In [53]:

```
import numpy as np
import scipy.stats as sps
```

Оптимизация по времени:

Подумай, необходима ли оптимизация!

1. На оптимизацию тратится время.
2. Скорее всего код станет непонятнее.
3. Не все оптимизации полезны. Оптимизируя по времени, вы можете увеличить расход памяти.

Перед оптимизацией стоит написать работающий код и тесты к нему.

Профилирование:

Профилирование - сбор характеристик работы программы.

Прежде чем приступать к оптимизации, нужно понять какой фрагмент кода нужно оптимизировать.

Профилирование по времени исполнения:

Инструменты, которые мы рассмотрим:

- cProfile
- line_profiler

Другие инструменты:

- py-spy
- pstats
- RunSnakeRun
- SnakeViz

py-spy позволяет визуализировать потребление времени во время выполнения программы без модификаций её кода

Измерение времени:

Иногда хочется измерить время исполнения участков кода целиком. При использовании IPython можно воспользоваться магическими функциями `%timeit` и `%%timeit`

`%timeit` позволяет измерить время исполнения одной строки

In [54]:

```
def slow_reverse(s):
    """
    :param s: list
    :return: reversed list
    """
    reversed_s = np.zeros(len(s))
    for i in range(len(s)):
        reversed_s[i] = s[len(s) - i - 1]
    return reversed_s
```

In [55]:

```
s = sps.randint(0, 100).rvs(100)
```

In [56]:

```
%timeit slow_reverse(s)
%timeit s[::-1]
```

```
44.2 µs ± 2.2 µs per loop (mean ± std. dev. of 7 runs, 10000 loops eac
h)
278 ns ± 2.42 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops e
ach)
```

In [57]:

```
%%timeit
s = sps.randint(0, 100).rvs(100)
reversed_s = np.zeros(len(s))
for i in range(len(s)):
    reversed_s[i] = s[len(s) - i - 1]
```

```
3.23 ms ± 422 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

Выводится среднее значение и среднеквадратичное отклонение.

Другой синтаксис:

In [58]:

```
import timeit

setup = """
import numpy as np
import scipy.stats as sps

def slow_reverse(s):
    reversed_s = np.zeros(len(s))
    for i in range(len(s)):
        reversed_s[i] = s[len(s) - i - 1]
    return reversed_s

s = sps.randint(0, 100).rvs(100)
"""

t = timeit.Timer("""slow_reverse(s)""", setup=setup)
```

Мы передаем все параметры `timeit.Timer` в строках из-за того, что `timeit` реализован в виде [шаблонноӥ строки (https://github.com/python/cpython/blob/master/Lib/timeit.py#L68)](), куда передаются параметры.
Это позволяет сэкономить время на вызове функции, если бы мы передавали её в качестве объекта.

In [59]:

```
?timeit.Timer
```

In [105]:

```
print(t.timeit(number=10))
print(t.timeit(number=100))
print(t.repeat(repeat=3, number=10))
```

```
0.0013150400191079825
0.010520849988097325
[0.0005955360247753561, 0.0014676569844596088, 0.0009530319948680699]
```

cProfiler

Позволяет собрать аналитику по вызовам функций:

- ncals - кол-во вызовов. Если в этой колонке стоит два числа $3/1$, то это значит, что функция рекурсивная. Первое число - общее кол-во вызовов, второе - кол-во нерекурсивных вызовов.
- totime - время исполнения функции без учета времени вызова подфункций
- cumtime - время исполнения функции с учетом времени вызова подфункций

In [27]:

```
def fib(n):
    if n == 0:
        return 1
    if n == 1:
        return 1
    return fib(n - 1) + fib(n - 2)
```

In [46]:

```
import cProfile
cProfile.run('fib(30)', sort='tottime')
```

```
         2692540 function calls (4 primitive calls) in 0.789 seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(functio
n)
2692537/1    0.789    0.000    0.789    0.789 <ipython-input-27-99a0d8
69b1b2>:1(fib)
        1    0.000    0.000    0.789    0.789 {built-in method builtin
s.exec}
        1    0.000    0.000    0.789    0.789 <string>:1(<module>)
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_l
sprof.Profiler' objects}
```

In [39]:

```
%%writefile slow_reverse.py

import numpy as np
import scipy.stats as sps

def slow_reverse(s):
    reversed_s = np.zeros(len(s))
    for i in range(len(s)):
        reversed_s[i] = s[len(s) - i - 1]
    return reversed_s

s = sps.randint(0, 100).rvs(1000)
slow_reverse(s)
```

```
Overwriting slow_reverse.py
```

In [40]:

```
import cProfile

code = open('slow_reverse.py', 'r')
cProfile.run(code.read(), sort='tottime')
```

```
         9306 function calls (9304 primitive calls) in 0.039 seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(functi
on)
        1    0.023    0.023    0.034    0.034 function_base.py:2154
(_vectorize_call)
     1000    0.004    0.000    0.011    0.000 _dtype.py:319(_name_ge
t)
     2000    0.003    0.000    0.004    0.000 numerictypes.py:293(is
subclass_)
     1000    0.003    0.000    0.007    0.000 numerictypes.py:365(is
subdtype)
     3000    0.002    0.000    0.002    0.000 {built-in method built
ins.issubclass}
        1    0.002    0.002    0.002    0.002 <string>:5(slow_revers
e)
        2    0.001    0.000    0.001    0.001 doccer.py:12(docforma
t)
```

line_profiler

Позволяет собрать построчную аналитику для нескольких функций

In [47]:

```
%load_ext line_profiler
def source(length=100):
    """
    A statement to execute under the line-by-line profiler.
    :param length: length of the list to reverse
    """
    s = sps.randint(0, 100).rvs(length)
    slow_reverse(s)
    fast_reverse(s)
```

In [48]:

```
def slow_reverse(s):
    reversed_s = np.zeros(len(s))
    for i in range(len(s)):
        reversed_s[i] = s[len(s) - i - 1]
    return reversed_s

def fast_reverse(s):
    return s[::-1]

%lprun -f slow_reverse -f fast_reverse source()
```

In [60]:

```
%lprun?
```

Профилирование по памяти:

- memory_profiler

Позволяет измерить общее и построчное потребление памяти.

In [64]:

```
def memory_func():
    x = [1] * 10 ** 4
    y = [2] * 10 ** 6
    del x
    return y
```

In [65]:

```
%load_ext memory_profiler
```

```
The memory_profiler extension is already loaded. To reload it, use:
  %reload_ext memory_profiler
```

Можно измерить общее потребление памяти (аналогично `%timeit`):

In [66]:

```
%memit memory_func()
```

```
peak memory: 77.99 MiB, increment: 7.35 MiB
```

`peak memory` - наибольшее значение расходуемой памяти системы во время работы программы. Нужно, чтобы посмотреть, насколько мы близки к тому, чтобы израсходовать всю RAM.

`increment = peak memory - starting memory`

In [67]:

```
%memit?
```

Можно измерить потребление памяти по строкам (аналогично `%lprun`). Однако `%mprun` не может работать с функциями из ноутбука, их нужно записывать в файл.

In [4]:

```
%%writefile memory_demo.py

def memory_func():
    x = [1] * 10 ** 4
    y = [2] * (2 * 10 ** 6)
    del y
    return x
```

```
Writing memory_demo.py
```

In [5]:

```
from memory_demo import memory_func
%mprun -f memory_func memory_func()
```