py-pal

Release latest

Lukas Jung

May 21, 2021

CONTENTS

1 Overview **1**
- 1.1 Installation . . . 1
- 1.2 Command line usage of the py-pal module . . . 2
- 1.3 Programmatic usage of the py-pal module . . . 2
- 1.4 Licensing Notes . . . 4

2 Project Structure **5**
- 2.1 Configuration files . . . 5

3 Development **7**
- 3.1 Development Environment Setup . . . 7
- 3.2 Building the *py-pal* module . . . 7
- 3.3 Testing . . . 8
- 3.4 Logging . . . 8

4 API Reference **9**
- 4.1 Subpackages . . . 9
- 4.2 py_pal.core . . . 13
- 4.3 py_pal.datagen . . . 13
- 4.4 py_pal.util . . . 14

5 Changelog **15**
- 5.1 What's New in Py-PAL 1.3.0 . . . 15
- 5.2 Py-PAL 1.2.0 . . . 15
- 5.3 Py-PAL 1.1.0 . . . 15
- 5.4 Py-PAL 1.0.0 . . . 16
- 5.5 Py-PAL 0.2.1 . . . 16
- 5.6 Py-PAL 0.1.6 . . . 16

Python Module Index **19**

Index **21**

CHAPTER

ONE

OVERVIEW

The *Python Performance Analysis Library* (*py-pal*) is a profiling tool for the Python programming language. With *py-pal* one can approximate the time complexity (big O notation) of Python functions in an empirical way. The arguments of the function and the executed opcodes serve as a basis for the analysis.

To the docs.

1.1 Installation

1.1.1 Requirements

- **An installation of the CPython implementation of the Python programming language of version greater or equal to 3.7**
  - For instance: https://www.python.org/ftp/python/3.7.9/python-3.7.9-amd64.exe
- **A compiler for the C/C++ programming language:**
  - **On Microsoft Windows, we use the *Buildtools für Visual Studio 2019*:** https://visualstudio.microsoft.com/de/thank-you-downloading-visual-studio/?sku=BuildTools&rel=16
  - On Linux, any C compiler supported by Cython e.g. g++

1.1.2 Install py-pal via pip by running:

This project requires CPython and a C compiler to run. Install CPython >= 3.7, then install py-pal by running:

```
pip install py-pal
```

or

```
python -m pip install py-pal
```

1.2 Command line usage of the py-pal module

python -m py_pal <target-module/file>

or

py-pal <target-module/file>

There are multiple aliases to the same command: *py-pal*, *py_pal* and *pypal*. If py-pal is executed this way, all functions called in the code are captured and analyzed. The output is in the form of a pandas data frame.

See the help message:

py-pal -h

py-pal can perform cost analysis on a line-by-line basis:

py-pal <file> -l/–line

The –separate flag can be used to examine the cost function of individual arguments (**caution**: this function assumes the independence of the arguments):

py-pal <file> -s/–separate

The output of the results can be noisy, to limit this you can use –filter-function to filter the desired functions from the result. Regular expressions are supported:

py-pal <file> -ff/–filter-function .*_sort

Similarly, the result can also be filtered by modules with –filter-module, e.g. to exclude importlib modules

py-pal <file> -fm/–filter-module "^(?!<frozen.*>).*"

To save the results in a specified folder use –out:

py-pal <file> -o/–out results

The output format can be changed with –format:

py-pal <file> -o/–out results –format json

With the additional specification of the –plot flag, the cost functions of the result set are stored as images:

py-pal <file> -o/–out results -p/–plot

For the –log-level flag see the development docs.

Example, creating plots for selected functions:

py-pal tests/examples/sort.py -o results -p -f sort

1.3 Programmatic usage of the py-pal module

To profile a single function and get the complexity estimate there is *profile_function*.

```
from py_pal.core import profile_function
from py_pal.data_collection.opcode_metric import OpcodeMetric
from py_pal.datagen import gen_random_growing_lists
from algorithms.sort import bubble_sort

profile_function(OpcodeMetric(), gen_random_growing_lists(), bubble_sort)
```

The *profile* decorator:

```
from py_pal.core import profile, DecoratorStore

@profile
def test():
    pass

# Must be called at some point
test()

estimator = AllArgumentEstimator(DecoratorStore.get_call_stats(), DecoratorStore.get_
↪opcode_stats())
res = estimator.export()
```

By using the *profile* decorator, it is possible to annotate Python functions such that only the annotated Python functions will be profiled. It acts similar to a whitelist filter.

Another possibility is to use the context-manager protocol:

```
from py_pal.analysis.estimator import AllArgumentEstimator
from py_pal.data_collection.tracer import Tracer

with Tracer() as t:
    pass

estimator = AllArgumentEstimator(t.get_call_stats(), t.get_opcode_stats())
res = estimator.export()

# Do something with the resulting DataFrame
print(res)
```

The most verbose way to use the *py-pal* API:

```
from py_pal.analysis.estimator import AllArgumentEstimator
from py_pal.data_collection.tracer import Tracer


t = Tracer()
t.trace()

# Your function
pass

t.stop()
estimator = AllArgumentEstimator(t.get_call_stats(), t.get_opcode_stats())
res = estimator.export()

# Do something with the resulting DataFrame
print(res)
```

All examples instantiate a tracer object that is responsible for collecting the data. After execution, the collected data is passed to the analysis module. Finally, an estimate of the asymptotic runtime of the functions contained in the code is obtained.

1.3.1 Modes

In the current version py-pal offers only the **profiling mode**. Although `py_pal.datagen` offers some functions for generating inputs, py-pal must be combined with appropriate test cases to realize a **performance testing mode**. An automatic detection and generation of appropriate test inputs does not exist at the moment.

1.3.2 Limitations

The profiling approach implemented by the py-pal modules does not distinguish between different threads executing a Python function. Actually it is a major problem to profile a Python script which makes use of threads. The bytecode counting strategy will increase all counters of Python functions on the current call stack no matter what threads is executing it. Thus, the data points will not be accurate to what really happened during the profiled execution of the script.

1.4 Licensing Notes

This work integrates some code from the big_O project. More specifically, most code in `py_pal.analysis.complexity`, `py_pal.datagen` and `py_pal.analysis.estimator.Estimator.infer_complexity` is adapted from bigO.

CHAPTER

TWO

PROJECT STRUCTURE

This document gives an overview of the project structure. Furthermore the repository contains a number of various configuration files whose functions are briefly introduced below.

- ***src/*** This is where the source code of the project resides.
- ***tests/* and *tests_cython/*** Here are the unittests of the project. There are tests for python and cython code.

2.1 Configuration files

- ***.ci/*, *.gitlab-ci.yml*** .gitlab-ci.yml defines the CI/CD structure for the project. The individual jobs are:
  - Linting
  - Style guide checking
  - Testing with tox
  - Test coverage generation
  - Building sources distribution packages
  - Creating binary distribution packages (Linux only)
  - Creating new releases
  - Updating the documentation

  The shell scripts in the `.ci` folder are used to create wheel binary packages. The Dockerfiles represent the images on which the CI/CD jobs are executed.
- ***.gitignore*** Git configuration file to manage the tracking of files within the version control system.
- ***.coveragrc*** Test coverage configuration file for the Coverage.py package.
- ***dev-requirements.txt*** Here the development dependencies of the project are specified.
- ***LICENSE*** The license under which the project is published.
- ***manifest.in*** This file specifies additional files that should be packaged with the source distribution of the package. Read more about packaging source distributions.
- ***setup.py*** The setup.py makes the repository a python package. It is used to provide the package with meta information, to include the C extensions and to build the project.
- ***README.md*, *CHANGELOG.rst*, *.readthedocs.yml* and *docs/*** All these files are used to describe the project. The `README.rst` acts as the landing page of the repository. the `CHANGELOG.rst` contains notable changes between different versions. and `.readthedocs.yml` controls the generation of documentation at https://readthedocs.org/.

The docs folder contains the detailed documentation of the project. The format of the files is reStructuredText and is used in combination with the Sphinx library to generate the documentation.

- ***setup.cfg*** This configuration file is a setup configuration file. It is used to define generic settings in the project. Among other things, settings for linting are stored here, but also for testing. The project uses pytest to execute the tests, which also reads settings from this file.
- ***tox.ini*** Tox is a test automation system intended to improve the testing workflow. Tox runs the *pytest* module to execute the tests. This project makes use of the test environment generation functionality to run tests against a combination of different requirement configurations.

CHAPTER

THREE

DEVELOPMENT

3.1 Development Environment Setup

To set up an environment for developing the py-pal module, the requirements mentioned in the section *Installation* must be met. Then

1. Clone this repository locally with git
2. Navigate to the cloned repository
3. Create a virtual environment

   python -m venv .venv

4. Activate the virtual environment

   On Microsoft Windows run: .venv\Scripts\activate.bat On Linux run: source venv/bin/activate

5. Install the dependencies for the development environment

   pip install -r dev-requirements.txt

   or

   python -m pip install -r dev-requirements

3.2 Building the *py-pal* module

python setup.py develop

With this command, the C extensions are compiled using Cython. Also, it packages all necessary files together and installs them in the current virtual environment.

Note, any change to a cython file (.pyx) requires recompilation, i.e. the above command must be executed again.

Attention, if it is not possible to install Cython by this command, the cython files (.pyx) are not taken into account. This results in the circumstance that the corresponding C/C++ files are not generated and thus, the old C/C++ files get used to build the C extensions. Directly speaking, changes to the cython files will have no effect because they are not processed!

3.3 Testing

The test execution is managed with *pytest*, read more about selecting specific tests and general usage here.

Run all regular and Cython tests together from the command line

pytest

Run all regular tests from the command line

pytest tests

Run all Cython tests from the command line

pytest tests_cython

Run a single test from tests/test_complexity_classification.py with plotting enabled. This will generate plots similar to running py-pal from the command line with the -p/–plot flag. The files are placed in the specified folder.

set PYPAL_SAVE_PLOTS=plots

set PYPAL_SAVE_STATISTICS=data

pytest tests\test_complexity_classification.py::TestComplexityClassification::test_bubble_sort

3.4 Logging

Logging can be helpful for understanding and debugging the *py-pal* module. *py-pal* supports logging and the different verbosity levels. The default log level is `logging.WARNING`.

To run *py-pal* with increased verbosity execute

py-pal <target> –log-level=info

py-pal <target> –log-level=debug

During testing you can configure the general logging level using the *set_log_level* function from py_pal.util

```
import logging
from py_pal.util import set_log_level

set_log_level(logging.INFO)
```

It is also possible to configure different logging levels per module with

```
import logging
from py_pal.data_collection import proxy

logging.getLogger('py_pal.data_collection.proxy').setLevel(logging.DEBUG)
logging.getLogger(proxy.__name__).setLevel(logging.DEBUG)
```

CHAPTER

FOUR

API REFERENCE

4.1 Subpackages

4.1.1 py_pal.analysis

py_pal.analysis.complexity

Definition of complexity classes.

class py_pal.analysis.complexity.**Complexity**
Bases: object

Abstract class that fits complexity classes to timing data.

compute(*n: numpy.ndarray*) → float
Compute the value of the fitted function at *n*.

fit(*n: numpy.ndarray*, *t: numpy.ndarray*) → float
Fit complexity class parameters to timing data.

Parameters

- **n** (numpy.ndarray) – Array of values of N for which execution time has been measured.
- **t** (numpy.ndarray) – Array of execution times for each N in seconds.

Returns Residuals, sum of square errors of fit

Return type numpy.ndarray

classmethod format_str()
Return a string describing the fitted function.

The string must contain one formatting argument for each coefficient.

transform_n(*n: numpy.ndarray*)
Terms of the linear combination defining the complexity class.

Output format: number of Ns x number of coefficients.

transform_y(*t: numpy.ndarray*)
Transform time as needed for fitting. (e.g., t->log(t)) for exponential class.

class py_pal.analysis.complexity.**Constant**
Bases: *py_pal.analysis.complexity.Complexity*

classmethod format_str()
Return a string describing the fitted function.

The string must contain one formatting argument for each coefficient.

transform_n(*n: numpy.ndarray*)
Terms of the linear combination defining the complexity class.

Output format: number of Ns x number of coefficients.

class py_pal.analysis.complexity.**Cubic**
Bases: *py_pal.analysis.complexity.Complexity*

classmethod format_str()
Return a string describing the fitted function.

The string must contain one formatting argument for each coefficient.

transform_n(*n: numpy.ndarray*)
Terms of the linear combination defining the complexity class.

Output format: number of Ns x number of coefficients.

class py_pal.analysis.complexity.**Exponential**
Bases: *py_pal.analysis.complexity.Complexity*

classmethod format_str()
Return a string describing the fitted function.

The string must contain one formatting argument for each coefficient.

transform_n(*n: numpy.ndarray*)
Terms of the linear combination defining the complexity class.

Output format: number of Ns x number of coefficients.

transform_y(*t: numpy.ndarray*)
Transform time as needed for fitting. (e.g., t->log(t)) for exponential class.

class py_pal.analysis.complexity.**Linear**
Bases: *py_pal.analysis.complexity.Complexity*

classmethod format_str()
Return a string describing the fitted function.

The string must contain one formatting argument for each coefficient.

transform_n(*n: numpy.ndarray*)
Terms of the linear combination defining the complexity class.

Output format: number of Ns x number of coefficients.

class py_pal.analysis.complexity.**Linearithmic**
Bases: *py_pal.analysis.complexity.Complexity*

classmethod format_str()
Return a string describing the fitted function.

The string must contain one formatting argument for each coefficient.

transform_n(*n: numpy.ndarray*)
Terms of the linear combination defining the complexity class.

Output format: number of Ns x number of coefficients.

class py_pal.analysis.complexity.**Logarithmic**
Bases: *py_pal.analysis.complexity.Complexity*

classmethod format_str()
Return a string describing the fitted function.

The string must contain one formatting argument for each coefficient.

transform_n(*n: numpy.ndarray*)
Terms of the linear combination defining the complexity class.

Output format: number of Ns x number of coefficients.

exception py_pal.analysis.complexity.**NotFittedError**
Bases: Exception

class py_pal.analysis.complexity.**Polynomial**
Bases: *py_pal.analysis.complexity.Complexity*

classmethod format_str()
Return a string describing the fitted function.

The string must contain one formatting argument for each coefficient.

transform_n(*n: numpy.ndarray*)
Terms of the linear combination defining the complexity class.

Output format: number of Ns x number of coefficients.

transform_y(*t: numpy.ndarray*)
Transform time as needed for fitting. (e.g., t->log(t)) for exponential class.

class py_pal.analysis.complexity.**Quadratic**
Bases: *py_pal.analysis.complexity.Complexity*

classmethod format_str()
Return a string describing the fitted function.

The string must contain one formatting argument for each coefficient.

transform_n(*n: numpy.ndarray*)
Terms of the linear combination defining the complexity class.

Output format: number of Ns x number of coefficients.

exception py_pal.analysis.complexity.**UnderDeterminedEquation**
Bases: Exception

py_pal.analysis.estimator

class py_pal.analysis.estimator.**AllArgumentEstimator**(**args*, *arg_selection_strategy: py_pal.analysis.estimator.ArgumentDataSelectionStrategy = <ArgumentDataSelectionStrategy.MEAN: 'mean'>*, *opcode_selection_strategy: py_pal.analysis.estimator.ArgumentDataSelectionStrategy = <ArgumentDataSelectionStrategy.MEAN: 'mean'>*, ***kwargs*)
Bases: *py_pal.analysis.estimator.Estimator*

py_pal.analysis.estimator implementation that treats all arguments as one by averaging out the proxy value of all arguments.

infer_complexity_per_argument(*data_frame: pandas.core.frame.DataFrame*, *arg_names: List[str]*) → Tuple[List[str], Union[Exception, *py_pal.analysis.complexity.Complexity*], pandas.core.frame.DataFrame]

View all argument axes together and sort argument proxy value ascending

class py_pal.analysis.estimator.**ArgumentDataSelectionStrategy**(*value*)

Bases: enum.Enum

An enumeration.

MAX = 'max'

MEAN = 'mean'

MIN = 'min'

class py_pal.analysis.estimator.**Estimator**(*call_stats: numpy.ndarray*, *opcode_stats: numpy.ndarray*, *per_line: bool = False*, *filter_function: str = False*, *filter_module: str = False*)

Bases: abc.ABC

Base class which provides functionality to transform statistics collected by the py_pal.data_collection.Tracer to pandas.DataFrame objects, ways to aggregate opcode statistics per function call and fit the *py_pal.analysis.complexity* classes.

aggregate_opcodes_per_target(*target: List[py_pal.settings.Columns]*) → pandas.core.frame.DataFrame

analyze() → Tuple[str, int, str, List[str], *py_pal.analysis.complexity.Complexity*, int, pandas.core.frame.DataFrame, float]

property calls: pandas.core.frame.DataFrame

export() → pandas.core.frame.DataFrame

Export results. The output order can be controlled with *df_analyze_order*.

group_opcodes_by_call(*data: pandas.core.frame.DataFrame*, *group_by: List[py_pal.settings.Columns]*, *result_columns: List[py_pal.settings.Columns]*) → Tuple[str, int, str, List[str], pandas.core.frame.DataFrame, float]

static infer_complexity(*data_frame: pandas.core.frame.DataFrame*, *arg_column: py_pal.settings.Columns*) → *py_pal.analysis.complexity.Complexity*

Derive the big O complexity class.

Parameters

- **arg_column** (py_pal.util.Columns) – Argument column to use as x-axis.
- **data_frame** (pandas.DataFrame) – Time series-like data, x-axis is argument size, y-axis is executed opcodes.

Returns Best fitting complexity class

Return type *py_pal.analysis.complexity.Complexity*

abstract infer_complexity_per_argument(*data_frame: pandas.core.frame.DataFrame*, *arg_names: List[str]*) → Tuple[List[str], *py_pal.analysis.complexity.Complexity*, pandas.core.frame.DataFrame]

Abstract method definition for the data transformation function. The implementation prepares the dataset for evaluation with the least squares method. The dataset is transformed with respect to the arguments (their proxy value) of the function. Finally, py_pal.analysis.estimator.infer_complexity() is executed and the result is returned.

Arguments: data_frame (pandas.DataFrame): dataset arg_names (List[str]): argument names of function

Returns: pandas.DataFrame: arguments considered in the analysis, estimated complexity class and the data considered in the analysis

property iterator: Tuple[str, int, str, List[str], pandas.core.frame.DataFrame, float]

property opcodes: pandas.core.frame.DataFrame

class py_pal.analysis.estimator.**SeparateArgumentEstimator**(*call_stats: numpy.ndarray*, *opcode_stats: numpy.ndarray*, *per_line: bool = False*, *filter_function: str = False*, *filter_module: str = False*)

Bases: *py_pal.analysis.estimator.Estimator*

Voodoo *py_pal.analysis.estimator* that tries to infer complexity for each argument separately. Even though the influence of arguments on each other is minimized this may not produce reliable results and therefore should be viewed as experimental.

analyze_args_separate_ascending(*data_frame: pandas.core.frame.DataFrame*)

infer_complexity_per_argument(*data_frame: pandas.core.frame.DataFrame*, *arg_names: List[str]*)
Try to look at each argument individually and sort the argument proxy value in ascending order.

static map_arg_names(*pos*, *names: List[str]*)

static unpack_tuples(*data_frame: pandas.core.frame.DataFrame*)

4.1.2 py_pal.data_collection

py_pal.data_collection.arguments

py_pal.data_collection.metric

py_pal.data_collection.opcode_metric

py_pal.data_collection.proxy

py_pal.data_collection.tracer

4.2 py_pal.core

4.3 py_pal.datagen

class py_pal.datagen.**KeySelectionType**
Bases: object

LAST = 'last'

MIDDLE = 'middle'

NOT_INCLUDED = 'not included'

RANDOM = 'random'

py_pal.datagen.**args_growing_graphs**(*_range=range(2, 20, 2)*, *directed=True*, *p=0.5*) → List[List[dict]]
Return list of arguments of random graphs with increasing amount of nodes.

py_pal.datagen.**args_growing_graphs_with_source**(*_range=range(2, 20, 2)*) → List[List[networkx.classes.graph.Graph]]
Return list of arguments of path graphs with increasing amount of nodes and the first node as source node.

py_pal.datagen.**args_growing_lists**(*interval=range(- 10000, 10000)*, *_range=range(10, 100, 10)*, *sort=False*) → List[List[Any]]
Return lists of arguments lists of random ints with increasing length.

py_pal.datagen.**args_growing_lists_with_search_key**(*interval=range(- 10000, 10000)*, *_range=range(10, 100, 10)*, *sort=False*, *key='not included'*)
Return lists of arguments lists of random ints with increasing length and a search key.

py_pal.datagen.**args_re_growing_strings**(*expression*, *_range=range(10, 1000, 100)*, *chars='abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'*) → List[List[str]]
Return lists of argument regular expression and strings with increasing length.

py_pal.datagen.**gen_growing_ints**(*_range=range(10, 100, 10)*)
Return lists of argument ints with increasing value.

py_pal.datagen.**gen_search_args**(*interval*, *length*, *sort*, *key*)

py_pal.datagen.**integers**(*n: int*, *_min: int*, *_max: int*) → List[int]
Return sequence of N random integers between _min and _max (included).

py_pal.datagen.**large_integers**(*n: int*) → List[int]
Return sequence of N large random integers.

py_pal.datagen.**n_**(*n: int*) → int
Return N.

py_pal.datagen.**range_n**(*n: int*, *start: int = 0*) → List[int]
Return the sequence [start, start+1, . . . , start+N-1].

py_pal.datagen.**strings**(*n: int*, *chars='abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'*) → str
Return random string of N characters, sampled at random from *chars*.

4.4 py_pal.util

py_pal.util.**check_positive**(*value*)

py_pal.util.**escape**(*filename*)

py_pal.util.**get_alt_path**(*path: pathlib.Path*)

py_pal.util.**normalize_column**(*data_frame: pandas.core.frame.DataFrame*) → Tuple[pandas.core.frame.DataFrame, float]

py_pal.util.**scale_column**(*data_frame: pandas.core.frame.DataFrame*, *scale: float*)

py_pal.util.**set_log_level**(*level*)

py_pal.util.**setup_logging**(*module*, *level=30*)

CHAPTER

FIVE

CHANGELOG

5.1 What's New in Py-PAL 1.3.0

5.1.1 Command line interface changes:

- Renamed -f/–function to -ff/–filter-function
- Added -fm/–filter-module functionality to filter results by module

5.2 Py-PAL 1.2.0

- Improved the statistics and plotting output

5.2.1 Command line interface changes:

- Deprecated *–save* flag in favor of *-o/–out*
- Renamed *-V/–visualize* to *-p/–plot*
- Change functionality of *-f/–function* from executing and profiling a specific function inside a python file to applying the analysis to a selected function. Regular expressions are suported.

5.3 Py-PAL 1.1.0

- Improved Data Collection: The heuristic for determining the size of function arguments has been improved.
- More tests
- More documentation
- More argument generation functions in *py_pal.datagen*
- Replaced command line option –debug with –log-level for more configurable log output

5.3.1 Refactoring

Project structure changes, overall CLI interface is unchanged. API changes:

- *py_pal.tracer* moved to *py_pal.data_collection.tracer*
- *py_pal.complexity* and *py_pal.estimator* moved to the *py_pal.analysis* package.
- *py_pal.analysis.estimator.Estimator* now takes call and opcode stats as arguments.

5.4 Py-PAL 1.0.0

- More thorough testing from different combinations of requirements and Python versions.
- Bug fixes

5.5 Py-PAL 0.2.1

5.5.1 Refactoring

The *estimator* module was refactored which introduces a slight change to the API. Classes inheriting from *Estimator* now only specify how to transform the collected data with respect to the arguments of the function.

Instead of *ComplexityEstimator* you should use the *AllArgumentEstimator* class. Additionally there is the *SeparateArgumentEstimator* which is experimental.

5.6 Py-PAL 0.1.6

5.6.1 More accurate Data Collection

The *Tracer* is enhanced by measuring builtin function calls with *AdvancedOpcodeMetric*.

Opcodes resembling a function call .e.g *FUNCTION_CALL* are filtered for built in function calls. If the called function is found in the complexity mapping a synthetic Opcode weight gets assigned. A builtin function call is evaluated using its argument and a pre-defined runtime complexity e.g. O(n log n) for *sort()*.

- The feature is enabled by default
- The calculation produces a performance overhead and can be disabled by providing a *OpcodeMetric* instance to the *Tracer*
- The *AdvancedOpcodeMetric* instance assigned to the *Tracer* provides statistics about how many builtin function calls were observed and how many were found in the complexity map

5.6.2 Bugfixes

- Cleaning data after normalization introduced wrong data points

PYTHON MODULE INDEX

p

py_pal.analysis.complexity, 9
py_pal.analysis.estimator, 11
py_pal.datagen, 13
py_pal.util, 14

INDEX

A

aggregate_opcodes_per_target() (*py_pal.analysis.estimator.Estimator method*), 12
AllArgumentEstimator (*class in py_pal.analysis.estimator*), 11
analyze() (*py_pal.analysis.estimator.Estimator method*), 12
analyze_args_separate_ascending() (*py_pal.analysis.estimator.SeparateArgumentEstimator method*), 13
args_growing_graphs() (*in module py_pal.datagen*), 13
args_growing_graphs_with_source() (*in module py_pal.datagen*), 14
args_growing_lists() (*in module py_pal.datagen*), 14
args_growing_lists_with_search_key() (*in module py_pal.datagen*), 14
args_re_growing_strings() (*in module py_pal.datagen*), 14
ArgumentDataSelectionStrategy (*class in py_pal.analysis.estimator*), 12

C

calls (*py_pal.analysis.estimator.Estimator property*), 12
check_positive() (*in module py_pal.util*), 14
Complexity (*class in py_pal.analysis.complexity*), 9
compute() (*py_pal.analysis.complexity.Complexity method*), 9
Constant (*class in py_pal.analysis.complexity*), 9
Cubic (*class in py_pal.analysis.complexity*), 10

E

escape() (*in module py_pal.util*), 14
Estimator (*class in py_pal.analysis.estimator*), 12
Exponential (*class in py_pal.analysis.complexity*), 10
export() (*py_pal.analysis.estimator.Estimator method*), 12

F

fit() (*py_pal.analysis.complexity.Complexity method*), 9
format_str() (*py_pal.analysis.complexity.Complexity class method*), 9
format_str() (*py_pal.analysis.complexity.Constant class method*), 9
format_str() (*py_pal.analysis.complexity.Cubic class method*), 10
format_str() (*py_pal.analysis.complexity.Exponential class method*), 10
format_str() (*py_pal.analysis.complexity.Linear class method*), 10
format_str() (*py_pal.analysis.complexity.Linearithmic class method*), 10
format_str() (*py_pal.analysis.complexity.Logarithmic class method*), 10
format_str() (*py_pal.analysis.complexity.Polynomial class method*), 11
format_str() (*py_pal.analysis.complexity.Quadratic class method*), 11

G

gen_growing_ints() (*in module py_pal.datagen*), 14
gen_search_args() (*in module py_pal.datagen*), 14
get_alt_path() (*in module py_pal.util*), 14
group_opcodes_by_call() (*py_pal.analysis.estimator.Estimator method*), 12

I

infer_complexity() (*py_pal.analysis.estimator.Estimator static method*), 12
infer_complexity_per_argument() (*py_pal.analysis.estimator.AllArgumentEstimator method*), 11
infer_complexity_per_argument() (*py_pal.analysis.estimator.Estimator method*), 12
infer_complexity_per_argument() (*py_pal.analysis.estimator.SeparateArgumentEstimator method*), 13
integers() (*in module py_pal.datagen*), 14
iterator (*py_pal.analysis.estimator.Estimator property*), 13

K

KeySelectionType (*class in py_pal.datagen*), 13

L

large_integers() (*in module py_pal.datagen*), 14
LAST (*py_pal.datagen.KeySelectionType attribute*), 13
Linear (*class in py_pal.analysis.complexity*), 10
Linearithmic (*class in py_pal.analysis.complexity*), 10
Logarithmic (*class in py_pal.analysis.complexity*), 10

M

map_arg_names() (*py_pal.analysis.estimator.SeparateArgumentEstimator static method*), 13
MAX (*py_pal.analysis.estimator.ArgumentDataSelectionStrategy attribute*), 12
MEAN (*py_pal.analysis.estimator.ArgumentDataSelectionStrategy attribute*), 12
MIDDLE (*py_pal.datagen.KeySelectionType attribute*), 13
MIN (*py_pal.analysis.estimator.ArgumentDataSelectionStrategy attribute*), 12
module
    py_pal.analysis.complexity, 9
    py_pal.analysis.estimator, 11
    py_pal.datagen, 13
    py_pal.util, 14

N

n_() (*in module py_pal.datagen*), 14
normalize_column() (*in module py_pal.util*), 14
NOT_INCLUDED (*py_pal.datagen.KeySelectionType attribute*), 13
NotFittedError, 11

O

opcodes (*py_pal.analysis.estimator.Estimator property*), 13

P

Polynomial (*class in py_pal.analysis.complexity*), 11
py_pal.analysis.complexity
    module, 9
py_pal.analysis.estimator
    module, 11
py_pal.datagen
    module, 13
py_pal.util
    module, 14

Q

Quadratic (*class in py_pal.analysis.complexity*), 11

R

RANDOM (*py_pal.datagen.KeySelectionType attribute*), 13
range_n() (*in module py_pal.datagen*), 14

S

scale_column() (*in module py_pal.util*), 14
SeparateArgumentEstimator (*class in py_pal.analysis.estimator*), 13
set_log_level() (*in module py_pal.util*), 14
setup_logging() (*in module py_pal.util*), 14
strings() (*in module py_pal.datagen*), 14

T

transform_n() (*py_pal.analysis.complexity.Complexity method*), 9
transform_n() (*py_pal.analysis.complexity.Constant method*), 10
transform_n() (*py_pal.analysis.complexity.Cubic method*), 10
transform_n() (*py_pal.analysis.complexity.Exponential method*), 10
transform_n() (*py_pal.analysis.complexity.Linear method*), 10
transform_n() (*py_pal.analysis.complexity.Linearithmic method*), 10
transform_n() (*py_pal.analysis.complexity.Logarithmic method*), 11
transform_n() (*py_pal.analysis.complexity.Polynomial method*), 11
transform_n() (*py_pal.analysis.complexity.Quadratic method*), 11
transform_y() (*py_pal.analysis.complexity.Complexity method*), 9
transform_y() (*py_pal.analysis.complexity.Exponential method*), 10
transform_y() (*py_pal.analysis.complexity.Polynomial method*), 11

U

UnderDeterminedEquation, 11
unpack_tuples() (*py_pal.analysis.estimator.SeparateArgumentEstimator static method*), 13