😊 Timing the Code

1. 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.
2. Both func_one and func_two generate a list of string representations of integers from 0 to n-1, but they achieve this in different ways:
    - **func_one(n)** uses a **list comprehension**, which iterates through numbers in range(n), converts each to a string using str(num), and returns the resulting list.
    - **func_two(n)** uses the **map function**, which applies str to each number in range(n), and then converts the mapped object into a list.
3. Both functions produce the same output, but func_two may be slightly more efficient in large-scale operations as map avoids the overhead of list comprehension by applying the function directly.

[2]:
```
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)]
```

[4]:
```
func_one(10)
```

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

[6]:
```
def func_two(n):
    '''
    Given a number n, returns a list of string integers
    ['0','1','2',...'n]
    '''
    return list(map(str,range(n)))
```

[8]:
```
func_two(10)
```

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

4. Now we have imported time, and we will use it to see how long it takes to run the code.
5. This code measures the execution time of func_one(1000000) and func_two(1000000) using the time.time() function.
    - **Step 1:** It records the starting time.
    - **Step 2:** It runs the function (func_one or func_two) with n = 1,000,000.
    - **Step 3:** It calculates the elapsed time by subtracting the recorded start time from the current time after execution.

6. The result shows which function is faster. Typically, func_two (using map) performs better in large-scale operations since it avoids creating a new list manually, whereas func_one (list comprehension) may consume more memory.

[10]:
```
import time
```

[12]:
```
# 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
```

[14]:
```
end_time
```

[14]:
```
0.09606242179870605
```

[16]:
```
# 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
```

[18]:
```
end_time
```

[18]:
```
0.1293478012084961
```

7. 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 faster. In this case, we can use the **timeit** module.
8. 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.

[20]:
```
import timeit
```

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

[22]:
```
setup = '''
def func_one(n):
    return [str(num) for num in range(n)]
'''
```

[24]:
```
stmt = 'func_one(100)'
```

[26]:
```
timeit.timeit(stmt,setup,number=100000)
```

[26]:
```
0.7216813000050024
```

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

[28]:
```
setup2 = '''
def func_two(n):
    return list(map(str,range(n)))
'''
```

[30]:
```
stmt2 = 'func_two(100)'
```

[32]:
```
timeit.timeit(stmt2,setup2,number=100000)
```

[32]:
```
0.961279300005117
```

10. It looks like func_two is more efficient. You can specify more number of runs if you want to clarify the difference for fast-performing functions.

[34]:
```
timeit.timeit(stmt,setup,number=1000000)
```

[34]:
```
7.424664099999063
```

[35]:
```
timeit.timeit(stmt2,setup2,number=1000000)
```

[35]:
```
9.916960700000345
```

11. Below, we have used the Jupyter “Magic” method to time our code. The only drawback of this code is that it only works in Jupyter notebook and the magic command needs to be at the top of the cell with nothing above it.

[36]:
```
%%timeit
func_one(100)
```

```
7.03 µs ± 62.1 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
```

[37]:
```
%%timeit
func_two(100)
```

```
10.1 µs ± 405 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
```