Nº6

- Encima de cada función usamos el decorador @profile(precision=6), indicando que queremos seis dígitos en los decimales.

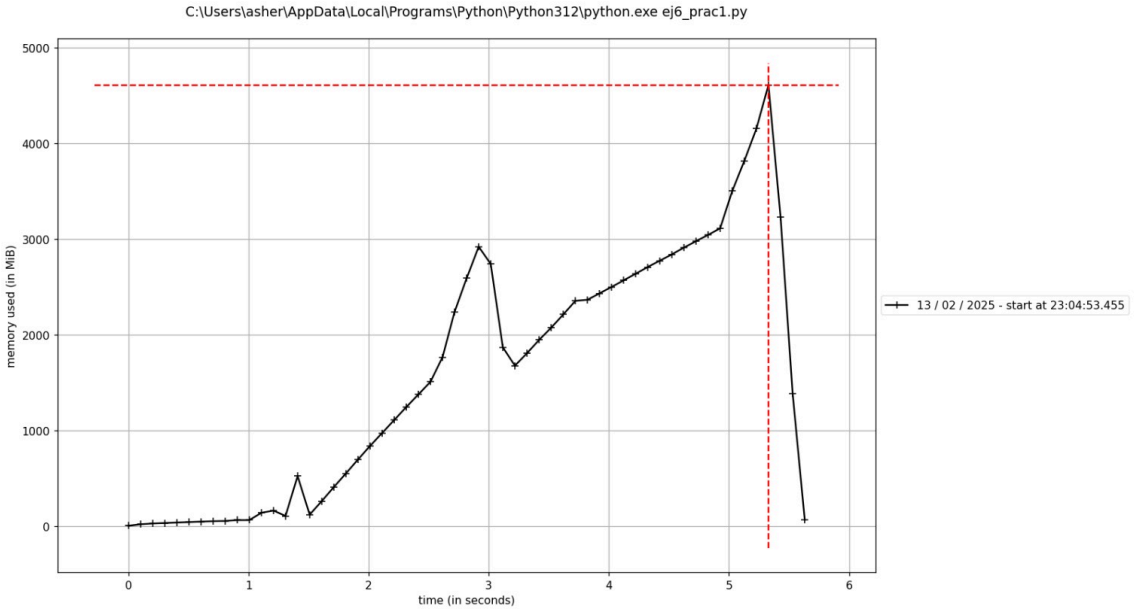
- *También se puede ver en el archivo txt adjunto:*

```
Line #    Mem usage    Increment  Occurrences   Line Contents
=============================================================
     6  61.132812 MiB  61.132812 MiB           1   @profile(precision=6)  # Se incrementa la memoria al definir la función (debido a la varibale precision)
     7                                          def types():
     8  82.378906 MiB  21.246094 MiB           1       a = np.random.rand(10**7).astype(np.float16)  # Se crea un array de 10 millones de elementos con tipo
de dato float16
     9  63.304688 MiB -19.074219 MiB           1       del a  # Se elimina la variable 'a', liberando su espacio en memoria
    10 101.453125 MiB  38.148438 MiB           1       a = np.random.rand(10**7).astype(np.float32)  # Se crea nuevamente 'a', ahora con tipo de dato
float32 (duplica el tamaño anterior)
    11  63.304688 MiB -38.148438 MiB           1       del a  # Se elimina 'a', liberando memoria
    12 139.601562 MiB  76.296875 MiB           1       a = np.random.rand(10**7).astype(np.float64)  # Se crea 'a' con tipo de dato float64 (doble del
tamaño anterior)
    13  63.304688 MiB -76.296875 MiB           1       del a  # Se elimina 'a' nuevamente
```

```
Line #    Mem usage    Increment  Occurrences   Line Contents
=============================================================
    15  63.351562 MiB  63.351562 MiB           1   @profile(precision=6)  # Se inicia el monitoreo de memoria en la función
    16                                         def listas():
    17 139.648438 MiB  76.296875 MiB           1       a = [1] * (10 ** 7)  # Se crea una lista con 10 millones de elementos
    18 292.238281 MiB 152.589844 MiB           1       b = [2] * (2 * 10 ** 7)  # Se crea otra lista con el doble de elementos que 'a'
    19 521.121094 MiB 228.882812 MiB           1       c = a + b  # Se crea 'c' sumando las listas 'a' y 'b', por lo que su tamaño es la suma de ambas
    20 521.121094 MiB   0.000000 MiB           1       d = c  # Se asigna 'c' a 'd', pero solo como referencia, sin ocupar memoria adicional
    21 368.531250 MiB -152.589844 MiB           1       del b  # Se elimina 'b', liberando su espacio en memoria
    22 292.234375 MiB -76.296875 MiB           1       a = None  # Se asigna 'None' a 'a', eliminando su contenido de memoria (es un singletone)
    23 292.234375 MiB   0.000000 MiB           1       del c  # Se elimina la referencia a 'c', pero como 'd' sigue apuntando a ella, la memoria aún no se
libera
    24  63.351562 MiB -228.882812 MiB           1       del d  # Se elimina 'd', y al no haber más referencias a los datos de 'c', la memoria se libera
```

```
Line #    Mem usage    Increment  Occurrences   Line Contents
=============================================================
    26  63.351562 MiB  63.351562 MiB           1   @profile(precision=6)  # Inicio de monitoreo de memoria para la función
    27                                         def numpy():
    28  63.351562 MiB   0.000000 MiB           1       SIZE = int(1e4)  # Se define el tamaño de la matriz (10,000 x 10,000 elementos)
    29 826.296875 MiB 762.945312 MiB           1       a = np.random.rand(SIZE, SIZE)  # Se crea la matriz 'a' con valores binarios aleatorios
    30 1589.238281 MiB 762.941406 MiB           1       b = np.random.rand(SIZE, SIZE)  # Se crea la matriz 'b', ocupando el mismo espacio que 'a'
    31 2352.179688 MiB 762.941406 MiB           1       b2 = b.copy()  # Se crea 'b2' como copia de 'b', lo que duplica su espacio en memoria
    32 2352.179688 MiB   0.000000 MiB           1       b3 = b  # 'b3' es solo una referencia a 'b', por lo que no ocupa memoria extra
    33 3115.128906 MiB 762.949219 MiB           1       c = a + b  # Se genera 'c' sumando las matrices 'a' y 'b', ocupando la misma cantidad de memoria
    34 2352.187500 MiB -762.941406 MiB           1       del b2  # Se elimina 'b2', liberando su espacio en memoria
    35 2352.187500 MiB   0.000000 MiB           1       del b3  # Se elimina la referencia 'b3', pero como 'b' sigue existiendo, no se libera memoria
    36 1589.246094 MiB -762.941406 MiB           1       a = None  # Se asigna 'None' a 'a', liberando su espacio en memoria
    37 2352.187500 MiB 762.941406 MiB           1       d = np.random.rand(SIZE, SIZE)  # Se crea otra matriz 'd' del mismo tamaño que 'a', ocupando lo
mismo
    38 2352.246094 MiB   0.058594 MiB           1       sum = np.sum(d)  # Se calcula la suma de los valores de 'd', ocupando un espacio muy bajo al ser un
entero
    39 2352.246094 MiB   0.000000 MiB           1       tr = d.T  # Se obtiene la matriz traspuesta de 'd', pero solo como referencia (vista), sin ocupar
memoria adicional
    40 3115.191406 MiB 762.945312 MiB           1       trcopy = d.T.copy()  # Se crea una copia de la matriz traspuesta, ocupando el mismo espacio que 'd'
    41 4641.074219 MiB 1525.882812 MiB           1       concat = np.concatenate((c, d), axis=0)  # Se concatenan 'c' y 'd', por lo que su memoria combinada
se suma
    42 4641.093750 MiB   0.019531 MiB           1       split = np.split(concat, 2, axis=0)  # Se divide la matriz en dos partes, pero solo como vistas de
los datos, sin ocupar más memoria
    43 826.386719 MiB -3814.707031 MiB           1       del c, d, tr, trcopy, concat, split  # Se eliminan todas estas variables, liberando su memoria
```

- Para ver un gráfico del uso de memoria usamos estos comandos mprof
    - run ej6_prac1.py
    - mprof plot



- Para saber cuanta memoría necesitamos para ejecutar el programa, basta con saber el máximo de memoria que el programa llega a usar. Para esto usamos el comando mprof peak, en este caso: 4606.184 MiB coincidiendo con la intersección de líneas rojas en el gráfico.