Memory usage in Python

There is nothing obvious in how Python consumes memory.

We'll demonstrate this below.

See [reference on how Python uses memory (https://towardsdatascience.com/the-strange-size-of-python-objects-in-memory-ce87bdfbb97f)](https://towardsdatascience.com/the-strange-size-of-python-objects-in-memory-ce87bdfbb97f) for more details.

In [1]:
```
import sys
```

Get size of empty list (i.e., overhead for list object)

In [2]:
```
sys.getsizeof([])
```

Out[2]: 72

Get size of list with 1 element

In [3]:
```
sys.getsizeof([0])
```

Out[3]:
```
80
```

Seems to be an extra 8 bytes for 1 extra element

Let's veryify by adding a second element

In [4]:
```
sys.getsizeof([0,1])
```

Out[4]:
```
88
```

Yes: 8 extra bytes per element

Let's try an "equivalent" construction, using `range` instead of manual enumeration

In [5]:
```
sys.getsizeof(range(2))
```

Out[5]:
```
48
```

Whoa ! Even smaller than an empty list !

Let's expand the range.

In [6]:
```
sys.getsizeof(range(1000))
```

Out[6]:
```
48
```

NO change in memory, even though we "added" many more elements.

What's going on here ?

Maybe the *type* of `range` is not a simple list ?

In [7]:
```
type( range(2) )
```

Out[7]:
```
range
```

Yes, it is of type `range` .

So the following is `False`

In [8]:
```
range(2) == [ 0 , 1 ]
```

Out[8]:
```
False
```

To make this true, we need to convert `range` into a `list`

In [9]:
```
list( range(2) ) == [ 0, 1 ]
```

Out[9]:
```
True
```

What about list comprehension ?

In [10]:
```
sys.getsizeof( [i for i in range(2)])
```

Out[10]:
```
104
```

In [11]:
```
sys.getsizeof( [i for i in range(4)])
```

Out[11]:
```
104
```

No change in size by adding even 2 elements.

What about adding 3 elements

In [12]:
```
sys.getsizeof( [i for i in range(5)])
```

Out[12]:
```
136
```

In [13]:
```
sys.getsizeof( [i for i in range(6)])
```

Out[13]:
```
136
```

In [14]:
```
sys.getsizeof( [i for i in range(9)])
```

Out[14]:
```
200
```

Memory doesn't seem to increase until you add more than 4 elements

Let's try a different way of enumeration: a generator

In [15]:
```
def gen(num_elements):
    for element in range(num_elements):
        yield element

print( gen(4))
```

```
<generator object gen at 0x7f174849ef50>
```

You can see that, rather than returning a list, a `generator` object is returned

To convert this into a list, use the `list` operator

In [16]:
```
print( list( gen(4)) )
```

```
[0, 1, 2, 3]
```

Let's examine the memory used

In [17]:
```
sys.getsizeof( gen(4) )
```

Out[17]: 128

In [18]:
```
sys.getsizeof( list(gen(4)) )
```

Out[18]: 128

It seems that taking the size implicitly causes enumeration to occurs

In [19]:
```
for elem in gen(4):
    print(f"Size of {elem} is {sys.getsizeof(elem)}")
```

```
Size of 0 is 24
Size of 1 is 28
Size of 2 is 28
Size of 3 is 28
```

In [20]:
```
sys.getsizeof(0), sys.getsizeof(1)
```

Out[20]:
```
(24, 28)
```

It seems that the size of integer 0 is smaller than other integers !

But we also see that enumerating the elements one at a time (rather than all simultaneously) might use much less memory

- constant size, assuming memory is freed in each iteration

Generators: Lazy evaluation

The answer is that generators (and their like) are "promises to the future"

- They have the *ability* to enumerate
- But only do so "on demand"
- You can force complete enumeration via the `list` operator

You probably *don't want to* force complete enumeration

- Will consume the maximum amount of memory

In Python, you can create a generator using a *generator function* (line `gen` , above) or inline

- *generator comprehension* using parentheses

In [35]:
```
num_elements = 100000

print( "Generator comprehension: ", sys.getsizeof( (i for i in range(num_element
s)) ))
print( "Generator function: ", sys.getsizeof( gen(num_elements)))
print( "Range: ", sys.getsizeof( range(num_elements) ) )
```

```
Generator comprehension:  128
Generator function:  128
Range:  48
```

This is in contrast to *list comprehension* which uses brackets

In [34]:
```
print( "List comprehension: ", sys.getsizeof( [i for i in range(num_elements)]
))
```

```
List comprehension:  824472
```

List comprehension uses *more memory* (full length of list) but, often, less time: (Example from [article (https://realpython.com/introduction-to-python-g)](https://realpython.com/introduction-to-python-g))

In [ ]:
```
import cProfile
```

In [48]:
```
cProfile.run( "sum( [i for i in range(num_elements)] )", sort=0)
```

```
         5 function calls in 0.020 seconds

   Ordered by: call count

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.020    0.020 {built-in method builtins.exec}
        1    0.002    0.002    0.002    0.002 {built-in method builtins.sum}
        1    0.015    0.015    0.015    0.015 <string>:1(<listcomp>)
        1    0.002    0.002    0.020    0.020 <string>:1(<module>)
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Pr
ofiler' objects}
```

The profile shows list comprehension is called a single time.

What above generator comprehension ?

In [50]:
```
cProfile.run( "sum( (i for i in range(num_elements)) )", sort=0)
```

```
         100005 function calls in 0.033 seconds

   Ordered by: call count

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
   100001    0.018    0.000    0.018    0.000 <string>:1(<genexpr>)
        1    0.000    0.000    0.033    0.033 {built-in method builtins.exec}
        1    0.015    0.015    0.033    0.033 {built-in method builtins.sum}
        1    0.000    0.000    0.033    0.033 <string>:1(<module>)
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Pr
ofiler' objects}
```

The generator is called *once per element*.

It seems that the `sum` function takes longer because the elements are being generated on demand during the summation.

So the "trick" in being able to use **big data** is to enumerate on demand

- Training loop consumes memory for only one mini-batch of data per iteration
- Rather than having complete training set in memory at once