# Урок 1. Типы профайлеров

Ранее в модуле, посвящённому дебаггингу и профилированию, мы разобрали, как обрабатывать ошибки и работать с иерархиями исключений, поработали с логированием и настройкой логгера на примере проекта на Flask. В текущем модуле мы продолжим знакомиться с инструментами, с помощью которых будем отлаживать и оптимизировать код.

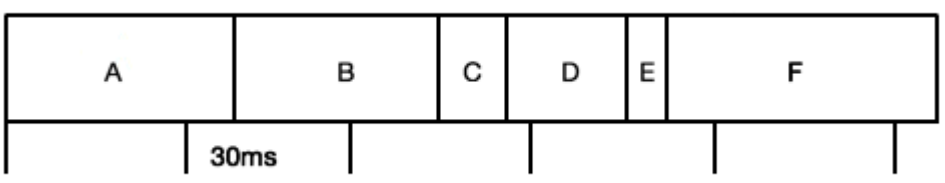
Первое занятие посвящено профилированию: мы разберём, для чего оно нужно и какие возможности у него есть. Мы познакомимся с разными методами профилирования и инструментами, с помощью которых эти методы реализуются.

Профилирование представляет собой способ поиска узких мест в процессе выполнения программы. Его результаты могут быть использованы для оптимизации кода и увеличения производительности системы. В каждом проекте наступает стадия, когда появляется необходимость анализа кода, который работает слишком медленно или тратит слишком много ресурсов.

Метрики выполнения кода:

- время выполнения функций,
- количество вызовов функций,
- дерево вызовов функций,
- загрузка CPU и потребление памяти,
- обращения к другим ресурсам сервера.

Первый тип профилировщика — статистический, принцип его работы простой: через установленные промежутки времени профайлер смотрит, какая функция выполняется в данный момент и сохраняет эти данные для последующего анализа.



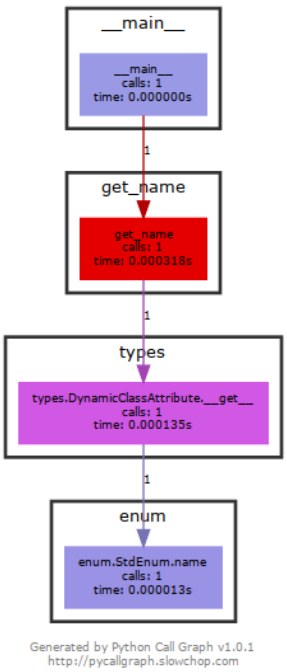
Плюсы:

- минимальное влияние на программу,
- простой и универсальный в использовании,
- определение тяжёлых и часто используемых функций.

Минусы:

- точность анализа зависит от количества измерений и временного интервала,
- результаты анализа содержат неполную информацию,
- для сбора точной статистики может потребоваться длительное время,
- малое количество инструментов для анализа.

Второй тип профилировщиков — профайлеры, основанные на событиях.




Если статистические профайлеры срабатывают с определённой временной частотой, то событийные профайлеры отслеживают вызов функции, выходы из них, исключения и измеряют интервалы между этими событиями. Измеренное время вместе с информацией о соответствующих участках кода и количестве

вызовов сохраняется для дальнейшего анализа. Логично, что событийные профайлеры намного точнее статистических, но при их использовании теряется скорость работы.

Плюсы:

- подробная информация о работе программы,
- не требуется написание кода для анализа,
- обширное количество инструментов профилирования.

Минусы:

- дополнительные накладные расходы.

```
if __name__ == '__main__':
    import time

    start = time.time()

    len = 10000000
    my_list = []
    for c in range(len):
        my_list.append(c)

    finish = time.time()
    c = finish - start

    print(c)
```

Существует и ручное профилирование.

```
import time

def profiler(func):
    def wrapper(*args, **kwargs):
        before = time.time()
        f = func(*args, **kwargs)
```

```
        after = time.time()
        print(after - before)

    return wrapper

@profiler
def hello_guys():
    print("Hello guys")

if __name__ == '__main__':
    hello_guys()
```

Оно осуществляется следующим образом: замеряем время работы функций, кидаем результаты в лог или в какую-либо систему сбора метрик, отрисовываем график и получаем результат.

Этот подход имеет больше минусов, чем плюсов: простота использования и реализации, выборочная проверка функций ведёт к небольшим накладным расходам на выполнение декоратора, который нужно сперва написать под конкретную задачу или систему. Самый главный недостаток — это отсутствие информации о входных данных: если функция зависит от входных данных, то результаты анализа будут совершенно неактуальны.