VizTracer

Release 0.16.3

Tian Gao

Jun 12, 2024

GETTING STARTED

VizTracer is a low-overhead logging/debugging/profiling tool that can trace and visualize your python code to help you intuitively understand your code and figure out the time consuming part of your code.

VizTracer can display every function executed and the corresponding entry/exit time from the beginning of the program to the end, which is helpful for programmers to catch sporadic performance issues.

CHAPTER

ONE

INSTALLATION

VizTracer requires python 3.6+ and can work on Linux/MacOs/Windows. No other dependency. For now, VizTracer only supports CPython.

The preferred way to install VizTracer is via pip

```
pip install viztracer
```

You can also download the source code and build it yourself.

Even though VizTracer functions without any other packages, it is still recommended to install the following packages to have a better performance.

- orjson: better json dump/load performance

```
pip install orjson
```

Or you can install *full* version of viztracer:

```
pip install viztracer[full]
```

CHAPTER

TWO

BASIC USAGE

2.1 Command Line

The easiest way to use VizTracer is through command line. Assume you have a python script to profile and the normal way to run it is:

```
python3 my_script.py
```

You can simply use VizTracer by

```
# These two commands are equivalent.
# In this docs, they might both be used, but you can choose either one that you prefer.
viztracer my_script.py
# OR
python3 -m viztracer my_script.py
```

which will generate a `result.json` file in the directory you run this command. You can open it with `vizviewer`

```
vizviewer result.json
```

If your script needs arguments like

```
python3 my_script.py arg1 arg2
```

Just feed it as it is to `viztracer`

```
viztracer my_script.py arg1 arg2
```

It's possible that there's a conflict or an ambiguity. `viztracer` takes `--` as a separator between arguments to `viztracer` and positional arguments to your script.

```
viztracer -o result.json -- my_script.py -o output_for_my_script.json
```

You can also run a module with VizTracer

```
viztracer -m your_module
```

You can specify the output file using `-o` or `--output_file` argument. The default output file is `result.json`. Three types of files are supported, html, json and gz(gzip of json file).

```
viztracer -o other_name.html my_script.py
viztracer -o other_name.json my_script.py
viztracer -o other_name.json.gz my_script.py
```

You can make viztracer to generate a unique name for the output file by using `-u` or `--unique_output_file`

```
viztracer -u my_script.py
viztracer --output_dir ./my_reports -u my_script.py
```

You can also show flamegraph from `result.json` file

```
vizviewer --flamegraph result.json
```

2.2 Inline

Sometimes the command line may not work as you expected, or you do not want to profile the whole script. You can manually start/stop the profiling in your script as well.

First of all, you need to import `VizTracer` class from the package

```
from viztracer import VizTracer
```

You can trace code with the `with` statement

```
with VizTracer(output_file="optional.json") as tracer:
    # Something happens here
```

Or you can create a `VizTracer` object and manually enable/disable the profile using `start()` and `stop()` function.

```
tracer = VizTracer()
tracer.start()
# Something happens here
tracer.stop()
tracer.save() # also takes output_file as an optional argument
```

2.3 Jupyter

If you are using Jupyter, you can use viztracer cell magics.

```
# You need to load the extension first
%load_ext viztracer
```

```
%%viztracer
# Your code after
```

```
# you can define arguments of VizTracer in magic
%%viztracer -p 8888
# Your code after
```

A `Show VizTracer Report` button will appear after the cell and you can click it to view the results.

Cell magic `%%viztracer` supports some of the command line arguments:

- `--port`
- `--output_file`

- `--max_stack_depth`
- `--ignore_c_function`
- `--ignore_frozen`
- `--log_func_args`
- `--log_print`
- `--log_sparse`

2.4 Display Report

VizTracer will generate a `result.json` by default, which could be opened with `vizviewer`

```
vizviewer result.json
```

You can also display all the files in a directory and open the reports in browser too. This is helpful when you have many files in one directory and want to check some or all of them.

This could also be used when you have a report directory where reports are frequently added. You can leave `vizviewer` in the background and browse your reports with pure browser.

```
vizviewer your_directory/
```

`vizviewer` will bring up webbrowser and open the report by default. You can disable this feature and only host an HTTP server on `localhost:9001`, which you can access through your browser

```
vizviewer --server_only result.json
```

If you do not want to host the HTTP server forever, you can use `--once` so the server will shut down after serving the trace file

```
vizviewer --once result.json
```

You can serve your HTTP server on a different port with `--port` or its equivalent `-p`

```
vizviewer --port 10000 result.json
```

You can also show flamegraph of the result

```
vizviewer --flamegraph result.json
```

You can use the external trace processor with `--use_external_processor`, which does not have the RAM limits as the browser. This is helpful when you try to open a large trace file.

```
vizviewer --use_external_processor result.json
```

`vizviewer` can also show standalone html report - it just host a simple HTTP server for the file

```
vizviewer result.html
```

Or, you can use `--open` for `viztracer`, it will then open the report after it generates it

```
viztracer --open my_script.py
viztracer -o result.html --open my_script.py
```

2.5 Circular Buffer Size

VizTracer uses a circular buffer to store the entries. When there are too many entries, it will only store the latest ones so you know what happened recently. The default buffer size is 1,000,000(number of entries), which takes about 150MiB disk space. You can specify this when you instantiate a `VizTracer` object

Notice it also takes a significant amount of RAM when VizTracer is tracing the program.

VizTracer will preallocate about `tracer_entries * 100B` RAM for circular buffer. It also requires about `1-2MB` per 10k entries to dump the json file.

```
viztracer --tracer_entries 500000 my_script.py
```

OR

```
tracer = VizTracer(tracer_entries=500000)
```

2.6 Configuration file

You can use a configuration file to set the default options for `viztracer`, which could help you avoid typing the same arguments for multiple runs.

The default filename for `viztracer` configuration file is `.viztracerrc`. *viztracer* will try to find `.viztracerrc` in current working directory. You can also specify your own configuration file with `viztracer --rcfile <your_config_file>`. The format of the configuration file is very similar to `ini` file, which could be parsed by built in `configparser`.

```
[default]
log_var = a.* latest
ignore_c_function = True
output_file = vizreport.json
max_stack_depth = 10
```

`[default]` can't be omitted and all the arguments should be in a key-value pair format, where the key is the argument name(without `--`) and the val is the value you need to pass in. Please notice that there are some arguments in `viztracer` that do not take parameters(like *–ignore_c_function* `), you need to pass `True` in the config file to make the config parser happy. If you need to pass multiple parameters to an argument(like `log_var`), just use space to separate the parameters like you do in cmdline interface.

2.7 Combine Reports

VizTracer can put multiple json reports together and generate a new trace file. This is especially helpful when you have multiple trace generators, for example, running multiple processes with VizTracer. As VizTracer uses Monotonic Clock, you can save reports with different VizTracer instances without worrying about timestamp alignment issue. You can even generate your own data and combine with VizTracer reports, like VizPlugins does.

```
viztracer --combine process1.json process2.json -o full_report.json
```

Another usage of combining reports would be to compare between different runs of the same program. Unlike combining from multiple sources, this requires a pre-alignment of all the trace data. VizTracer also provides a way to align the start of all reports for this usage.

```
viztracer --align_combine run1.json run2.json -o compare_report.json
```

2.8 Compress Your Report

VizTracer supports compressing your json report. The general compression ratio is about 50:1 to 100:1 for a large report.

You can compress your report with `--compress`.

```
viztracer --compress result.json -o result.cvf
```

You can also decompress your report with `--decompress`

```
viztracer --decompress result.cvf -o result.json
```

CHAPTER

THREE

GLOBAL TRACER OBJECT

Some features in VizTracer require a `VizTracer` object so it's helpful to make the tracer object accessible globally.

When you are using command line entry `viztracer your_script.py`, you don't need to worry about it. The tracer will be automatically registered and you can access it from any file.

When you instantiate the `VizTracer` object like `tracer = VizTracer()` in your script, it will be automatically registered globally. It is *not* recommended to have multiple tracer objects in a single script. However, you can turn off the global register by `tracer = VizTracer(register_global=False)`

To access the tracer, do

```
from viztracer import get_tracer
# get_tracer() will return None if no tracer is registered
tracer = get_tracer()
```

When you use `VizLoggingHandler` or `VizCounter` or `VizObject`, setting their tracer to `None` will make the logging a `NOP`. This will enable you to leave the instrumentation code as it is and run your program both regularly and with `viztracer`

You can do things like:

```
from viztracer import VizLoggingHandler, get_tracer

handler = VizLoggingHandler()

handler.setTracer(get_tracer())
```

```
from viztracer import get_tracer, VizObject

obj = VizObject(get_tracer(), "my variable")
```

CHAPTER

FOUR

LIMITATIONS

VizTracer uses `sys.setprofile()` for its profiler capabilities, so it will conflict with other profiling tools which also use this function. Be aware of it when using VizTracer

The clock resolution and latency on WSL1 are very bad, so if you are using WSL1, you may experience extra overhead. There's no solution for it, except for upgrading to WSL2.

VizTracer, like other python tools that need to execute arbitrary code inside the module, may conflict with code that check for top module or have other structural requirements. For example, `unittest.main()` won't work if you use VizTracer from command line. There are ways to avoid it. You can use inline VizTracer, which will always work. Or you can specify modules to `unittest.main()`, which is not a general solution but could work without too much code changes.

CHAPTER

FIVE

FILTER

Sometimes, you may have too many data entries and you want to filter some of them out to either improve the performance or make the report clearer. VizTracer provides multiple filters for your needs.

5.1 Min Duration

You can ask VizTracer to only record entries that last longer than a period of time

```
viztracer --min_duration 0.2ms my_script.py
```

OR

```
tracer = VizTracer(min_duration=200)
```

Notice that with command line interface, `viztracer` expects a string representing a period of time, which should be in format of <val><unit>. ex. `0.2ms`, `300ns`, `5.5us`. You can also omit the unit and it would be parsed as `us`, ex. `0.5` is the same as `0.5us`.

But as an argument to `VizTracer`, it should be a number of `us`.

The default value of `min_duration` is `0`, meaning every function entry is recorded.

5.2 Max Stack depth

You can limit maximum stack depth to trace by

```
viztracer --max_stack_depth 10 my_script.py
```

OR

```
tracer = VizTracer(max_stack_depth=10)
```

5.3 Include Files

You can include only certain files/folders to trace by

```
# -- is used to resolve ambiguity
viztracer --include_files ./src -- my_script.py
```

OR

```
tracer = VizTracer(include_files=["./src"])
```

5.4 Exclude Files

Similarly, you can exclude certain files/folders to trace by

```
# -- is used to resolve ambiguity
viztracer --exclude_files ./not_interested.py -- my_script.py
```

OR

```
tracer = VizTracer(exclude_files=["./not_interested.py"])
```

5.5 Ignore C Function

By default, VizTracer will trace both python and C functions. You can turn off tracing C functions by

```
viztracer --ignore_c_function my_script.py
```

OR

```
tracer = VizTracer(ignore_c_function=True)
```

Since most of the builtin functions(like `append` or `len`) are C functions which are frequently called, ignoring C functions often improves the overhead and file size significantly.

5.6 Ignore Non File

You can ask VizTracer not to trace any functions that are not in a valid file(mostly import stuff) using `ignore_frozen`

```
viztracer --ignore_frozen my_script.py
```

OR

```
tracer = VizTracer(ignore_frozen=True)
```

5.7 Ignore Function

It's possible that you want to ignore some arbitrary functions and their descendants. You can do it using `@ignore_function` decorator

```
from viztracer import ignore_function
# This only works when there's a globally registered tracer
@ignore_function
def some_function():
    # nothing inside will be traced
```

5.8 Log Sparse

You can make VizTracer log only certain functions using `--log_sparse`. This is helpful when you are only interested in the time spent on specific functions for a big picture on larger projects.

First, you need to add decorator `@log_sparse` on the function you want to log

```
from viztracer import log_sparse

# @log_sparse will only log this function
@log_sparse
def function_you_want_to_log():
    # function body

# @log_sparse(stack_depth=5) will log this function and its descendants
# with a limit stack depth of 5
# Nested @log_sparse with stack_depth won't work
# (only the outermost function and its stack will be logged)
@log_sparse(stack_depth=5)
def function_you_want_to_log():
    # function body

# Use dynamic_tracer_check=True if you use tracer as a context manager (or with %
↪%viztracer).
@log_sparse(dynamic_tracer_check=True)
def function_you_want_to_log():
    # function body

with VizTracer(log_sparse=True):
    function_you_want_to_log()
```

Then just call viztracer with `--log_sparse`

```
viztracer --log_sparse your_script.py
```

When you are using `--log_sparse`, due to the nature of the recording, some advanced features may not work with it.

You can leave `@log_sparse` as it is when you are not running the script with VizTracer. It will be like a no-op

If you want to log a piece of code, rather than a full function, please check *Duration Event*. Duration Event is compatible with `log_sparse`

To use `@log_sparse` in conjunction with a context manager, you must define decorating functions within the created context, or set the *dynamic_tracer_check=True* ` argument of decorator. The second option leads to runtime checks, so it increases the overhead.

CHAPTER

SIX

CUSTOM EVENTS

You may want to insert custom events to the report while you are tracing the program.

VizTracer supports three kinds of custom events:

- Instant Event
- Variable Event
- Duration Event

6.1 Instant Event

Instant Event is a log at a specific timestamp, showing as an arrow. It's useful to log a transient event. You need to give it a `name` which is a string, and an argument `args`. They will be displayed in the report

`args` has to be a jsonifiable object, normally a string, or a combination of dict, list, string and number.

`scope` can be set to `t``(default), ``p` or `g`, for thread, process and global.

```
tracer.log_instant(f"Event1", args=args, scope="p")
```

6.2 Variable Event

Variable Event is a way to log a specific variable in your program and display it in the report.

If the variable you log is a number, VizTracer will use a counter event to display it, otherwise instant event will be used.

A `name` should be given for the variable, then the variable itself

```
trace.log_var("name for the var", var)
```

6.3 Magic Comment

You can use magic comment to log instant events and variable events.

In this way, you'll have 0 overhead and side effect when you run your program normally, and log the events when you use viztracer to trace it

```
# !viztracer: log_instant("start logging")
a = 3
# !viztracer: log_var("a", a)
```

Or you can use inline magic comment `# !viztracer: log`, which will log the assigned value if the statement is an assign or it will log an instant event indicating this line is executed

```
# This will log an instant event with name "f()"
f()  # !viztracer: log

# This will log the variable a
a = 3  # !viztracer: log
```

You can also do conditional log with `if`

```
# This will log the variable a
a = 3  # !viztracer: log if a == 3
# This has the same effect
# !viztracer: log_var("a", a)
```

You need `--magic_comment` option for `viztracer` to trigger the magic comment

```
viztracer --magic_comment your_program.py
```

6.4 Duration Event

Duration Event is almost the same as function call event that normally being logged automatically, with the only exception that it does not have to be a function.

You can log any piece of code using duration event and it will look like a function call event in your final report.

```
from viztracer import get_tracer

with get_tracer().log_event("my event name"):
    # some code running here
```

You should use `log_event` method of your tracer, which is accessible through `get_tracer()` function when you are using CLI, or just pass the tracer if you are using inline.

This feature is especially helpful when you are using *Log Sparse*.

CHAPTER

SEVEN

EXTRA LOG

VizTracer features with many extra log possibilities **without even changing your source code**. You can start VizTracer from command line and use command line arguments to control what you need to log.

7.1 Log Variable

You can log any variable using regex matching to the variable name. This is like adding `print` after assigning the variable without actually writing the code. The log will appear in the report as an Instant Event, and the variables `repr` will be showed

```
viztracer --log_var <var_name> -- my_script.py
```

`--` is added to resolve the ambiguity. Every time a variable matches regex `<var_name>` is assigned a value, it will be logged. If you don't know what regex is, simply using the full name of the variable as `<var_name>` will allow you to log the variable

7.2 Log Number

Similar to *Log Variable*, you can log any variable as a number, which will be logged as Counter Event. The report will visualize the number through time as a separate signal like `VizCounter` did.

```
viztracer --log_number <var_name> -- my_script.py
```

`--` is added to resolve the ambiguity. Every time a variable matches regex `<var_name>` is assigned a value, it will be logged. If you don't know what regex is, simply using the full name of the variable as `<var_name>` will allow you to log the variable

Using `--log_number` on non-numeric variables will raise an exception.

7.3 Log Attribute

You can log writes to attributes based on the name of the attribute. This is useful when you want to track an attribute of an object, but there are just too many entries to it. It could be `self.attr_name`, `obj.attr_name` or even `obj_list[0].attr_name`. With `log_attr` you can log the attributes matching the regex as an Instant Event.

```
viztracer --log_attr <attr_name> -- my_script.py
```

-- is added to resolve the ambiguity. Every time an attribute matches regex <attr_name> is assigned a value, it will be logged. If you don't know what regex is, simply using the full name of the attribute as <attr_name> will allow you to log the attribute

7.4 Log Function Entry

You can log when a function is called. This is helpful to label the timeline for some crucial function. The log will be displayed as an Instant Event.

```
viztracer --log_func_entry <func_name> -- my_script.py
```

-- is added to resolve the ambiguity. Every time an function matches regex <func_name> is called, it will be logged. If you don't know what regex is, simply using the full name of the function as <func_name> will allow you to log the function

7.5 Log Function Execution

You can log function execution details. VizTracer will record all the assignments in specified functions and display them in the detailed information of the generated report. The log will be showed in function entry's `args` list.

```
viztracer --log_func_exec <func_name> -- my_script.py
```

-- is added to resolve the ambiguity. Every time an function matches regex <func_name> is called, its execution will be logged. If you don't know what regex is, simply using the full name of the function as <func_name> will allow you to log the function

7.6 Log Audit

You can log audit events introduced since python 3.8. The audit events will be logged as instant events.

```
# -- is added to resolve ambiguity
viztracer --log_audit import builtins.input -- my_script.py

# regex is supported
viztracer --log_audit os.* -- my_script.py

# If no argument is given, log every audit
viztracer --log_audit -- my_script.py

# You sometimes need to quote the regex to avoid command line ambiguity
# (Linux terminal would think that you are passing files starts with '.')
viztracer --log_audit ".*" -- my_script.py
```

7.7 Log Exception

You can log raised exceptions. All raised exceptions, whether caught or not, will be displayed as an Instant Event in the report.

```
viztracer --log_exception my_script.py
```

7.8 Log Function Arguments

You can log every function's arguments as `string`, aka their `__repr__`. The arguments will be stored in each python function entry under `args["func_args"]`. You can overwrite the object's `__repr__` function to log the object as you need.

You can enable this feature in command line or using inline.

```
viztracer --log_func_args my_script.py
```

```
tracer = VizTracer(log_func_args=True)
```

This feature will introduce a very large overhead(depends on your argument list), so be aware of it

You can log additional arbitrary (key, value) pairs for your function entry using `add_func_args()`. Refer to *VizTracer* for it's usage

7.9 Log Function Return Value

VizTracer can log every function's return value as `string`, aka it's `__repr__`. The return value will be stored in each python function entry under `args["return_value"]`. You can overwrite the object's `__repr__` function to log the object as you need.

You can enable this feature in command line or using inline.

```
viztracer --log_func_retval my_script.py
```

```
tracer = VizTracer(log_func_retval=True)
```

7.10 Log Print

You can intercept `print()` function and record the data it prints to the report as an Instant Event. This is like doing print debug on timeline.

You can do this simply by:

```
viztracer --log_print my_script.py
```

OR

```
tracer = VizTracer(log_print=True)
```

7.11 Log Garbage Collector

You can log the optional garbage collector module in Python. Notice that in CPython, most garbage collection is done using reference count. The garbage collector module is only responsible for the cycle reference. So this feature is mainly used to detect cycle reference collection status, and the time consumed by running the optional garbage collector.

You can do this simply by:

```
viztracer --log_gc my_script.py
```

OR

```
tracer = VizTracer(log_gc=True)
```

7.12 Log Exit data

Normally VizTracer only logs the executed code in "execution phase", or "within exec() function". You can, however, log functions in `atexit` or other on-exit hooks.

```
viztracer --log_exit my_script.py
```

7.13 Work with `logging` module

VizTracer can work with python builtin `logging` module by adding a handler to it. The report will show logging data as Instant Events.

```
from viztracer import VizTracer, VizLoggingHandler

tracer = VizTracer()
handler = VizLoggingHandler()
handler.setTracer(tracer)
# A handler is added to logging so logging will dump data to VizTracer
logging.basicConfig(handlers = [handler])
```

CHAPTER

EIGHT

CONCURRENCY

VizTracer supports concurrency tracing, including asyncio, multi-thread and multi-process.

8.1 asyncio

VizTracer supports `asyncio` module natively. However, you can use `--log_async` to make the report clearer.

Under the rug, asyncio is a single-thread program that's scheduled by Python built-ins. With `--log_async`, you can visualize different tasks as "threads", which could separate the real work from the underlying structure, and give you a more intuitive understanding of how different tasks consume the runtime.

```
viztracer --log_async my_script.py
```

8.2 threading

VizTracer supports python native `threading` module without the need to do any modification to your code. Just start `VizTracer` before you create threads and it will just work.

8.3 other multi-thread

If you are using multi-thread via other mechanism, for example, PyQt thread, VizTracer can't support it out of the box. However, you can notice VizTracer that you are in a separate thread and enable tracing in that thread with `enable_thread_tracing`

```
from viztracer import get_tracer

class YourThread:
    def run(self):
        # This will tell VizTracer to trace the thread
        get_tracer().enable_thread_tracing()
```

8.4 subprocess

VizTracer supports `subprocess`. You need to make sure the main process exits after subprocesses finish.

```
viztracer my_script_using_subprocess.py
```

This will generate an HTML file for all processes. There are a couple of things you need to be aware though.

VizTracer patches subprocess module(to be more specific, `subprocess.Popen`) to make this work like a magic. However, it will only patch when the args passed to `subprocess.Popen` is a list(`subprocess.Popen(["python", "subscript.py"])`) and the first argument starts with `python`. This covers most of the cases, but if you do have a situation that can't be solved, you can raise an issue and we can talk about solutions.

8.5 multiprocessing and concurrent.futures

VizTracer supports `multiprocessing` and `concurrent.futures`, and it will make the main process wait for all the other processes to finish so the report can include all processes. You can skip the waiting using Ctrl+C.

```
viztracer my_script_using_multiprocess.py
```

This feature is available on all platforms and for both `fork` and `spawn` type `Process`.

However, on Windows, `multiprocessing.Pool` won't work with VizTracer because there's no way to gracefully catch the exit of the process

8.6 os.fork()

VizTracer supports `os.fork`, with some caveats.

On Python3.8+, it works well, the main process will wait for forked processes to finish. You can even use `os.exec()` and its other forms after you fork the process. Of course VizTracer only records what happens before `os.exec()`, you need *generic multi process support* to record what happens after.

On Python3.6/3.7, VizTracer is not able to wait for the forked process to finish. It would be user's responsibility to wait for the forked process to finish if they want to see both processes in the report.

8.7 loky

VizTracer supports `loky>=3.0.0` as `loky` implemented the `viztracer` initializer. You can log `loky` processes just as easy as builtin `multiprocessing`

8.8 generic multi process support

VizTracer has a simple instrumentation for all the third party libraries to integrate VizTracer to their multi process code.

First, your main process has to be executed by `viztracer`. Inline VizTracer won't work. In your program, you need `get_tracer().init_kwargs`, which is a `Dict` that can be easily serializable with `pickle` or other libraries.

Then, pass this argument to your sub-process, and instantiate a VizTracer object with it

```
# init_kwargs is the argument from main process
tracer = VizTracer(**init_kwargs)
tracer.register_exit()
tracer.start()
```

And you are good to go. The main process should collect the data from sub-processes automatically and put together a report.

8.9 combine reports

You can generate json reports from different processes and combine them manually as well. It is recommended to use `--pid_suffix` so the report will be saved as a json file ending with the pid of the process. You can specify your own file name using `-o` too.

```
viztracer --pid_suffix single_process.py
# or
viztracer -o process1.json single_process.py
```

You can specify the output directory if you want to

```
viztracer --pid_suffix --output_dir ./temp_dir single_process.py
```

After generating `json` files, you need to combine them

```
viztracer --combine ./temp_dir/*.json
```

This will generate the HTML report with all the process info. You can specify `--output_file` when using `--combine`.

CHAPTER

NINE

REMOTE ATTACH

9.1 Attach

VizTracer supports remote attach so you don't need to start the process with VizTracer. This is helpful when you don't want to restart the process to trace it. You can run the process once and forever, and only attach VizTracer when you want to trace it. The process will run normally without performance hit when you are not attaching VizTracer.

This feature does not support Windows

To attach to the process and trace it, you have two ways:

1. You can attach to an arbitrary Python process, as long as `viztracer` is importable in that process
2. You can attach to a Python process that already installed VizTracer

The **first** way is more flexible - it will inject code into the process to load `viztracer`. You can even pass arguments from `viztracer` command line.

```
viztracer --attach <pid> -o result.json
```

viztracer has to be importable in the attached process otherwise it will raise an exception

This means, you need to install `viztracer` in the environment(venv, pyenv etc.) of the attached process. You don't have to use the `viztracer` from the same environment to attach though.

gdb is required on Linux, and lldb is required on MacOS

Notice that, if there is already a globally registered `VizTracer` object in the attached process, that object will be used for tracing. If it's already running, then attaching won't do anything. Otherwise all the arguments will be sent to the attached process to instantiate a `VizTracer` object.

By default, you need to Ctrl+C out of viztracer to save the report. Be aware that it is the attached process rather than attaching process(viztracer) that is saving the report, so it's the attached process's resource that is being spent.

You can also trace for a period of time using `-t`

```
viztracer --attach <pid> -t <seconds>
```

Even though this looks decent, there are some dark magic going under the rug and you may want to do something cleaner, which brings up the **second** way - pre-install viztracer in the process you want to profile. Another good thing about this way is that it's thread-aware. Even if you attach after spawning threads, you can still get profile data from the other threads.

```
from viztracer import VizTracer
tracer = VizTracer()
tracer.install()
```

`tracer.install()` will basically add handlers for `SIGUSR1` and `SIGUSR2` which are only available on Unix. This also requires the program not to use these two signals.

Then when you are running this process, you can attach to it with VizTracer, using it's pid

```
viztracer --attach_installed <pid>
```

If you need other options, you should specify them in `VizTracer` instance in attached process.

9.2 Uninstall

There could be time when you want to "uninstall" VizTracer from a process. For example, for some reason, the attach process failed and VizTracer is left on in the process. You can do that with:

```
viztracer --uninstall <pid>
```

CHAPTER

TEN

PLUGINS

VizTracer supports third party plugins that comply to the specification of VizTracer.

To use a plugin, you need to install the plugin first. For example, if you want to use a plugin named `vizplugins`

```
pip install vizplugins
```

Then you need to pass the plugin to `viztracer` using `--plugins`

```
viztracer --plugins vizplugins -- my_script.py
```

There could be multiple plugins to use in a package, which are differentiate by modules. You can specify the module where plugin lives(you should refer to the plugin's doc for detailed usage)

```
viztracer --plugins vizplugins.cpu_time -- my_script.py
```

You can even pass arguments to the plugin, but you need double quotes to pack them together.

```
viztracer --plugins "vizplugins.cpu_time -f 100" -- my_script.py
```

You can also do it inline, just pass the string or the plugin object itself in a list to VizTracer

```
tracer = VizTracer(plugins=["vizplugins.cpu_time"])
# Or
tracer = VizTracer(plugins=[vizplugins.CpuTimePlugin()])

# To gracefully terminate all plugins, you need to do terminate
tracer.terminate()
```

```
# You can use with statement to avoid explicitly terminate
with VizTracer(plugins=["vizplugins.cpu_time"]):
    # Do your stuff here
```

If you want to develop your own plugin for VizTracer, take a look at *VizPlugin*

CHAPTER

ELEVEN

VIZTRACER

class VizTracer(*self*, *tracer_entries=1000000*, *verbose=1*, *max_stack_depth=-1*, *include_files=None*, *exclude_files=None*, *ignore_c_function=False*, *ignore_frozen=False*, *log_func_retval=False*, *log_func_args=False*, *log_print=False*, *log_gc=False*, *log_async=False*, *pid_suffix=False*, *register_global=True*, *min_duration=0*, *output_file='result.json'*)

tracer_entries: integer = 1000000

Size of circular buffer. The user can only specify this value when instantiate `VizTracer` object or if they use command line

Please be aware that a larger number of entries also means more disk space, RAM usage and loading time. Be familiar with your computer's limit.

`tracer_entries` means how many entries `VizTracer` can store. It's not a byte number.

```
viztracer --tracer_entries 500000
```

verbose: integer = 1

Verbose level of VizTracer. Can be set to `0` so it won't print anything while tracing

Setting it to `0` is equivalent to

```
viztracer --quiet
```

max_stack_depth: integer = -1

Specify the maximum stack depth VizTracer will trace. `-1` means infinite.

Equivalent to

```
viztracer --max_stack_depth <val>
```

include_files: list of string or None = None

Specify the files or folders that VizTracer will trace. If it's not empty, VizTracer will function in whitelist mode, any files/folders not included will be ignored.

Because converting code filename in tracer is too expensive, we will only compare the input and its absolute path against code filename, which could be a relative path. That means, if you run your program using relative path, but gives the `include_files` an absolute path, it will not be able to detect.

Can't be set with `exclude_files`

Equivalent to

```
viztracer --include_files file1[ file2 [file3 ...]]
```

NOTICE

In command line, `--include_files` takes multiple arguments, which will be ambiguous about the command that actually needs to run, so you need to explicitly specify command using `--`

```
viztracer --include_files file1 file2 -- my_scrpit.py
```

exclude_files: list of string or None = None

Specify the files or folders that VizTracer will not trace. If it's not empty, VizTracer will function in blacklist mode, any files/folders not included will be ignored.

Because converting code filename in tracer is too expensive, we will only compare the input and its absolute path against code filename, which could be a relative path. That means, if you run your program using relative path, but gives the `exclude_files` an absolute path, it will not be able to detect.

Can't be set with `include_files`

Equivalent to

```
viztracer --exclude_files file1[ file2 [file3 ...]]
```

NOTICE

In command line, `--exclude_files` takes multiple arguments, which will be ambiguous about the command that actually needs to run, so you need to explicitly specify command using `--`

```
viztracer --exclude_files file1 file2 -- my_scrpit.py
```

ignore_c_function: boolean = False

Whether trace c function

Setting it to `True` is equivalent to

```
viztracer --ignore_c_function
```

ignore_frozen: boolean = False

Whether trace functions from frozen functions(mostly import stuff)

Setting it to `True` is equivalent to

```
viztracer --ignore_frozen
```

log_func_retval: boolean = False

Whether log the return value of the function as string in report entry

Setting it to `True` is equivalent to

```
viztracer --log_func_retval
```

log_func_args: boolean = False

Whether log the arguments of the function as string in report entry

Setting it to `True` is equivalent to

```
viztracer --log_func_args
```

log_print: boolean = False

Whether replace the `print` function to log in VizTracer report

Setting it to `True` is equivalent to

```
viztracer --log_print
```

log_gc: boolean = False

Whether log garbage collector

Setting it to `True` is equivalent to

```
viztracer --log_gc
```

log_async: boolean = False

Whether log async tasks as separate "thread" in vizviewer

Setting it to `True` is equivalent to

```
viztracer --log_async
```

register_global: boolean = True

whether register the tracer globally, so every file can use `get_tracer()` to get this tracer. When command line entry is used, the tracer will be automatically registered. When `VizTracer()` is manually instantiated, it will be registered as well by default.

Some functions may require a globally registered tracer to work.

This attribute will only be effective when the object is initialized:

```
tracer = VizTracer(register_global=False)
```

min_duration: float = 0

Minimum duration of a function to be logged. The value is in unit of `us`.

output_file: string = "result.json"

Default file path to write report

Equivalent to

```
viztracer -o <filepath>
```

run(*command*, *output_file=None*)

run `command` and save report to `output_file`

save(*output_file=None*)

parse data and save report to `output_file`. If `output_file` is `None`, save to default path.

start()

start tracing

stop()

stop tracing

clear()

clear all the collected data

parse()

parse the data collected, return number of total entries

enable_thread_tracing()

enable tracing in the current thread, useful when you use multi-thread without builtin threading module

add_instant(*name*, *scope='g'*)

Parameters

- **name** (*str*) – name of this instant event
- **scope** (*str*) – one of `g`, `p` or `t` for global, process or thread level event

Add instant event to the report.

add_func_args(*name*, *key*, *value*)

Parameters

- **key** (*str*) – key to display in the report
- **value** (*object*) – a jsonifiable object

This method allows you to attach args to the current function, which will show in the report when you click on the function

log_event(*event_name*)

Parameters
event_name (*str*) – name of this event that will appear in the result

Returns
VizEvent object that should only be used with `with` statement

Return type
VizEvent

```
with get_tracer().log_event("event name"):
    # some code here
```

set_afterfork(*callback*, **args*, ***kwargs*)

Parameters

- **callback** (*callable*) – the callback function after fork, should take a `VizTracer` object as the first argument
- **args** (*list*) – positional arguments to `callback`
- **kwargs** (*dict*) – keyword arguments to `callback`

This method will register a callback function after the process is forked. If you want different behavior on child processes with `multiprocessing`, you can utilize this method

Notice that the `callback` argument should be a `callable` that takes a `VizTracer` object as the first argument

```
from viztracer import get_tracer

def afterfork_callback(tracer):
    tracer.max_stack_depth = 10
```

(continues on next page)

(continued from previous page)

```
get_tracer().set_afterfork(afterfork_callback)
```

CHAPTER

TWELVE

CUSTOM EVENT

_EventBase is the base class of VizCounter and VizObject. It should never be used directly.

class _EventBase(*tracer*, *name=None*, *trigger_on_change=True include_attributes=[] exclude_attributes=[]*)

tracer: *VizTracer*

an object of VizTracer

tracer can be set to None so the logging operation will be NOP. Your program will run normally with the instrumented code even when you are not using viztracer.

name: string = None

name of the event which will show on trace viewer. If not specified, class name will be used

trigger_on_change: boolean = True

whether to trigger log every time a public attribute is changed

include_attributes: list of string = []

a list of attributes that will trigger the log and be included in the report. If not empty, _EventBase will behave like whitelist

exclude_attributes: list of string = []

a list of attributes that will not trigger the log and not be included in the report. If not empty, _EventBase will behave like blacklist

log()

_viztracer_log()

manually log the current attributes

config()

_viztracer_set_config(*key*, *value*)

Parameters

- **key** (*str*) – "trigger_on_change", "include_attributes" or "exclude_attribtues"
- **value** – the value you want to set on corresponding config

@triggerlog(*when='after'*)

Parameters

when (*str*) – "after", "before" or "both" to specify when the log() function is called

triggerlog is a decorator for class methods to do auto-log when the method is called.

class VizCounter(*_EventBase*)

`VizCounter` should be used to track a numeric variable through time. You can track CPU usage, memory usage, or any numeric variable you are interested in using `VizCounter`

```
from viztraer import VizTracer, VizCounter
tracer = VizTracer()
tracer.start()
counter = VizCounter(tracer, "counter name")
```

Because `VizCounter` has `trigger_on_change` on by default, any writes to its public attributes(does not start with _) will be automatically logged

```
counter.a = 2
counter.b = 1.2
```

You can turn `trigger_on_change` off and manually decide when to log

```
counter = VizCounter(tracer, "counter name", trigger_on_change=False)
# OR
counter = VizCounter(tracer, "counter name")
counter.config("trigger_on_change", False)
```

```
counter.a = 1
counter.b = 1
# Until here, nothing happens
counter.log() # trigger the log
```

class VizObject(*_EventBase*)

`VizObject` is almost exactly the same as `VizCounter`, with the exception that `VizObject` can log jsonifiable objects(`dict`, `list`, `string`, `int`, `float`)

12.1 Inheritance

In practice, you can inherit from `VizCounter` or `VizObject` class and build your own class so it will be much easier to track the data in your class. Remember you need to do `__init__` function of the base class! If your class has a lot of attributes and they are frequently being written to, it is wise to turn off `trigger_on_change`

```
class MyClass(VizObject):
    def __init__(self, tracer):
        super().__init__(tracer, "my name", trigger_on_change=False)
```

You can manually do log by

```
obj = MyClass(tracer)
obj.log()
```

or you can decorate your class method with `triggerlog` to trigger log on function call

```
class MyClass(VizObject):
    @VizObject.triggerlog
    def log_on_this_function():
        #function
```

CHAPTER

THIRTEEN

DECORATOR

@ignore_function

`@ignore_function` can tell VizTracer to skip on functions you specified.

```
# This only works when there's a globally registered tracer
@ignore_function
def function_you_want_to_ignore():
    # function body

# You can specify tracer if no global tracer is registered
@ignore_function(tracer=tracer)
def function_you_want_to_ignore():
    # function body
```

@trace_and_save(*method=None*, *output_dir='./'*, ***viztracer_kwargs*)

Parameters

- **method** (*function*) – trick to make both `@trace_and_save` and `@trace_and_save(**kwargs)` work
- **output_dir** (*str*) – output directory you want to put your logs in
- **viztracer_kwargs** (*dict*) – kwargs for VizTracer

`@trace_and_save` can be used to trace a certain function and save the result as the program goes. This can be very helpful when you are running a very long program and just want to keep recording something periodically. You won't drain your memory and the parsing/dumping will be done in a new process, which can minimize the performance impact to your main process.

You can pass any argument you want to `VizTracer` by giving it to the decorator

```
@trace_and_save(output_dir="./mylogs", ignore_c_function=True)
def function_you_want_to_trace():
    # function body

# this works as well
@trace_and_save
def function_you_want_to_trace():
    # function body
```

@log_sparse(*func=None*, *stack_depth=0*, *dynamic_tracer_check=False*)

You can make VizTracer log only certain functions using `--log_sparse` mode.

Parameters

- **func** (*function*) – callable to decorate
- **stack_depth** (*int*) – log the function and its descendants with a limit stack depth
- **dynamic_tracer_check** (*bool*) – run time check of tracer

```
from viztracer import log_sparse

# @log_sparse will only log this function
@log_sparse
def function_you_want_to_log():
    # function body

# @log_sparse(stack_depth=5) will log this function and its descendants
# with a limit stack depth of 5
# Nested @log_sparse with stack_depth won't work
# (only the outermost function and its stack will be logged)
@log_sparse(stack_depth=5)
def function_you_want_to_log():
    # function body

# Use dynamic_tracer_check=True if you use tracer as a context manager (or with %
↪%viztracer).
@log_sparse(dynamic_tracer_check=True)
def function_you_want_to_log():
    # function body

with VizTracer(log_sparse=True):
    function_you_want_to_log()
```

CHAPTER

FOURTEEN

VIZPLUGIN

In this doc, we will show how to build your own plugin for VizTracer.

Every plugin should inherit `VizPluginBase` from `viztracer.vizplugin`

class VizPluginBase(*self*)

support_version(*self*)

Returns
the string for the latest version of viztracer supported to have API compatibility

Return type
dict

You must have this method overloaded and return the version, otherwise viztracer will raise an exception

```
def support_version(self):
    return "0.11.0"
```

message(*self*, *m_type*, *payload*)

Parameters

- **m_type** (*str*) – type of message
- **payload** (*dict*) – a very flexible payload with the message

Returns
the corresponding return value. Could be an action, or a respond.

Return type
dict

As of now, `message()` only supports two kinds of `m_type` - `"event"` and `"command"`.

`"event"` has a `payload` that has key `"when"`, to indicate when the event happens.

`"when"` could be `initialize`, `pre-start`, `post-stop` or `pre-save`.

When `"event"` type message is received, the plugin can return an action, with key `"action"` set to the action it needs. For now the only one supported is `"handle_data"`, with which you also needs to provide a data handler `"handler"`

return `{}` if you don't want to do anything

`"command"` has a payload that has key `"cmd_type"`, to indicate what VizTracer expects the plugin to do.

`"cmd_type"` could only be `"terminate"` for now. `"terminate"` command is guaranteed before viztracer exits when commandline interface is used. However, if you are using viztracer inline, you will have to explicitly run `tracer.terminate()` unless you are using `with VizTracer()`.

When "command" type message is received, the plugin has to return a dict like {"success": True} to inform VizTracer that the plugin received the command and did what it asked.

14.1 Possible Actions

This sections lists all the possible actions you can take after you received an event. You should return {"action": "the action name" ...} with specific payload for each action

"handle_data"

When you want to modify the data, which is a common way to change the result viztracer has, you need to use "handle_data" action.

You should return {"action": "handle_data", "handler": my_handler} where my_handler should be a function with a prototype def my_handler(data). VizTracer will call this handler to modify the original data before it saves the report

14.2 Make Your Plugin Accessible

You will also need a magic function defined to enable VizTracer to load your plugins from command line. The function has to be named `get_vizplugin(arg)`, which will return an instance of your plugin object based on the arg given when it's called. You also need to put this function in the module that you want your use to use on command line.

For example, if I want my user to use my plugin as viztracer --plugins vizplugins -- my_script.py, I should put get_vizplugin() function in vizplugins/__init__.py. Or if I want them to use as viztracer --plugins vizplugins.cpu_usage -- my_script.py, I should put the function in vizplugins/cpu_usage.py

Be aware that **arg is an unparsed string** like "vizplugins.cpu_time f 100". You can split it yourself and parse it the way you like. Or you can specify something special for your own plugin

CHAPTER

FIFTEEN

CONTACT US

If you have any questions, bug reports or feature requests, please go to github issue

CHAPTER

SIXTEEN

LICENSE

INDEX

Symbols

A

B

C

E

I

L

M

N

O

P

R

S

T

V