```
$ time python3 particulas-x.py 7 superdir results.txt 2000 2000 100

real    2m14,961s
user    2m08,996s
sys     0m2,899s
```

Comparemos estos resultados con los anteriores:

Programa	Tiempo	Mejora	Contra C++
C++ puro	2m11	NA	NA
Py 2: Optimizado	3m 09s	NA	+44 %
Py 5: Cython	2m 37s	-17 %	+20 %
Py 7: Extendido en C	2m 15s	-29 %	+3 %

Como vemos, obtenemos una performance equivalente a haber hecho todo el programa en C++ pero sólo con la función que realmente era responsable de la mayor parte del tiempo de ejecución. En realidad sólo llevamos hasta el extremo lo que habíamos empezado con Mypyc y Cython: ahora hicimos completamente el módulo en C++, a mano, minimizando también el costo del pasaje entre Python y C++.

Siempre va a ser menos trabajo para la persona que desarrolla un programa el hacer toda la parte de alto nivel en Python (manejo de archivos, interfaz de usuario, generación de reportes, etc) y usar C++ para la parte crítica en velocidad, que realizar absolutamente todo el programa en C++. Y como vimos podemos terminar teniendo una performance equivalente.

Entonces, como conclusión general, recomendamos fuertemente esta forma de trabajo: arrancar con el programa en Python, si es lento hacer *profiling*, y optimizar eso. Y si sigue siendo lento utilizar algunas de las herramientas de compilado automático del código, y de última construir la parte crítica en C o C++ (a menos que esta parte ya esté construida desde antes, que es de lo que habla la siguiente subsección).

Primero hacerlo bien, luego hacerlo rápido (es mucho mejor que hacerlo rápido y después tratar de que funcione correctamente).

Y siempre, en todos estas refactorizaciones de código (como en cualquier otra que hagamos mientras programamos), lo recomendable es tener pruebas de unidad para validad que luego de cada cambio que hagamos sigamos teniendo un sistema funcionando correctamente.

1.3.2. Usando directamente código previamente compilado

Por otro lado, si lo único que necesitamos es utilizar una librería ya hecha, ¡ni siquiera hace falta crear código intermedio! Podemos usar código previamente compilado directamente desde Python.

La performance del programa no es la única razón que podemos tener para tomar este camino, también puede pasar que tengamos un código compilado pero no las fuentes (muy normal si usamos software comercial o con restricciones de copyright), es un subproducto de otro proceso, código nuestro del que tenemos las fuentes pero su proceso de compilación es complicado y lleva tiempo, etc.

En esta subsección veremos entonces dos maneras de acceder a bibliotecas previamente compiladas, que normalmente encontramos en Linux con la extensión `.so` (por *shared object*) o en Windows como `.dll` (por *dynamic link library*).

Como ejemplo usaremos la *GNU Scientific Library* ("GSL"), una biblioteca de procesamiento numérico para programadores C y C++. Posee más de mil funciones en total (más info aquí), pero para nuestro caso buscamos algo relativamente sencillo, `gsl_hypot`, una función que calcula el valor de $\sqrt{x^2 + y^2}$ en una manera que evita el *overflow*.

Obviamente tenemos que tener acceso a dicha biblioteca ya compilada; este binario lo tenemos en la mayoría de las distribuciones de Linux; al momento de escribir el libro, usando el sistema operativo Ubuntu, la encontramos en `/usr/lib/x86_64-linux-gnu/libgsl.so.27`.

La primer herramienta que veremos es el módulo `ctypes` de la biblioteca estándar de Python. Como estamos en Linux usaremos la función `cdll` (por *C dynamic library loader*), pero cuidado que en Windows se deben utilizar `windll` u `oledll`.

```
>>> from ctypes import cdll
>>> libgsl = cdll.LoadLibrary("libgsl.so.27")
>>> libgsl
<CDLL 'libgsl.so.27', handle 564375d67b50 at 0x7ff8355a44c0>
```

Con dicha función cargamos la biblioteca que vamos a usar como ejemplo, y vemos que tenemos una instancia de `CDLL`, que representa la biblioteca ya cargada.

Ya mencionamos que íbamos a operar con una función específica dentro de esta biblioteca:

```
>>> libgsl.gsl_hypot
<_FuncPtr object at 0x7ff83544d900>
```

Vemos que no tenemos un "objeto función" de Python, sino un puntero a una función en C; es evidente que aunque operemos desde Python estamos trabajando directamente con el código compilado. Relacionado con eso, tenemos que tener el cuidado que las funciones en `CDLL` usan la convención estándar de C para ser llamadas, y se asume que van a devolver enteros (`int`).

¿Cómo aplica eso a nuestra función ejemplo? Para saberlo tenemos que buscar su signatura; la encontramos como parte de la documentación mencionada arriba (puntualmente acá), donde vemos:

```
double gsl_hypot(const double x, const double y)
```

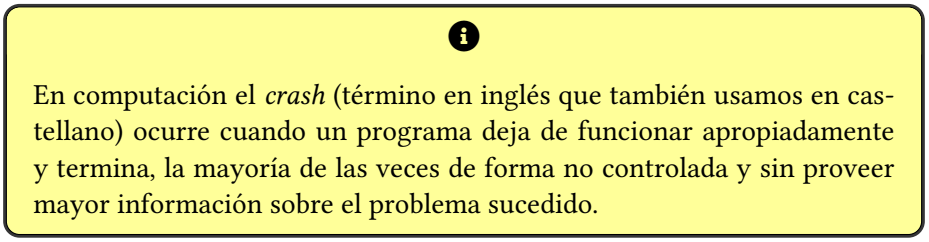
¿Podemos pasarle a la función objetos `float` de Python y recibir eso mismo como respuesta? No directamente: todos los tipos de Python (excepto enteros, cadenas y bytes) tienen que ser envueltos en su tipo específico de `ctypes` para que lleguen como un tipo de datos de C a la función:

```
>>> from ctypes import c_double
>>> libgsl.gsl_hypot(c_double(5.2), c_double(3.1))
-858993459
```

Pero cuidado, ¡ese resultado es incorrecto! Recuerden que habíamos mencionado que se asumía que la respuesta es un entero, lo cual no aplica en este caso. Debemos especificarle a la función qué tipo de dato esperamos como resultado (para que lo interprete correctamente):

```
>>> libgsl.gsl_hypot.restype = c_double
>>> libgsl.gsl_hypot(c_double(5.2), c_double(3.1))
6.053924347066125
```

`ctypes` es una herramienta poderosa, y nos permite trabajar directamente en bajo nivel con las bibliotecas compiladas, pero debemos tener cuidado de estar interpretando bien en todo momento la información ya que podemos tener resultados equivocados (o si los errores son mayores, incluso provocarle un *crash* al intérprete).



En computación el *crash* (término en inglés que también usamos en castellano) ocurre cuando un programa deja de funcionar apropiadamente y termina, la mayoría de las veces de forma no controlada y sin proveer mayor información sobre el problema sucedido.

La segunda herramienta que vamos a ver mejora mucho esa experiencia, ya que es capaz de usar la declaración de interfaz de la función y realizar automáticamente las conversiones necesarias. Se llama CFFI (por *C Foreign Function Interface*, "interfaz para funciones foráneas en C") pero sirve también para trabajar con otros lenguajes, como Fortran.

Luego de instalar `cffi` desde PyPI, vemos que su utilización es muy similar al de `ctypes`, con la gran diferencia de que le especificamos la interfaz y luego la usamos directamente:

```
>>> from cffi import FFI
>>> ffi = FFI()
>>> ffi.cdef("""
...   double gsl_hypot(const double x, const double y);
... """)
>>> libgsl = ffi.dlopen("libgsl.so.27")
>>> libgsl.gsl_hypot(5.2, 3.1)
6.053924347066125
```

CFFI también nos permite ir un paso más allá y armar directamente un módulo compilado accesible desde Python, con acceso a esa biblioteca (para lo que tenemos que tener acceso a las fuentes de la biblioteca; en Debian/Ubuntu las instalamos con `sudo apt install libgsl-dev`). El procedimiento normal es tener algún pequeño script que realice la compilación, ya que luego esto se puede incorporar al mecanismo de distribución de nuestro programa; para nuestro ejemplo:

```
from cffi import FFI
ffi = FFI()
ffi.cdef("""
    double gsl_hypot(const double x, const double y);
""")

ffi.set_source("gsl", """
    #include "gsl/gsl_math.h"   // the C header of the library
""", libraries=['gsl'])

ffi.compile(verbose=True)
```

Al correr ese script nos deja un `gsl.cpython-310-x86_64-linux-gnu.so` similar a los que ya conocemos de las secciones anteriores. Lo usamos directamente:

```
>>> import gsl
>>> gsl.lib.gsl_hypot
<built-in method gsl_hypot of _cffi_backend.Lib object at 0x7f1c04a47240>
>>> gsl.lib.gsl_hypot(5.2, 3.1)
6.053924347066125
```

Como ya habíamos declarador la interfaz antes, CFFI armó automáticamente la capa que permite a nuestro código en Python utilizar el código ya compilado, entonces el uso es directo.

Esto contrasta con `ctypes` ya que, aunque tengamos que armar ciertas estructuras para tener una interfaz compilada, luego en el uso propiamente dicho no hay ningún detalle que necesitemos considerar.

VERSIÓN PRELIMINAR

Parte II
Temas específicos

En esta Parte desarrollaremos capítulos que abordan temas específicos de aplicación de herramientas de Python. Cada capítulo está autocontenido, de forma que el lector o lectora puede acceder directamente al tema de interés sin realizar un recorrido secuencial de los capítulos, aunque serán utilizados conceptos y herramientas tratados en la Parte I.

Parte III
Apéndices

A | Zen de Python

Incluimos aquí las frases traducidas correspondientes al Zen de Python [**zen-de-python**].

- Bello es mejor que feo.
- Explícito es mejor que implícito.
- Simple es mejor que complejo.
- Complejo es mejor que complicado.
- Plano es mejor que anidado.
- Espaciado es mejor que denso.
- La legibilidad es importante.
- Los casos especiales no son lo suficientemente especiales como para romper las reglas.
- Sin embargo la practicidad le gana a la pureza.
- Los errores nunca deberían pasar silenciosamente.
- A menos que se silencien explícitamente.
- Frente a la ambigüedad, evitar la tentación de adivinar.
- Debería haber una, y preferiblemente solo una, manera obvia de hacerlo.
- A pesar de que esa manera no sea obvia a menos que seas Holandés.
- Ahora es mejor que nunca.
- A pesar de que nunca es muchas veces mejor que *ahora* mismo.
- Si la implementación es difícil de explicar, es una mala idea.
- Si la implementación es fácil de explicar, puede que sea una buena idea.
- Los espacios de nombres son una gran idea, ¡tengamos más de esos!