Советы по производительности

Этот документ посвящен различным советам и трюкам, которые помогают повысить производительность ваших программ на Python. Оригинал на англ.

Введение

Жизненный цикл создания приложения:

1. Напишите программу.
2. Протестируйте её.
3. Если она слишком медленная, воспользуйтесь профилировщиком.
4. Оптимизируйте программу.
5. Перейдите к пункту №2.

Если программа слишком медленная, профилирование может показать, какие части программы занимают большую часть времени.

Некоторые оптимизации являются показателем хорошего стиля программирования, и поэтому их следует применять по мере изучения языка. Примером может служить перемещение вычислений значений, которые не изменяются внутри цикла, за пределы цикла.

Сортировка

Сортировка списков, как правило, довольно эффективна. Метод сортировки для списков принимает необязательную функцию сравнения в качестве аргумента, который можно использовать для изменения поведения сортировки. Это довольно удобно, хотя и может значительно замедлить сортировку, так как функция сравнения будет вызываться много раз.

Конкатенация строк

Строки в Python неизменяемы. Этот факт часто настигает врасплох начинающих программистов Python. Неизменность дает некоторые преимущества и недостатки.

Преимуществом является то, что строки могут использоваться в качестве ключей в словарях, а отдельные копии могут совместно использоваться несколькими связанными переменных. Недостатком является то, что вы не можете написать что-то вроде: измените все символы «а» на «б» в заданной строке. Вместо этого вы должны создать новую строку. Это постоянное копирование может привести к ухудшению эффективности.

Распространенная и катастрофическая ошибка при построении больших строк:

```
s = ""
for substring in list:
    s += substring
```

Вместо этого используйте:

```
s = "".join(list)
```

Аналогично, если вы генерируете последовательности строк.

```
s = ""
for x in list:
    s += some_function(x)
```

Вместо этого используйте:

```
slist = [some_function(elt) for elt in somelist]
s = "".join(slist)
```

Вместо:

```
out = "<html>" + head + prologue + query + tail + "</html>"
```

Используйте:

```
out = "<html>%s%s%s%s</html>" % (head, prologue, query, tail)
```

Еще лучше, для удобства чтения (это не имеет ничего общего с эффективностью) используйте словарь:

```
out = "<html>%(head)s%(prologue)s%(query)s%(tail)s</html>" % locals()
```

Последние два примера просты в понимании, и что не маловажно, они выполняются намного быстрее, особенно при большом количестве вызовов. Не забывайте, что Python выполняет весь поиск методов во время выполнения.

Циклы

Python поддерживает несколько циклических конструкций. Наиболее часто используется оператор *for*. Если тело вашего цикла простое, то накладные расходы интерпретатора самого цикла *for* могут составлять значительную часть накладных расходов. Здесь очень удобна функция *map*. Единственное ограничение заключается в том, что «тело цикла» *map* должно быть вызовом функции. Оптимальным способом перебора последовательностей является генератор списка (*list comprehensions*). Помимо синтаксического преимущества, он быстрее, чем эквивалентное использование *map*.

Рассмотрим следующий пример. Вместо того чтобы зацикливаться на списке слов и преобразовывать их в верхний регистр:

```
newlist = []
for word in oldlist:
    newlist.append(word.upper())
```

вы можете использовать *map*, чтобы выполнить цикл из скомпилированного кода на C:

```
newlist = map(str.upper, oldlist)
```

Генератор списка обеспечивает синтаксически более компактный и более эффективный способ написания вышеупомянутого цикла *for*:

```
newlist = [s.upper() for s in oldlist]
```

Выражение-генератор функционирует примерно как генератор списка или *map*, но избегает накладных расходов на создание всего списка сразу. Вместо этого он возвращает объект генератора, который можно перебирать шаг за шагом:

```
iterator = (s.upper() for s in oldlist)
```

Выбор подходящего метода будет зависеть от используемой версии Python и свойств обрабатываемых данных. Гвидо ван Россум написал гораздо более подробное (и сжатое) исследование оптимизации циклов, которое определенно стоит прочитать.

Избегайте точки...

Предположим, вы не можете использовать *map* или генератор списка. Вам приходится использовать цикл *for*.

Рассмотрим следующий пример:

```
newlist = []
for word in oldlist:
    newlist.append(word.upper())
```

newlist.append и word.upper – это ссылки на функции, которые проверяются каждый раз в цикле. Исходный цикл можно заменить на:

```
upper = str.upper
newlist = []
append = newlist.append
for word in oldlist:
    append(upper(word))
```

Этот трюк следует применять с осторожностью. Данный подход становится все труднее поддерживать, если цикл большой. Если вы не очень хорошо знакомы с этим фрагментом кода, вам необходимо изучить его, чтобы понять определения append и upper.

Локальные переменные

Финальное ускорение заключается в использовании локальных переменных везде, где это возможно. Если приведенный выше цикл преобразовать в функцию, то append и upper становятся локальными переменными. Python обращается к локальным переменным гораздо эффективнее, чем к глобальным.

```
def func():
    upper = str.upper
    newlist = []
    append = newlist.append
    for word in oldlist:
        append(upper(word))
    return newlist
```

Протестируем производительность циклов:

```
Время в секундах

Обычный цикл  3.47
Без использования точек внутри цикла (no dots)  2.45
Локальные переменные  +  no dots 1.79
Использование функции map  0.54
```

Инициализация элементов словаря

Предположим, вы создаете словарь частот слов, и вы уже разбили свой текст на список слов. Вы можете выполнить что-то вроде:

```
wdict = {}
for word in words:
    if word not in wdict:
        wdict[word] = 0
    wdict[word] += 1
```

Каждый раз, когда слово уже находится в словаре, *if* терпит неудачу. Если вы обрабатывается большое количество слов, многие из них, вероятно, встречаются несколько раз. В ситуации, когда инициализация значения происходит только один раз и инкремент этого значения происходит много раз, то оптимальным способом будет использовать оператор *try*:

```
wdict = {}
for word in words:
    try:
        wdict[word] += 1
    except KeyError:
        wdict[word] = 1
```

Важно ловить ожидаемое исключение *KeyError* и не использовать исключение по умолчанию *except*, чтобы избежать попытки обработать исключения, которые вы действительно не можете обработать с помощью оператора(ов) в предложении *try*.

Ещё одна альтернатива стала доступна с выходом Python 2.x. Словари теперь имеют метод get(), который возвращает значение по умолчанию, если нужный ключ не найден в словаре. Это упрощает цикл:

```
wdict = {}
get = wdict.get
for word in words:
    wdict[word] = get(word, 0) + 1
```

Кроме того, если значение, хранящееся в словаре, является объектом или (изменяемым) списком, вы также можете использовать метод dict.setdefault, например:

```
wdict.setdefault(key, []).append(new_element)
```

Вы можете подумать, что это позволит избежать необходимости искать ключ дважды. На самом деле это не так (даже в python 3.0), но, по крайней мере, двойной поиск выполняется на C.

Другой вариант-использовать класс defaultdict:

```
from collections import defaultdict

wdict = defaultdict(int)

for word in words:
    wdict[word] += 1
```

Импорт

Операторы импорта могут выполняться практически в любом месте. Часто бывает полезно поместить их внутри функций, чтобы ограничить их видимость и/или сократить начальное время запуска. Хотя интерпретатор Python оптимизирован для того, чтобы не импортировать один и тот же модуль несколько раз, повторное выполнение инструкции *import* может серьезно повлиять на производительность в некоторых обстоятельствах.

Рассмотрим следующие два фрагмента кода:

```
def doit1():
    import string ###### импорт внутри функции
    string.lower('Python')

for num in range(100000):
    doit1()
```

и

```
import string ###### импорт вне функции
def doit2():
    string.lower('Python')

for num in range(100000):
    doit2()
```

Второй пример будет работать намного быстрее, даже если ссылка на модуль *string* является глобальной в методе doit2. Приведенный выше пример, очевидно, немного надуман, но общий принцип остается в силе.

Обратите внимание, что включение импорта внутри функции может ускорить начальную загрузку модуля, особенно если импортированный модуль может не потребоваться. Как правило, это случай «ленивой» оптимизации – избегание работы

(импорт модуля, который может быть очень дорогим) до тех пор, пока вы не будете уверены, что это необходимо.

Это позволит существенно сэкономить только в тех случаях, когда модуль вообще не был бы импортирован (из любого модуля) – если модуль уже загружен (как это будет иметь место для многих стандартных модулей, таких как *string* или *re*), избегание импорта ничего не спасет. Чтобы увидеть, какие модули загружены в систему, посмотрите в sys.modules.

Хороший способ сделать ленивый импорт – это:

```
email = None

def parse_email():
    global email
    if email is None:
        import email
    ...
```

Таким образом, модуль электронной почты будет импортирован только один раз, при первом вызове функции parse_email().

Агрегирование данных

Накладные расходы на вызов функций в Python относительно высоки, особенно по сравнению со скоростью выполнения встроенной функции. Это свидетельствует о том, что там, где это уместно, функции должны обрабатывать агрегаты данных. Вот «искуственный» пример, написанный на Python:

```
import time
x = 0
def doit1(i):
    global x
    x = x + i

list = range(100000)
t = time.time()
for i in list:
    doit1(i)

print "%.3f" % (time.time()-t)
```

и

```
import time
x = 0
def doit2(list):
    global x
    for i in list:
        x = x + i

list = range(100000)
t = time.time()
doit2(list)
print "%.3f" % (time.time()-t)
```

Второй пример работает примерно в четыре раза быстрее, чем первый.

Системный параметры

Интерпретатор Python выполняет некоторые периодические проверки. В частности, он решает, следует ли разрешить запуск другого потока и следует ли запускать задержанный вызов [*pending call*] (обычно это вызов, установленный обработчиком сигнала). Большую часть времени выполнять нечего, поэтому каждый проход вокруг цикла интерпретатора может замедлить работу. В модуле *sys* есть функция *setcheckinterval*, которую можно вызвать, чтобы сообщить интерпретатору, как часто следует выполнять эти периодические проверки. До выпуска Python 2.3 он по умолчанию был равен 10. В 2.3 этот показатель был повышен до 100. Если вы не работаете с потоками и не ожидаете, что будете ловить много сигналов, установка большего значения может улучшить производительность интерпретатора, иногда существенно.

Python это не C

Это также не Perl, Java, C++ или Haskell. Будьте осторожны при использовании ваших знаний о том, как работают другие языки, в Python. Простой пример служит для демонстрации:

```
% timeit.py -s 'x = 47' 'x * 2'
loops, best of 3: 0.574 usec per loop
% timeit.py -s 'x = 47' 'x << 1'
loops, best of 3: 0.524 usec per loop
% timeit.py -s 'x = 47' 'x + x'
loops, best of 3: 0.382 usec per loop
```

Теперь рассмотрим аналогичные программы на языке C:

```
#include <stdio.h>

int main (int argc, char *argv[]) {
 int i = 47;
 int loop;
 for (loop=0; loop<500000000; loop++)
  i + i;
 return 0;
}
```

и время исполнения:

```
% for prog in mult add shift ; do
< for i in 1 2 3 ; do
< echo -n "$prog: "
< /usr/bin/time ./$prog
< done
< echo
< done
mult: 6.12 real 5.64 user 0.01 sys
mult: 6.08 real 5.50 user 0.04 sys
mult: 6.10 real 5.45 user 0.03 sys

add: 6.07 real 5.54 user 0.00 sys
add: 6.08 real 5.60 user 0.00 sys
add: 6.07 real 5.58 user 0.01 sys

shift: 6.09 real 5.55 user 0.01 sys
shift: 6.10 real 5.62 user 0.01 sys
shift: 6.06 real 5.50 user 0.01 sys
```

Обратите внимание, что в Python есть значительное преимущество в добавлении числа к самому себе вместо того, чтобы умножать его на два или сдвигать влево на один бит. В языке C на всех современных компьютерных архитектурах каждая из трех арифметических операций преобразуется в единую машинную инструкцию, которая выполняется за один цикл, поэтому на самом деле не имеет значения, какую из них вы выберете.

Распространенный «тест», который часто выполняют джуны-программисты Python, – это перевод синтаксиса Perl

```
while (<>) {
    print;
}
```

в код Python, который выглядит примерно так

```
import fileinput

for line in fileinput.input():
    print line,
```

и делают вывод о том, что Python должен быть намного медленнее, чем Perl. Как уже неоднократно указывали другие, Python медленнее Perl для некоторых вещей и быстрее для других. Относительная производительность также часто зависит от вашего опыта работы с этими двумя языками.

Повторное использование функции во время исполнения скрипта

Допустим, у вас есть функция:

```
class Test:
   def check(self,a,b,c):
     if a == 0:
       self.str = b*100
     else:
       self.str = c*100

 a = Test()
 def example():
   for i in range(0,100000):
     a.check(i,"b","c")

 import profile
 profile.run("example()")
```

И предположим, что эта функция вызывается откуда-то еще много раз.

Ваш метод *check* будет иметь оператор *if*, замедляющий вас все время, кроме первой итерации, так что вы можете сделать это:

```
class Test2:
   def check(self,a,b,c):
     self.str = b*100
     self.check = self.check_post
   def check_post(self,a,b,c):
     self.str = c*100

 a = Test2()
 def example2():
   for i in range(0,100000):
     a.check(i,"b","c")

 import profile
 profile.run("example2()")
```

Этот пример «искусственный», но если *if* является довольно сложным выражением (или чем-то с большим количеством точек), вы можете сэкономить на его вычислении, если знаете, что оно будет истинным только в первый раз.

Профилирование кода

Первый шаг к ускорению вашей программы – это изучение того, где находятся узкие места. Вряд ли имеет смысл оптимизировать код, который никогда не выполняется или уже работает быстро.

Информация про профилирование: https://docs.python.org/3/library/profile.html