Benchmarking

In this module, we will learn how benchmarking help in addressing performance issues.

Suppose we had written a code and it is giving the desired result too but what if we want to run this code a bit faster because the needs have changed. In this case, we need to find out what parts of our code are slowing down the entire program. In this case, benchmarking and profiling can be useful.

What is Benchmarking?

Benchmarking aims at evaluating something by comparison with a standard. However, the question that arises here is that what would be the benchmarking and why we need it in case of software programming. Benchmarking the code means how fast the code is executing and where the bottleneck is. One major reason for benchmarking is that it optimizes the code.

How does benchmarking work?

If we talk about the working of benchmarking, we need to start by benchmarking the whole program as one current state then we can combine micro benchmarks and then decompose a program into smaller programs. In order to find the bottlenecks within our program and optimize it. In other words, we can understand it as breaking the big and hard problem into series of smaller and a bit easier problem for optimizing them.

Python module for benchmarking

In Python, we have a by default module for benchmarking which is called **timeit**. With the help of the **timeit** module, we can measure the performance of small bit of Python code within our main program.

Example

In the following Python script, we are importing the **timeit** module, which further measures the time taken to execute two functions – **functionA** and **functionB** –

Benchmarking01.py

Line	Code
1	`import timeit`
2	
3	`def functionA():`
4	`    print("Function A starts the execution:")`
5	`    print("Function A completes the execution:")`
6	`def functionB():`
7	`    print("Function B starts the execution")`
8	`    print("Function B completes the execution")`
9	
10	`start_time = timeit.default_timer()`
11	`functionA()`
12	`print(timeit.default_timer() - start_time)`
13	`start_time = timeit.default_timer()`

```
14 functionB()
15 print(timeit.default_timer() - start_time)
```

After running the above script, we will get the execution time of both the functions as shown below.

Output

```
Function A starts the execution:
Function A completes the execution:
0.0014599495514175942
Function B starts the execution
Function B completes the execution
0.0017024724827479076
```

Writing our own timer using the decorator function

In Python, we can create our own timer, which will act just like the **timeit** module. It can be done with the help of the **decorator** function. Following is an example of the custom timer –

Benchmarking02.py

```
1  import random
2  import time
3
4  def timer_func(func):
5      def function_timer(*args, **kwargs):
6          start = time.time()
7          value = func(*args, **kwargs)
8          end = time.time()
9          runtime = end - start
10         msg = "{func} took {time} seconds to complete its execution."
11         print(msg.format(func=func.__name__, time=runtime))
12         return value
13     return function_timer
14
15 @timer_func
16 def Myfunction():
17     for x in range(5):
18         sleep_time = random.choice(range(1,3))
19         time.sleep(sleep_time)
20
21 if __name__ == '__main__':
22    Myfunction()
```

The above python script helps in importing random time modules. We have created the timer_func() decorator function. This has the function_timer() function inside it. Now, the nested function will grab the time before calling the passed in function. Then it waits for the function to return and grabs the end time. In this way, we can finally make python script print the execution time. The script will generate the output as shown below.

Output

```
Myfunction took 8.000457763671875 seconds to complete its execution.
```