benchit Documentation

Release 0.0.6

Divakar Roy

Jun 04, 2021

Contents

Note: This package is under active development. API changes are very likely.

CHAPTER 1

Installation

Latest PyPI stable release (alongwith dependencies) :

```
pip install benchit
```

Pull latest development release on GitHub and install in the current directory :

```
pip install -e git+https://github.com/droyed/benchit.git@master#egg=benchit
```

CHAPTER 2

Dependencies

- cpuinfo
- matplotlib
- numpy
- pandas
- psutil
- tqdm
- ipython

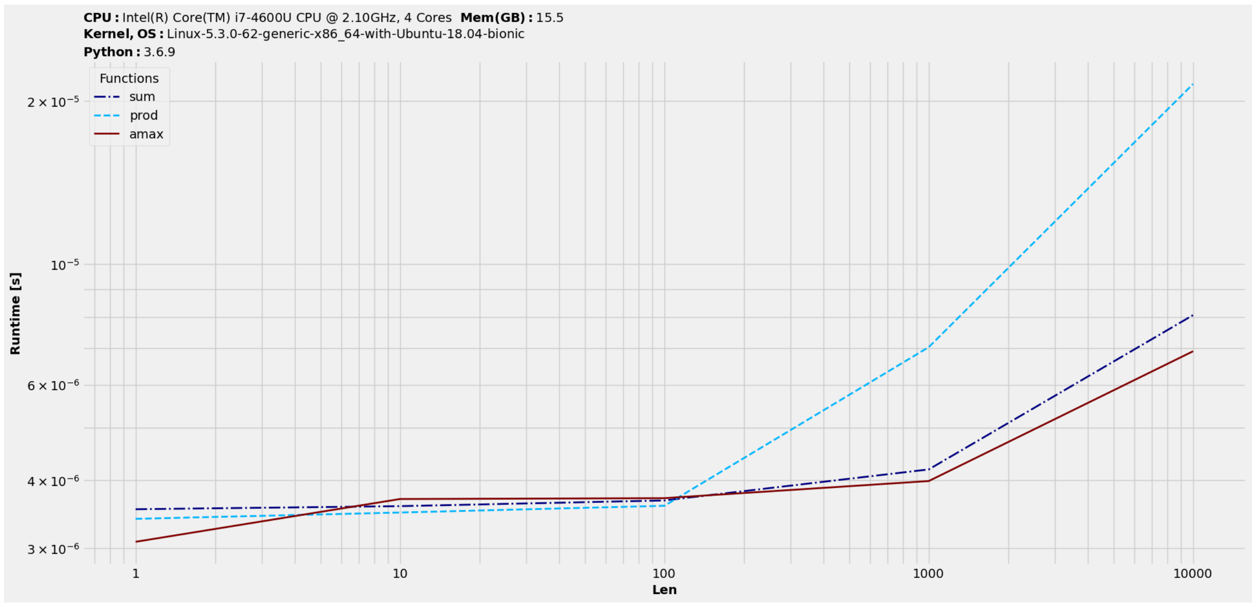
CHAPTER 3

Quick start

Let's benchmark NumPy ufuncs - sum, prod, max on *1D* arrays -

```
# Setup input functions and datasets
>>> import numpy as np
>>> funcs = [np.sum,np.prod,np.max]
>>> inputs = [np.random.rand(i) for i in 10**np.arange(5)]

# Benchmark and plot
>>> import benchit
>>> t = benchit.timings(funcs, inputs)
>>> t.plot(logy=True, logx=True, save='index_timings.png')
```



Though these perform entirely different operations, it was meant to showcase a basic usage. For a detailed explanation on the usage and more realistic scenarios, jump over to - *Benchmarking steps*.

CHAPTER 4

Benchit - Contents

4.1 Introduction

We often end up with more than one way to solve a problem and at times we need to compare them based on certain criteria, which could be memory-efficiency or performance. Comparative analysis is an essential process to evaluate different methods on those criteria. Usually the problem setup involves various datasets that in some way represent various possible intended use-cases. Such a problem setup helps us present an in-depth analysis of the available methods across those cases. Please note that with this package, we are solely focusing on benchmarking pertaining to Python.

4.1.1 Relevant scenarios

Many times we use different Python modules to solve a problem. Python modules like NumPy, Numba, SciPy, etc. are built on different philosophies and hence fair differently on different datasets. Often one of the requirements is runtime performance when evaluating solutions with them or even with Vanilla Python. With this package, we are primarily focusing on evaluating runtime performance with different methods across different datasets.

The benchmarking process should cover all Python supported data, but the main motivation with this package has been to perform benchmarking on NumPy ndarrays, Pandas dataframe, Python lists and scalars.

4.2 Benchmarking steps

A minimal workflow employing this package would basically involve three steps :

- Setup functions : A list or dictionary of functions to be benchmarked. It supports both single and multiple arguments.
- Setup datasets : A list or dictionary of datasets to be benchmarked.
- Benchmark to get timings in a dataframe-like object. Each row holds one dataset and each header represents one function each. Dataframe has been the design choice, as it supports plotting directly from it and additionally benchmarking setup information could be stored as name values for index and columns.

We will study these with the help of a sample setup in *Minimal workflow*.

We will study about setting up functions and datasets in detail later in this document.

Note: Prior to Python 3.6, dictionary keys are not maintained in the order they are inserted. So, when working with those versions and with input dataset being defined as a dictionary, to keep the order, collections.OrderedDict could be used.

To get more out of it, we could optionally do the following :

- Plot the timings.
- Get speedups or scaled-timings of all functions with respect to one among them.
- Rank the functions based on various performance-metrics.

A detailed study with examples in the next section should clear up things.

We will try to take a hands-on approach and explore the features available with this package. We will start off with the minimal steps to benchmarking on a setup and then explore other utilities to cover most common features.

Rest of the documentation will use the module's methods. So, let's import it once :

```
>>> import benchit
```

4.2.1 Minimal workflow

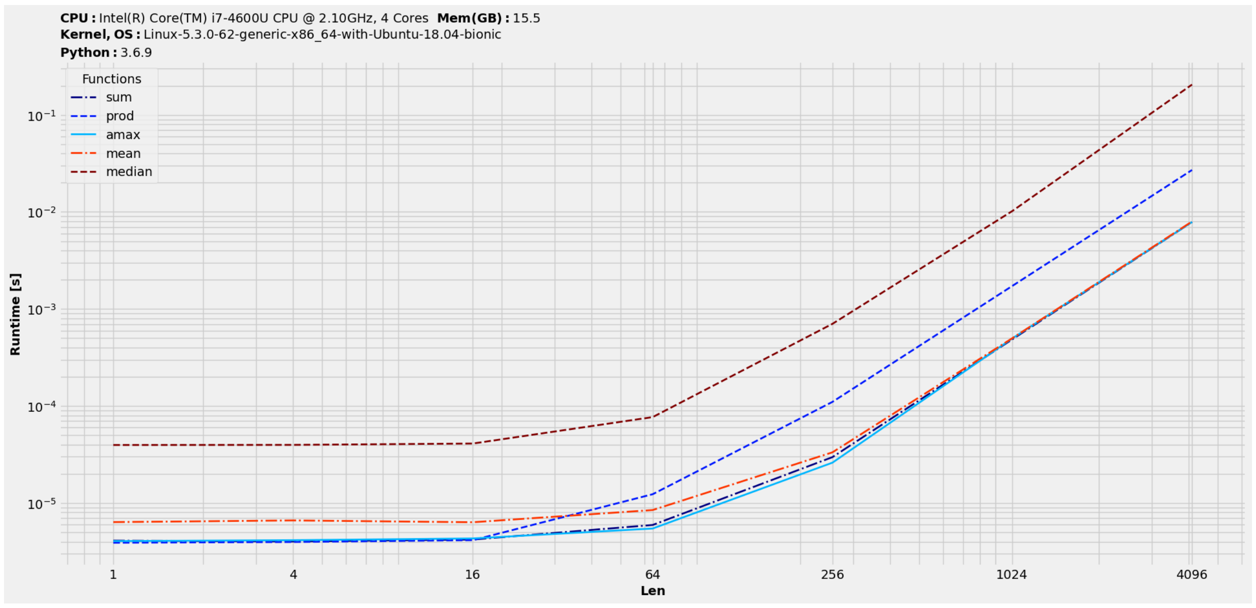
We will study a case of single argument with default parameters. Let's take a sample case where we try to benchmark the five most common NumPy ufuncs - sum, prod, max, mean, median on arrays varying in their sizes. To keep it simple, let's consider *1D* arrays. Thus, the benchmarking steps would look something like this :

```
>>> import numpy as np
>>> funcs = [np.sum,np.prod,np.max,np.mean,np.median]
>>> inputs = [np.random.rand(i,i) for i in 4**np.arange(7)]
>>> t = benchit.timings(funcs, inputs)
>>> t
Functions        sum      prod      amax      mean    median
Len
1           0.000005  0.000004  0.000005  0.000007  0.000046
4           0.000005  0.000004  0.000005  0.000007  0.000047
16          0.000005  0.000005  0.000005  0.000007  0.000049
64          0.000007  0.000014  0.000007  0.000009  0.000094
256         0.000035  0.000131  0.000030  0.000038  0.000845
1024        0.000511  0.002050  0.000512  0.000522  0.011525
4096        0.008208  0.032582  0.008257  0.008274  0.261838
```

It's a *dataframe-like* object, called *BenchmarkObj*. We can plot it, which automatically adds in system configuration into the title area to convey all the available benchmarking information :

```
>>> t.plot(logy=True, logx=True, save='timings.png')
```

Resultant plot would look something like this :



These *4* lines of codes would be enough for most of the benchmarking workflows.

Extract dataframe & construct back

The underlying benchmarking data is stored as a pandas dataframe that could be extracted with :

```
>>> df = t.to_dataframe()
```

As we shall see in the next sections, this would be useful in our benchmarking quest to extend the capabilities.

There's a benchmarking object construct function *benchit.bench* that accepts dataframe alongwith *dtype*. So, we can construct it, like so :

```
>>> t = benchit.bench(df, ...)
```

4.2.2 Setup functions

This would be a list or dictionary of functions to be benchmarked.

A general syntax for list version would look something like this :

```
>>> funcs = [func1, func2, ...]
```

We already saw a sample of it in *Minimal workflow*.

A general syntax for dictionary version would look something like this :

```
>>> funcs = {'func1_name':func1, 'func2_name':func2, ...}
```

Mixing in lambdas

Lambda functions could also be mixed into our functions for benchmarking with a dictionary. So, the general syntax would be :

```
>>> funcs = {'func1_name':func1, 'lambda1_name':lamda1, 'func2_name':func2, ...}
```

This is useful for directly incorporating one-liner solutions without the need of defining them beforehand.

Let's take a sample setup where we will tile a *1D* array twice with various solutions as lambda and regular functions mixed in :

```
import numpy as np

def numpy_concat(a):
    return np.concatenate([a, a])

# We need a dictonary to give each lambda an unique name, through keys
funcs = {'r_':lambda a:np.r_[a, a],
         'stack+reshape':lambda a:np.stack([a, a]).reshape(-1),
         'hstack':lambda a:np.hstack([a, a]),
         'concat':numpy_concat,
         'tile':lambda a:np.tile(a,2)}
```

4.2.3 Setup datasets

This would be a list or dictionary of datasets to be benchmarked.

A general syntax for list version would look something like this :

```
>>> in_ = [dataset1, dataset2, ...]
```

For such list type *inputs*, based on the datasets and additional argument *indexby* to *benchit.timings*, each dataset is assigned an *index*.

A general syntax for dictionary version would look something like this :

```
>>> in_ = {'argument_value1':dataset1, 'argument_value2':dataset2, ...}
```

For such dictionary type *inputs*, index values would be the dictionary keys.

For both lists and dicts, these index values are used for plotting, etc. With single argument cases, this is pretty straightforward.

Now, we might have functions that accept more than one argument, let's call those as *multivar* cases and focus on those. Please keep in mind that for those *multivar* cases, we need to feed in *multivar=True* into *benchit.timings*.

Pseudo code would look something like this :

```
>>> in_ = {m:generate_inputs(m,k1,k2) for m in m_list} # k1, k2 are constants
>>> t = benchit.timings(fncs, in_, multivar=True, input_name='arg0')
```

Groupings

Groupings are applicable for both single and multiple variable cases.

There are essentially two rules to form groupings :

- Use dictionary as *inputs*.
- Use a nested loop structure to form the input datasets with tuples of input parameters as the dictionary keys. These keys could be derived from the input arguments or otherwise. Essentially, we would have two or more sources of forming that input argument(s) and those are to be listed as the keys.

Thus, considering two sources, a general structure would be :

```
>>> in_ = {(source1_value1, source2_value1): dataset1,
           (source1_value2, source2_value2): dataset2, ...}
```

As stated earlier, with multiple arguments case, as the most common scenario, we would have the input arguments put in as the key elements. Thus, with functions that accept two arguments, it would be :

```
>>> in_ = {(argument1_value1, argument2_value1): dataset1,
           (argument1_value2, argument2_value2): dataset2, ...}
```

Example :

Let's take a complete example to understand groupings :

```
>>> in_ = {(argument1_value1, argument2_value1): dataset1,
           (argument1_value1, argument2_value2): dataset2,
           (argument1_value1, argument2_value3): dataset3,
           (argument1_value2, argument2_value1): dataset4,
           (argument1_value2, argument2_value2): dataset5,
           (argument1_value2, argument2_value3): dataset6, ...}
```

Thus,

- Considering *argument1* values as reference, we would have *2* groups - *(dataset1, 2, 3)* and *(dataset4, 5, 6)*.
- Considering *argument2* values as reference, we would have *3* groups - *(dataset1, 4)*, *(dataset2, 5)* and *(dataset3, 6)*.

Optionally, to finalize the groupings with proper names, we can assign names to each argument with *input_name* argument to *benchit.timings*. So, *input_name* would be a list or tuple specifying the names for each argument as its elements as strings. These would be picked up for labelling purpose when plotting.

Thus, a complete pseudo code to form groupings with a two-level nested loop would look something like this :

```
>>> in_ = {(m,n):generate_inputs(m,n) for m in m_list for n in n_list}
>>> t = benchit.timings(fncs, in_, input_name=['arg0', 'arg1'])
```

Plots on groupings would result in subplots. More on this with examples is shown later in this document. Note that we can have a *n-level* nested loop structure and the subplots would take care of the plotting.

4.3 Features

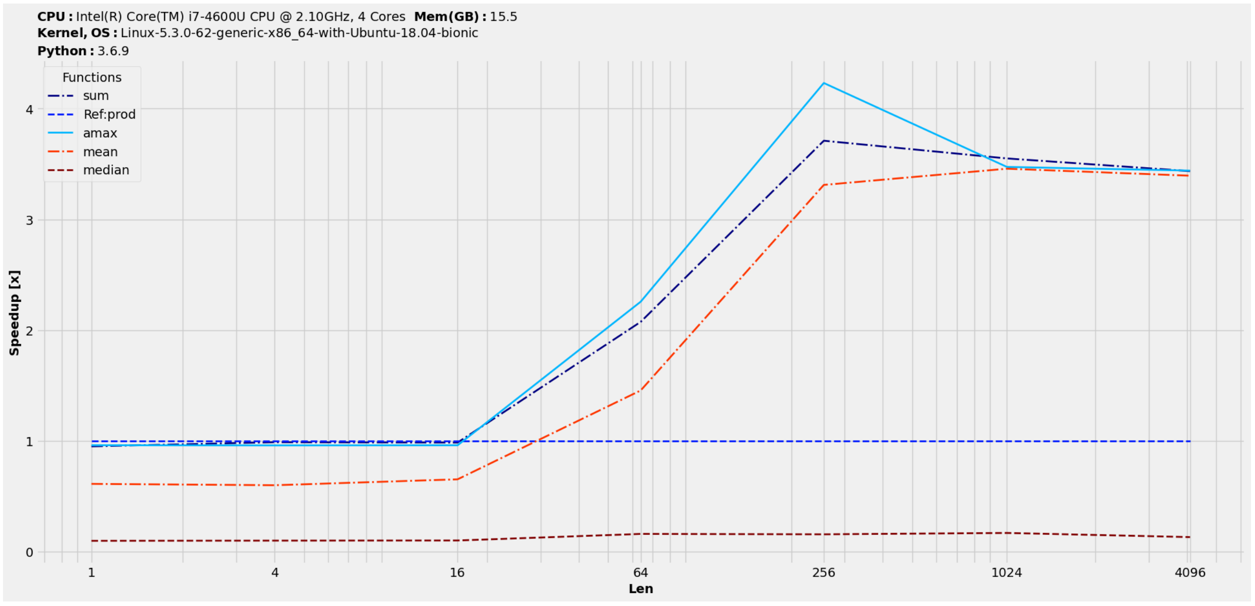
Apart from getting the timings, we can further process the underlying data to study various aspects of the benchmarking.

4.3.1 Speedups & scaled-timings

Benchmarking results could be stored as two more datatypes with *BenchmarkObj*, namely *speedups* and *scaled_timings* (timings numbers simply scaled by one reference function). So, these alongwith the entry datatype of *timings* form the essential three datatypes of this package. All asssociated class methods and utility functions revolve around them.

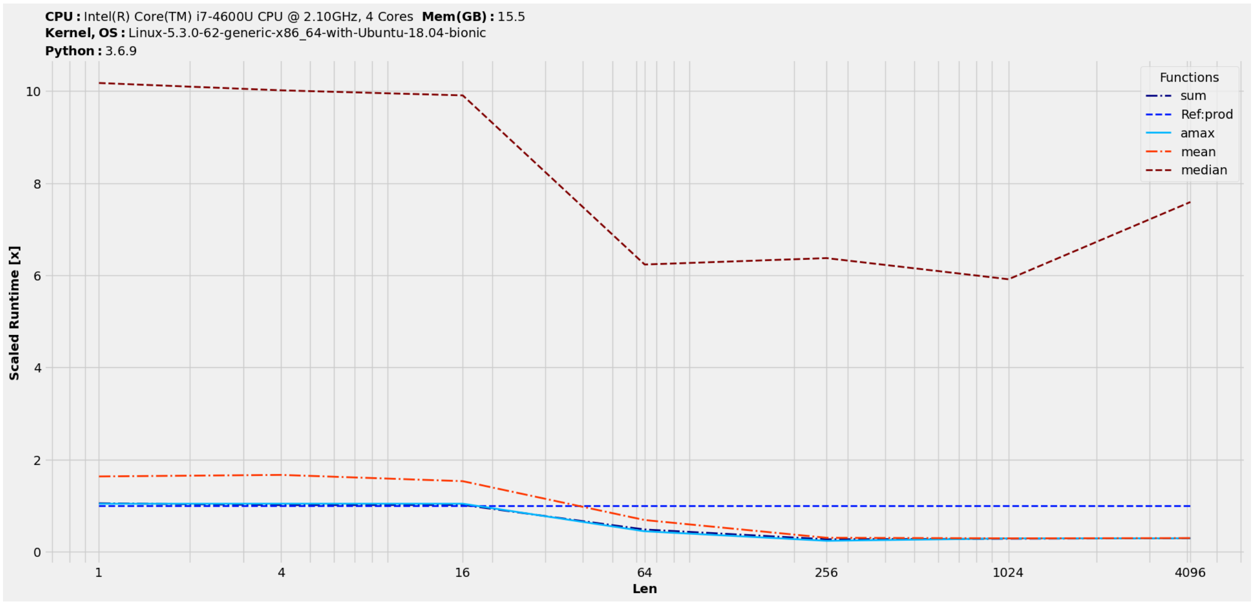
Let's study the speedups of all w.r.t *prod* alongwith ranking :

```
>>> s = t.speedups(ref=1) # prod's location index in t is 1
>>> s.plot(logy=False, logx=True, save='speedups_by_prod.png')
```



Finally, the scaled-timings :

```
>>> st = t.scaled_timings(ref=1) # prod's location index in t is 1
>>> st.plot(logy=False, logx=True, save='scaledtimings_by_prod.png')
```



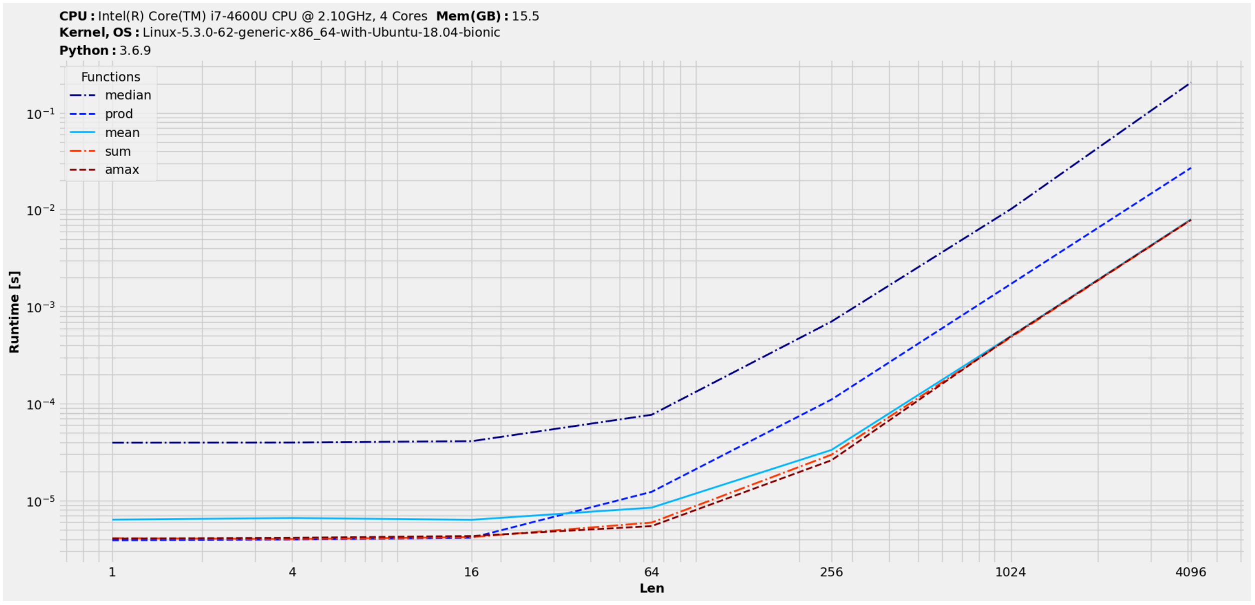
The input argument to methods *speedups* and *scaled_timings* i.e. *ref* accepts three types of arguments for indexing - *int* as the location index, *str* as the function name string and *function* itself that was input into *funcs*.

Let's explore the other available tools with this package. As mentioned earlier, all of these are applicable to all the three datatypes with *BenchmarkObj*. We will re-use the numbers obtained with the *Minimal workflow* discussed earlier.

4.3.2 Rank & reset_columns

The order of the plot legend by default would be same as the order in *funcs* argument. With many competing solutions in *funcs*, for an easy correlation between them and the plot lines, we might want to sort the legend based on their performance and hence comes the idea of ranking. There are various criteria on which performance could be ranked. Let's explore the usage with the default one :

```
>>> t.rank()
>>> t.plot(logy=True, logx=True, save='timings_ranked.png')
```



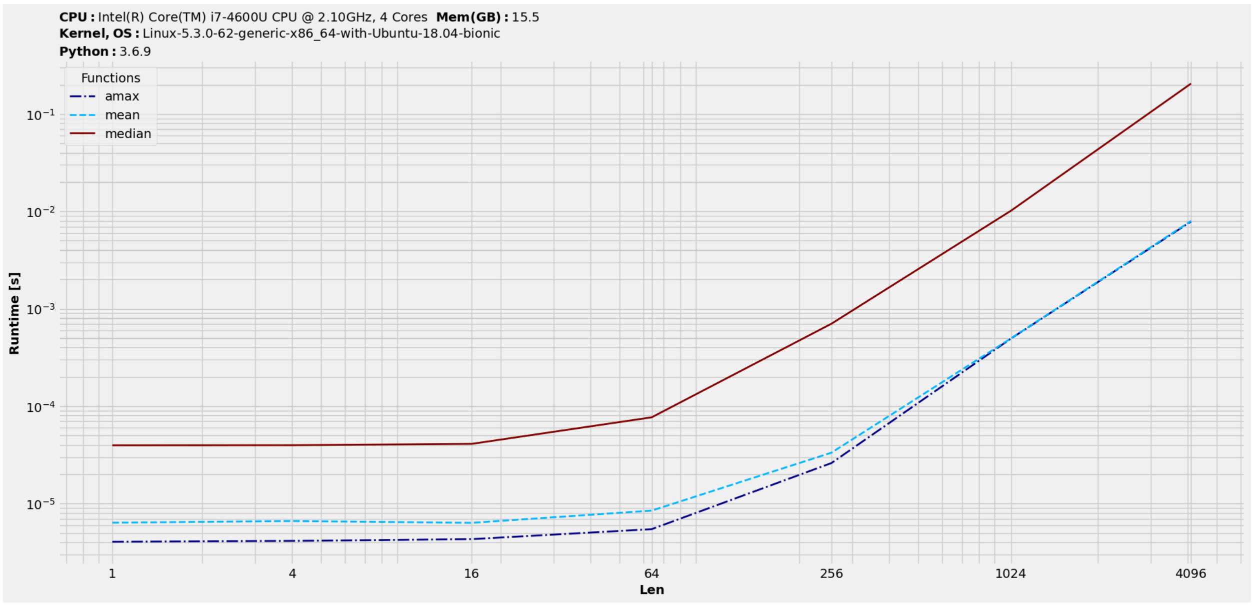
Note that ranking would have changed the columns order. To revert to the original order, use :

```
>>> t.reset_columns()
```

4.3.3 Drop

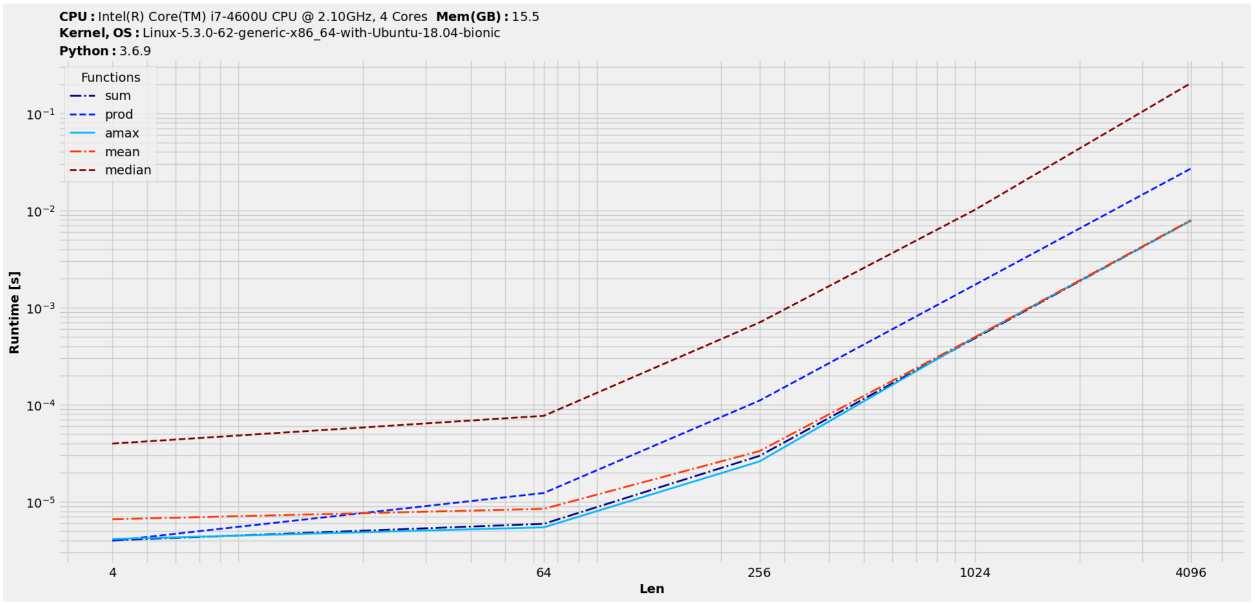
Let's say we might want to focus on few functions and hence drop the others or even drop some input datasets. This method does the job, as we can drop by the column and index values. Note that this works in-place. So, if we want to drop *median* and *prod*, it would be :

```
>>> t.drop(['sum', 'prod'], axis=1)
>>> t.plot(logy=True, logx=True, save='timings_dropfuncs.png')
```



To drop certain datasets (starting with original *t*) :

```
>>> t.drop([1,16], axis=0)
>>> t.plot(logy=True, logx=True, save='timings_dropdata.png')
```



4.3.4 Copy

As the name suggests, we can make a copy of the benchmarking object with it. It should be useful when we are trying out stuffs and need a backup of benchmarking results.

4.4 Expose to pandas-world

Earlier we saw how we can go back and forth between *benchit.BenchmarkObj* and *pandas.DataFrame*. We also studied how that could be used to extend plot functionality. In this section, let's study how we can extend it to manipulate benchmarking results. We will continue with the hands-on method of explanation.

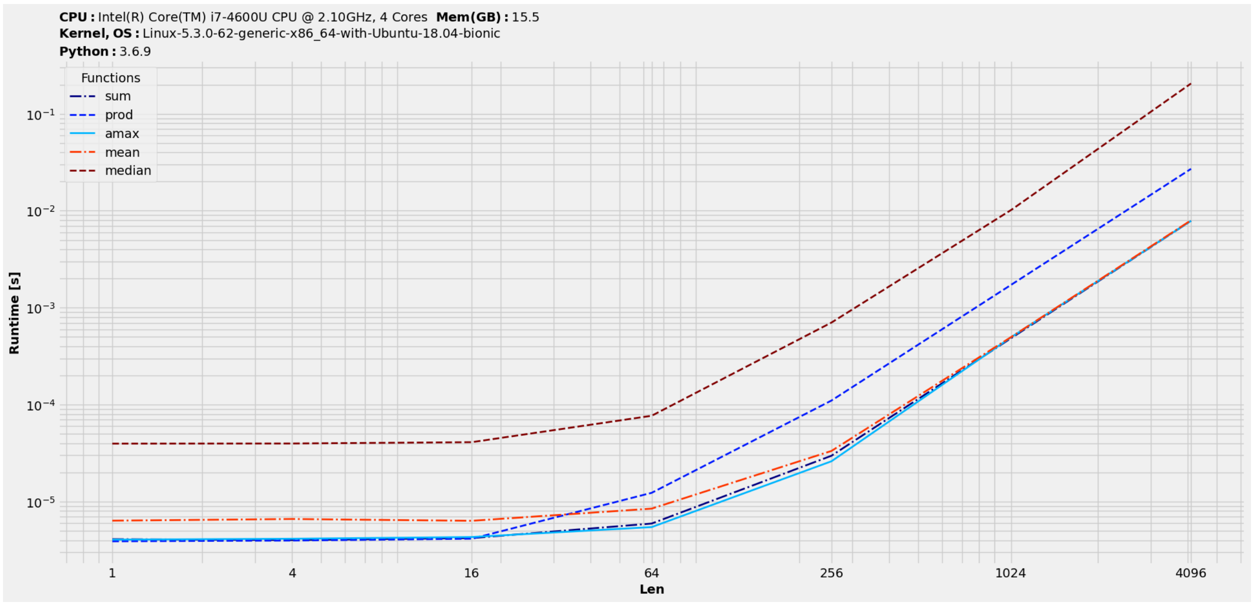
4.4.1 General syntax

For a given benchmarking object *t*, the general syntax on working with the underlying dataframe would be something like this :

```
>>> df = t.to_dataframe()
>>> df_new = # some operation on df to result in a new dataframe, df_new
>>> benchit.bench(df_new, dtype=t.dtype)
```

4.4.2 Examples

We will take over from *Minimal workflow* with the *timings* plot and look at few cases. For reference, the timings plot looked something like this :



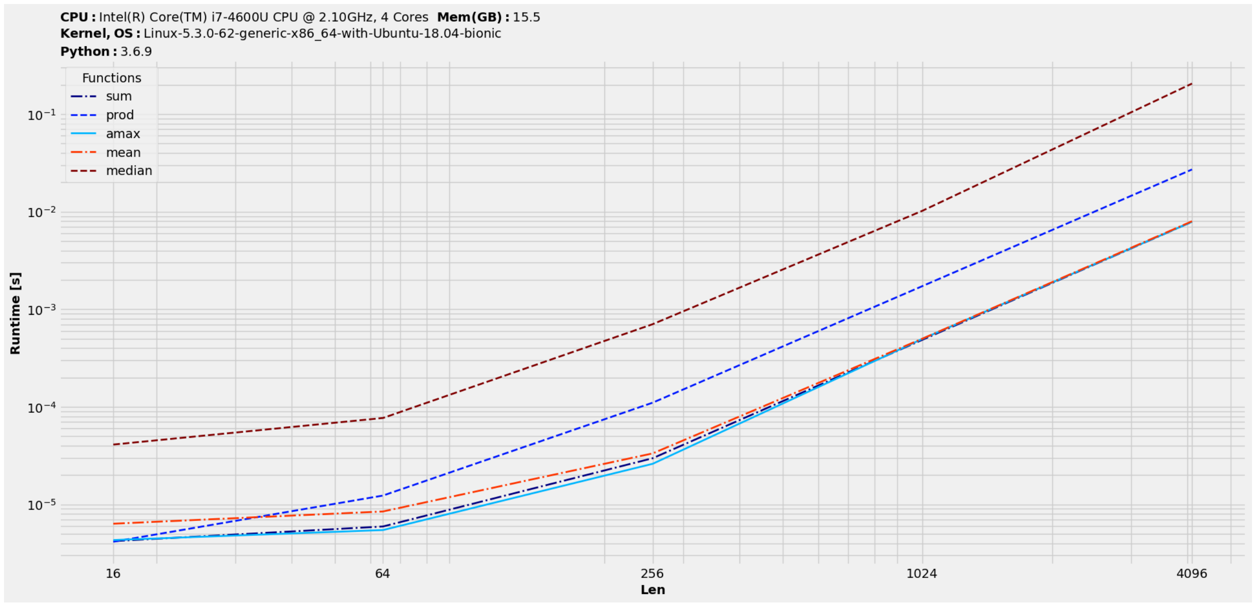
Crop

As an example, just to emphasize on the ease to do this *business*, a typical way of dropping the first two datasets would be :

```
>>> benchit.bench(t.to_dataframe().iloc[2:],dtype=t.dtype)
```

Default *dtype* argument for *benchit.bench* is set for *timings*. So, it becomes simpler with :

```
>>> benchit.bench(t.to_dataframe().iloc[2:]).plot(logx=True, save='timings_cropdata.
↪png')
```

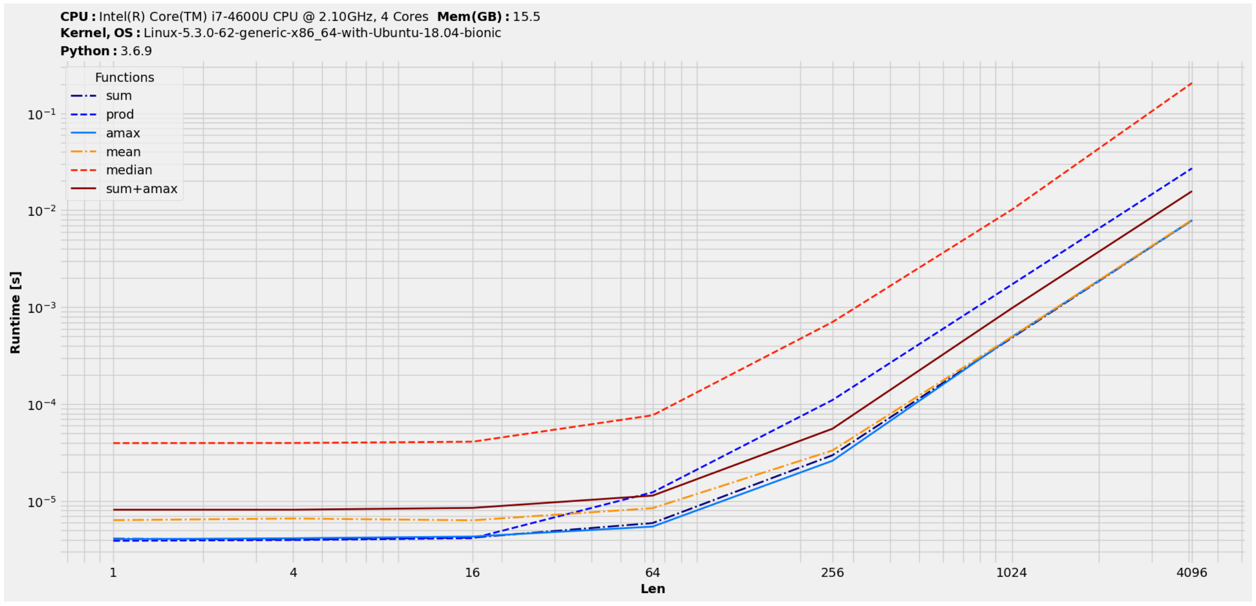


Combine

Back to the same *Minimal benchmarking workflow*, let's say we want to see the combined timings for two functions and how would it fare against other individual functions. *Dataframe* format makes it easy :

```
# Create a new column with combined data
>>> df['sum+amax'] = df['sum'] + df['amax']

# Create a new function-column with combined data and plot
>>> benchit.bench(df).plot(logx=True, save='timings_comb.png')
```



At least one interesting observation could be made there. If we compare combined one of *sum & max* against *prod*, the former wins on lower timings only with larger datasets.

Earlier listed *Drop* is based on this strategy of working with the inherent dataframe data. There are endless possibilities and scenarios where having a dataframe data could be useful and necessary!

4.5 Real-world examples

We will take few realistic scenarios and also study how arguments could be setup differently.

4.5.1 Multiple arg

Adding arrays

We will study a multiple argument case. This was inspired by a Stack Overflow question on adding two arrays. We will study the case of functions that accept two arguments. The two functions in consideration are :

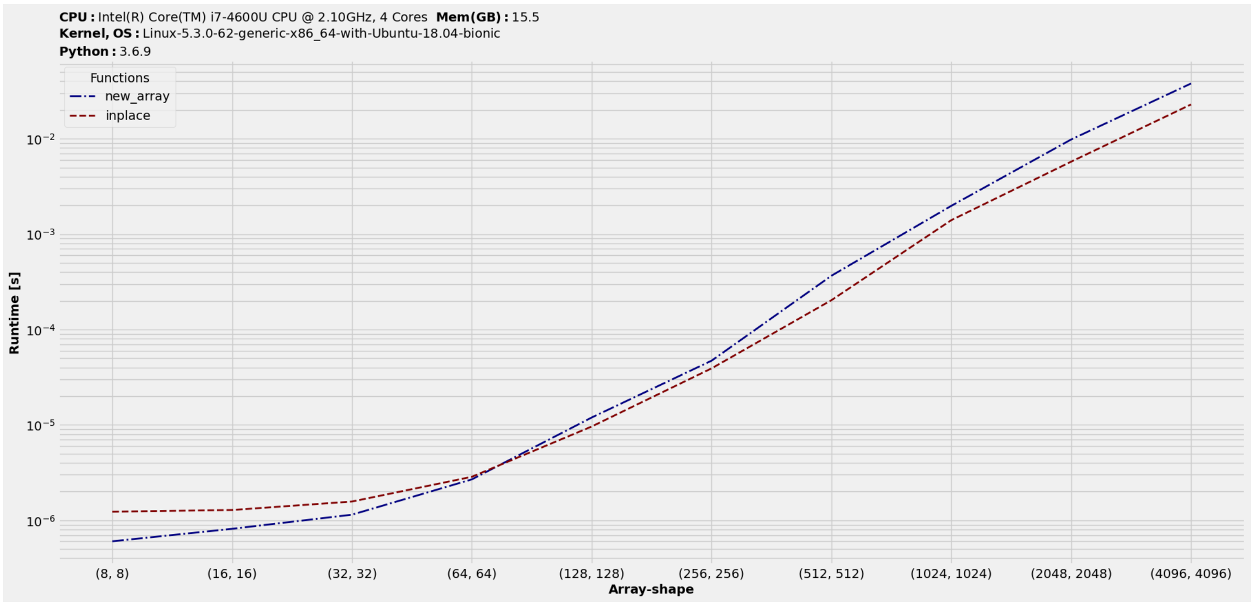
```
def new_array(a1, a2):
    a1 = a1 + a2

def inplace(a1, a2):
    a1 += a2
```

These accept NumPy array data and thus would perform those summations and write-back in a vectorized way. The first one does summation stores in temporary buffers and then pushes back the result to a new memory space, while the second method writes back the addition result to first array's memory space. We are investigating, which one's better and by how much. Let's put them to the test using our tools!

Now, as mentioned earlier, for multiple argument cases, we need to feed in each of those input datasets as a tuple each. We could setup the inputs as a list. But, let's setup in a dictionary, so that datasets are assigned labels with its keys. Let's get the timings and hence plot them :

```
>>> R = np.random.rand
>>> inputs = {(i,i):(R(i,i),R(i,i)) for i in 2**np.arange(3,13)}
>>> t = benchit.timings([new_array,inplace], inputs, multivar=True, input_name='Array-
↪shape')
>>> t.plot(logy=True, logx=False, save='multivar_addarrays_timings.png')
```



Looking at the plot, we can conclude that the *write-back* one is better for larger arrays, which makes sense given its memory efficiency.

Euclidean distance

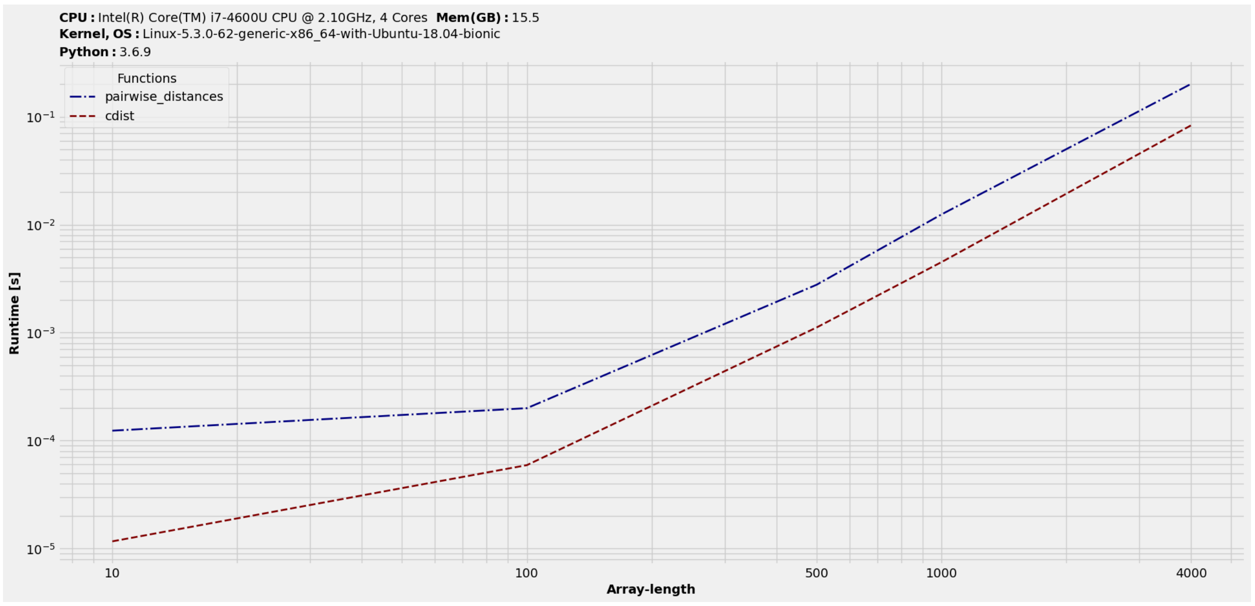
Single variable

We will study another multiple argument case. The setup involves euclidean distances between two *2D* arrays. We will feed in arrays with varying number of rows and 3 columns to represent data in 3D Cartesian coordinate system and benchmark two commonly used functions in Python.

```
# Setup input functions
>>> from sklearn.metrics.pairwise import pairwise_distances
>>> from scipy.spatial.distance import cdist
>>> fns = [pairwise_distances, cdist]

# Setup input datasets
>>> import numpy as np
>>> in_ = {n:[np.random.rand(n,3), np.random.rand(n,3)] for n in [10,100,500,1000,
↪4000]}

# Get benchmarking object (dataframe-like) and plot results
>>> t = benchit.timings(fns, in_, multivar=True, input_name='Array-length')
>>> t.plot(save='multivar_euclidean_timings.png')
```



Multi-variable Groupings

We will simply extend previous test-case to cover for the second argument to the distance functions, i.e. with varying number of columns. We will re-use most of that earlier setup.

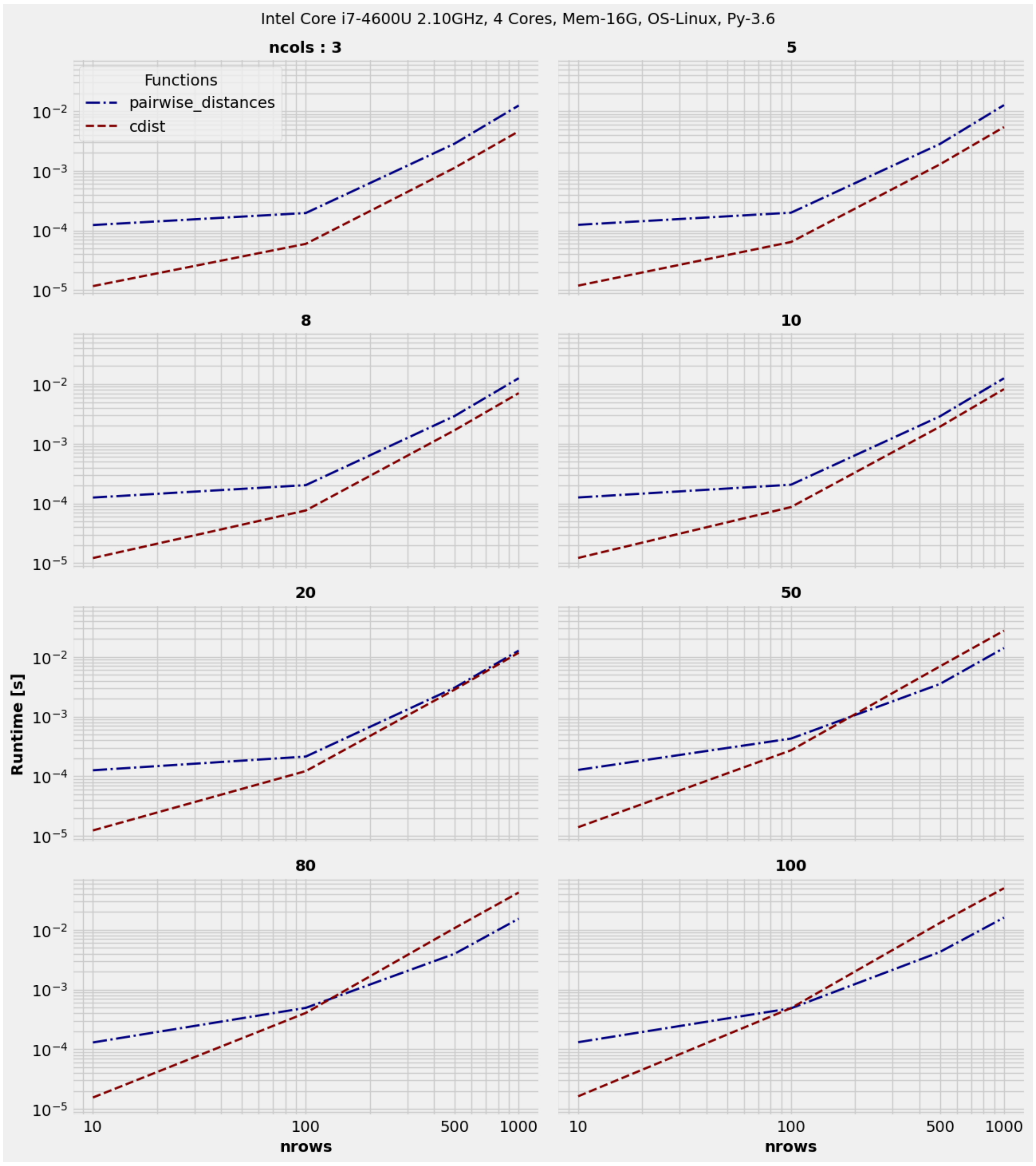
Also, we will explore subplot specific arguments available with *plot*. These are marked with prefix as : *sp_*, short for *subplot_*.

```
>>> R = np.random.rand
>>> nrows_list = [10, 100, 500, 1000]  # list of number of rows
>>> ncols_list = [3, 5, 8, 10, 20, 50, 80, 100]  # list of number of cols
>>> in_ = {(nr,nc):[R(nr,nc), R(nr,nc)] for nr in nrows_list for nc in ncols_list}
>>> t = benchit.timings(fns, in_, multivar=True, input_name=['nrows', 'ncols'])
```

Now, let's do the groupings to study the behaviour w.r.t. to each argument.

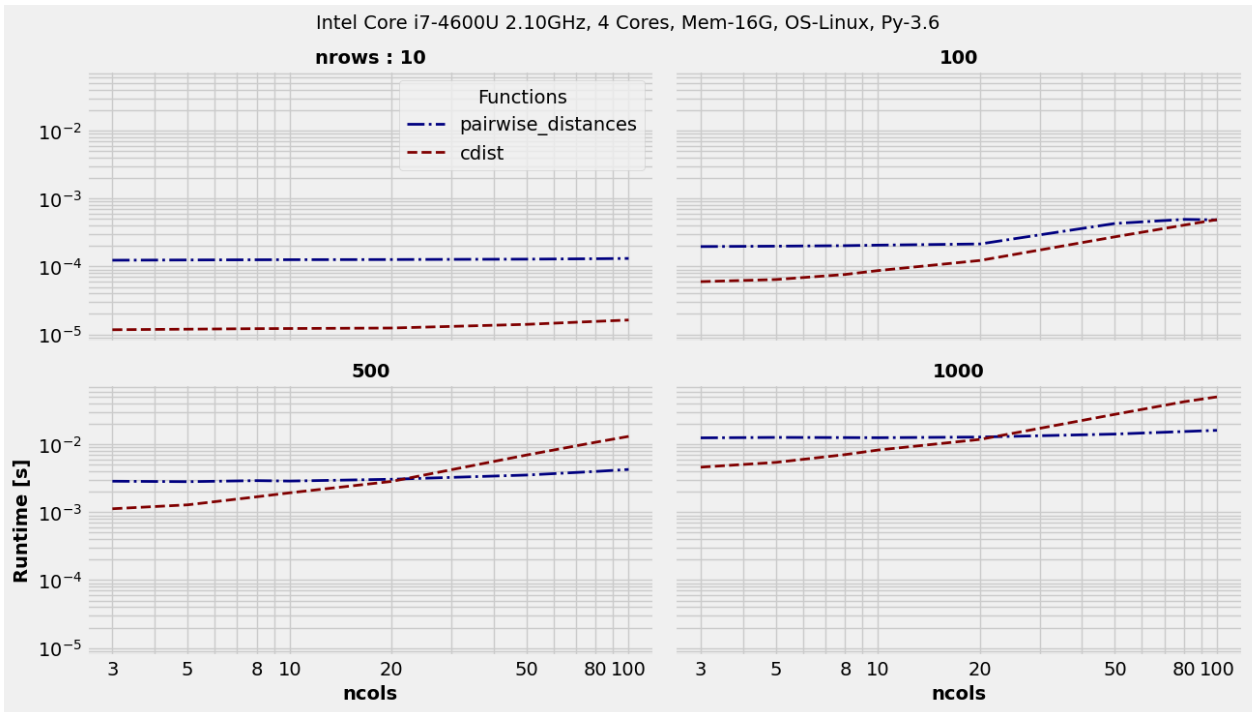
Grouping based on *argID = 0* :

```
>>> t.plot(logx=True, sp_ncols=2, sp_argID=0, sp_sharey='g', save='multigrp_id0_
↪euclidean_timings.png')
```



Grouping based on *argID = 1* :

```
>>> t.plot(logx=True, sp_ncols=2, sp_argID=1, sp_sharey='g', save='multigrp_id1_
↪euclidean_timings.png')
```



Some interesting obseravtions could be made there. The implementations are obviously different. This is resulting in *pairwise_distances* winning as we move to higher number of columns. Though, on smaller datasets or with smaller number of rows, *cdist* is clearly ahead.

4.5.2 Single arg

Forward-fill on mask

Single-variable Groupings

Let's manufacture a simple forward-filling scheme based on indices of *True* values in a boolean-array, whereas *False* should keep previous values. Also, the values would be kept as *0s* until the first *True*.

To give it a better understanding, two examples should clarify with a solution based on *np.maximum.accumulate* :

```
>>> b
array([ True, False,  True,  True, False, False, False,  True, False, False])
>>> np.maximum.accumulate(np.where(b,np.arange(len(b)), 0))
array([0, 0, 2, 3, 3, 3, 3, 7, 7, 7])

>>> b
array([False, False, False,  True, False, False, False,  True, False, False])
>>> np.maximum.accumulate(np.where(b,np.arange(len(b)), 0))
array([0, 0, 0, 3, 3, 3, 3, 7, 7, 7])
```

We could also solve it with *np.repeat* based on the counts between consecutive *True* ones. Let's benchmark these two methods :

```
# Functions
def repeat(b):
    idx = np.flatnonzero(np.r_[b,True])
```

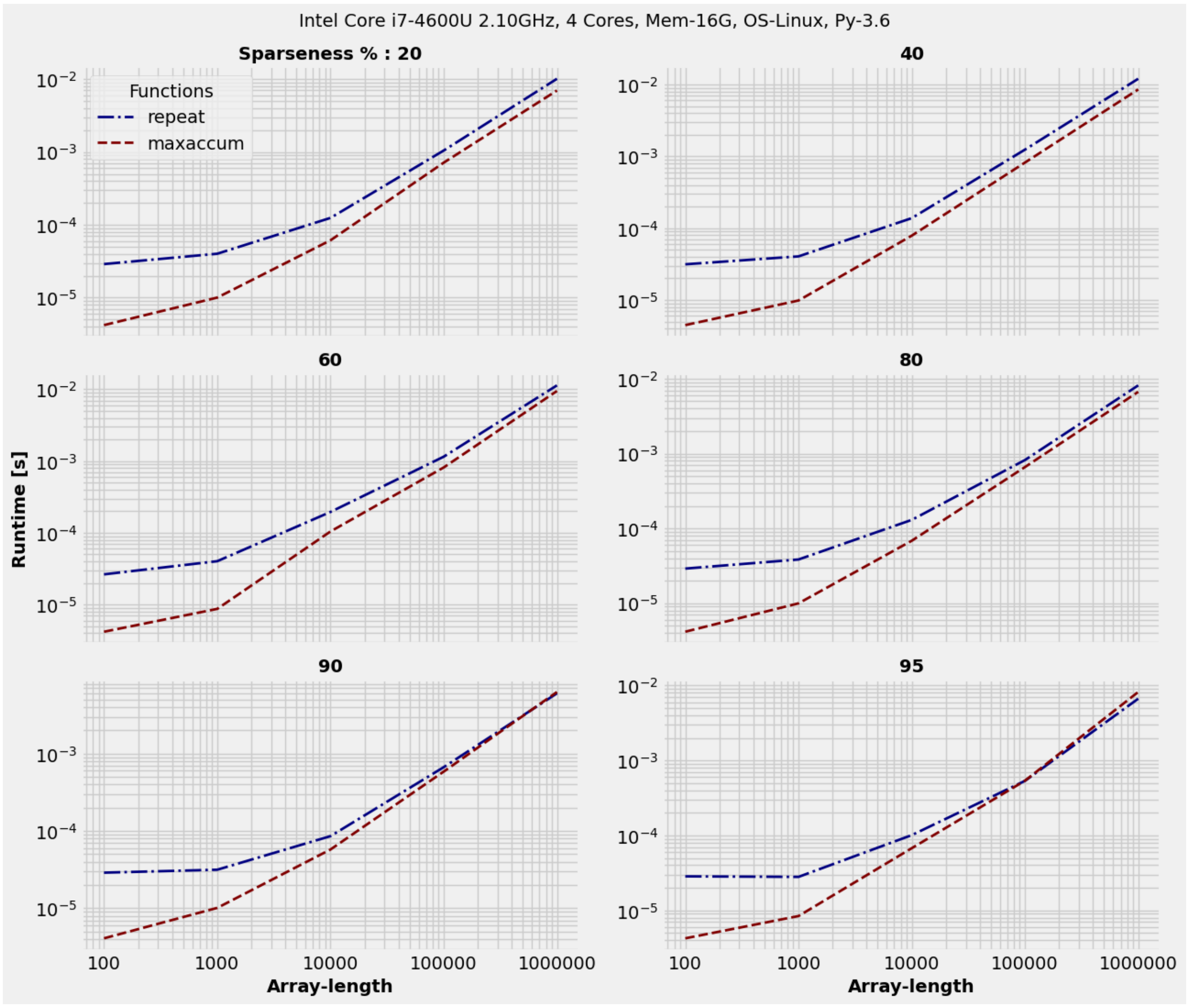
(continues on next page)

(continued from previous page)

```
    return np.repeat(idx[:-1], np.diff(idx))

def maxaccum(b):
    return np.maximum.accumulate(np.where(b,np.arange(len(b)), 0))

in_ = {(n,sf): np.random.rand(n)<(100-sf)/100. for n in [100,1000,10000,100000,
↪1000000] for sf in [20, 40, 60, 80, 90, 95]}
t = benchit.timings([repeat, maxaccum], in_, input_name=['Array-length','Sparseness %
↪'])
t.plot(logx=True, sp_ncols=2, save='singlegrp_id0_ffillmask_timings.png')
```



As always, some interesting inferences could be derived there. Seems *np.maximum.accumulate* is winning on most occasions, whereas *repeat* based one is doing better on highly sparse big datasets.

4.5.3 No argument

Random sampling

Finally, there might be cases when input functions have external no argument required. To create one such scenario, let's consider a setup where we compare numpy.random.choice against random.sample to get samples without replacement. We will consider an input data of *1000,000* elements and use those functions to extract *1000* samples. We will test out *random.sample* with two kinds of data - array and list, while feeding only array data to *numpy.random.choice*. Thus, in total we have three solutions, as listed in the full benchmarking shown below :

```
# Global inputs
import numpy as np
ar = np.arange(1000000)
l = ar.tolist()
sample_num = 1000

# Setup input functions with no argument
# NumPy random choice on array data
def np_noreplace():
    return np.random.choice(ar, sample_num, replace=False)

from random import sample

# Random sample on list data
def randsample_on_list():
    return sample(l, sample_num)

# Random sample on array data
def randsample_on_array():
    return sample(ar.tolist(), sample_num)

# Benchmark
t = benchit.timings(funcs=[np_noreplace, randsample_on_list, randsample_on_array])
>>> t
Functions  np_noreplace  randsample_on_list  randsample_on_array
Case
NoArg           0.02528            0.000653             0.033294
```

One interesting observation there - With array data *numpy.random.choice* is slightly better than *random.sample*. But, if we allow the flexibility of choosing between list and array data, *random.sample* turns the table in a big way. That's the whole point with benchmarking, which is to get insights into how different modules compare on the same functionality and how different data formats affect those runtime numbers. This in turn, should help the end-user decide on choosing methods depending on the available setup.

4.6 Plotting schemes

This is a brief discussion on various plotting schemes and tips that would be helpful in consideration, while customizing plots and plotting in different environments.

4.6.1 Plot features

For most of the plotting purposes, we can stick to *benchit*'s *plot* method - benchit.BenchmarkObj.plot. This method enables *kwargs* to pandas.DataFrame.plot Also, *pandas.DataFrame.plot* has its own *kwargs* that traces back to matplotlib.pyplot.plot. In essence, within *benchit*'s *plot* we can explore all plot arguments available to *pandas* and *matplotlib* plot versions. This should be sufficient for most plotting requirements.

To go the full hog, we can employ two more methods, which could be used individually or in combination.

Method #1 : Modify matplotlib rc settings

All of the *matplotlib* plot settings are stored in a dictionary-like variable called matplotlib.rcParams, which is global to the matplotlib package. More info on this is available at - Customizing Matplotlib with style sheets and rcParams. We can modify these to suit our plotting requirements. This setup is to be done before plotting.

Method #2 : Use axes methods

benchit's *plot* method returns an object of class *matplotlib.axes.Axes*. This has methods to change certain plot parameters. These could be located at matplotlib.axes.Axes. Most of those would be named as *Axes.set_[property]*. This is an after-plot adjustment and only applicable on interactive matplotlib backends.

4.6.2 Notebook plots

Plotting in IPython notebooks or Jupyter notebooks is supported for different *matplotlib* backends. Simply tell *benchit* to set the environment accordingly before plotting, with :

```
benchit.setparams(environ='notebook')
```

Note that this could also be used for non-interactive backends for better visualization. Matplotlib backends lists these backends and provides some general information on backends.

Sample notebook run.

4.6.3 Plot tips

When plotting with *benchit.BenchmarkObj.plot*, following tips could come in handy :

- If *xticks* seem congested, we can pass over the setting up for them to pandas version with *set_xticks_from_index* set as *False*. Another way would be to rotate *xticks* using its *rot* argument.

4.7 Changelog

4.7.1 0.0.6 (2021-06-04)

Changes :

- Added single-var groupings.
- Added props method to *BenchmarkObj* to list meta information about it.

Bug fixes :

- Fixed *ipython* dependency.

4.7.2 0.0.5 (2020-10-15)

Changes :

- Changes were applied at various levels to accommodate multiple arguments input with combinations cases mostly generated through nested loops over those arguments. Such combinations are led to generate subplots. This sub-plotting workflow is integrated to usual plotting mechanism to provide a one-stop solution. As such, the interface to the user stays the same, but with additional feature of sub-plotting for the combinations case.
- New one-line *specs* information.
- Framing feature added for plotting.
- Now we can quickly check out the layout and a general plot trend with a new function named *setparams*. This also replaces old function named *set_environ*, as we have few more arguments added to it.

Bug fixes :

- Fix for *logx* set as *True* and *set_xticks_from_index* set as *True* when dealing with various inputs on plotting.

4.7.3 0.0.4 (2020-08-06)

Changes :

- Progress bar fix for IPython/notebook runs to limit them to loop for datasets only.
- Documentation re-organized to show the minimal workflow and features separately.
- Introduced *set_environ* to set parameters for plotting, etc. according to *matplotlib* backend.
- Plot code reorganised to use specifications as title at dataframe plot level. This has helped in having an universal code to work for all backends. Makes use of *matplotlib rcparams* to setup environment before invoking dataframe plot method. The inspiration has been with notebook plotting. This has led to code cleanup to push more work in *main.py*.
- Owing to previous change, now deprecated *_add_specs_as_title*, *_add_specs_as_textbox*.

4.7.4 0.0.3 (2020-07-05)

Focus has been to make plots work across different matplotlib backends and few other plot improvements.

Changes :

- Added documentation support for lambdas.
- Added matplotlib inline plot support.
- Added different modes of argument for *speedups* and *scaled_timings* functions.
- Set customized default *logy* values based on *BenchmarkObj* datatypes.
- Added support for matplotlib backends other than *Qt5Agg* including notebook and inlining cases. Fix included a generic exception handling process to get fullscreen plots as is needed to put specifications as plot titles.

Bug fixes :

- *reset_columns* after *drop* works.

4.7.5 0.0.2 (2020-05-19)

With this release the focus has been to move the workflow from a tools perspective to a platform one. There's work done on a much tighter integration with *pandas-dataframe* format that should help to keep things on a platform-specific workflow.

Added methods to *BenchmarkObj* :

- *rank* to rank data based on certain performance metrics. This helps on making easier conclusions off the plots.
- *reset_columns* to retrieve original columns order. Intended to be used in conjunction with rank method.
- *copy* to retrieve original benchmarking results. This is to be used in cases where we might need to play around with the results and need a backup.
- *drop* to drop some functions or datasets to handle scenarios when we might want to focus on fewer ones.

Other changes :

- Added constructor for *BenchmarkObj* as *bench*.
- Renamed method *to_pandas_dataframe* for *BenchmarkObj* to *to_dataframe*.
- Nested calls to *speedups* and *scaled_timings* for *BenchmarkObj* disabled that are not applicable.
- Plot takes on automatic *ylabel* based on benchmarking object datatype.
- Introduced truncated colormap to avoid lighter colors, given the lighter background with the existing plotting scheme.
- Documentation pages re-arranged and improved to follow a workflow-based documentation.
- Renamed *bench.py* to *main.py* to avoid any conflicts with the new constructor to *BenchmarkObj* called *bench*.

4.7.6 0.0.1 (2020-04-10)

- Initial release.

4.8 API Reference

class `benchit.`**`BenchmarkObj`**(*df_timings*, *dtype='timings'*, *multivar=False*, *multiindex=False*)

Bases: `object`

Class that holds various methods to benchmark solutions on various aspects of benchmarking metrics. This also includes timing and plotting methods. The basic building block is a pandas dataframe that lists timings off various methods. The index has the various datasets and headers are functions. This class is intended to hold timings data. It is the central building block to benchmarking workflow..

`copy`()

Make a copy.

Returns Copy of input BenchmarkObj object.

Return type *BenchmarkObj*

`drop`(*labels*, *axis=1*)

Drop functions or datasets off the benchmarking object based on column or index values. It is an in-place operation.

Parameters **labels** (*Any scalar or list or tuple of scalars*) – Column or index value(s) to be dropped.

Returns NA.

Return type None

plot(*set_xticks_from_index=True*, *xlabel=None*, *ylabel=None*, *colormap='jet'*, *logx=False*, *logy=None*, *grid=True*, *linewidth=2*, *rot=None*, *dpi=None*, *fontsize=14*, *specs_fontsize=None*, *tick_fontsize=None*, *label_fontsize=None*, *legend_fontsize=None*, *figsize=None*, *specs_position='left'*, *debug_plotfs=False*, *pause_timefs=0.1*, *modules=None*, *use_frame=False*, *sp_argID=0*, *sp_ncols=-1*, *sp_sharey=None*, *sp_title_position='center'*, *sp_title_fontsize=None*, *sp_show_specs=True*, *save=None*, ***kwargs*)

Plot dataframe using given input parameters.

Parameters

- **set_xticks_from_index** (*bool, optional*) – Flag to use dataframe's index to set set_xticklabels or not.
- **xlabel** (*str, optional*) – Xlabel string.
- **ylabel** (*str, optional*) – Ylabel string.
- **colormap** (*str, optional*) – String that decides the colormap for plotting
- **logx** (*bool, optional*) – Flag to set x-axis scale as log or linear.
- **logy** (*None or bool, optional*) – If set as None, it detects default boolean flag using input Object datatype to be used as logy argument for plotting that decides the y-axis scale. With True and False, the scale is log and linear respectively. If set as boolean, it is used directly as logy argument.
- **grid** (*bool, optional*) – Flag to show grid or not.
- **linewidth** (*int, optional*) – Width of line to be used for plotting.
- **rot** (*int or None, optional*) – Rotation for ticks (xticks for vertical, yticks for horizontal plots).
- **dpi** (*float or None, optional*) – The resolution of the figure in dots-per-inch.
- **fontsize** (*float or int or None, optional*) – Fontsize used across specs_fontsize, tick_fontsize and label_fontsize if they are not set.
- **specs_fontsize** (*float or int or None, optional*) – Fontsize for specifications text displayed as title.
- **tick_fontsize** (*float or int or None, optional*) – Fontsize for xticks and yticks.
- **label_fontsize** (*float or int or None, optional*) – Fontsize for xlabel and ylabel.
- **figsize** (*tuple of two integers or None, optional*) – Tuple with syntax (figure_width, figure_height) for the figure window. This is applied only for environemnts where full-screen viewing is not possible.
- **specs_position** (*None or str, optional*) – str that decides where to print specs information. Options are : None(default), 'left', 'right' and 'center'.
- **debug_plotfs** (*bool, optional*) – Flag to decide whether to display debug info on fullscreen showing of plot. This is used only for interactive backends.
- **pause_timefs** (*float, optional*) – This is a pause number in seconds, used for plot to be rendered in fullscreen before saving it.
- **modules** (*dict, optional*) – Dictionary of modules.
- **use_frame** (*bool, optional*) – This indicates whether to use a frame or not. For subplot, this applies a frame to each subplot.

- **sp_argID** (*int, optional*) – This is specific to subplot case, when applicable (combinations are possible). This represents argument index for the input datasets to be used as the base (for x-axis labelling). This is based on 0-based indexing. Default argument index is 0, i.e. the first argument.
- **sp_ncols** (*int, optional*) – This is specific to subplot case, when applicable (combinations are possible). This denotes the number of columns used to create subplot grid.
- **sp_sharey** (*str or None, optional*) – This is specific to subplot case, when applicable (combinations are possible). This is used to indicate if and how the y-values are to be shared. Accepted values and their respective functionalities are listed below :

  None : y-values are not shared. 'row' or 'r': y-values are shared among same row of subplots. 'global' or 'g': y-values are shared globally across all subplots.
- **sp_title_position** (*str, optional*) – This is specific to subplot case, when applicable (combinations are possible). This indicates where to place the title for each subplot. Available values are - 'left', 'center' or 'right' respective to their positions.
- **sp_title_fontsize** (*float or int or None, optional*) – This is specific to subplot case, when applicable (combinations are possible). Fontsize for title for subplots that shows the grouping argument(s).
- **sp_show_specs** (*bool, optional*) – This decides whether to show specifications or not. Default is True, i.e show specifications.
- **save** (*str or None, optional*) – Path to save plot.
- ****kwargs** – Options to pass to pandas plot method, including kwargs for matplotlib plotting method.

Returns Plot of data from object's dataframe.

Return type matplotlib.axes._subplots.AxesSubplot

Notes

All subplot specific arguments have prefix of "**sp_**".

props()

Show object properties without the dataframe.

Parameters **None**

Returns NA

Return type None

rank(*mode='range'*)

Rank different functions based on their performance number and rank them by changing the columns order accordingly. It is an in-place operation.

Parameters **mode** (*str, optional*) – Sets the ranking criteria to rank different functions. It must be one among - 'range', 'constant', 'index'.

Returns NA.

Return type None

reset_columns()

Reset columns to original order.

scaled_timings(*ref*)

Evaluate scaled timings for all function calls with respect to one among them.

Parameters **ref** (*int or str or function*) – Input value represents one of the headers in the input BenchmarkObj. The scaled timings for all function calls are computed with respect to this reference.

Returns Scaled timings.

Return type *BenchmarkObj*

show_columns()

Get reference to inherent dataframe columns.

Parameters **None**

Returns Array of inherent dataframe columns.

Return type pandas.core.indexes.base.Index

show_index()

Get reference to inherent dataframe columns.

Parameters **None**

Returns Array of inherent dataframe columns.

Return type pandas.core.indexes.base.Index

speedups(*ref*)

Evaluate speedups for all function calls with respect to one among them.

Parameters **ref** (*int or str or function*) – Same as with scaled_timings.

Returns Speedups.

Return type *BenchmarkObj*

to_dataframe(*copy=False*)

Return underlying pandas dataframe object.

benchit.**bench**(*df*, *dtype='t'*, *copy=False*, *multivar=False*, *multiindex=False*)

Constructor function for creating BenchmarkObj object from a pandas dataframe. With input arguments, it could set as a timings or speedups or scaled-timings object. Additionally, the dataframe could be copied so that source dataframe stays unaffected.

Parameters

- **df** (*pandas dataframe*) – Dataframe listing the timings or speedups or scaled-timings or just any 2D data, i.e. number of levels with rows and columns is 1. Also, the dataframe should have the benchmarking information setup in the standardized setup way. Columns represent function names, alongwith df.columns.name assigned as 'Functions'. Index values represent dataset IDs, alongwith df.index.name assigned based on dataset type.
- **dtype** (*str, optional*) – Datatype value that decides between timings or speedups or scaled-timings. Mapping strings are : 't' -> timings, 'st' -> scaled-timings, 's' -> speedups.
- **copy** (*bool, optional*) – Decides whether to copy data when constructing benchamrking object.

Returns Data stored in BenchmarkObj.

Return type *BenchmarkObj*

`benchit.`**`timings`**(*funcs*, *inputs=None*, *multivar=False*, *input_name=None*, *indexby='auto'*)

Evaluate function calls on given input(s) to compute the timing. Puts out a dataframe-like object with the input properties being put into the header and index names and values.

Parameters

- **funcs** (*list or tuple*) – Contains the functions to be timed.
- **inputs** (*list or tuple or None, optional*) – Each elements of it represents one dataset each.
- **multivar** (*bool, optional*) – Decides whether to consider single or multiple variable input for feeding into the functions. As such it expects all functions to accept inputs in the same format. With the value as False, it assumes that every function accepts only one input. Hence, each element in inputs is considered as the only input to every function call. With the value as True, it assumes that every function accepts more than one input. Hence, each element in inputs is unpacked and fed to all functions.
- **input_name** (*str, optional*) – String that sets the index name for the output timings dataframe. This is used later on with plots to automatically assign x-label.
- **indexby** (*str, optional*) – String that sets the index properties for the output timings dataframe. Argument value must be one of - *'len'*, *'shape'*, *'item'*, *'scalar'*.

Returns Timings stored in a dataframe-like object with each row for each dataset and each column represents a function call.

Return type *BenchmarkObj*

`benchit.`**`extract_modules_from_globals`**(*glb*, *mode='valid'*)

Get modules from globals dict.

Parameters

- **glb** (*dict*) – Dictionary containing the modules.
- **mode** (*str, optional*) – Must be one of - *'valid'*, *'all'*.

Returns Extracted modules in a list

Return type list

`benchit.`**`specs_print`**(*modules=None*)

Print system specifications.

Parameters **modules** (*dict, optional*) – Dictionary containing the modules. These are optionally included to setup python modules info and printing it.

Returns NA.

Return type None

`benchit.`**`specs_short`**()

Get short-formatted one-line specifications as a string.

Parameters **None** – NA

Returns Specs information as a one-line string.

Return type str

`benchit.`**`setparams`**(*timeout=0.2*, *rep=5*, *environ='normal'*)

Set parameters for benchit.

Parameters

- **timeout** (*float or int, optional*) – Sets up timeout while looping with timeit that decides when to exit benchmarking for current iteration setup.
- **rep** (*float or int, optional*) – Sets up number of repetitions as needed to select the best timings among them as final runtime number for current iteration setup.
- **environ** (*str, optional*) – String that sets up environment given the current setup with global variable _ENVIRON.

Returns NA.

Return type None

CHAPTER 5

Appendix

- genindex
- modindex

Python Module Index

Index