Профилирование в Python для ускорения вычислений

Практикум на ЭВМ 317 группы

Никита Юдин

We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil.
Donald Knuth

Профилирование — сбор статистики во время работы программы.
В дальнейшем будет рассмотрено использование памяти и измерение времени работы.

Содержание презентации

Pympler profiler

Pympler - средство разработки для измерения, наблюдения и анализа поведения объектов Python в работающем приложении Python.
Установка: $pip install pympler

Пример использования функции **asizeof**:

```
>>>from pympler import asizeof
>>>obj = [1, 2, (3, 4), 'text']
>>>asizeof.asizeof(obj)
176
>>>print asizeof.asized(obj, detail=1).format()
[1, 2, (3, 4), 'text'] size=176 flat=48
     (3, 4) size=64 flat=32
     'text' size=32 flat=32
     1 size=16 flat=16
     2 size=16 flat=16
```

Pympler profiler

Для детектирования утечек памяти у **Pympler** есть модуль **muppy**. Пример использования трекера изменения памяти:

```
>>> from pympler import tracker
>>> tr = tracker.SummaryTracker()
```

Точка выполнения, до которой ничего не изменилось

```
>>> tr.print_diff()
  types |   # objects |   total size
======= | =========== | ============
```

Pympler profiler

Добавим объектов

```
>>> i = 1
>>> l = [1,2,3,4]
>>> d = {}
>>> tr.print_diff()
  types |   # objects |   total size
======= | =========== | ============
   dict |           1 |    280      B
   list |           1 |    192      B
```

Память, занимаюмую числами, не отображает — для них память была выделена раньше в другом месте интерпретации.

Pympler profiler

В случае объектов, у которых есть утечки памяти, можно посмотреть дерево ссылок этих объектов:

```
>>> from pympler import refbrowser
>>> root = "some root object"
>>> root_ref1 = [root]
>>> root_ref2 = (root, )
>>> ib = refbrowser.InteractiveBrowser(root)
>>> ib.main()
```

Pympler profiler

Вот такое графическое представление имеет дерево ссылок для объекта root:

>>

Содержание презентации

Стандартные средства

Python имеет соответствующие модули для профилирования:

1) **cProfile**
2) **profile**
3) **pstats** - обработка статистики

```
1  import cProfile, pstats, io
2  pr = cProfile.Profile()
3  pr.enable()
4  # ...do something ...
5  pr.disable()
6  s = io.StringIO()
7  ps = pstats.Stats(pr, stream=s).sort_stats('↩
       cumulative')
8  ps.print_stats()
9  print(s.getvalue())
```

kernprof

Главная особенность — построчное профилирование программы. Есть возможность использования **Cprofile**. Установка:
$pip install line_profiler. Пример использования в командной строке:
$ kernprof -l script_to_profile.py

В самом скрипте нужно профилируемые функции обернуть декоратором **@profile**

```
1  @profile
2  def slow_function(a, b, c):
3          ...
```

Просмотр результатов:
$ python -m line_profiler script_to_profile.py.lprof
$ python -m pstats script_to_profile.py.lprof

Профилирование в **IPython Notebook**:

```
%prun -s cumulative <expr>
```

Построчное профилирование в **IPython Notebook**:

```
%load_ext line_profiler
%lprun -f <function> <expression>
%lprun -f my_func my_func(1, 2, 'text')
```

Вывод построчного профилирования **line_prof**:

```
>>
         Time  Per Hit   % Time  Line Contents
==============================================
                                      @wraps(f)
                                      def modified_f(*args, **kwargs):
           3      3.0      0.0          X = args[chunked_arg]
           1      1.0      0.0          pre_args = args[:chunked_arg]
           1      1.0      0.0          post_args = args[chunked_arg+1:]

           5      5.0      0.0          real_n_chunks = min(n_chunks, X.sh
         111    111.0      0.0          X_chunks = np.array_split(X, real_

           2      2.0      0.0          if not parallel:
    16251324 16251324.0     97.2              result_chunks = list(map(lam
                                      else:
```

vprof

vprof — пакет, предоставляющий интерактивные визуализации Python программ для таких характеристик, как время выполнения и использование памяти. Установка: $pip install vprof.
Использование: $vprof -c <modes> <program>

- c граф вызовов функций
- m статистика использования памяти
- h тепловая подсветка кода (часто выполняемые участки ярче)

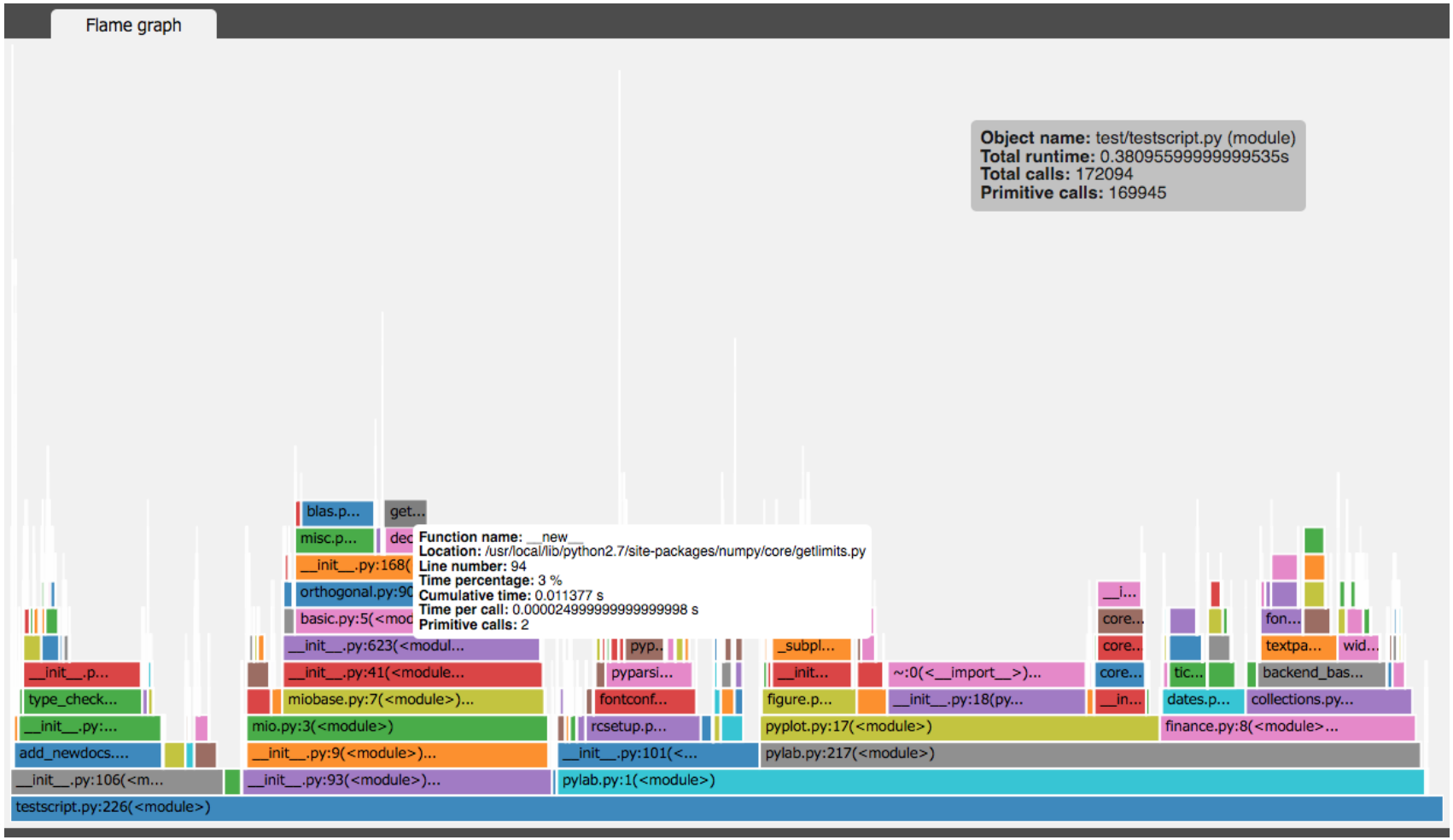
Использование **vprof** в Python коде

```
1  from vprof import profiler
2  def foo(arg1, arg2):
3      ...
4  profiler.run(foo, 'cmh', args=(arg1, arg2), ↩
       host='localhost', port=8000)
```

'cmh' = c | m | h - режимы профилирования

Результат профилирования будет выведен в новой вкладке браузера

Вызовы функций:



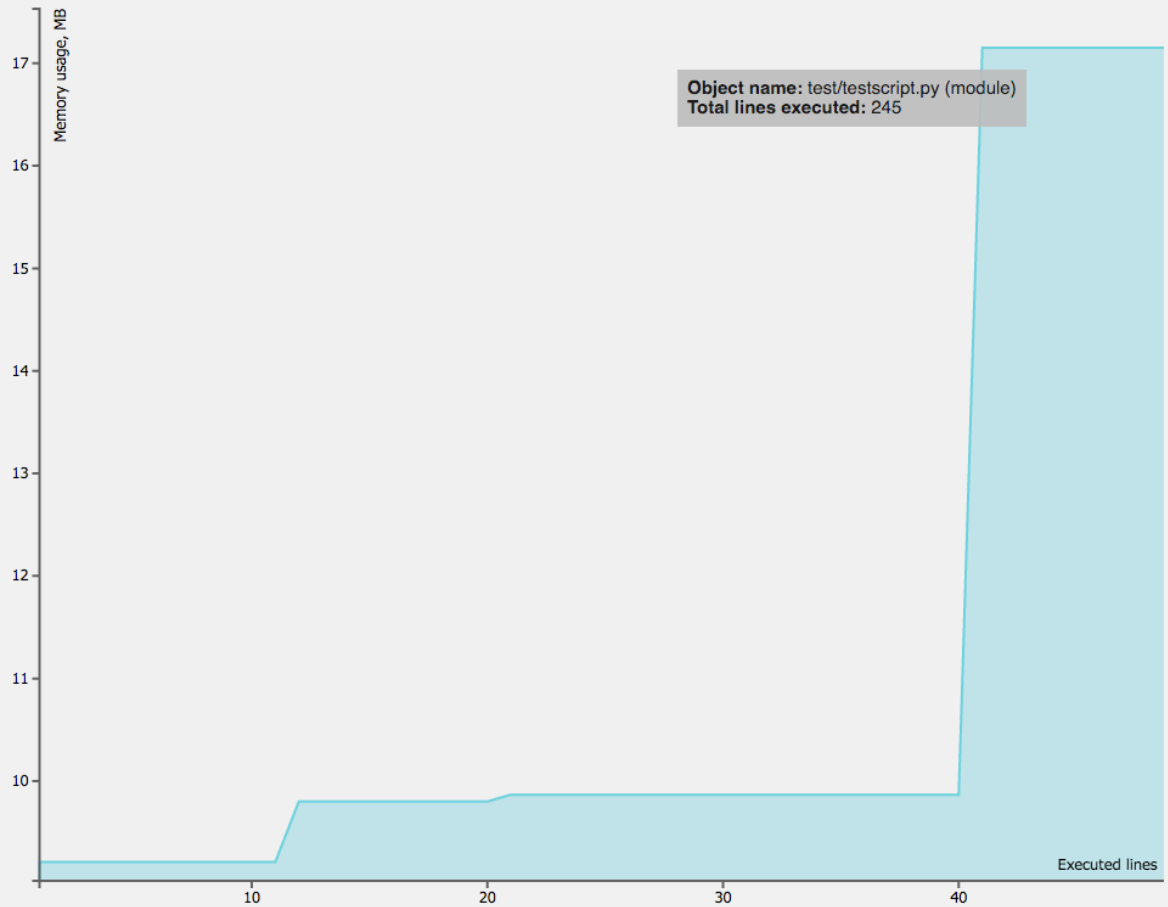
>>

Использование памяти:

Memory stats

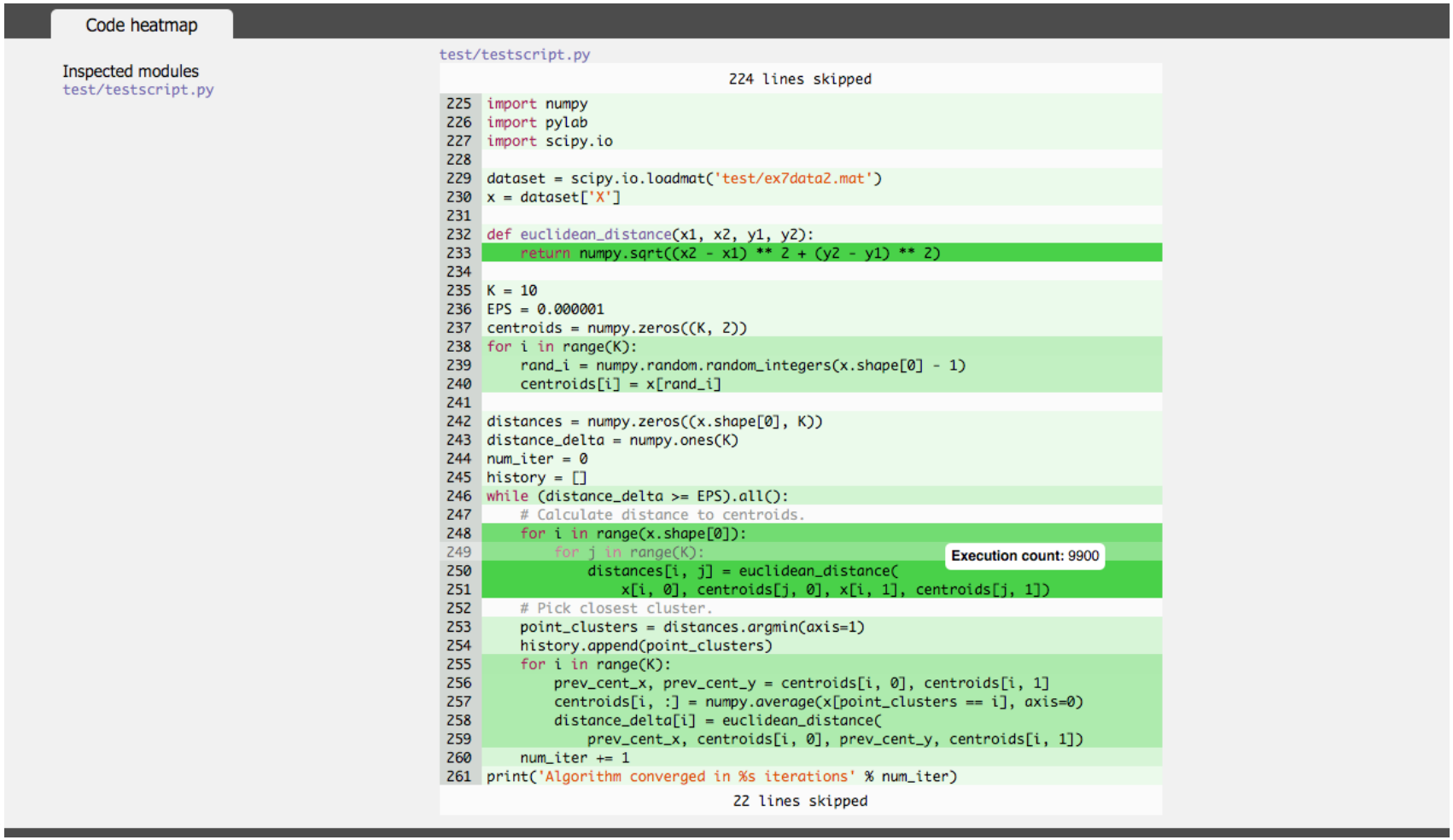
Objects left in memory

Object type	Count
function	2286
dict	589
tuple	471
wrapper_descriptor	434
getset_descriptor	404
weakref	312
method_descriptor	247
builtin_function_or_method	228
list	175
type	174
_frozen_importlib.ModuleSpec	101
module	101
_frozen_importlib_external.SourceFileLoader	92
staticmethod	84
cell	64
property	62
set	62
StgDict	37
_ctypes.PyCSimpleType	27
numpy.ma.core._MaskedUnaryOperation	26
numpy.ma.core._frommethod	23
method	21
member_descriptor	20
numpy.testing.nosetester.NoseTester	19
numpy.ma.core._MaskedBinaryOperation	18
classmethod	16
_weakrefset.WeakSet	15
_frozen_importlib_external.FileFinder	14
numpy.ma.extras._fromnxfunction	9
frozenset	9
_frozen_importlib_external.ExtensionFileLoader	8
numpy.ma.core._convert2ma	8
operator.itemgetter	8
_ctypes.PyCFuncPtrType	7
numpy.ma.core._DomainedBinaryOperation	6
numpy.core.numerictypes._typedict	6
numpy.ma.core._DomainSafeDivide	6



>>

Вызовы функций:



pprofile

pprofile — профилировщик со спецификацией для работы с нитями, написанный на чистом Python (https://github.com/vpelletier/pprofile).
Использование в командной строке:
$pprofile some_python_executable arg1 ...
Сохранение в формат **Callgrind Profile Format** для просмотра в **kcachegrind**(или в **qcachegrind**):
$pprofile –format callgrind –out cachegrind.out.threads demo/threads.py

pprofile

```
1  import pprofile
2  def someHotSpotCallable():# Deterministic
3      prof = pprofile.Profile()
4      with prof():
5          # Code to profile
6      prof.print_stats()
7  def someOtherHotSpotCallable():# Statistic
8      prof = pprofile.StatisticalProfile()
9      with prof(
10         period=0.001, # Sample every 1ms
11         single=True, # Only sample current ↩
               thread
12     ):
13         # Code to profile
14     prof.print_stats()
```

Содержание презентации

profilestats2.0

Декоратор для профилирования отдельных функций и преобразования результатов в формат, пригодный для визуализации с помощью **kcachegrind/qcachegrind(Qt)** (анализатор дерева вызовов)
Загрузить данный декоратор можно из https://pypi.python.org/pypi/profilestats/

Пример использования:

```
1  from profilestats import profile
2
3  @profile(print_stats=10, dump_stats=True)
4  def my_function(args, etc):
5      pass
```

Передаваемые аргументы:

1) cumulative (default: True) - собирать данные для нескольких вызовов в один отчёт
2) print_stats (default: 0) - длина вывода
3) sort_stats (default: ‘cumulative’) - тип сортировки статистики
4) dump_stats (default: False) - сохранять в файл
5) profile_filename (default: ‘profilestats.out’) - отчёт в формате Python
6) callgrind_filename (default: ‘callgrind.out’) - отчёт в формате kqcachegrind

pyprof2calltree

pyprof2calltree — пакет, содержащий в себе скрипт, позволяющий визуализировать результаты профилирования **cProfile** с помощью графического анализатора дерева вызовов **kcachegrind**.
Загрузить пакет можно из https://pypi.python.org/pypi/pyprof2calltree/

pyprof2calltree

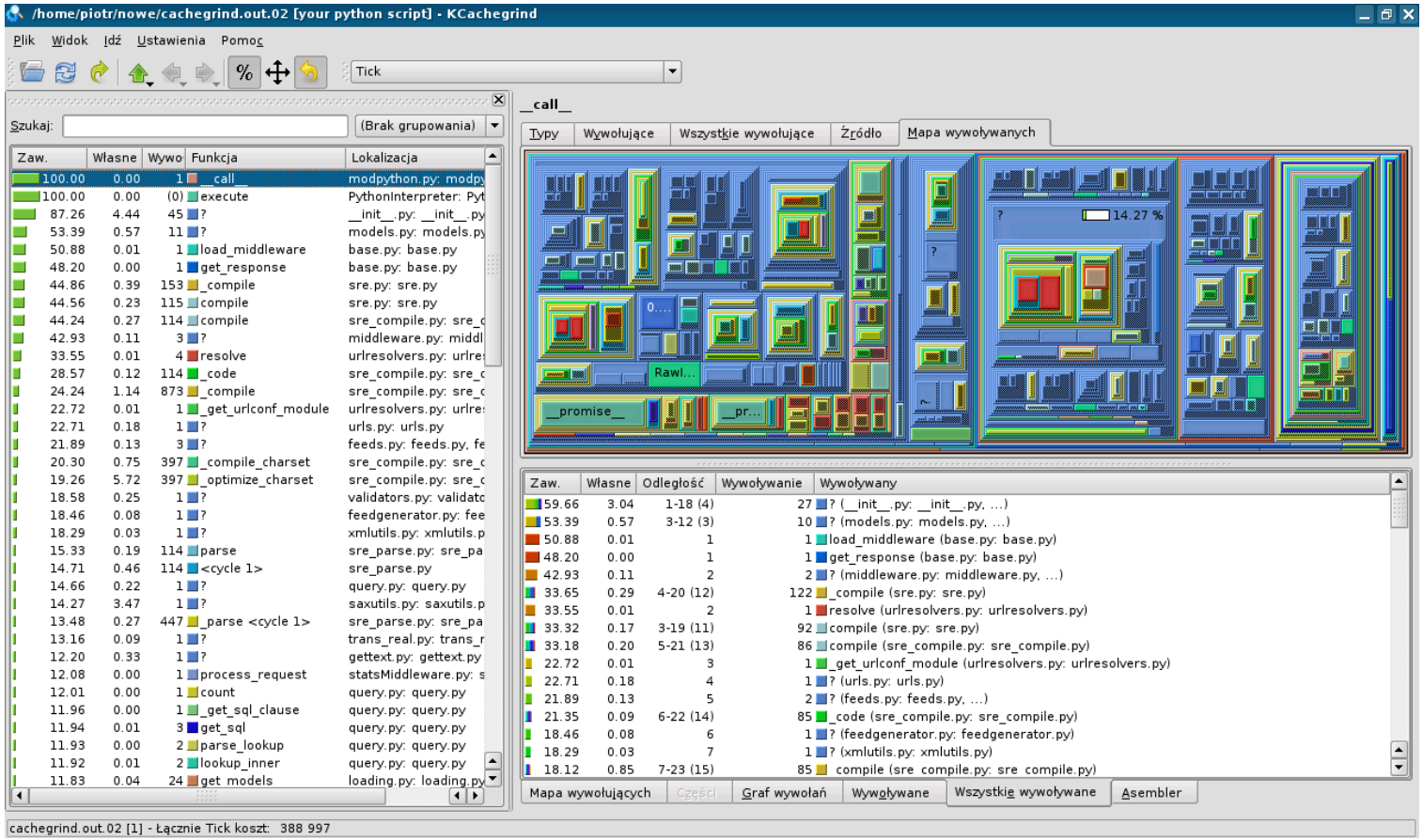
Пример использования в интерпретаторе Python:

```
>>> from xml.etree import ElementTree
>>> from cProfile import Profile
>>> xml_content = '<a>\n' + '\t<b/><c><d>text</d></c>\n' * 100 + '</a>'
>>> profiler = Profile()
>>> profiler.runctx("ElementTree.fromstring(xml_content)", locals(), globals())
>>> from pyprof2calltree import convert, visualize
>>> visualize(profiler.getstats())
>>> convert(profiler.getstats(), 'profiling_results.kgrind')
```

pyprof2calltree в IPython

```
In [1]: %doctest_mode
Exception reporting mode: Plain
Doctest mode is: ON
>>> from xml.etree import ElementTree
>>> xml_content = '<a>\n' + '\t<b/><c><d>text</d></c>\n' * 100 + '</a>'
>>> %prun -D out.stats ElementTree.fromstring(xml_content)
*** Profile stats marshalled to file 'out.stats'
>>> from pyprof2calltree import convert, visualize
>>> visualize('out.stats')
>>> convert('out.stats', 'out.kgrind')
>>> results = %prun -r ElementTree.fromstring(xml_content)
>>> visualize(results)
```

pyprof2calltree



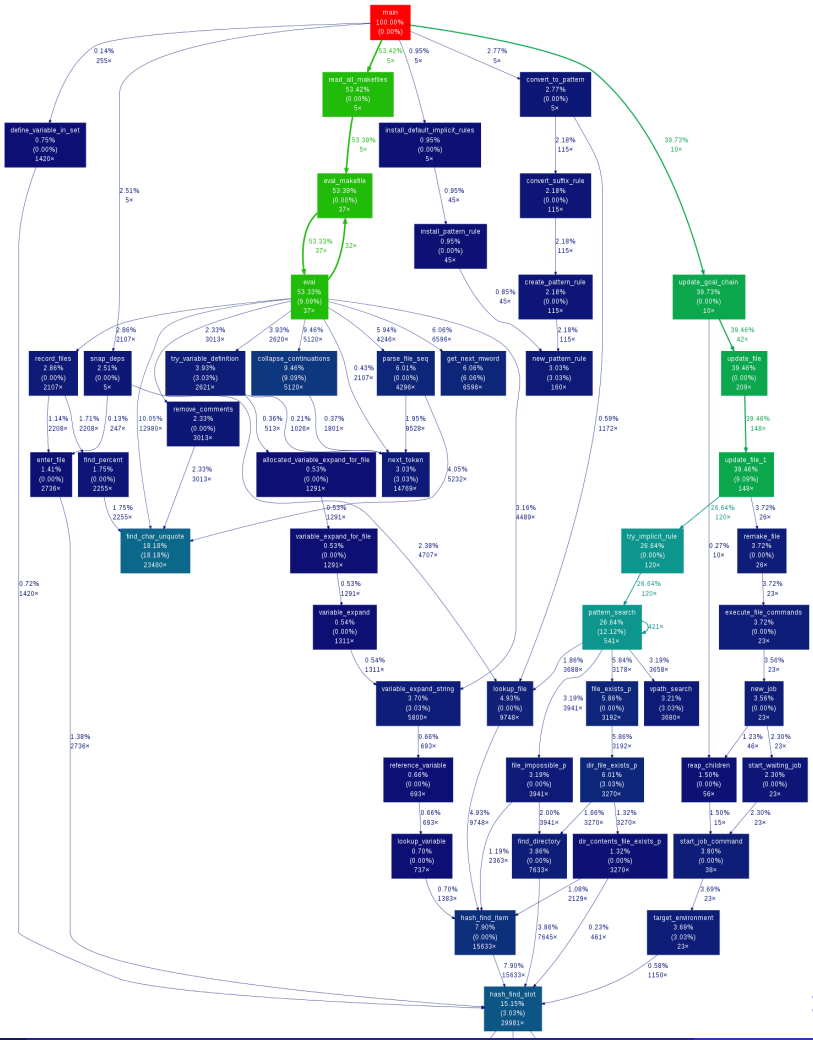
>>

gprof2dot

gprof2dot — Python скрипт, преобразующий вывод из многих профилировщиков в формат dot graph.
Установка: $pip install gprof2dot
Использование: $gprof2dot.py [options] [file] ...
Пример визуализации: (dot из пакета graphviz)
gprof:
$/path/to/your/executable arg1 arg2
$gprof path/to/your/executable | gprof2dot.py | dot -Tpng -o output.png
python profile:
$python -m profile -o output.pstats path/to/your/script arg1 arg2
$gprof2dot.py -f pstats output.pstats | dot -Tpng -o output.png

gprof2dot

Для интерактивного просмотра графа вызовов существует скрипт **xdot.py** (https://pypi.python.org/pypi/xdot). Использование:
$dot.py [file]



RunSnakeRun

RunSnakeRun — простая программа для просмотра результатов профилирования **cProfile**, **Profile**.
Особенности:

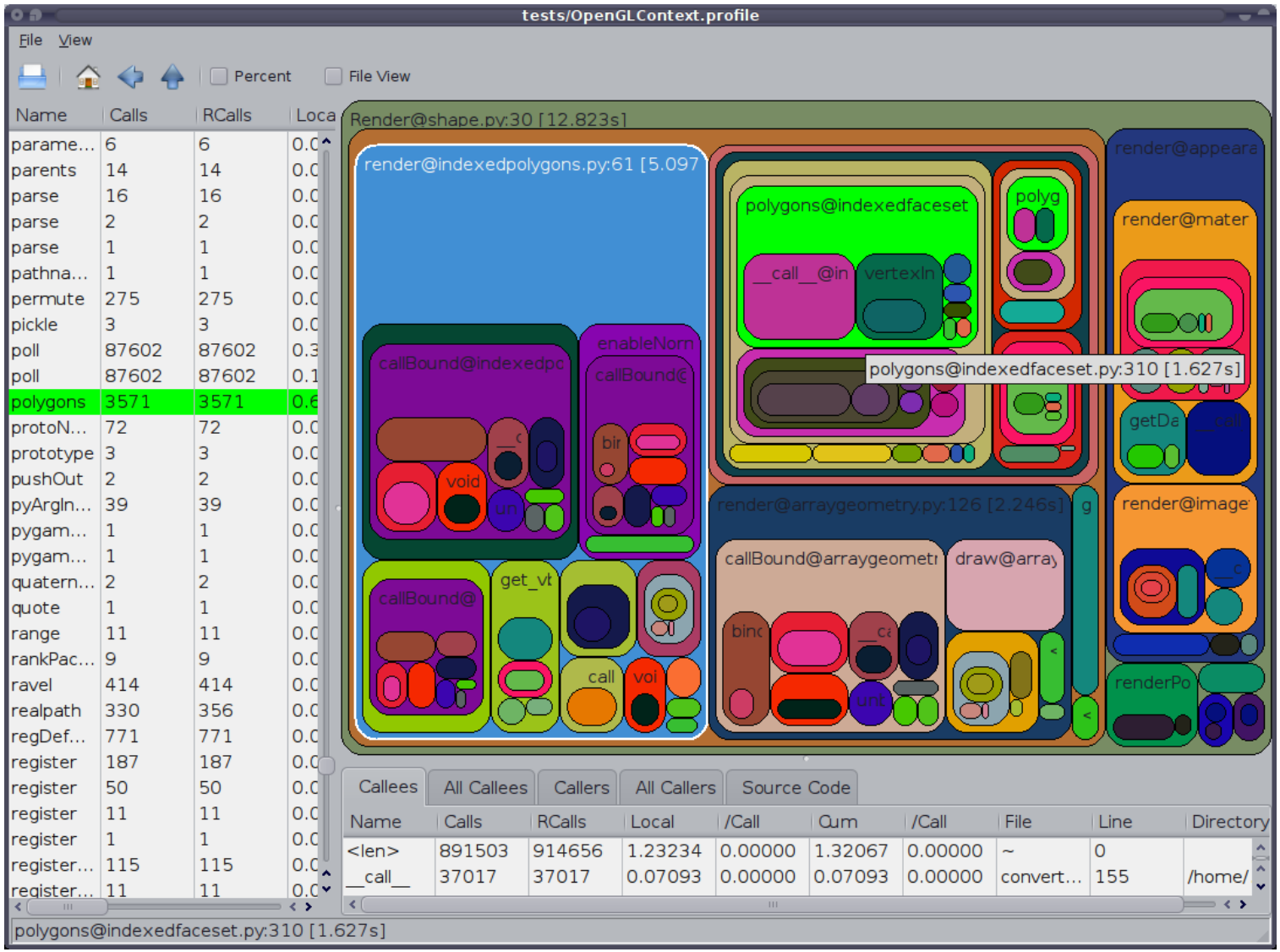
- Сортируемая сетка данных
- Дерево вызовов
- Просмотр пакетов, модулей, функций

Пример визуализации вывода **cProfile**:

```
1  import cProfile
2  command = """reactor.run()"""
3  cProfile.runctx( command, globals(), locals()↩
       , filename="OpenGLContext.profile" )
```

$python runsnake.py OpenGLContext.profile

RunSnakeRun



SnakeViz

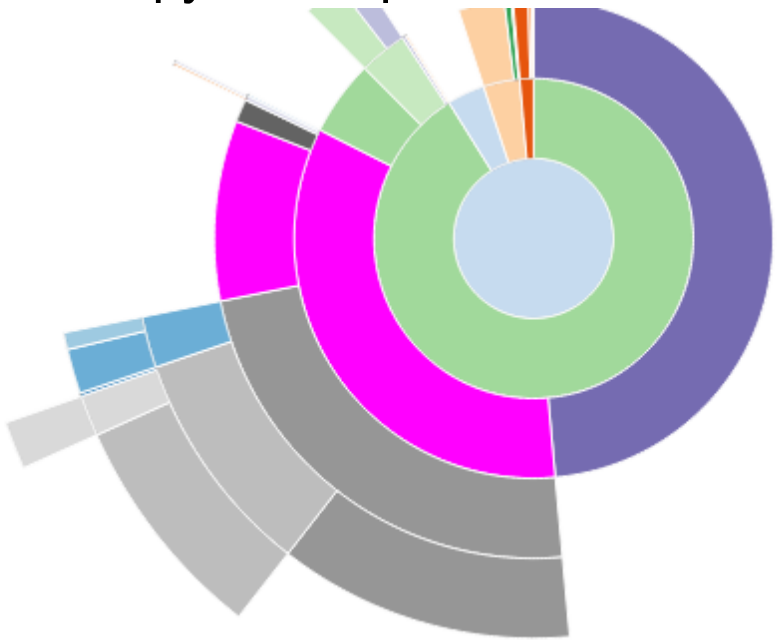
SnakeViz — браузерный визуализатор вывода профилировщика **cProfile**. Создание данного пакета было вдохновлено программой **RunSnakeRun**. Установка: $pip install snakeviz.
Пример использования: $snakeviz program.prof
В **SnakeViz** входит также **IPython magic**.

IPython magic:

```
%load_ext snakeviz
% snakeviz glob.glob('*.txt')#line-by-line
%%snakeviz#whole block of code
files = glob.glob('*.txt')
for file in files:
    with open(file) as f:
        print(hashlib.md5(f.read().encode('↩
            utf-8')).hexdigest())
```

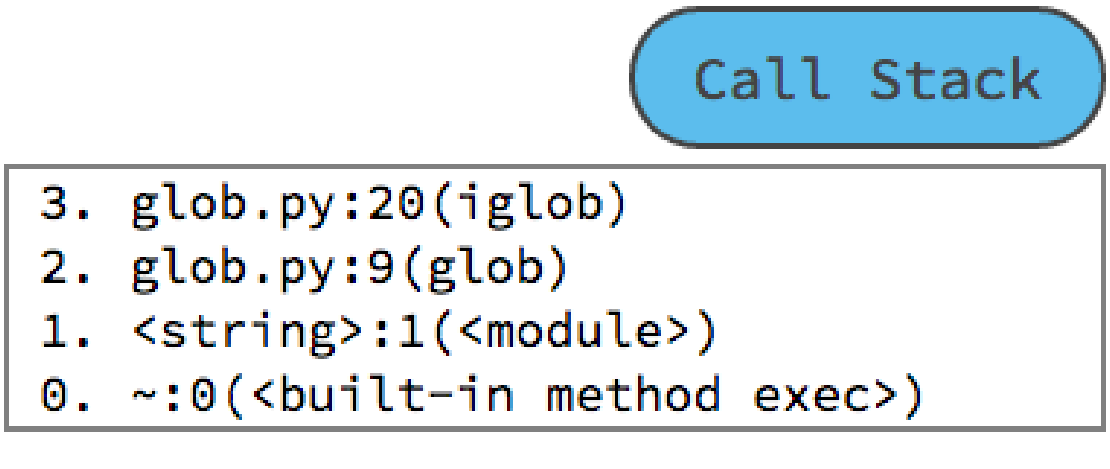
Сектора — функции, центральный круг — первая вызванная функция

Name:
filter
Cumulative Time:
0.000294 s (31.78 %)
File:
fnmatch.py
Line:
48
Directory:
/Users/jiffyclub/miniconda3/en
vs/snakevizdev/lib/python3.4/



>>

Стек вызовов (появляется при нажатии правой кнопки «мыши»):



Статистика вызовов функций:

Search:

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
1	0.000421	0.000421	0.000421	0.000421	~:0(<built-in method listdir>)
1	0.000104	0.000104	0.000202	0.000202	functools.py:441(wrapper)
1	7.9e-05	7.9e-05	0.000294	0.000294	fnmatch.py:48(filter)
1	6.7e-05	6.7e-05	8e-05	8e-05	functools.py:342(_make_key)
1	4.4e-05	4.4e-05	0.00079	0.00079	glob.py:61(glob1)

>>

Источники

- https://pythonhosted.org/Pympler/
- https://docs.python.org/3/library/profile.html
- https://github.com/rkern/line_profiler
- https://github.com/nvdv/vprof
- https://github.com/vpelletier/pprofile
- https://pypi.python.org/pypi/profilestats/
- https://pypi.python.org/pypi/pyprof2calltree/
- https://github.com/jrfonseca/gprof2dot
- http://www.vrplumber.com/programming/runsnakerun/
- https://jiffyclub.github.io/snakeviz/