Python Call Graph

Release 1.0.1

Gerald Kaszuba

March 29, 2016

Contents

1 Screenshots **3**

2 Project Status **5**

3 Features **7**

4 Quick Start **9**

5 Documentation Index **11**

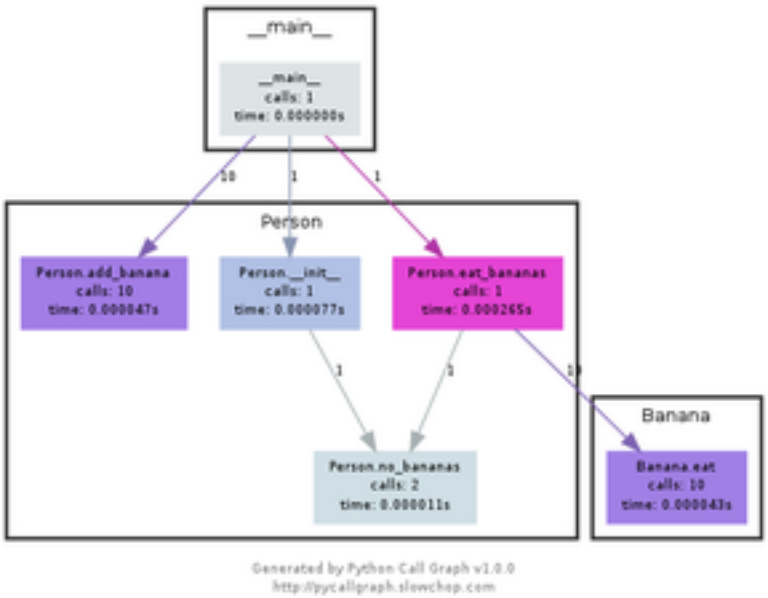
5.1 Usage Guide . . . 11
5.1.1 Intro . . . 11
5.1.2 Outputs . . . 11
5.1.3 Filtering . . . 12
5.1.4 Command-line Usage . . . 15
5.1.5 Custom Outputs . . . 17
5.2 Examples . . . 17
5.2.1 Basic . . . 17
5.2.2 Regular Expressions (Grouped) . . . 18
5.2.3 Regular Expressions (Ungrouped) . . . 20
5.3 API Classes . . . 22
5.3.1 `PyCallGraph` — Main interface to Python Call Graph . . . 22
5.3.2 `output.Output` — Base class for all output modules . . . 22
5.3.3 `globbing_filter.GlobbingFilter` — Class used for filtering methods . . . 23
5.4 Internal Classes . . . 23
5.4.1 `SyncronousTracer` . . . 23

Welcome! Python Call Graph is a Python module that creates call graph visualizations for Python applications.

CHAPTER 1

Screenshots

Click on the images below to see a larger version and the source code that generated them.



CHAPTER 2

Project Status

The latest version is **1.0.1** which was released on 2013-09-17, and is a backwards incompatbile from the previous release.

The project lives on GitHub, where you can report issues, contribute to the project by forking the project then creating a pull request, or just browse the source code.

The documentation needs some work stiil. Feel free to contribute :)

CHAPTER 3

Features

- Support for Python 2.7+ and Python 3.3+.
- Static visualizations of the call graph using various tools such as Graphviz and Gephi.
- Execute pycallgraph from the command line or import it in your code.
- Customisable colors. You can programatically set the colors based on number of calls, time taken, memory usage, etc.
- Modules can be visually grouped together.
- Easily extendable to create your own output formats.

CHAPTER 4

Quick Start

Installation is easy as:

```
pip install pycallgraph
```

You can either use the *command-line interface* for a quick visualization of your Python script, or the *pycallgraph module* for more fine-grained settings.

The following examples specify graphviz as the outputter, so it's required to be installed. They will generate a file called **pycallgraph.png**.

The command-line method of running pycallgraph is:

```
$ pycallgraph graphviz -- ./mypythonscript.py
```

A simple use of the API is:

```
from pycallgraph import PyCallGraph
from pycallgraph.output import GraphvizOutput

with PyCallGraph(output=GraphvizOutput()):
    code_to_profile()
```

CHAPTER 5

Documentation Index

5.1 Usage Guide

5.1.1 Intro

Python Call Graph was made to be a visual profiling tool for Python applications. It uses a debugging Python function called sys.set_trace() which makes a callback every time your code enters or leaves function. This allows Python Call Graph to track the name of every function called, as well as which function called which, the time taken within each function, number of calls, etc.

It is able to generate different types of *outputs and visualizations*. Initially Python Call Graph was only used to generate DOT files for GraphViz, and as of version 1.0.0, it can also generate JSON files, and GDF files for Gephi. Creating *custom outputs* is fairly easy by subclassing the *Output* class.

You can either use the *command-line interface* for a quick visualization of your Python script, or the *pycallgraph module* for more fine-grained settings.

Todo

Add some examples and screenshots

5.1.2 Outputs

Graphviz

This output leverages the GraphViz graph generation tool. You'll need it to be installed before attempting to use it.

Gephi

This output generates a GDF file that can be used with Gephi.

Todo

Expand this section with screenshots and examples.

5.1.3 Filtering

Banana

Filtering is sometimes needed when the output of Python Call Graph is overwhelming, or if you want to only measure a small portion of your program. The filtering guide below is based on the filter.py example.

Let's demonstrate with a class that can eat a banana:

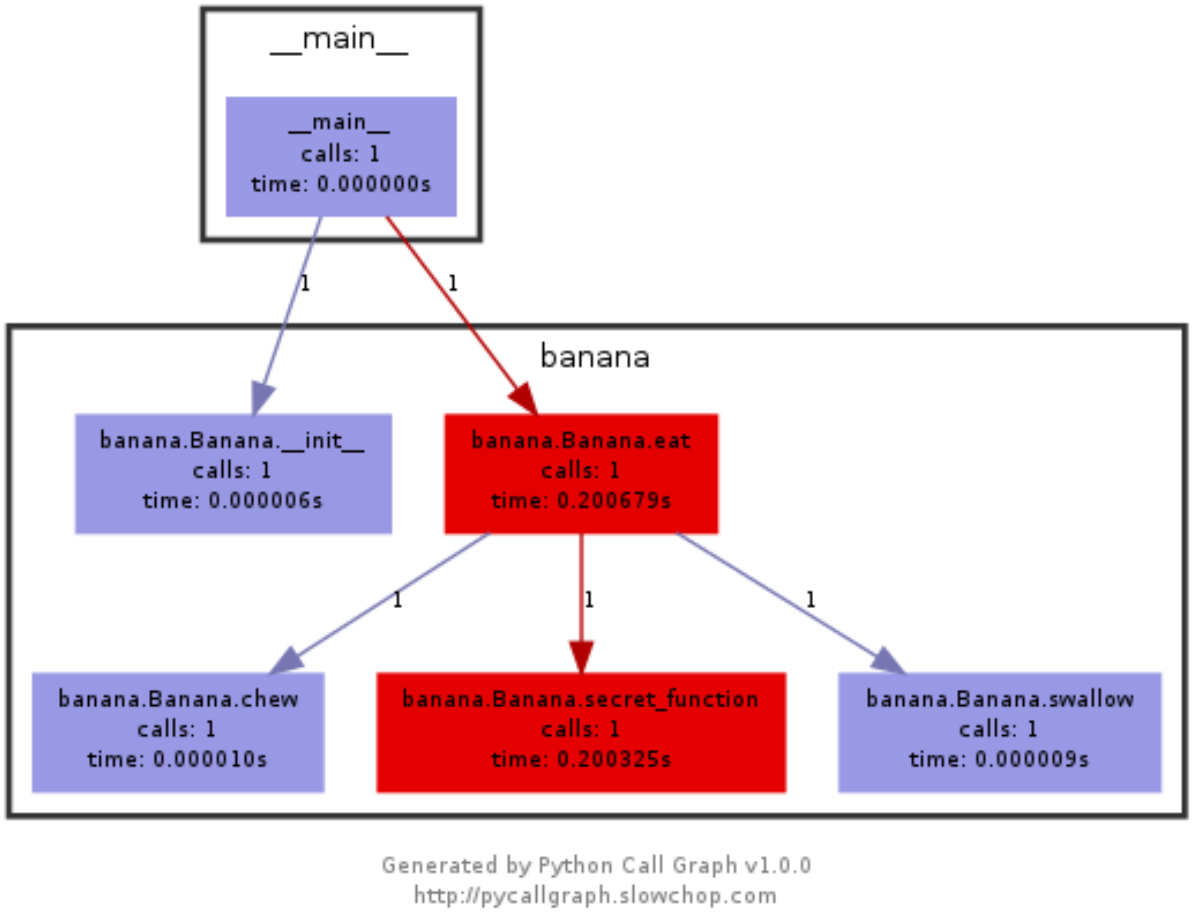
```
import time


class Banana:

    def __init__(self):
        pass

    def eat(self):
        self.secret_function()
        self.chew()
        self.swallow()

    def secret_function(self):
        time.sleep(0.2)

    def chew(self):
        pass

    def swallow(self):
        pass
```

No Filter

The code to measure it without any configuration, apart from the output file:

```
#!/usr/bin/env python

from pycallgraph import PyCallGraph
from pycallgraph.output import GraphvizOutput

from banana import Banana


graphviz = GraphvizOutput(output_file='filter_none.png')

with PyCallGraph(output=graphviz):
    banana = Banana()
    banana.eat()
```

The Graphviz output after running the measurement code:



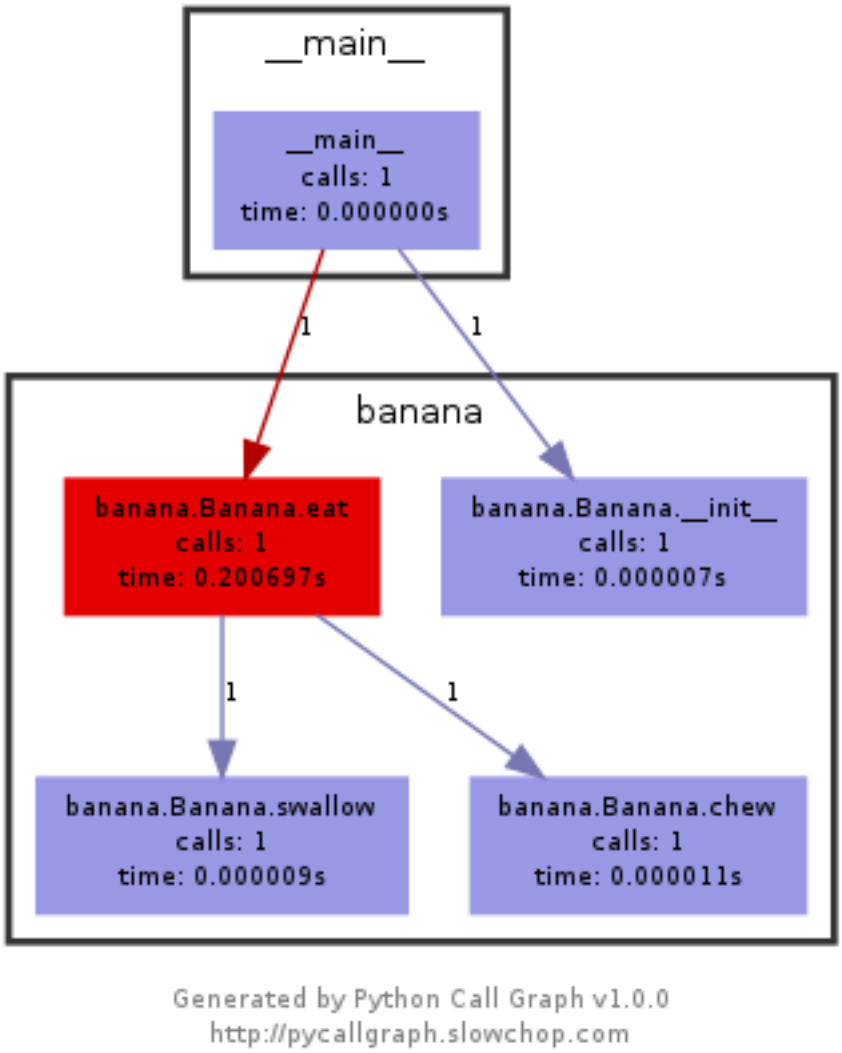
Hide the secret

Probably need to hide that **secret_function**. Create a *GlobbingFilter* which excludes **secret_function** along with **pycallgraph** so we don't see the internals. Add that filter to the config option called **trace_filter**:

```
#!/usr/bin/env python

from pycallgraph import PyCallGraph
from pycallgraph import Config
from pycallgraph import GlobbingFilter
from pycallgraph.output import GraphvizOutput

from banana import Banana


config = Config()
config.trace_filter = GlobbingFilter(exclude=[
    'pycallgraph.*',
    '*.secret_function',
])

graphviz = GraphvizOutput(output_file='filter_exclude.png')

with PyCallGraph(output=graphviz, config=config):
```

```
    banana = Banana()
    banana.eat()
```

And the output:



You can also use "include" as well as "exclude" in the *GlobbingFilter*.

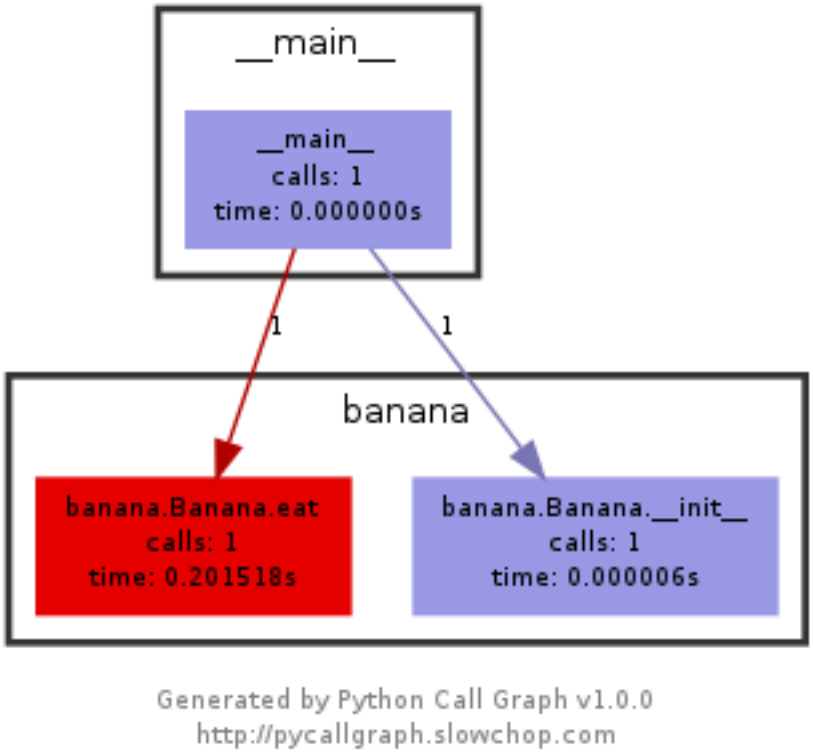
Maximum Depth

Let's say you're only interested in the first level of calls. You can specify this using **config.max_depth**:

```
#!/usr/bin/env python

from pycallgraph import PyCallGraph
from pycallgraph import Config
from pycallgraph.output import GraphvizOutput

from banana import Banana
```

```
config = Config(max_depth=1)
graphviz = GraphvizOutput(output_file='filter_max_depth.png')

with PyCallGraph(output=graphviz, config=config):
    banana = Banana()
    banana.eat()
```

And the output:



5.1.4 Command-line Usage

Synopsis

pycallgraph [*OPTION*]... *OUTPUT_MODE* [*OUTPUT_OPTIONS*] *python_file.py*

Description

OUTPUT_MODE can be one of graphviz, gephi and json. *python_file.py* is a python script that will be traced and afterwards, a call graph visualization will be generated.

General Arguments

<OUTPUT_MODE>
A choice of graphviz, gephi and json.

-h, --help
Shows a list of possible options for the command line.

-v, --verbose
Turns on verbose mode which will print out information of pycallgraph's state and processing.

-d, --debug
Turns on debug mode which will print out debugging information such as the raw Graphviz generated files.

-ng, --no-groups
Do not group modules in the results. By default this is turned on and will visually group together methods of the same module. The technique of grouping does rely on the type of output used.

-s, --stdlib
When running a trace, also include the Python standard library.

-m, --memory
An experimental option which includes memory tracking in the trace.

-t, --threaded
An experimental option which processes the trace in another thread. This may or may not be faster.

Filtering Arguments

-i, --include <pattern>
Wildcard pattern of modules to include in the output. You can have multiple include arguments.

-e, --exclude <pattern>
Wildcard pattern of modules to exclude in the output. You can have multiple include arguments.

--include-pycallgraph
By default pycallgraph filters itself out of the trace. Enabling this will include pycallgraph in the trace.

--max-depth
Maximum stack depth to trace. Any calls made past this stack depth are not included in the trace.

Graphviz Arguments

-l <tool>, **--tool** <tool>
Modify the default Graphviz tool used by pycallgraph. It uses "dot", but can be changed to either neato, fdp, sfdp, twopi, or circo.

Examples

Create a call graph image called pycallgraph.png on myprogram.py:

```
pycallgraph graphviz -- ./myprogram.py
```

Create a call graph of a standard Python installation script with command line parameters:

```
pycallgraph graphviz --output-file=setup.png -- setup.py --dry-run install
```

Run Django's *manage.py* script, but since there are many calls within Django, and will cause a massively sized generated image, we can filter it to only trace the core Django modules:

```
pycallgraph -v --stdlib --include "django.core.*" graphviz -- ./manage.py syncdb --noinput
```

5.1.5 Custom Outputs

Todo

Sorry, this section needs some work :) Feel free to contribute!

5.2 Examples

5.2.1 Basic

A simple Python example with two classes.

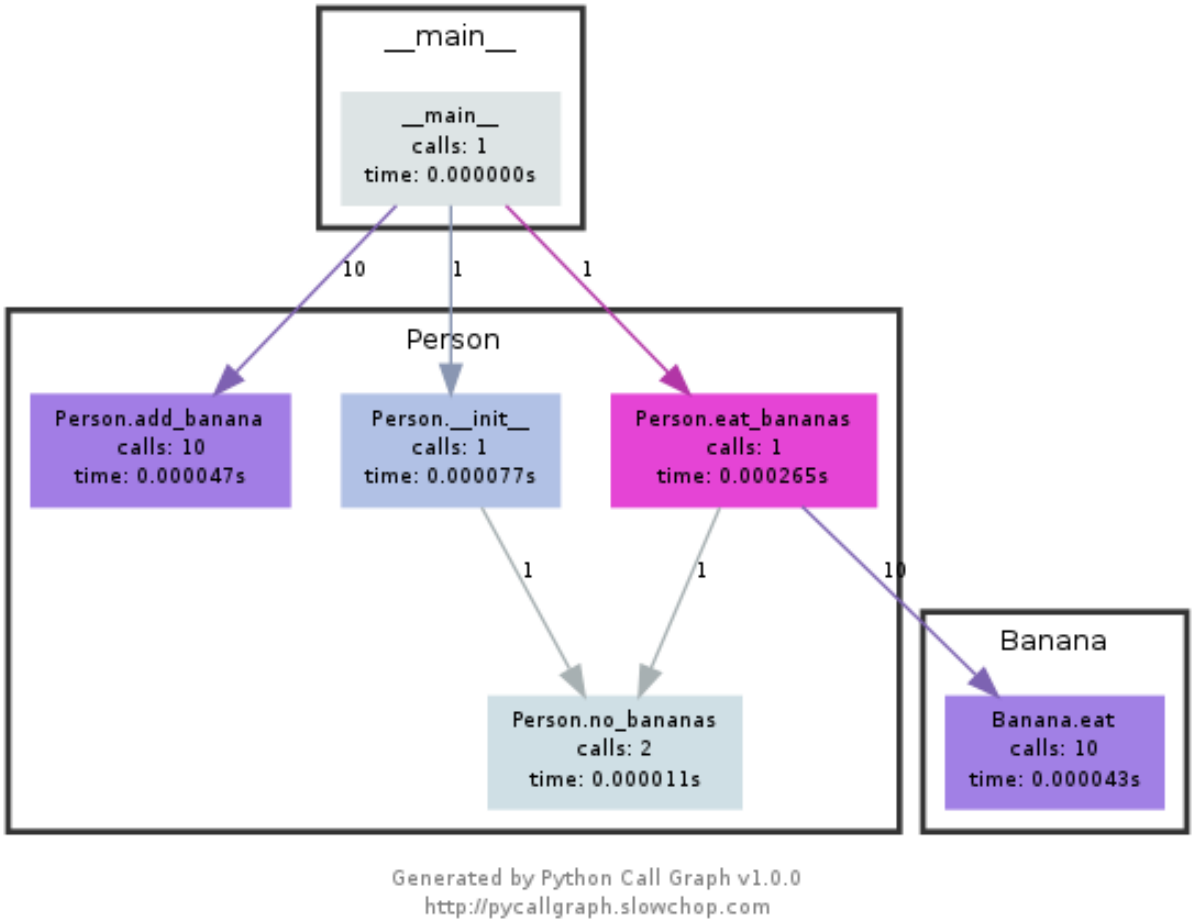
Source Code

```
#!/usr/bin/env python
'''
This example demonstrates a simple use of pycallgraph.
'''
from pycallgraph import PyCallGraph
from pycallgraph.output import GraphvizOutput


class Banana:

    def eat(self):
        pass


class Person:

    def __init__(self):
        self.no_bananas()

    def no_bananas(self):
        self.bananas = []

    def add_banana(self, banana):
        self.bananas.append(banana)

    def eat_bananas(self):
        [banana.eat() for banana in self.bananas]
        self.no_bananas()


def main():
    graphviz = GraphvizOutput()
    graphviz.output_file = 'basic.png'

    with PyCallGraph(output=graphviz):
        person = Person()
        for a in xrange(10):
            person.add_banana(Banana())
        person.eat_bananas()
```

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

Generated Image

Below is the generated image from the code above. If you're having issues with the image below, try the `direct` link to image.



5.2.2 Regular Expressions (Grouped)

See how a regular expression is constructed and matched. The example also shows the comparison between creating a regular expression object before matching, versus matching a "new" regular expression every iteration. See also *Regular Expressions (Ungrouped)*.

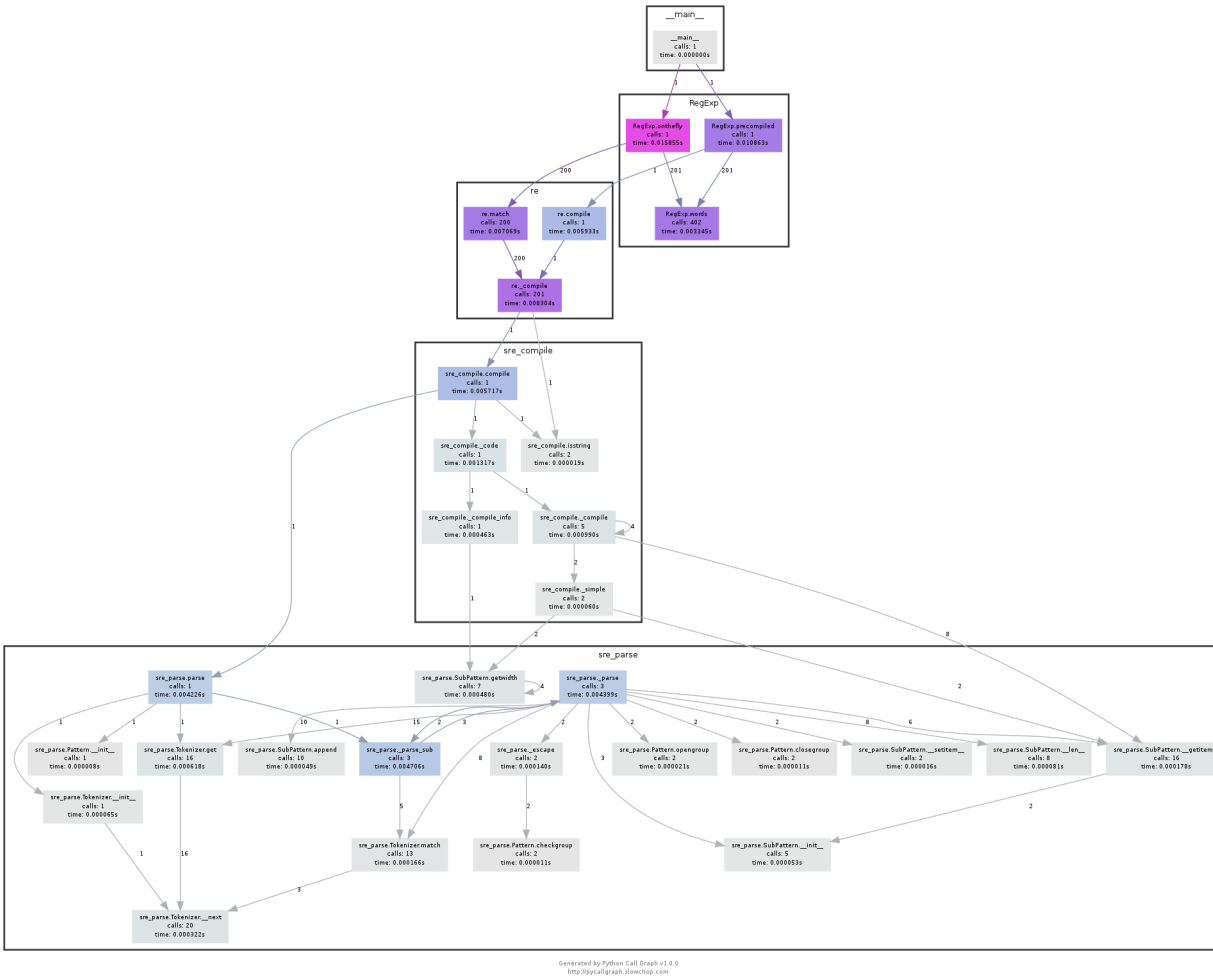
Source Code

```
#!/usr/bin/env python
'''
Runs a regular expression over the first few hundred words in a dictionary to
find if any words start and end with the same letter, and having two of the
```

```
same letters in a row.
'''
import argparse
import re

from pycallgraph import PyCallGraph
from pycallgraph import Config
from pycallgraph.output import GraphvizOutput


class RegExp(object):

    def main(self):
        parser = argparse.ArgumentParser()
        parser.add_argument('--grouped', action='store_true')
        conf = parser.parse_args()

        if conf.grouped:
            self.run('regexp_grouped.png', Config(groups=True))
        else:
            self.run('regexp_ungrouped.png', Config(groups=False))

    def run(self, output, config):
        graphviz = GraphvizOutput()
        graphviz.output_file = output
        self.expression = r'^([^s]).*(.)\2.*\1$'

        with PyCallGraph(config=config, output=graphviz):
            self.precompiled()
            self.onthefly()

    def words(self):
        a = 200
        for word in open('/usr/share/dict/words'):
            yield word.strip()
            a -= 1
            if not a:
                return


    def precompiled(self):
        reo = re.compile(self.expression)
        for word in self.words():
            reo.match(word)

    def onthefly(self):
        for word in self.words():
            re.match(self.expression, word)


if __name__ == '__main__':
    RegExp().main()
```

Generated Image

Below is the generated image from the code above. If you're having issues with the image below, try the `direct link to image`.



5.2.3 Regular Expressions (Ungrouped)

Similar to the *Regular Expressions (Grouped)* example, but without grouping turned on.

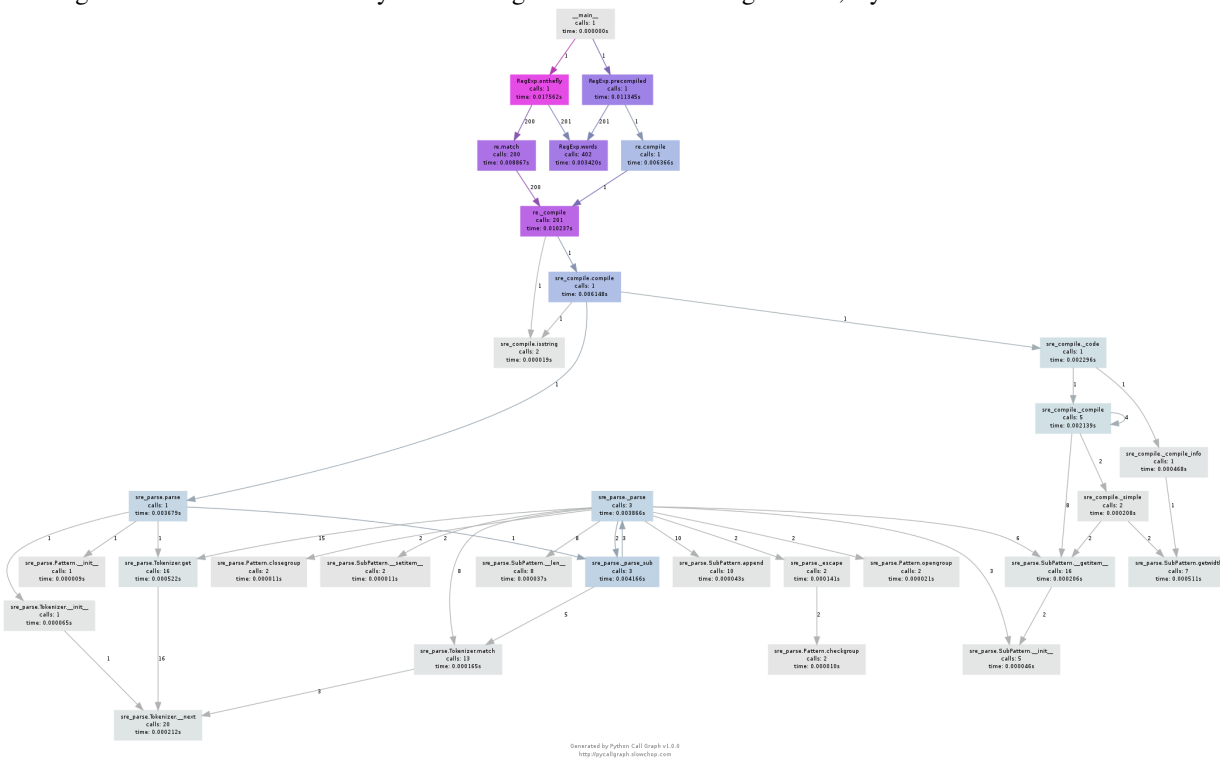
Source Code

```
#!/usr/bin/env python
'''
Runs a regular expression over the first few hundred words in a dictionary to
find if any words start and end with the same letter, and having two of the
same letters in a row.
'''
import argparse
import re

from pycallgraph import PyCallGraph
from pycallgraph import Config
from pycallgraph.output import GraphvizOutput
```

```
class RegExp(object):

    def main(self):
        parser = argparse.ArgumentParser()
        parser.add_argument('--grouped', action='store_true')
        conf = parser.parse_args()

        if conf.grouped:
            self.run('regexp_grouped.png', Config(groups=True))
        else:
            self.run('regexp_ungrouped.png', Config(groups=False))

    def run(self, output, config):
        graphviz = GraphvizOutput()
        graphviz.output_file = output
        self.expression = r'^([^s]).*(.)\2.*\1$'

        with PyCallGraph(config=config, output=graphviz):
            self.precompiled()
            self.onthefly()

    def words(self):
        a = 200
        for word in open('/usr/share/dict/words'):
            yield word.strip()
            a -= 1
            if not a:
                return


    def precompiled(self):
        reo = re.compile(self.expression)
        for word in self.words():
            reo.match(word)

    def onthefly(self):
        for word in self.words():
            re.match(self.expression, word)


if __name__ == '__main__':
    RegExp().main()
```

Generated Image

Below is the generated image from the code above. If you're having issues with the image below, try the `direct link to image`.



5.3 API Classes

5.3.1 `PyCallGraph` — Main interface to Python Call Graph

class `pycallgraph.`**`PyCallGraph`**(*output=None*, *config=None*)

`done`()
Stops the trace and tells the outputters to generate their output.

`reset`()
Resets all collected statistics. This is run automatically by start(reset=True) and when the class is initialized.

`start`(*reset=True*)
Begins a trace. Setting reset to True will reset all previously recorded trace data.

`stop`()
Stops the currently running trace, if any.

5.3.2 `output.Output` — Base class for all output modules

class `pycallgraph.output.`**`Output`**(***kwargs*)
Base class for all outputters.

done()
Called when the trace is complete and ready to be saved.

sanity_check()
Basic checks for certain libraries or external applications. Raise or warn if there is a problem.

set_config(*config*)
This is a quick hack to move the config variables set in Config into the output module config variables.

should_update()
Return True if the update method should be called periodically.

start()
Initialise variables after initial configuration.

update()
Called periodically during a trace, but only when should_update is set to True.

5.3.3 globbing_filter.GlobbingFilter — Class used for filtering methods

class pycallgraph.globbing_filter.**GlobbingFilter**(*include=None*, *exclude=None*)
Filter module names using a set of globs.

Objects are matched against the exclude list first, then the include list. Anything that passes through without matching either, is excluded.

5.4 Internal Classes

5.4.1 SyncronousTracer

Symbols

–include-pycallgraph
command line option, 16
–max-depth
command line option, 16
-d, –debug
command line option, 16
-e, –exclude <pattern>
command line option, 16
-h, –help
command line option, 15
-i, –include <pattern>
command line option, 16
-l <tool>, –tool <tool>
command line option, 16
-m, –memory
command line option, 16
-ng, –no-groups
command line option, 16
-s, –stdlib
command line option, 16
-t, –threaded
command line option, 16
-v, –verbose
command line option, 16

C

command line option
–include-pycallgraph, 16
–max-depth, 16
-d, –debug, 16
-e, –exclude <pattern>, 16
-h, –help, 15
-i, –include <pattern>, 16
-l <tool>, –tool <tool>, 16
-m, –memory, 16
-ng, –no-groups, 16
-s, –stdlib, 16
-t, –threaded, 16
-v, –verbose, 16

D

done() (pycallgraph.output.Output method), 22
done() (pycallgraph.PyCallGraph method), 22

G

GlobbingFilter (class in pycallgraph.globbing_filter), 23

O

Output (class in pycallgraph.output), 22

P

PyCallGraph (class in pycallgraph), 22

R

reset() (pycallgraph.PyCallGraph method), 22

S

sanity_check() (pycallgraph.output.Output method), 23
set_config() (pycallgraph.output.Output method), 23
should_update() (pycallgraph.output.Output method), 23
start() (pycallgraph.output.Output method), 23
start() (pycallgraph.PyCallGraph method), 22
stop() (pycallgraph.PyCallGraph method), 22

U

update() (pycallgraph.output.Output method), 23