Замерим время создания множества экземляров классов, которые мы определили. Попробуем создать по сто тысяч экземпляров на каждый класс:

```
Время создания экземпляров различных классов (n = 100000)
    Класс Course: 1.056108
    Класс CourseWithSlots: 1.158427
    Класс CourseWithWeakref: 1.108676
```

```
Время доступа и чтения атрибутов у различных классов (n = 100000)
    Класс Course. Чтение: 0.065404, запись: 0.071034
    Класс CourseWithSlots. Чтение: 0.060217, запись: 0.068282
    Класс CourseWithWeakref. Чтение: 0.068127, запись: 0.072400
```

Можно заметить, что классы со слотами чуть быстрее позволяют читать и изменять атрибуты. Чтобы проверить это, увеличим количество измерений.

Увеличим количество экземпляров до 500 тысяч:

```
Время создания экземпляров различных классов (n = 500000)
    Класс Course: 4.521500
    Класс CourseWithSlots: 4.386944
    Класс CourseWithWeakref: 4.543547
```

```
Время доступа и чтения атрибутов у различных классов (n = 500000)
    Класс Course. Чтение: 0.377064, запись: 0.350836
    Класс CourseWithSlots. Чтение: 0.267044, запись: 0.342073
    Класс CourseWithWeakref. Чтение: 0.289550, запись: 0.371747
```

Видим, что время создания экземпляров не сильно отличается. Но также видим, что у класса со __slots__ заметно быстрее происходит чтение и изменение атрибутов.

Посмотрим на то, что нам выводит модуль memory_profiler:

```
   48
   49    131.0 MiB      2.4 MiB           2       t1 = timeit.timeit(
   50    128.6 MiB      0.0 MiB           2           "[Course(f'title_{i}', Teacher(f'teacher_{i}',
f'title_{i}'))"
   51    128.6 MiB      0.0 MiB           1           + f"for i in range({n})]",
   52    128.6 MiB      0.0 MiB           1           setup="from __main__ import Teacher, Course",
   53    128.6 MiB      0.0 MiB           1           number=1,
   54                                               )
   55
   56    132.2 MiB      1.2 MiB           2       t2 = timeit.timeit(
   57    131.0 MiB      0.0 MiB           2           "[CourseWithSlots(f'title_{i}', Teacher
(f'teacher_{i}', f'title_{i}'))"
   58    131.0 MiB      0.0 MiB           1           + f"for i in range({n})]",
   59    131.0 MiB      0.0 MiB           1           setup="from __main__ import Teacher, CourseWithSlots",
   60    131.0 MiB      0.0 MiB           1           number=1,
   61                                               )
   62
   63    132.2 MiB     -1.6 MiB           2       t3 = timeit.timeit(
   64    132.2 MiB      0.0 MiB           2           "[CourseWithWeakref(f'title_{i}', Teacher
(f'teacher_{i}', f'title_{i}'))"
```

Видим, что память выделяется как раз в те моменты, когда мы выполняем создание объектов. Но в какой-то момент, при создании объектов класса CourseWithWeakref, память освободилась. Произошло это, по моему предположению, из-за срабатывания встроенного в интерпретатор сборщика мусора.

Посмотрим на вывод модуля cProfile:

```
         301735 function calls (301729 primitive calls) in 0.138 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
      7/1    0.000    0.000    0.479    0.479 {built-in method builtins.exec}
        1    0.000    0.000    0.479    0.479 <string>:1(<module>)
        1    0.000    0.000    0.479    0.479 memory_profiler.py:1185(wrapper)
        1    0.000    0.000    0.466    0.466 memory_profiler.py:759(f)
        1    0.002    0.002    0.466    0.466 time_comparison.py:76(attribute_access_comparison)
        6    0.000    0.000    0.464    0.077 timeit.py:231(timeit)
        6    0.000    0.000    0.462    0.077 timeit.py:164(timeit)
        1    0.000    0.000    0.081    0.081 <timeit-src>:2(inner)
        1    0.080    0.080    0.080    0.080 <timeit-src>:6(<listcomp>)
   300000    0.040    0.000    0.040    0.000 {built-in method builtins.setattr}
        1    0.000    0.000    0.009    0.009 memory_profiler.py:853(show_results)
       48    0.009    0.000    0.009    0.000 {method 'write' of '_io.TextIOWrapper' objects}
        1    0.000    0.000    0.004    0.004 memory_profiler.py:713(__call__)
        1    0.000    0.000    0.004    0.004 memory_profiler.py:728(add_function)
        1    0.000    0.000    0.004    0.004 memory_profiler.py:645(add)
```

Строки были отсортированы по cumtime - общему времени + время выполнения вложенных вызовов функций. Видим, что большинство инструкций выполняется в среднем меньше чем за 2 миллисекунды, но из-за большого количества вызовов этих инструкций, время выполнения значительно растёт. При этом запуске программы параметр N был равен 100000. Соответственно для трех классов было создано 300000 экземпляров и у каждого из них был вызван метод setattr.