***Профилирование времени работы алгоритмов***

Введение

Успешно освоив материал урока 1, мы теперь знаем, как оценить время работы алгоритма без получения цифровых значений. Мы узнали, что существует так называемая «О-нотация», которая предусматривает механизм оценки тенденции роста времени работы алгоритма с ростом объема обрабатываемых данных. При использовании О-нотации каждый алгоритм описывается некоторой формулой.

Но есть ситуации, когда все же требуется (или приходится) профилировать время в абсолютном его выражении, например, когда:

1. Алгоритм перенесен на код языка программирования, но определить сложность в О-нотации представляется трудоемкой задачей.
2. Требуется собрать статистику времени работы алгоритмов для формирования отчетов.
3. Сложность (в О-нотации) нескольких алгоритмов одинаковая.
4. Требуется оперативно оценить время работу отдельных блоков кода.

Вы помните, что вычислять абсолютные значения времени мы тоже умеем. Для этого в уроке 1 мы применяли модуль time. И все же существуют более эффективные, более функциональные и более подходящие для таких задач инструменты: модули **timeit** и **cProfile**.

И главное: конечная цель профилирования – поиск узких мест в работе программы и их устранение для ускорения работы кода.

Модуль timeit

Предлагает разработчику простые средства замера быстродействия фрагментов Python-кода в секундах.

**Первый (традиционный) способ замеров**

obj\_t = timeit.Timer(stmt = 'pass', setup = 'pass')

stmt – замеряемая функция, однострочное или многострочное выражение.

setup – настройки, принимаемые перед запуском замеров.

obj\_t.timeit(*[number=1000000]*)

Команда выполнит setup-выражение один раз и вернет время, необходимое для выполнения основного stmt-выражения number-раз. По умолчанию number=1000000.

**Листинг 1. task\_1.py**

|  |
| --- |
| *"""Генерация списков"""* **from** timeit **import** Timer   *# итератор с конкатенацией* **def** test\_concat():  my\_lst = []  **for** i **in** range(1000):  my\_lst = my\_lst + [i]   *# итератор с функцией append* **def** test\_cycle():  my\_lst = []  **for** i **in** range(1000):  my\_lst.append(i)   *# списковое включение* **def** test\_lst\_comp():  my\_lst = [i **for** i **in** range(1000)]   *# встроенная функция range* **def** test\_range():  my\_lst = list(range(1000))   t1 = Timer(**"test\_concat()"**, **"from \_\_main\_\_ import test\_concat"**) print(**"list concat "**, t1.timeit(number=1000), **"milliseconds"**)  t2 = Timer(**"test\_cycle()"**, **"from \_\_main\_\_ import test\_cycle"**) print(**"list append "**, t2.timeit(number=1000), **"milliseconds"**)  t3 = Timer(**"test\_lst\_comp()"**, **"from \_\_main\_\_ import test\_lst\_comp"**) print(**"list comprehension "**, t3.timeit(number=1000), **"milliseconds"**)  t4 = Timer(**"test\_range()"**, **"from \_\_main\_\_ import test\_range"**) print(**"list range "**, t4.timeit(number=1000), **"milliseconds"**)  **""" concat 1.1779784 milliseconds append 0.0715625000000002 milliseconds comprehension 0.033750200000000063 milliseconds range 0.011227300000000273 milliseconds  Вы можете объяснить получение таких результатов? """** |

obj\_t.repeat([repeat=3[,number=1000000]])

Выполнение вызова **timeit()** c указанным repeat (количеством повторений замеров). По умолчанию значение repeat составляет 3 для Python 3.6 и более ранних версий и 5 – для Python 3.7 и более поздних версий.

**Второй (лаконичный) способ замеров**

Начиная с Python 2.6 разработчику доступны более удобные функции:

timeit.**timeit**(*stmt[, setup[, timer[, number=1000000]]]]*)

Создает экземпляр класса **Timer**, передает в конструктор входные параметры, вызывает метод **timeit()**, возвращает результат в секундах типа float.

timeit.**repeat**(*stmt[, setup[, timer[, repeat=3[, number=1000000]]]]*)

Создает экземпляр класса **Timer**, передает в конструктор входные параметры, вызывает метод **repeat()**, возвращает список результатов.

На практике использование этих функций можно встретить чаще, чем предыдущий вариант.

Замерять мы можем не только код в виде функций, но и просто блоки кода. И выявлять таким образом проблемные участки кода. Однострочные выражения обозначаются одинарными кавычками. Многострочные – тройными.

**Листинг 2. task\_2.py**

|  |
| --- |
| *"""Замеряем время работы блоков кода"""* **from** timeit **import** timeit  print(timeit(**"x = 2 + 2"**)) print(timeit(**"x = sum(range(10))"**))  print(timeit(**""" for i in range(3):  y = i + 2  a = 4  if a == y:  1/2 """**))  **""" 0.010643799999999981 0.3782346000000001 0.40320979999999995 """** |

Рассмотрим пример посложнее.

**Листинг 3. task\_3.py**

|  |
| --- |
| *"""Замеряем время выполнения блоков кода"""* **from** timeit **import** repeat, default\_timer  *# до запуска замеров выполняем построение массива* setup = **"elems=range(2000)"** *# выражения, время которых мы замеряем* statements = [  **'[el\*el for el in elems]'**,  **'''res=[] for el in elems:  res.append(el\*el)'''**,  **'map(lambda el: el\*el, elems)'**]  *# обходим список выражений* **for** st **in** statements:  *# для каждого выражение вычисляем время и переносим его на 10000 запусков этого кода  # делаем по три замера* print(repeat(st, setup, default\_timer, 3, 10000))  **""" [1.3041752, 1.3166882000000002, 1.356252] [2.4010173999999997, 2.4017775000000006, 2.421308] [0.00282190000000071, 0.0021614999999997053, 0.0019029000000010399] """** |

В данном случае мы сделали по три замера каждого выражения. Соответственно, получили три массива.

Продолжим практиковаться с замерами.

**Листинг 4. task\_4.py**

|  |
| --- |
| *"""Еще замеры с timeit"""* **from** timeit **import** timeit  **""" Эта команда выполнит выражение setup один раз,  а затем возвратит время в секундах типа float,  которое требуется что бы выполнить основное выражение number раз. """   def** concat\_test():  my\_lst = []  **for** i **in** range(1000):  my\_lst += [i]   **def** append\_test():  my\_lst = []  **for** i **in** range(1000):  my\_lst.append(i)   *# через строку кода* STR\_CODE = **''' l = []  for i in range(1000):   l += [i] '''** *# еще через строку кода* STR\_CODE\_2 = **''' j = sum(range(1, 1000)) '''** print(timeit(**"concat\_test()"**, setup=**"from \_\_main\_\_ import concat\_test"**, number=1000)) print(timeit(**"append\_test()"**, setup=**"from \_\_main\_\_ import append\_test"**, number=1000)) print(timeit(STR\_CODE, number=1000)) print(timeit(STR\_CODE\_2, number=1000))  print()  print(timeit(**"concat\_test()"**, **"from \_\_main\_\_ import concat\_test"**, number=1000)) print(timeit(**"append\_test()"**, **"from \_\_main\_\_ import append\_test"**, number=1000)) print(timeit(STR\_CODE, number=1000)) print(timeit(STR\_CODE\_2, number=1000)) |

В этом примере мы используем лаконичный вариант проведения замеров. При котором в функцию **timeit** передаем замеряемое выражение. Для выражения в виде функции необходимо также выполнить ее импорт.

Само слово «setup» можно убрать, оставив только его значение.

Еще один простой пример на понимание важности импортов.

**Листинг 5. task\_5.py**

|  |
| --- |
| *"""Еще замеры с timeit"""* **import** timeit  import\_comp = **"import random"** test\_code = **'''  def my\_func():   return random.uniform(10, 100) '''** print(timeit.repeat(stmt=test\_code, setup=import\_comp))  *# еще вариант записи* print(timeit.repeat(test\_code, import\_comp)) |

Проверим возможности модуля на примере замеров относительно эффективности способов форматирования строк.

**Листинг 6. task\_6.py**

|  |
| --- |
| **from** timeit **import** timeit   **def** format\_concate():  param = 123  string = **"first "** + str(param) + **" second"   def** format\_percent():  param = 123  string = **"first %s second"** % param   **def** format\_format():  param = 123  string = **"first {} second"**.format(param)   *# читаемость и поддерживаемость* **def** format\_f():  param = 123  string = **f"first {**param**} second"** print(timeit(**"format\_concate()"**, **"from \_\_main\_\_ import format\_concate"**))  print(timeit(**"format\_percent()"**, **"from \_\_main\_\_ import format\_percent"**))  print(timeit(**"format\_format()"**, **"from \_\_main\_\_ import format\_format"**))  print(timeit(**"format\_f()"**, **"from \_\_main\_\_ import format\_f"**))  **""" 0.33543220000000007 0.31619379999999997 0.37685690000000016 0.22341030000000006 """** |

Выполнение замеров также осуществляется для проектов на ООП.

**Листинг 7. task\_7.py**

|  |
| --- |
| **from** time **import** sleep **from** timeit **import** repeat, default\_timer   **class** TestClass:   @staticmethod  **def** some\_slow\_method(loop\_count):  **for** i **in** range(loop\_count):  sleep(1)   @staticmethod  **def** some\_quick\_method(loop\_count):  **for** i **in** range(loop\_count):  sleep(0.1)   **if** \_\_name\_\_ == **'\_\_main\_\_'**:   setup = **""" from \_\_main\_\_ import TestClass test = TestClass()  """** statements = [**'test.some\_slow\_method(5)'**,  **'test.some\_slow\_method(3)'**,  **'test.some\_quick\_method(5)'**,  **'test.some\_quick\_method(3)'**]   **for** st **in** statements:  print(**f'{**st**}, {**min(repeat(st, setup, default\_timer, 3, 1))**}'**)  **""" test.some\_slow\_method(5), 5.002561600000001 test.some\_slow\_method(3), 3.000246899999997 test.some\_quick\_method(5), 0.5015900000000002 test.some\_quick\_method(3), 0.30135129999999677 """** *# если нет необходимости создавать объект класса* **'''  setup = "from \_\_main\_\_ import TestClass"  statements = ['TestClass.some\_slow\_method(5)',  'TestClass.some\_slow\_method(3)',  'TestClass.some\_quick\_method(5)',  'TestClass.some\_quick\_method(3)'] '''** |

В этом примере для массива полученных цифр применяется функция **min()**. Что по ней говорится в документации?

“Изначально, получив массив результатов замеров, мы можем сделать расчет среднего значения времени. Но большого смысла в этом нет, т.к. минимальное значения в массиве даст тот результат, который нам нужен – насколько быстро компьютер выполнит данный код. Остальные значения в результирующем списке обычно вызваны другими процессами. Так что значения функции **min()** нам будет достаточно”.

В этом примере также мы можем увидеть объект **default\_timer**, соответствующий фиксации времени старта замеров.

Теперь попробуем самостоятельно определить время, но уже немного другим путем. Кстати, в этом примере, мы сделали свой декоратор из пользовательской функции **time\_it** на базе возможностей модуля **timeit**.

**Листинг 8. task\_8.py**

|  |
| --- |
| *"""Генерация целых чисел"""* **from** timeit **import** default\_timer, timeit   **def** time\_it(func):  **def** wrapper(numb):  start\_time = default\_timer()  func(numb)  *# правая отсечка времени и результат* print(default\_timer() - start\_time)  *# логгирование  # и любые другие действия* **return** wrapper   @time\_it **def** gen\_prime(x):  multiples = []  results = []  **for** i **in** range(2, x+1):  **if** i **not in** multiples:  results.append(i)  **for** j **in** range(i\*i, x+1, i):  multiples.append(j)   **return** results   gen\_prime(3000)  print(timeit(**"gen\_prime(3000)"**, **"from \_\_main\_\_ import gen\_prime"**, number=1))  **""" # левая отсечка времени start\_time = default\_timer() # запуск функции gen\_prime(3000) # правая отсечка времени и результат print(default\_timer() - start\_time)  # сравним с привычным вариантом замеров print(timeit("gen\_prime(3000)", "from \_\_main\_\_ import gen\_prime", number=1))   0.0578244 0.05766979999999999  Существенных расхождений не выявлено """** |

Вернемся к задаче вычисления чисел Фибоначчи и применим возможности модуля **timeit** к ее решению через цикл, рекурсию и оптимизированную рекурсию.

Цикл:

**Листинг 9. task\_9.py**

|  |
| --- |
| *"""Фибо через цикл"""* **from** timeit **import** timeit   **def** func(n\_val):  **if** n\_val < 2:  **return** n\_val  pp = 0  p = 1  **for** i **in** range(n\_val-1):  pp, p = p, pp + p  **return** p   n = 8  print(timeit(**"func(n)"**, **"from \_\_main\_\_ import func, n"**)) *# print(timeit.timeit("f(n)", "from \_\_main\_\_ import f")) -> вот так не сработает* print(timeit(**"func(n)"**, globals=globals()))  **""" 0.6502667 0.653676 """** |

В этом примере мы видим еще один интересный параметр, принимаемый функцией **timeit**. **globals** указывает пространство имен для выполнения кода. Мы его используем. Чтобы «показать» программе, что аргументы замеряемых функций нужно искать в глобальной области видимости. Кроме того, мы «ушли» от импорта функций. Функции становятся «видны» благодаря параметру **globals**.

Рекурсия:

**Листинг 9. task\_9.py**

|  |
| --- |
| *"""Фибо через рекурсию"""* **from** timeit **import** timeit   **def** func(n\_val):  **if** n\_val < 2:  **return** n\_val  **return** func(n\_val - 1) + func(n\_val - 2)   n = 8  print(timeit(**"func(n)"**, globals=globals()))  **"""8.6776222"""** |

Рекурсия c оптимизацией через мемоизацию. Об этом механизме мы уже знаем.

**Листинг 9. task\_9.py**

|  |
| --- |
| *"""Фибо через рекурсию с мемоизацией через декоратор"""* **from** timeit **import** timeit   **def** memorize(func):  **def** wrapper(n\_val, memory={}):  res = memory.get(n\_val)  **if** res **is None**:  res = func(n\_val)  memory[n\_val] = res  **return** res  **return** wrapper   @memorize **def** func(n\_val):  **if** n\_val < 2:  **return** n\_val  **return** func(n\_val - 1) + func(n\_val - 2)   n = 8  print(timeit(**"func(n)"**, globals=globals()))  **"""0.19176139999999997"""** |

Обычный рекурсивный алгоритм имеет очень высокую временную сложность:

1. Каждый вызов функции порождает следующие два;
2. Последующие вызовы функции влекут за собой свои два вызова;
3. Так будет происходить, пока входной параметр функции не достигнет значения единицы.

Получим дерево вызова функции, наибольшая длина которого будет равна N, а число вызовов функции – 2^N.

Запишем получившееся дерево вызова функции:

1. 2^0 = 1 вызов: f(n)
2. 2^1 = 2 вызова: f(n-1), f(n-2)
3. 2^2 = 4 вызова: f(n-1-1), f(n-1-2), f(n-2-1), f(n-2-2)
4. 2^3 = 8 вызовов: f(n-3), f(n-4), f(n-4), f(n-5), f(n-4), f(n-5), f(n-5), f(n-6)
5. 2^4 = 16 вызовов: f(n-4), f(n-5), f(n-5), f(n-6), f(n-5), f(n-6), f(n-6), f(n-7), f(n-5), f(n-6), f(n-6), f(n-7), f(n-6), f(n-7), f(n-7), f(n-8)
6. 2^5 = 32 вызова: f(n-5), f(n-6), f(n-6), f(n-7), f(n-6), f(n-7), f(n-7), f(n-8), f(n-6), f(n-7), f(n-7), f(n-8), f(n-7), f(n-8), f(n-8), f(n-9), f(n-6), f(n-7), f(n-7), f(n-8), f(n-7), f(n-8), f(n-8), f(n-9), f(n-7), f(n-8), f(n-8), f(n-9), f(n-8), f(n-9), f(n-9), f(n-10)
7. …
8. 2^k вызовов
9. …
10. f(n-m)==f(1), f(n-m)==f(1), ... , f(n-m)==f(1)

Оценка вычислительной сложности алгоритма в 2^n поверхностная, но достаточная, чтобы заключить, что алгоритм будет затрачивать существенное время даже на относительно малых значениях n. Поэтому для практического использования его требуется оптимизировать.

Функция многократно вызывается для одного и того же значения параметра n. Поэтому на первом этапе применим мемоизацию. Мемоизация — это подход, который позволяет нам сохранять результаты промежуточных решений, для того чтобы в следующих расчетах не повторять тоже самое.

Теперь повторный вызов функции с одинаковым значением не будет приводить к рекурсии. Вместо этого результат будет возвращаться из словаря memory, в который записываются результаты предыдущих вычислений.  
  
Хронология вызовов будет такой:

f(n), f(n-1), f(n-2), f(n-3), f(n-4), …, f(1), f(0), f(1), …, f(n-5), f(n-4), f(n-3), f(n-2).  
  
Добавляя мемоизирующий декоратор, удалось снизить сложность алгоритма с экспоненциальной (2n) до линейной (n).

Модуль cProfile

Встроенный компонент, обеспечивающий профилирование кода на Python. Регистрирует функции и время их выполнения в секундах, предоставляет разработчику подробную статистику замеров.

Профилировка – это измерение производительности всей программы и ее фрагментов. Цель – найти «горячие точки»: участки кода, на выполнение которых расходуется больше всего времени.

Профайлер – основной инструмент оптимизатора программ. Бывает так, что программа работает медленно из-за единственной машинной инструкции.

*Например, инструкции деления, которая многократно выполняется в глубоко вложенном цикле*.

Программист, приложивший титанические усилия для улучшения остального кода, окажется очень удивлен, что в результате производительность приложения возросла едва ли на 10%-15%.

Большинство профайлеров поддерживают следующий набор базовых операций:

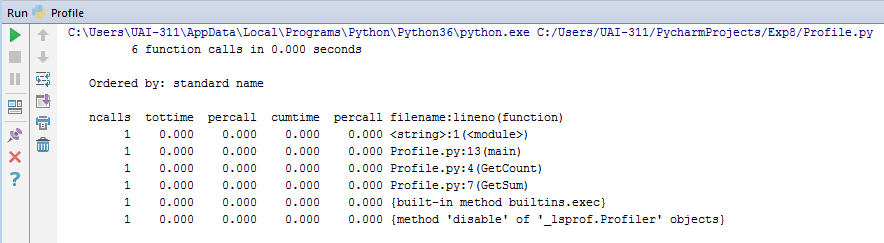
* Определение общего времени исполнения каждой точки программы;
* Определение удельного времени исполнения каждой точки программы;
* Определение причины и/или источника конфликтов;
* Определение количества вызовов точки программы;
* Определение степени покрытия программы.

Профилировщиком **cProfile** также легко пользоваться, как **timeit**, но он дает больше подробной информации о том, на что тратится время при выполнении программы. Профилирование кода с **cProfile** это достаточно просто. Все что вам нужно сделать, это [импортировать модуль](https://python-scripts.com/import-modules-python) и вызвать его функцию **run**.

**Листинг 10. task\_10.py**

|  |
| --- |
| *"""Профилировка"""* **from** cProfile **import** run   **def** get\_count(items):  **return** items.\_\_len\_\_()   **def** get\_sum(items):  sum = 0  **for** i **in** items:  sum += i  **return** sum   **def** main():  my\_lst = [3, 5, 6, 7]  res\_count = get\_count(my\_lst)  res\_sum = get\_sum(my\_lst)   run(**'main()'**) |

В результате увидим информацию о главном процессе:



Первая строка показывает, что в ней 6 вызова функций. На запуск всех функций ушло 0 секунд. Слабых мест нет. Следующая строка говорит нам, в каком порядке результаты выдачи. Здесь есть несколько столбцов.

* **ncalls** – это количество совершенных вызовов;
* **tottime** – это все время, потраченное в данной функции;
* **percall** – ссылается на коэффициент tottime, деленный на ncalls;
* **cumtime** – совокупное время, потраченное как в данной функции, так и наследуемых функциях (те, которые внутри профилируемых функций). Это работает также и с рекурсивными функциями!
* Второй столбец **percall** – это коэффициент cumtime деленный на примитивные вызовы;
* **filename:lineno(function)** предоставляет соответствующие данные о каждой функции.

Рассмотрим пример, в котором создадим проблемные ситуации самостоятельно.

**Листинг 11. task\_11.py**

|  |
| --- |
| *"""Профилировка времени выполнения функций"""  """Профилировка времени выполнения функций"""* **from** cProfile **import** run **from** time **import** sleep   **def** fast():  print(**"Я быстрая функция"**)   **def** slow():  sleep(3)  print(**"Я очень медленная функция"**)   **def** medium():  sleep(0.5)  print(**"Я средняя функция..."**)   **def** main():  fast()  slow()  medium()   run(**'main()'**) |