Timing your code

Sometimes it's important to know how long your code is taking to run, or at least know if a particular line of code is slowing down your entire project. Python has a built-in timing module to do this.

Example Function or Script

Here we have two functions that do the same thing, but in different ways. How can we tell which one is more efficient? Let's time it!

In [10]:
```
def func_one(n):
    '''
    Given a number n, returns a list of string integers
    ['0','1','2',...'n]
    '''
    return [str(num) for num in range(n)]
```

In [11]:
```
func_one(10)
```

Out[11]:
```
['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
```

In [12]:
```
def func_two(n):
    '''
    Given a number n, returns a list of string integers
    ['0','1','2',...'n]
    '''
    return list(map(str,range(n)))
```

In [13]:
```
func_two(10)
```

Out[13]:
```
['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
```

Timing Start and Stop

We can try using the time module to simply calculate the elapsed time for the code. Keep in mind, due to the time module's precision, the code needs to take **at least** 0.1 seconds to complete.

In [57]:
```
import time
```

In [58]:
```
# STEP 1: Get start time
start_time = time.time()
# Step 2: Run your code you want to time
result = func_one(1000000)
```

```
# Step 3: Calculate total time elapsed
end_time = time.time() - start_time
```

In [59]:
```
end_time
```

Out[59]:
```
0.18550348281860352
```

In [60]:
```
# STEP 1: Get start time
start_time = time.time()
# Step 2: Run your code you want to time
result = func_two(1000000)
# Step 3: Calculate total time elapsed
end_time = time.time() - start_time
```

In [61]:
```
end_time
```

Out[61]:
```
0.1496279239654541
```

Timeit Module

What if we have two blocks of code that are quite fast, the difference from the time.time() method may not be enough to tell which is fater. In this case, we can use the timeit module.

The timeit module takes in two strings, a statement (stmt) and a setup. It then runs the setup code and runs the stmt code some n number of times and reports back average length of time it took.

In [18]:
```
import timeit
```

The setup (anything that needs to be defined beforehand, such as def functions.)

In [39]:
```
setup = '''
def func_one(n):
    return [str(num) for num in range(n)]
'''
```

In [40]:
```
stmt = 'func_one(100)'
```

In [41]:
```
timeit.timeit(stmt,setup,number=100000)
```

Out[41]:
```
1.3161248000000114
```

Now let try running func_two 10,000 times and compare the length of time it took.

In [42]:
```
setup2 = '''
def func_two(n):
    return list(map(str,range(n)))
'''
```

In [43]:
```
stmt2 = 'func_two(100)'
```

In [44]:
```
timeit.timeit(stmt2,setup2,number=100000)
```

Out[44]:
```
1.0892171000000417
```

It looks like func_two is more efficient. You can specify more number of runs if you want to clarify the

different for fast performing functions.

In [45]:
```
timeit.timeit(stmt,setup,number=1000000)
```

Out[45]:
```
13.129837899999984
```

In [46]:
```
timeit.timeit(stmt2,setup2,number=1000000)
```

Out[46]:
```
10.894090699999992
```

Timing you code with Jupyter "magic" method

NOTE: This method is ONLY available in Jupyter and the magic command needs to be at the top of the cell with nothing above it (not even commented code)

In [63]:
```
%%timeit
func_one(100)
```

```
100000 loops, best of 3: 13.4 µs per loop
```

In [64]:
```
%%timeit
func_two(100)
```

```
100000 loops, best of 3: 10.9 µs per loop
```

Great! Check out the documentation for more information: https://docs.python.org/3/library/timeit.html