line$_{profiler}$ $Documentation$

Release 4.1.2

Robert Kern

Dec 05, 2023

PACKAGE LAYOUT

The line_profiler module for doing line-by-line profiling of functions

Github	https://github.com/pyutils/line_profiler
Pypi	https://pypi.org/project/line_profiler
ReadTheDocs	https://kernprof.readthedocs.io/en/latest/

CHAPTER

ONE

INSTALLATION

Releases of *line_profiler* and *kernprof* can be installed using pip

```
pip install line_profiler
```

The package also provides extras for optional dependencies, which can be installed via:

```
pip install line_profiler[all]
```

CHAPTER

TWO

LINE PROFILER BASIC USAGE

To demonstrate line profiling, we first need to generate a Python script to profile. Write the following code to a file called `demo_primes.py`.

```
from line_profiler import profile


@profile
def is_prime(n):
    '''
    Check if the number "n" is prime, with n > 1.

    Returns a boolean, True if n is prime.
    '''
    max_val = n ** 0.5
    stop = int(max_val + 1)
    for i in range(2, stop):
        if n % i == 0:
            return False
    return True


@profile
def find_primes(size):
    primes = []
    for n in range(size):
        flag = is_prime(n)
        if flag:
            primes.append(n)
    return primes


@profile
def main():
    print('start calculating')
    primes = find_primes(100000)
    print(f'done calculating. Found {len(primes)} primes.')


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

In this script we explicitly import the `profile` function from `line_profiler`, and then we decorate function of interest with `@profile`.

By default nothing is profiled when running the script.

```
python demo_primes.py
```

The output will be

```
start calculating
done calculating. Found 9594 primes.
```

The quickest way to enable profiling is to set the environment variable `LINE_PROFILE=1` and running your script as normal.

```
LINE_PROFILE=1 python demo_primes.py
```

This will output 3 files: profile_output.txt, profile_output_<timestamp>.txt, and profile_output.lprof and stdout will look something like:

```
start calculating
done calculating. Found 9594 primes.
Timer unit: 1e-09 s

  0.65 seconds - demo_primes.py:4 - is_prime
  1.47 seconds - demo_primes.py:19 - find_primes
  1.51 seconds - demo_primes.py:29 - main
Wrote profile results to profile_output.txt
Wrote profile results to profile_output_2023-08-12T193302.txt
Wrote profile results to profile_output.lprof
To view details run:
python -m line_profiler -rtmz profile_output.lprof
```

For more control over the outputs, run your script using *kernprof*. The following invocation will run your script, dump results to `demo_primes.py.lprof`, and display results.

```
python -m kernprof -lvr demo_primes.py
```

Note: the `-r` flag will use "rich-output" if you have the `rich` module installed.

See also:

- autoprofiling usage in: *line_profiler.autoprofile*

CHAPTER

THREE

LINE_PROFILER PACKAGE

3.1 Subpackages

3.1.1 line_profiler.autoprofile package

3.1.1.1 Submodules

3.1.1.1.1 line_profiler.autoprofile.ast_profle_transformer module

line_profiler.autoprofile.ast_profle_transformer.**ast_create_profile_node**(*modname*, *profiler_name='profile'*, *attr='add_imported_function_or_modu*

Create an abstract syntax tree node that adds an object to the profiler to be profiled.

An abstract syntax tree node is created which calls the attr method from profile and passes modname to it. At runtime, this adds the object to the profiler so it can be profiled. This node must be added after the first instance of modname in the AST and before it is used. The node will look like:

```
>>> # xdoctest: +SKIP
>>> import foo.bar
>>> profile.add_imported_function_or_module(foo.bar)
```

Parameters

- **script_file** (*str*) – path to script being profiled.
- **prof_mod** (*List[str]*) – list of imports to profile in script. passing the path to script will profile the whole script. the objects can be specified using its dotted path or full path (if applicable).

Returns

expr
AST node that adds modname to profiler.

Return type
(_ast.Expr)

class line_profiler.autoprofile.ast_profle_transformer.**AstProfileTransformer**(*profile_imports=False*, *profiled_imports=None*, *profiler_name='profile'*)

Bases: NodeTransformer

Transform an abstract syntax tree adding profiling to all of its objects.

Adds profiler decorators on all functions & methods that are not already decorated with the profiler. If profile_imports is True, a profiler method call to profile is added to all imports immediately after the import.

Initializes the AST transformer with the profiler name.

Parameters

- **profile_imports** (*bool*) – If True, profile all imports.
- **profiled_imports** (*List[str]*) – list of dotted paths of imports to skip that have already been added to profiler.
- **profiler_name** (*str*) – the profiler name used as decorator and for the method call to add to the object to the profiler.

`visit_FunctionDef`(*node*)

Decorate functions/methods with profiler.

Checks if the function/method already has a profile_name decorator, if not, it will append profile_name to the end of the node's decorator list. The decorator is added to the end of the list to avoid conflicts with other decorators e.g. @staticmethod.

Parameters
(_ast.FunctionDef) – node function/method in the AST

Returns

node
function/method with profiling decorator

Return type
(_ast.FunctionDef)

`_visit_import`(*node*)

Add a node that profiles an import

If profile_imports is True and the import is not in profiled_imports, a node which calls the profiler method, which adds the object to the profiler, is added immediately after the import.

Parameters
node (*Union[_ast.Import,_ast.ImportFrom]*) – import in the AST

Returns

node

if profile_imports is False:
returns the import node

if profile_imports is True:
returns list containing the import node and the profiling node

Return type
(Union[Union[_ast.Import,_ast.ImportFrom],List[Union[_ast.Import,_ast.ImportFrom,_ast.Expr]]])

`visit_Import`(*node*)

Add a node that profiles an object imported using the "import foo" sytanx

Parameters
node (*_ast.Import*) – import in the AST

Returns

node

if profile_imports is False:
returns the import node

if profile_imports is True:
returns list containing the import node and the profiling node

Return type
(Union[_ast.Import,List[Union[_ast.Import,_ast.Expr]]])

`visit_ImportFrom`(*node*)

Add a node that profiles an object imported using the "from foo import bar" syntax

Parameters
node (*_ast.ImportFrom*) – import in the AST

Returns

node

if profile_imports is False:
returns the import node

if profile_imports is True:
returns list containing the import node and the profiling node

Return type
(Union[_ast.ImportFrom,List[Union[_ast.ImportFrom,_ast.Expr]]])

3.1.1.1.2 line_profiler.autoprofile.ast_tree_profiler module

`class` line_profiler.autoprofile.ast_tree_profiler.`AstTreeProfiler`(*script_file*, *prof_mod*, *profile_imports*, *ast_transformer_class_handler=<class 'line_profiler.autoprofile.ast_profle_transformer.A*, *prof-mod_extractor_class_handler=<class 'line_profiler.autoprofile.profmod_extractor.Profi*

Bases: `object`

Create an abstract syntax tree of a script and add profiling to it.

Reads a script file and generates an abstract syntax tree, then adds nodes and/or decorators to the AST that adds the specified functions/methods, classes & modules in prof_mod to the profiler to be profiled.

Initializes the AST tree profiler instance with the script file path

Parameters

- **script_file** (*str*) – path to script being profiled.
- **prof_mod** (*List[str]*) – list of imports to profile in script. passing the path to script will profile the whole script. the objects can be specified using its dotted path or full path (if applicable).
- **profile_imports** (*bool*) – if True, when auto-profiling whole script, profile all imports aswell.
- **ast_transformer_class_handler** (*Type*) – the AstProfileTransformer class that handles profiling the whole script.

- **profmod_extractor_class_handler** (*Type*) – the ProfmodExtractor class that handles mapping prof_mod to objects in the script.

`static _check_profile_full_script(script_file, prof_mod)`

Check whether whole script should be profiled.

Checks whether path to script has been passed to prof_mod indicating that the whole script should be profiled

Parameters

- **script_file** (*str*) – path to script being profiled.
- **prof_mod** (*List[str]*) – list of imports to profile in script. passing the path to script will profile the whole script. the objects can be specified using its dotted path or full path (if applicable).

Returns

profile_full_script
if True, profile whole script.

Return type
(bool)

`static _get_script_ast_tree(script_file)`

Generate an abstract syntax from a script file.

Parameters
script_file (*str*) – path to script being profiled.

Returns
abstract syntax tree of the script.

Return type
tree (_ast.Module)

`_profile_ast_tree(tree, tree_imports_to_profile_dict, profile_full_script=False, profile_imports=False)`

Add profiling to an abstract syntax tree.

Adds nodes to the AST that adds the specified objects to the profiler. If profile_full_script is True, all functions/methods, classes & modules in the script have a node added to the AST to add them to the profiler. If profile_imports is True as well as profile_full_script, all imports are have a node added to the AST to add them to the profiler.

Parameters

- **tree** (*_ast.Module*) – abstract syntax tree to be profiled.
- **tree_imports_to_profile_dict** (*Dict[int,str]*) –

  dict of imports to profile

  key (int):
  index of import in AST

  value (str):
  alias (or name if no alias used) of import
- **profile_full_script** (*bool*) – if True, profile whole script.
- **profile_imports** (*bool*) – if True, and profile_full_script is True, profile all imports aswell.

Returns

tree
abstract syntax tree with profiling.

Return type
(_ast.Module)

`profile()`

Create an abstract syntax tree of a script and add profiling to it.

Reads a script file and generates an abstract syntax tree. Then matches imports in the script's AST with the names in prof_mod. The matched imports are added to the profiler for profiling. If path to script is found in prof_mod, all functions/methods, classes & modules are added to the profiler. If profile_imports is True as well as path to script in prof_mod, all the imports in the script are added to the profiler.

Returns

tree
abstract syntax tree with profiling.

Return type
(_ast.Module)

3.1.1.1.3 line_profiler.autoprofile.autoprofile module

3.1.1.1.3.1 AutoProfile Script Demo

The following demo is end-to-end bash code that writes a demo script and profiles it with autoprofile.

```
# Write demo python script to disk
python -c "if 1:
    import textwrap
    text = textwrap.dedent(
        '''
        def plus(a, b):
            return a + b

        def fib(n):
            a, b = 0, 1
            while a < n:
                a, b = b, plus(a, b)

        def main():
            import math
            import time
            start = time.time()

            print('start calculating')
            while time.time() - start < 1:
                fib(10)
                math.factorial(1000)
            print('done calculating')

        main()
        '''
    ).strip()
```

(continues on next page)

(continued from previous page)

```
    with open('demo.py', 'w') as file:
        file.write(text)
"

echo "---"
echo "## Profile With AutoProfile"
python -m kernprof -p demo.py -l demo.py
python -m line_profiler -rmt demo.py.lprof
```

line_profiler.autoprofile.autoprofile.**_extend_line_profiler_for_profiling_imports**(*prof*)

Allow profiler to handle functions/methods, classes & modules with a single call.

Add a method to LineProfiler that can identify whether the object is a function/method, class or module and handle it's profiling accordingly. Mainly used for profiling objects that are imported. (Workaround to keep changes needed by autoprofile separate from base LineProfiler)

Parameters

prof (*LineProfiler*) – instance of LineProfiler.

line_profiler.autoprofile.autoprofile.**run**(*script_file*, *ns*, *prof_mod*, *profile_imports=False*)

Automatically profile a script and run it.

Profile functions, classes & modules specified in prof_mod without needing to add @profile decorators.

Parameters

- **script_file** (*str*) – path to script being profiled.
- **ns** (*dict*) – "locals" from kernprof scope.
- **prof_mod** (*List[str]*) – list of imports to profile in script. passing the path to script will profile the whole script. the objects can be specified using its dotted path or full path (if applicable).
- **profile_imports** (*bool*) – if True, when auto-profiling whole script, profile all imports aswell.

3.1.1.1.4 line_profiler.autoprofile.line_profiler_utils module

line_profiler.autoprofile.line_profiler_utils.**add_imported_function_or_module**(*self*, *item*)

Method to add an object to LineProfiler to be profiled.

This method is used to extend an instance of LineProfiler so it can identify whether an object is function/method, class or module and handle it's profiling accordingly.

Parameters

item (*Callable | Type | ModuleType*) – object to be profiled.

3.1.1.1.5 line_profiler.autoprofile.profmod_extractor module

class line_profiler.autoprofile.profmod_extractor.**ProfmodExtractor**(*tree*, *script_file*, *prof_mod*)

Bases: object

Map prof_mod to imports in an abstract syntax tree.

Takes the paths and dotted paths in prod_mod and finds their respective imports in an abstract syntax tree.

Initializes the AST tree profiler instance with the AST, script file path and prof_mod

Parameters

- **tree** (*_ast.Module*) – abstract syntax tree to fetch imports from.
- **script_file** (*str*) – path to script being profiled.
- **prof_mod** (*List[str]*) – list of imports to profile in script. passing the path to script will profile the whole script. the objects can be specified using its dotted path or full path (if applicable).

static **_is_path**(*text*)

Check whether a string is a path.

Checks if a string contains a slash or ends with .py indicating it is a path.

Parameters
text (*str*) – string to check whether it is a path or not

Returns
bool indicating whether the string is a path or not

Return type
ret (bool)

classmethod **_get_modnames_to_profile_from_prof_mod**(*script_file*, *prof_mod*)

Grab the valid paths and all dotted paths in prof_mod and their subpackages and submodules, in the form of dotted paths.

First all items in prof_mod are converted to a valid path. if unable to convert, check if the item is an invalid path and skip it, else assume it is an installed package. The valid paths are then converted to dotted paths. The converted dotted paths along with the items assumed to be installed packages are added a list of modnames_to_profile. Then all subpackages and submodules under each valid path is fetched, converted to dotted path and also added to the list. if script_file is in prof_mod it is skipped to avoid name collision with othe imports, it will be processed elsewhere in the autoprofile pipeline.

Parameters

- **script_file** (*str*) – path to script being profiled.
- **prof_mod** (*List[str]*) – list of imports to profile in script. passing the path to script will profile the whole script. the objects can be specified using its dotted path or full path (if applicable).

Returns
list of dotted paths to profile.

Return type
modnames_to_profile (List[str])

static **_ast_get_imports_from_tree**(*tree*)

Get all imports in an abstract syntax tree.

Parameters
tree (*_ast.Module*) – abstract syntax tree to fetch imports from.

Returns

list of dicts of all imports in the tree, containing:

name (str):
the real name of the import. e.g. foo from "import foo as bar"

alias (str):
the alias of an import if applicable. e.g. bar from "import foo as bar"

tree_index (int):
the index of the import as found in the tree

Return type
module_dict_list (List[Dict[str,Union[str,int]]])

`static _find_modnames_in_tree_imports(modnames_to_profile, module_dict_list)`

Map modnames to imports from an abstract sytax tree.

Find imports in modue_dict_list, created from an abstract syntax tree, that match dotted paths in modnames_to_profile. When a submodule is imported, both the submodule and the parent module are checked whether they are in modnames_to_profile. As the user can ask to profile "foo" when only "from foo import bar" is imported, so both foo and bar are checked. The real import name of an import is used to map to the dotted paths. The import's alias is stored in the output dict.

Parameters

- **modnames_to_profile** (*List[str]*) – list of dotted paths to profile.
- **module_dict_list** (*List[Dict[str,Union[str,int]]]*) – list of dicts of all imports in the tree.

Returns

dict of imports found

key (int):
index of import in AST

value (str):
alias (or name if no alias used) of import

Return type
modnames_found_in_tree (Dict[int,str])

`run()`

Map prof_mod to imports in an abstract syntax tree.

Takes the paths and dotted paths in prod_mod and finds their respective imports in an abstract syntax tree, returning their alias and the index they appear in the AST.

Returns

tree_imports_to_profile_dict

dict of imports to profile

key (int):
index of import in AST

value (str):
alias (or name if no alias used) of import

Return type
(Dict[int,str])

3.1.1.1.6 line_profiler.autoprofile.util_static module

This file was autogenerated based on code in ubelt and xdoctest via dev/maintain/port_utilities.py in the line_profiler repo.

line_profiler.autoprofile.util_static.**package_modpaths**(*pkgpath*, *with_pkg=False*, *with_mod=True*, *followlinks=True*, *recursive=True*, *with_libs=False*, *check=True*)

Finds sub-packages and sub-modules belonging to a package.

Parameters

- **pkgpath** (*str*) – path to a module or package
- **with_pkg** (*bool*) – if True includes package __init__ files (default = False)
- **with_mod** (*bool*) – if True includes module files (default = True)
- **exclude** (*list*) – ignores any module that matches any of these patterns
- **recursive** (*bool*) – if False, then only child modules are included
- **with_libs** (*bool*) – if True then compiled shared libs will be returned as well
- **check** (*bool*) – if False, then then pkgpath is considered a module even if it does not contain an __init__ file.

Yields
str – module names belonging to the package

References

http://stackoverflow.com/questions/1707709/list-modules-in-py-package

Example

```
>>> from xdoctest.static_analysis import *
>>> pkgpath = modname_to_modpath('xdoctest')
>>> paths = list(package_modpaths(pkgpath))
>>> print('\n'.join(paths))
>>> names = list(map(modpath_to_modname, paths))
>>> assert 'xdoctest.core' in names
>>> assert 'xdoctest.__main__' in names
>>> assert 'xdoctest' not in names
>>> print('\n'.join(names))
```

line_profiler.autoprofile.util_static.**_parse_static_node_value**(*node*)

Extract a constant value from a node if possible

line_profiler.autoprofile.util_static.**_extension_module_tags**()

Returns valid tags an extension module might have

Returns
List[str]

line_profiler.autoprofile.util_static.**_static_parse**(*varname*, *fpath*)

Statically parse the a constant variable from a python file

Example

```
>>> # xdoctest: +SKIP("ubelt dependency")
>>> dpath = ub.Path.appdir('tests/import/staticparse').ensuredir()
>>> fpath = (dpath / 'foo.py')
>>> fpath.write_text('a = {1: 2}')
>>> assert _static_parse('a', fpath) == {1: 2}
>>> fpath.write_text('a = 2')
>>> assert _static_parse('a', fpath) == 2
>>> fpath.write_text('a = "3"')
>>> assert _static_parse('a', fpath) == "3"
>>> fpath.write_text('a = ["3", 5, 6]')
>>> assert _static_parse('a', fpath) == ["3", 5, 6]
>>> fpath.write_text('a = ("3", 5, 6)')
>>> assert _static_parse('a', fpath) == ("3", 5, 6)
>>> fpath.write_text('b = 10' + chr(10) + 'a = None')
>>> assert _static_parse('a', fpath) is None
>>> import pytest
>>> with pytest.raises(TypeError):
>>>     fpath.write_text('a = list(range(10))')
>>>     assert _static_parse('a', fpath) is None
>>> with pytest.raises(AttributeError):
>>>     fpath.write_text('a = list(range(10))')
>>>     assert _static_parse('c', fpath) is None
```

line_profiler.autoprofile.util_static.**_platform_pylib_exts**()

Returns .so, .pyd, or .dylib depending on linux, win or mac. On python3 return the previous with and without abi (e.g. .cpython-35m-x86_64-linux-gnu) flags. On python2 returns with and without multiarch.

Returns
tuple

line_profiler.autoprofile.util_static.**_syspath_modname_to_modpath**(*modname*, *sys_path=None*, *exclude=None*)

syspath version of modname_to_modpath

Parameters

- **modname** (*str*) – name of module to find
- **sys_path** (*None | List[str | PathLike]*) – The paths to search for the module. If unspecified, defaults to `sys.path`.
- **exclude** (*List[str | PathLike] | None*) – If specified prevents these directories from being searched. Defaults to None.

Returns
path to the module.

Return type
str

Note: This is much slower than the pkgutil mechanisms.

There seems to be a change to the editable install mechanism: https://github.com/pypa/setuptools/issues/3548 Trying to find more docs about it.

TODO: add a test where we make an editable install, regular install, standalone install, and check that we always find the right path.

Example

```
>>> print(_syspath_modname_to_modpath('xdoctest.static_analysis'))
...static_analysis.py
>>> print(_syspath_modname_to_modpath('xdoctest'))
...xdoctest
>>> # xdoctest: +REQUIRES(CPython)
>>> print(_syspath_modname_to_modpath('_ctypes'))
..._ctypes...
>>> assert _syspath_modname_to_modpath('xdoctest', sys_path=[]) is None
>>> assert _syspath_modname_to_modpath('xdoctest.static_analysis', sys_path=[]) is␣
↪None
>>> assert _syspath_modname_to_modpath('_ctypes', sys_path=[]) is None
>>> assert _syspath_modname_to_modpath('this', sys_path=[]) is None
```

Example

```
>>> # test what happens when the module is not visible in the path
>>> modname = 'xdoctest.static_analysis'
>>> modpath = _syspath_modname_to_modpath(modname)
>>> exclude = [split_modpath(modpath)[0]]
>>> found = _syspath_modname_to_modpath(modname, exclude=exclude)
>>> if found is not None:
>>>     # Note: the basic form of this test may fail if there are
>>>     # multiple versions of the package installed. Try and fix that.
>>>     other = split_modpath(found)[0]
>>>     assert other not in exclude
>>>     exclude.append(other)
>>>     found = _syspath_modname_to_modpath(modname, exclude=exclude)
>>> if found is not None:
>>>     raise AssertionError(
>>>         'should not have found {}.'.format(found) +
>>>         ' because we excluded: {}.'.format(exclude) +
>>>         ' cwd={} '.format(os.getcwd()) +
>>>         ' sys.path={} '.format(sys.path)
>>>     )
```

line_profiler.autoprofile.util_static.**modname_to_modpath**(*modname*, *hide_init=True*, *hide_main=False*, *sys_path=None*)

Finds the path to a python module from its name.

Determines the path to a python module without directly import it

Converts the name of a module (__name__) to the path (__file__) where it is located without importing the module. Returns None if the module does not exist.

Parameters

- **modname** (*str*) – The name of a module in `sys_path`.
- **hide_init** (*bool*) – if False, __init__.py will be returned for packages. Defaults to True.
- **hide_main** (*bool*) – if False, and `hide_init` is True, __main__.py will be returned for packages, if it exists. Defaults to False.
- **sys_path** (*None | List[str | PathLike]*) – The paths to search for the module. If unspecified, defaults to `sys.path`.

Returns
modpath - path to the module, or None if it doesn't exist

Return type
str | None

Example

```
>>> modname = 'xdoctest.__main__'
>>> modpath = modname_to_modpath(modname, hide_main=False)
>>> assert modpath.endswith('__main__.py')
>>> modname = 'xdoctest'
>>> modpath = modname_to_modpath(modname, hide_init=False)
>>> assert modpath.endswith('__init__.py')
>>> # xdoctest: +REQUIRES(CPython)
>>> modpath = basename(modname_to_modpath('_ctypes'))
>>> assert 'ctypes' in modpath
```

line_profiler.autoprofile.util_static.**split_modpath**(*modpath*, *check=True*)

Splits the modpath into the dir that must be in PYTHONPATH for the module to be imported and the modulepath relative to this directory.

Parameters

- **modpath** (*str*) – module filepath
- **check** (*bool*) – if False, does not raise an error if modpath is a directory and does not contain an `__init__.py` file.

Returns
(directory, rel_modpath)

Return type
Tuple[str, str]

Raises
ValueError – if modpath does not exist or is not a package

Example

```
>>> from xdoctest import static_analysis
>>> modpath = static_analysis.__file__.replace('.pyc', '.py')
>>> modpath = abspath(modpath)
>>> dpath, rel_modpath = split_modpath(modpath)
>>> recon = join(dpath, rel_modpath)
>>> assert recon == modpath
>>> assert rel_modpath == join('xdoctest', 'static_analysis.py')
```

line_profiler.autoprofile.util_static.**normalize_modpath**(*modpath*, *hide_init=True*, *hide_main=False*)

Normalizes __init__ and __main__ paths.

Parameters

- **modpath** (*str | PathLike*) – path to a module
- **hide_init** (*bool*) – if True, always return package modules as __init__.py files otherwise always return the dpath. Defaults to True.
- **hide_main** (*bool*) – if True, always strip away main files otherwise ignore __main__.py. Defaults to False.

Returns

a normalized path to the module

Return type

str | PathLike

Note: Adds __init__ if reasonable, but only removes __main__ by default

---

Example

```
>>> from xdoctest import static_analysis as module
>>> modpath = module.__file__
>>> assert normalize_modpath(modpath) == modpath.replace('.pyc', '.py')
>>> dpath = dirname(modpath)
>>> res0 = normalize_modpath(dpath, hide_init=0, hide_main=0)
>>> res1 = normalize_modpath(dpath, hide_init=0, hide_main=1)
>>> res2 = normalize_modpath(dpath, hide_init=1, hide_main=0)
>>> res3 = normalize_modpath(dpath, hide_init=1, hide_main=1)
>>> assert res0.endswith('__init__.py')
>>> assert res1.endswith('__init__.py')
>>> assert not res2.endswith('.py')
>>> assert not res3.endswith('.py')
```

line_profiler.autoprofile.util_static.**modpath_to_modname**(*modpath*, *hide_init=True*, *hide_main=False*, *check=True*, *relativeto=None*)

Determines importable name from file path

Converts the path to a module (__file__) to the importable python name (__name__) without importing the module.

The filename is converted to a module name, and parent directories are recursively included until a directory without an __init__.py file is encountered.

Parameters

- **modpath** (*str*) – module filepath
- **hide_init** (*bool, default=True*) – removes the __init__ suffix
- **hide_main** (*bool, default=False*) – removes the __main__ suffix
- **check** (*bool, default=True*) – if False, does not raise an error if modpath is a dir and does not contain an __init__ file.
- **relativeto** (*str | None, default=None*) – if specified, all checks are ignored and this is considered the path to the root module.

Todo:

- **[ ] Does this need modification to support PEP 420?**
  https://www.python.org/dev/peps/pep-0420/

Returns
modname

Return type
str

Raises
ValueError – if check is True and the path does not exist

Example

```
>>> from xdoctest import static_analysis
>>> modpath = static_analysis.__file__.replace('.pyc', '.py')
>>> modpath = modpath.replace('.pyc', '.py')
>>> modname = modpath_to_modname(modpath)
>>> assert modname == 'xdoctest.static_analysis'
```

Example

```
>>> import xdoctest
>>> assert modpath_to_modname(xdoctest.__file__.replace('.pyc', '.py')) == 'xdoctest
↪'
>>> assert modpath_to_modname(dirname(xdoctest.__file__.replace('.pyc', '.py'))) ==
↪'xdoctest'
```

Example

```
>>> # xdoctest: +REQUIRES(CPython)
>>> modpath = modname_to_modpath('_ctypes')
>>> modname = modpath_to_modname(modpath)
>>> assert modname == '_ctypes'
```

Example

```
>>> modpath = '/foo/libfoobar.linux-x86_64-3.6.so'
>>> modname = modpath_to_modname(modpath, check=False)
>>> assert modname == 'libfoobar'
```

3.1.1.2 Module contents

3.1.1.2.1 Auto-Profiling

In cases where you want to decorate every function in a script, module, or package decorating every function with `@profile` can be tedious. To make this easier "auto-profiling" was introduced to `line_profiler` in Version 4.1.0.

The "auto-profile" feature allows users to specify a script or module. This is done by passing the name of script or module to kernprof.

To demonstrate auto-profiling, we first need to generate a Python script to profile. Write the following code to a file called `demo_primes2.py`.

```
def is_prime(n):
    '''
    Check if the number "n" is prime, with n > 1.

    Returns a boolean, True if n is prime.
    '''
    max_val = n ** 0.5
    stop = int(max_val + 1)
    for i in range(2, stop):
        if n % i == 0:
            return False
    return True


def find_primes(size):
    primes = []
    for n in range(size):
        flag = is_prime(n)
        if flag:
            primes.append(n)
    return primes


def main():
    print('start calculating')
```

(continues on next page)

(continued from previous page)

```
    primes = find_primes(100000)
    print(f'done calculating. Found {len(primes)} primes.')


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

Note that this is the nearly the same "primes" example from the "Basic Usage" section in *line_profiler*, but none of the functions are decorated with `@profile`.

To run this script with auto-profiling we invoke kernprof with `-p` or `--prof-mod` and pass the names of the script. When running with `-p` we must also run with `-l` to enable line profiling. In this example we include `-v` as well to display the output after we run it.

```
python -m kernprof -lv -p demo_primes2.py demo_primes2.py
```

The output will look like this:

```
start calculating
done calculating. Found 9594 primes.
Wrote profile results to demo_primes2.py.lprof
Timer unit: 1e-06 s

Total time: 0.677348 s
File: demo_primes2.py
Function: is_prime at line 4

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     4                                           def is_prime(n):
     5                                               '''
     6                                               Check if the number "n" is prime,␣
↪with n > 1.
     7
     8                                               Returns a boolean, True if n is␣
↪prime.
     9                                               '''
    10    100000      19921.6      0.2      2.9      math.floor(1.3)
    11    100000      17743.6      0.2      2.6      max_val = n ** 0.5
    12    100000      23962.7      0.2      3.5      stop = int(max_val + 1)
    13   2745693     262005.7      0.1     38.7      for i in range(2, stop):
    14   2655287     342216.1      0.1     50.5          if n % i == 0:
    15     90406      10401.4      0.1      1.5              return False
    16      9594       1097.2      0.1      0.2      return True


Total time: 1.56657 s
File: demo_primes2.py
Function: find_primes at line 19

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    19                                           def find_primes(size):
```

(continues on next page)

(continued from previous page)

```
    20         1          0.3      0.3      0.0      primes = []
    21    100000      11689.5      0.1      0.7      for n in range(size):
    22    100000    1541848.0     15.4     98.4          flag = is_prime(n)
    23     90406      10260.0      0.1      0.7          if flag:
    24      9594       2775.9      0.3      0.2              primes.append(n)
    25         1          0.2      0.2      0.0      return primes


Total time: 1.61013 s
File: demo_primes2.py
Function: main at line 28

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    28                                           def main():
    29         1         17.6     17.6      0.0      print('start calculating')
    30         1    1610081.3    2e+06    100.0      primes = find_primes(100000)
    31         1         26.6     26.6      0.0      print(f'done calculating. Found
↪{len(primes)} primes.')
```

3.2 Submodules

3.2.1 line_profiler.__main__ module

3.2.2 line_profiler._line_profiler module

This is the Cython backend used in *line_profiler.line_profiler*.

class line_profiler._line_profiler.**LineProfiler**

Bases: object

Time the execution of lines of Python code.

This is the Cython base class for *line_profiler.line_profiler.LineProfiler*.

Example

```
>>> import copy
>>> import line_profiler
>>> # Create a LineProfiler instance
>>> self = line_profiler.LineProfiler()
>>> # Wrap a function
>>> copy_fn = self(copy.copy)
>>> # Call the function
>>> copy_fn(self)
>>> # Inspect internal properties
>>> self.functions
>>> self.c_last_time
>>> self.c_code_map
>>> self.code_map
```

(continues on next page)

(continued from previous page)

```
>>> self.last_time
>>> # Print stats
>>> self.print_stats()
```

add_function(*func*)

Record line profiling information for the given Python function.

c_code_map

A Python view of the internal C lookup table.

c_last_time

code_hash_map

code_map

line_profiler 4.0 no longer directly maintains code_map, but this will construct something similar for backwards compatibility.

disable()

disable_by_count()

Disable the profiler if the number of disable requests matches the number of enable requests.

dupes_map

enable()

enable_by_count()

Enable the profiler if it hasn't been enabled before.

enable_count

functions

get_stats()

Return a LineStats object containing the timings.

last_time

line_profiler 4.0 no longer directly maintains last_time, but this will construct something similar for backwards compatibility.

threaddata

timer_unit

class line_profiler._line_profiler.**LineStats**(*timings*, *unit*)

Bases: object

Object to encapsulate line-profile statistics.

Variables

- **timings** (*dict*) – Mapping from (filename, first_lineno, function_name) of the profiled function to a list of (lineno, nhits, total_time) tuples for each profiled line. total_time is an integer in the native units of the timer.
- **unit** (*float*) – The number of seconds per timer unit.

line_profiler._line_profiler.**__pyx_unpickle_LineProfiler**(*__pyx_type*, *__pyx_checksum*, *__pyx_state*)

line_profiler._line_profiler.**_code_replace**(*func*, *co_code*)

Implements CodeType.replace for Python < 3.8

line_profiler._line_profiler.**label**(*code*)

Return a (filename, first_lineno, func_name) tuple for a given code object.

This is the same labelling as used by the cProfile module in Python 2.5.

3.2.3 line_profiler.explicit_profiler module

New in `line_profiler` version 4.1.0, this module defines a top-level `profile` decorator which will be disabled by default **unless** a script is being run with *kernprof*, if the environment variable `LINE_PROFILE` is set, or if `--line-profile` is given on the command line.

In the latter two cases, the `atexit` module is used to display and dump line profiling results to disk when Python exits.

If none of the enabling conditions are met, then `line_profiler.profile` is a noop. This means you no longer have to add and remove the implicit `profile` decorators required by previous version of this library.

Basic usage is to import line_profiler and decorate your function with line_profiler.profile. By default this does nothing, it's a no-op decorator. However, if you run with the environment variable `LINE_PROFILER=1` or if `'--profile' in sys.argv`, then it enables profiling and at the end of your script it will output the profile text.

Here is a minimal example that will write a script to disk and then run it with profiling enabled or disabled by various methods:

```
# Write demo python script to disk
python -c "if 1:
    import textwrap
    text = textwrap.dedent(
        '''
        from line_profiler import profile

        @profile
        def plus(a, b):
            return a + b

        @profile
        def fib(n):
            a, b = 0, 1
            while a < n:
                a, b = b, plus(a, b)

        @profile
        def main():
            import math
            import time
            start = time.time()

            print('start calculating')
            while time.time() - start < 1:
                fib(10)
```

(continues on next page)

(continued from previous page)

```
                math.factorial(1000)
            print('done calculating')

        main()
        '''
    ).strip()
    with open('demo.py', 'w') as file:
        file.write(text)
"

echo "---"
echo "## Base Case: Run without any profiling"
python demo.py

echo "---"
echo "## Option 0: Original Usage"
python -m kernprof -l demo.py
python -m line_profiler -rmt demo.py.lprof

echo "---"
echo "## Option 1: Enable profiler with the command line"
python demo.py --line-profile

echo "---"
echo "## Option 1: Enable profiler with an environment variable"
LINE_PROFILE=1 python demo.py
```

The explicit `line_profiler.profile` decorator can also be enabled and configured in the Python code itself by calling `line_profiler.profile.enable()`. The following example demonstrates this:

```
# In-code enabling
python -c "if 1:
    import textwrap
    text = textwrap.dedent(
        '''
        from line_profiler import profile
        profile.enable(output_prefix='customized')

        @profile
        def fib(n):
            a, b = 0, 1
            while a < n:
                a, b = b, a + b

        fib(100)
        '''
    ).strip()
    with open('demo.py', 'w') as file:
        file.write(text)
"
echo "## Configuration handled inside the script"
python demo.py
```

Likewise there is a `line_profiler.profile.disable()` function that will prevent any subsequent functions decorated with `@profile` from being profiled. In the following example, profiling information will only be recorded for `func2` and `func4`.

```
# In-code enabling / disable
python -c "if 1:
    import textwrap
    text = textwrap.dedent(
        '''
        from line_profiler import profile

        @profile
        def func1():
            return list(range(100))

        profile.enable(output_prefix='custom')

        @profile
        def func2():
            return tuple(range(100))

        profile.disable()

        @profile
        def func3():
            return set(range(100))

        profile.enable()

        @profile
        def func4():
            return dict(zip(range(100), range(100)))

        print(type(func1()))
        print(type(func2()))
        print(type(func3()))
        print(type(func4()))
        '''
    ).strip()
    with open('demo.py', 'w') as file:
        file.write(text)
"

echo "---"
echo "## Configuration handled inside the script"
python demo.py

# Running with --line-profile will also profile ``func1``
python demo.py --line-profile
```

The core functionality in this module was ported from `xdev`.

class line_profiler.explicit_profiler.**GlobalProfiler**

Bases: `object`

Manages a profiler that will output on interpreter exit.

The `line_profile.profile` decorator is an instance of this object.

Variables

- **setup_config** (*Dict[str, List[str]]*) – Determines how the implicit setup behaves by defining which environment variables / command line flags to look for.
- **output_prefix** (*str*) – The prefix of any output files written. Should include a part of a filename. Defaults to "profile_output".
- **write_config** (*Dict[str, bool]*) – Which outputs are enabled. All default to True. Options are lprof, text, timestamped_text, and stdout.
- **show_config** (*Dict[str, bool]*) – Display configuration options. Some outputs force certain options. (e.g. text always has details and is never rich).
- **enabled** (*bool | None*) – True if the profiler is enabled (i.e. if it will wrap a function that it decorates with a real profiler). If None, then the value defaults based on the `setup_config`, `os.environ`, and `sys.argv`.

Example

```
>>> from line_profiler.explicit_profiler import *  # NOQA
>>> self = GlobalProfiler()
>>> # Setting the _profile attribute prevents atexit from running.
>>> self._profile = LineProfiler()
>>> # User can personalize the configuration
>>> self.show_config['details'] = True
>>> self.write_config['lprof'] = False
>>> self.write_config['text'] = False
>>> self.write_config['timestamped_text'] = False
>>> # Demo data: a function to profile
>>> def collatz(n):
>>>     while n != 1:
>>>         if n % 2 == 0:
>>>             n = n // 2
>>>         else:
>>>             n = 3 * n + 1
>>>     return n
>>> # Disabled by default, implicitly checks to auto-enable on first wrap
>>> assert self.enabled is None
>>> wrapped = self(collatz)
>>> assert self.enabled is False
>>> assert wrapped is collatz
>>> # Can explicitly enable
>>> self.enable()
>>> wrapped = self(collatz)
>>> assert self.enabled is True
>>> assert wrapped is not collatz
>>> wrapped(100)
>>> # Can explicitly request output
>>> self.show()
```

_kernprof_overwrite(*profile*)

Kernprof will call this when it runs, so we can use its profile object instead of our own. Note: when kernprof overwrites us we wont register an atexit hook. This is what we want because kernprof wants us to use another program to read its output file.

_implicit_setup()

Called once the first time the user decorates a function with `line_profiler.profile` and they have not explicitly setup the global profiling options.

enable(*output_prefix=None*)

Explicitly enables global profiler and controls its settings.

disable()

Explicitly initialize and disable this global profiler.

show()

Write the managed profiler stats to enabled outputs.

If the implicit setup triggered, then this will be called by `atexit`.

line_profiler.explicit_profiler.**_python_command**()

Return a command that corresponds to `sys.executable`.

3.2.4 line_profiler.ipython_extension module

3.2.5 line_profiler.line_profiler module

This module defines the core *LineProfiler* class as well as methods to inspect its output. This depends on the *line_profiler._line_profiler* Cython backend.

line_profiler.line_profiler.**load_ipython_extension**(*ip*)

API for IPython to recognize this module as an IPython extension.

line_profiler.line_profiler.**is_coroutine**(*f*)

line_profiler.line_profiler.**is_generator**(*f*)

Return True if a function is a generator.

line_profiler.line_profiler.**is_classmethod**(*f*)

class line_profiler.line_profiler.**LineProfiler**

Bases: *LineProfiler*

A profiler that records the execution times of individual lines.

This provides the core line-profiler functionality.

Example

```
>>> import line_profiler
>>> profile = line_profiler.LineProfiler()
>>> @profile
>>> def func():
>>>     x1 = list(range(10))
>>>     x2 = list(range(100))
>>>     x3 = list(range(1000))
>>> func()
>>> profile.print_stats()
```

wrap_classmethod(*func*)

Wrap a classmethod to profile it.

wrap_coroutine(*func*)

Wrap a Python 3.5 coroutine to profile it.

wrap_generator(*func*)

Wrap a generator to profile it.

wrap_function(*func*)

Wrap a function to profile it.

dump_stats(*filename*)

Dump a representation of the data to a file as a pickled LineStats object from *get_stats()*.

print_stats(*stream=None*, *output_unit=None*, *stripzeros=False*, *details=True*, *summarize=False*, *sort=False*, *rich=False*)

Show the gathered statistics.

run(*cmd*)

Profile a single executable statment in the main namespace.

runctx(*cmd*, *globals*, *locals*)

Profile a single executable statement in the given namespaces.

runcall(*func*, **args*, ***kw*)

Profile a single function call.

add_module(*mod*)

Add all the functions in a module and its classes.

line_profiler.line_profiler.**is_ipython_kernel_cell**(*filename*)

Return True if a filename corresponds to a Jupyter Notebook cell

line_profiler.line_profiler.**show_func**(*filename*, *start_lineno*, *func_name*, *timings*, *unit*, *output_unit=None*, *stream=None*, *stripzeros=False*, *rich=False*)

Show results for a single function.

Parameters

- **filename** (*str*) – path to the profiled file
- **start_lineno** (*int*) – first line number of profiled function
- **func_name** (*str*) – name of profiled function
- **timings** (*List[Tuple[int, int, float]]*) – measurements for each line (lineno, nhits, time).

- **unit** (*float*) – The number of seconds used as the cython LineProfiler's unit.
- **output_unit** (*float | None*) – Output unit (in seconds) in which the timing info is displayed.
- **stream** (*io.TextIOBase | None*) – defaults to sys.stdout
- **stripzeros** (*bool*) – if True, prints nothing if the function was not run
- **rich** (*bool*) – if True, attempt to use rich highlighting.

Example

```
>>> from line_profiler.line_profiler import show_func
>>> import line_profiler
>>> # Use a function in this file as an example
>>> func = line_profiler.line_profiler.show_text
>>> start_lineno = func.__code__.co_firstlineno
>>> filename = func.__code__.co_filename
>>> func_name = func.__name__
>>> # Build fake timeings for each line in the example function
>>> import inspect
>>> num_lines = len(inspect.getsourcelines(func)[0])
>>> line_numbers = list(range(start_lineno + 3, start_lineno + num_lines))
>>> timings = [
>>>     (lineno, idx * 1e13, idx * (2e10 ** (idx % 3)))
>>>     for idx, lineno in enumerate(line_numbers, start=1)
>>> ]
>>> unit = 1.0
>>> output_unit = 1.0
>>> stream = None
>>> stripzeros = False
>>> rich = 1
>>> show_func(filename, start_lineno, func_name, timings, unit,
>>>           output_unit, stream, stripzeros, rich)
```

line_profiler.line_profiler.**show_text**(*stats*, *unit*, *output_unit=None*, *stream=None*, *stripzeros=False*, *details=True*, *summarize=False*, *sort=False*, *rich=False*)

Show text for the given timings.

line_profiler.line_profiler.**load_stats**(*filename*)

Utility function to load a pickled LineStats object from a given filename.

line_profiler.line_profiler.**main**()

The line profiler CLI to view output from `kernprof -l`.

3.3 Module contents

3.3.1 Line Profiler

The line_profiler module for doing line-by-line profiling of functions

Github	https://github.com/pyutils/line_profiler
Pypi	https://pypi.org/project/line_profiler
ReadTheDocs	https://kernprof.readthedocs.io/en/latest/

3.3.2 Installation

Releases of *line_profiler* and *kernprof* can be installed using pip

```
pip install line_profiler
```

The package also provides extras for optional dependencies, which can be installed via:

```
pip install line_profiler[all]
```

3.3.3 Line Profiler Basic Usage

To demonstrate line profiling, we first need to generate a Python script to profile. Write the following code to a file called `demo_primes.py`.

```
from line_profiler import profile


@profile
def is_prime(n):
    '''
    Check if the number "n" is prime, with n > 1.

    Returns a boolean, True if n is prime.
    '''
    max_val = n ** 0.5
    stop = int(max_val + 1)
    for i in range(2, stop):
        if n % i == 0:
            return False
    return True


@profile
def find_primes(size):
    primes = []
    for n in range(size):
        flag = is_prime(n)
        if flag:
            primes.append(n)
```

(continues on next page)

(continued from previous page)

```
    return primes


@profile
def main():
    print('start calculating')
    primes = find_primes(100000)
    print(f'done calculating. Found {len(primes)} primes.')


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

In this script we explicitly import the `profile` function from `line_profiler`, and then we decorate function of interest with `@profile`.

By default nothing is profiled when running the script.

```
python demo_primes.py
```

The output will be

```
start calculating
done calculating. Found 9594 primes.
```

The quickest way to enable profiling is to set the environment variable `LINE_PROFILE=1` and running your script as normal.

```
LINE_PROFILE=1 python demo_primes.py
```

This will output 3 files: profile_output.txt, profile_output_<timestamp>.txt, and profile_output.lprof and stdout will look something like:

```
start calculating
done calculating. Found 9594 primes.
Timer unit: 1e-09 s

  0.65 seconds - demo_primes.py:4 - is_prime
  1.47 seconds - demo_primes.py:19 - find_primes
  1.51 seconds - demo_primes.py:29 - main
Wrote profile results to profile_output.txt
Wrote profile results to profile_output_2023-08-12T193302.txt
Wrote profile results to profile_output.lprof
To view details run:
python -m line_profiler -rtmz profile_output.lprof
```

For more control over the outputs, run your script using *kernprof*. The following invocation will run your script, dump results to `demo_primes.py.lprof`, and display results.

```
python -m kernprof -lvr demo_primes.py
```

Note: the `-r` flag will use "rich-output" if you have the `rich` module installed.

See also:

- autoprofiling usage in: *line_profiler.autoprofile*

class line_profiler.**LineProfiler**

Bases: *LineProfiler*

A profiler that records the execution times of individual lines.

This provides the core line-profiler functionality.

Example

```
>>> import line_profiler
>>> profile = line_profiler.LineProfiler()
>>> @profile
>>> def func():
>>>     x1 = list(range(10))
>>>     x2 = list(range(100))
>>>     x3 = list(range(1000))
>>> func()
>>> profile.print_stats()
```

wrap_classmethod(*func*)

Wrap a classmethod to profile it.

wrap_coroutine(*func*)

Wrap a Python 3.5 coroutine to profile it.

wrap_generator(*func*)

Wrap a generator to profile it.

wrap_function(*func*)

Wrap a function to profile it.

dump_stats(*filename*)

Dump a representation of the data to a file as a pickled LineStats object from *get_stats()*.

print_stats(*stream=None*, *output_unit=None*, *stripzeros=False*, *details=True*, *summarize=False*, *sort=False*, *rich=False*)

Show the gathered statistics.

run(*cmd*)

Profile a single executable statment in the main namespace.

runctx(*cmd*, *globals*, *locals*)

Profile a single executable statement in the given namespaces.

runcall(*func*, **args*, ***kw*)

Profile a single function call.

add_module(*mod*)

Add all the functions in a module and its classes.

line_profiler.**load_ipython_extension**(*ip*)

API for IPython to recognize this module as an IPython extension.

line_profiler.**load_stats**(*filename*)

Utility function to load a pickled LineStats object from a given filename.

line_profiler.**main**()

The line profiler CLI to view output from `kernprof -l`.

line_profiler.**show_func**(*filename*, *start_lineno*, *func_name*, *timings*, *unit*, *output_unit=None*, *stream=None*, *stripzeros=False*, *rich=False*)

Show results for a single function.

Parameters

- **filename** (*str*) – path to the profiled file
- **start_lineno** (*int*) – first line number of profiled function
- **func_name** (*str*) – name of profiled function
- **timings** (*List[Tuple[int, int, float]]*) – measurements for each line (lineno, nhits, time).
- **unit** (*float*) – The number of seconds used as the cython LineProfiler's unit.
- **output_unit** (*float | None*) – Output unit (in seconds) in which the timing info is displayed.
- **stream** (*io.TextIOBase | None*) – defaults to sys.stdout
- **stripzeros** (*bool*) – if True, prints nothing if the function was not run
- **rich** (*bool*) – if True, attempt to use rich highlighting.

Example

```
>>> from line_profiler.line_profiler import show_func
>>> import line_profiler
>>> # Use a function in this file as an example
>>> func = line_profiler.line_profiler.show_text
>>> start_lineno = func.__code__.co_firstlineno
>>> filename = func.__code__.co_filename
>>> func_name = func.__name__
>>> # Build fake timeings for each line in the example function
>>> import inspect
>>> num_lines = len(inspect.getsourcelines(func)[0])
>>> line_numbers = list(range(start_lineno + 3, start_lineno + num_lines))
>>> timings = [
>>>     (lineno, idx * 1e13, idx * (2e10 ** (idx % 3)))
>>>     for idx, lineno in enumerate(line_numbers, start=1)
>>> ]
>>> unit = 1.0
>>> output_unit = 1.0
>>> stream = None
>>> stripzeros = False
>>> rich = 1
>>> show_func(filename, start_lineno, func_name, timings, unit,
>>>           output_unit, stream, stripzeros, rich)
```

line_profiler.**show_text**(*stats*, *unit*, *output_unit=None*, *stream=None*, *stripzeros=False*, *details=True*, *summarize=False*, *sort=False*, *rich=False*)

Show text for the given timings.

CHAPTER

FOUR

KERNPROF MODULE

Script to conveniently run profilers on code in a variety of circumstances.

To profile a script, decorate the functions of interest with `@profile`

```
echo "if 1:
    @profile
    def main():
        1 + 1
    main()
" > script_to_profile.py
```

Note: New in 4.1.0: Instead of relying on injecting `profile` into the builtins you can now `import line_profiler` and use `line_profiler.profile` to decorate your functions. This allows the script to remain functional even if it is not actively profiled. See *line_profiler* for details.

Then run the script using kernprof:

```
kernprof -b script_to_profile.py
```

By default this runs with the default `cProfile` profiler and does not require compiled modules. Instructions to view the results will be given in the output. Alternatively, adding `-v` to the command line will write results to stdout.

To enable line-by-line profiling, then *line_profiler* must be available and compiled. Add the `-l` argument to the kernprof invocation.

```
kernprof -lb script_to_profile.py
```

For more details and options, refer to the CLI help. To view kernprof help run:

```
kenprof --help
```

which displays:

```
usage: kernprof [-h] [-V] [-l] [-b] [-o OUTFILE] [-s SETUP] [-v] [-r] [-u UNIT] [-z] [-i␣
↪[OUTPUT_INTERVAL]] [-p PROF_MOD] [--prof-imports] script ...

Run and profile a python script.

positional arguments:
  script                The python script file to run
  args                  Optional script arguments
```

(continues on next page)

(continued from previous page)

```
options:
  -h, --help            show this help message and exit
  -V, --version         show program's version number and exit
  -l, --line-by-line    Use the line-by-line profiler instead of cProfile. Implies --
↪builtin.
  -b, --builtin         Put 'profile' in the builtins. Use 'profile.enable()'/'.disable()
↪', '@profile' to decorate functions, or 'with profile:' to profile a section of code.
  -o OUTFILE, --outfile OUTFILE
                        Save stats to <outfile> (default: 'scriptname.lprof' with --line-
↪by-line, 'scriptname.prof' without)
  -s SETUP, --setup SETUP
                        Code to execute before the code to profile
  -v, --view            View the results of the profile in addition to saving it
  -r, --rich            Use rich formatting if viewing output
  -u UNIT, --unit UNIT  Output unit (in seconds) in which the timing info is displayed␣
↪(default: 1e-6)
  -z, --skip-zero       Hide functions which have not been called
  -i [OUTPUT_INTERVAL], --output-interval [OUTPUT_INTERVAL]
                        Enables outputting of cumulative profiling results to file every␣
↪n seconds. Uses the threading module. Minimum value is 1 (second). Defaults to␣
↪disabled.
  -p PROF_MOD, --prof-mod PROF_MOD
                        List of modules, functions and/or classes to profile specified␣
↪by their name or path. List is comma separated, adding the current script path profiles
                        full script. Only works with line_profiler -l, --line-by-line
  --prof-imports        If specified, modules specified to `--prof-mod` will also␣
↪autoprofile modules that they import. Only works with line_profiler -l, --line-by-line
```

kernprof.**execfile**(*filename*, *globals=None*, *locals=None*)

Python 3.x doesn't have 'execfile' builtin

kernprof.**is_generator**(*f*)

Return True if a function is a generator.

class kernprof.**ContextualProfile**(**args*, ***kwds*)

Bases: `Profile`

A subclass of Profile that adds a context manager for Python 2.5 with: statements and a decorator.

enable_by_count(*subcalls=True*, *builtins=True*)

Enable the profiler if it hasn't been enabled before.

disable_by_count()

Disable the profiler if the number of disable requests matches the number of enable requests.

wrap_generator(*func*)

Wrap a generator to profile it.

wrap_function(*func*)

Wrap a function to profile it.

class kernprof.**RepeatedTimer**(*interval*, *dump_func*, *outfile*)

Bases: `object`

Background timer for outputting file every n seconds.

Adapted from [SO474528].

References

_run()

start()

stop()

kernprof.**find_script**(*script_name*)

Find the script.

If the input is not a file, then $PATH will be searched.

kernprof.**_python_command**()

Return a command that corresponds to `sys.executable`.

kernprof.**main**(*args=None*)

Runs the command line interface

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex

BIBLIOGRAPHY

[SO474528] https://stackoverflow.com/questions/474528/execute-function-every-x-seconds/40965385#40965385

PYTHON MODULE INDEX

k

l

INDEX

Symbols

A

C

D

N

P

R

S

T

V

W