asv_runner

Release 0.2.0

asv Developers

Feb 11, 2024

CONTENTS

Welcome to ASV Runner, the pure Python core of ASV (Airspeed-Velocity). ASV Runner provides essential functionality for benchmarking Python packages with ease and efficiency.

Key Features:

- **Benchmark Classes**: ASV Runner includes the core benchmark classes for `asv` that enable accurate measurement of performance metrics such as runtime, memory consumption, and peak memory usage.
- **Minimal Dependencies**: ASV Runner has minimal dependencies, relying primarily on pure Python for timing operations.
- **Flexible Integration**: ASV Runner is a core component of ASV, enabling comprehensive benchmarking of Python packages throughout their development lifecycle.

ASV runner is a powerful framework for leveraging `asv`'s rich plugin ecosystem. Explore the documentation and discover how ASV Runner can help you accurately measure and analyze the performance of your Python packages.

CHAPTER

ONE

API REFERENCE

This page contains auto-generated API reference documentation[1].

1.1 asv_runner

1.1.1 Subpackages

asv_runner.benchmarks

Variables

pkgname (str)
The name of the current package.

pkgpath (_frozen_importlib_external._NamespacePath)
The path of the current package.

module_names (List[str])
The names of all submodules in the current package that don't contain an underscore.

benchmark_types (List[Type])
A list to hold all benchmark classes from the submodules.

Raises

NotRequired (Exception)
If a submodule raises a `NotRequired` exception during import, it is ignored.

Notes

This module first identifies all submodules in the current package that don't contain an underscore in their names. It then iterates over these submodules, imports each one, and checks if it contains an attribute named "export_as_benchmark".

If such an attribute exists, its contents (which should be a list of benchmark classes) are added to the `benchmark_types` list. If a submodule raises a `NotRequired` exception during the import, it is ignored, and the loop continues with the next submodule.

This code is useful in a benchmarking suite where new benchmarks can be added simply by adding a new submodule with an "export_as_benchmark" attribute.

[1] Created with sphinx-autodoc2

Automatically discovers and imports benchmark classes from all submodules in the current package.

Submodules

asv_runner.benchmarks._maxrss

Module Contents

Data

ON_PYPY

API

asv_runner.benchmarks._maxrss.**ON_PYPY**

None

asv_runner.benchmarks._base

Module Contents

Classes

Benchmark	Class representing a single benchmark. The class encapsulates functions and methods that can be marked as benchmark

Functions

_get_attr	Retrieves an attribute from a source by its name.
_get_all_attrs	Yields attributes from a list of sources by their name.
_get_first_attr	Retrieves the first attribute from a list of sources by its name.
get_setup_cache_key	Retrieves the cache key for a function's setup.
get_source_code	Extracts, concatenates, and dedents the source code of the given items.
_get_sourceline_info	Retrieves the source file and line number information of the given object.
_check_num_args	Verifies if the function under benchmarking accepts a correct number of arguments.
_repr_no_address	Returns a string representing the object, but without its memory address.
_validate_params	Validates the params and param_names attributes and returns validated lists.
_unique_param_ids	Processes the params list to handle duplicate names within parameter sets, ensuring unique IDs.

API

asv_runner.benchmarks._base._get_attr(*source*, *name*, *ignore_case=False*)

Retrieves an attribute from a source by its name.

Parameters

source (object)
The source from which to get the attribute.

name (str)
The name of the attribute.

ignore_case (bool, optional)
Whether to ignore case when comparing attribute names. Defaults to `False`.

Returns

attr (object or None)
The attribute if it is found, else `None`.

Raises

ValueError
If more than one attribute with the given name exists and `ignore_case` is `True`.

asv_runner.benchmarks._base._get_all_attrs(*sources*, *name*, *ignore_case=False*)

Yields attributes from a list of sources by their name.

Parameters

sources (List[object])
The list of sources from which to get the attribute.

name (str)
The name of the attribute.

ignore_case (bool, optional)
Whether to ignore case when comparing attribute names. Defaults to `False`.

Yields

val (object)
The attribute if it is found in the source.

asv_runner.benchmarks._base._get_first_attr(*sources*, *name*, *default*, *ignore_case=False*)

Retrieves the first attribute from a list of sources by its name.

Parameters

sources (List[object])
The list of sources from which to get the attribute.

name (str)
The name of the attribute.

default (object)
The default value to return if no attribute is found.

ignore_case (bool, optional)
Whether to ignore case when comparing attribute names. Defaults to `False`.

Returns

attr (object)
The first attribute found or the default value if no attribute is found.

asv_runner.benchmarks._base.**get_setup_cache_key**(*func*)

Retrieves the cache key for a function's setup.

Parameters

func (function)
The function for which to get the cache key.

Returns

cache_key (str or None)
The cache key if the function is not `None`, else `None`.

Notes

The cache key is a string composed of the function's module name and the line number where the function's source code starts.

asv_runner.benchmarks._base.**get_source_code**(*items*)

Extracts, concatenates, and dedents the source code of the given items.

Parameters

items (Iterable[object])
An iterable of items, typically functions or methods, for which to extract the source code.

Returns

source_code (str)
The concatenated and dedented source code of the items.

Notes

The function retrieves the source code of each item. If the item has a `pretty_source` attribute, it uses that as the source code. Otherwise, it attempts to use the `inspect` module's `getsourcelines` function to extract the source code.

The function also adds class names to methods and properly indents the source code. If the source code belongs to a method, the function retrieves the class name and prepends it to the source code, properly indenting it to reflect its position within the class. If the source code belongs to the same class as the previous item, only the indentation is adjusted.

`asv_runner.benchmarks._base._get_sourceline_info`(*obj*, *basedir*)

Retrieves the source file and line number information of the given object.

Parameters

obj (object)
The object for which to retrieve source file and line number information. This is typically a function or a method.

basedir (str)
The base directory relative to which the source file path should be expressed.

Returns

sourceline_info (str)
A string containing the relative path of the source file and the line number where the object is defined, in the format `' in {filename}:{lineno}'`. If the source file or line number cannot be determined, an empty string is returned.

Notes

The function uses the `inspect` module's `getsourcefile` and `getsourcelines` functions to determine the source file and line number of the object, respectively. The source file path is converted to a path relative to `basedir` using `os.path.relpath`.

`asv_runner.benchmarks._base._check_num_args`(*root*, *benchmark_name*, *func*, *min_num_args*, *max_num_args=None*)

Verifies if the function under benchmarking accepts a correct number of arguments.

Parameters

root (str)
The root directory for the function's source file (used to print detailed error messages).

benchmark_name (str)
The name of the benchmark for which the function is being checked (used in error messages).

func (function)
The function to check for correct number of arguments.

min_num_args (int)
The minimum number of arguments the function should accept.

max_num_args (int, optional)
The maximum number of arguments the function should accept. If not provided, `max_num_args` is assumed to be the same as `min_num_args`.

Returns

validity (bool)
True if the function accepts a correct number of arguments, False otherwise.

Notes

The function uses the `inspect` module's `getfullargspec` function to determine the number of arguments the function accepts. It correctly handles functions, methods, variable argument lists, and functions with default argument values. In case of any error or if the function does not accept a correct number of arguments, an error message is printed to standard output.

`asv_runner.benchmarks._base._repr_no_address`(*obj*)

Returns a string representing the object, but without its memory address.

Parameters

obj (object)
The object to represent.

Returns

representation (str)
A string representation of the object without its memory address.

Notes

When Python's built-in `repr` function is used on an object, it often includes the memory address of the object. In some cases, this might not be desirable (for example, when comparing object representations in unit tests, where the memory address is not relevant). This function provides a way to get a string representation of an object without its memory address.

The function works by first getting the `repr` of the object, then using a regular expression to detect and remove the memory address if it's present. To avoid false positives, the function also gets the `repr` of the object using

the `object` class's `__repr__` method (which always includes the address), and only removes the address from the original `repr` if it matches the address in the `object.__repr__`.

Please note, this function is not guaranteed to remove the memory address for all objects. It is primarily intended to work for objects that have a `repr` similar to the default one provided by the `object` class.

`asv_runner.benchmarks._base._validate_params`(*params*, *param_names*, *name*)

Validates the params and param_names attributes and returns validated lists.

Parameters

params (`list`)
: List of parameters for the function to be benchmarked.

param_names (`list`)
: List of names for the parameters.

name (`str`)
: The name of the benchmark.

Returns

params, param_names (`list, list`)
: The validated parameter and parameter name lists.

`asv_runner.benchmarks._base._unique_param_ids`(*params*)

Processes the params list to handle duplicate names within parameter sets, ensuring unique IDs.

Parameters

params (`list`)
: List of parameters. Each entry is a list representing a set of parameters.

Returns

params (`list`)
: List of parameters with duplicate names within each set handled. If there are duplicate names, they are renamed with a numerical suffix to ensure unique IDs.

`class` `asv_runner.benchmarks._base.Benchmark`(*name*, *func*, *attr_sources*)

Class representing a single benchmark. The class encapsulates functions and methods that can be marked as benchmarks, along with setup and teardown methods, timing and other configuration.

Notes

The class uses regex to match method names that will be considered as benchmarks. The matched functions are then processed for benchmarking using various helper methods.

By default, a benchmark's timeout is set to 60 seconds.

Initialization

Initialize a new instance of `Benchmark`.

Parameters

name (`str`)
The name of the benchmark.

func (`function`)
The function to benchmark.

attr_sources (`list`)
List of sources from which attributes of the benchmark will be drawn. These attributes include setup, teardown, timeout, etc.

Attributes

pretty_name (`str`)
A user-friendly name for the function being benchmarked, if available.

_setups (`list`)
List of setup methods to be executed before the benchmark.

_teardowns (`list`)
List of teardown methods to be executed after the benchmark.

_setup_cache (`function`)
A special setup method that is only run once per parameter set.

setup_cache_key (`str`)
A unique key for the setup cache.

setup_cache_timeout (`float`)
The time after which the setup cache should be invalidated.

timeout (`float`)
The maximum time the benchmark is allowed to run before it is aborted.

code (`str`)
The source code of the function to be benchmarked and its setup methods.

version (`str`)
A version string derived from a hash of the code.

_params (`list`)
List of parameters for the function to be benchmarked.

param_names (`list`)
List of names for the parameters.

_current_params (tuple)
The current set of parameters to be passed to the function during the benchmark.

params (list)
The list of parameters with unique representations for exporting.

_skip_tuples (list)
List of tuples representing parameter combinations to be skipped before calling the setup method.

Raises

ValueError
If `param_names` or `_params` is not a list or if the number of parameters does not match the number of parameter names.

name_regex
None

__repr__()

set_param_idx(*param_idx*)
Set the current parameter values for the benchmark based on a parameter index.

This method updates the `_current_params` attribute with the set of parameter values that correspond to the provided parameter index.

Parameters

param_idx (int)
The index of the desired parameter set in the Cartesian product of `_params` attribute list.

Raises

ValueError
If the provided parameter index is not valid. This could occur if the index does not correspond to any element in the Cartesian product of the `_params` list.

insert_param(*param*)
Inserts a parameter at the beginning of the current parameter list.

This method modifies the `_current_params` attribute, inserting the provided parameter value at the front of the parameter tuple.

Parameters

param (Any)
The parameter value to insert at the front of `_current_params`.

check(*root*)
Checks call syntax (argument count) for benchmark's setup, call, and teardown.

Parameters

root (Any)
The root context for checking argument count in setup, call and teardown.

Returns

result (bool)
`True` if correct argument count is used in all methods, `False` otherwise.

Notes

The call syntax is checked only based on the number of arguments. It also sets the current parameters for the benchmark if they exist. The number of arguments required by setup, call, and teardown methods may increase if a setup cache is defined.

`do_setup()`

`redo_setup()`

`do_teardown()`

`do_setup_cache()`

`do_run()`

`do_profile(`*filename=None*`)`
Executes the benchmark's function with profiling using `cProfile`.

Parameters

filename (str, optional)
The name of the file where the profiling data should be saved. If not provided, the profiling data will not be saved.

Raises

RuntimeError
If the `cProfile` module couldn't be imported.

Notes

The method uses an inner function `method_caller` to call the function to be profiled. The function and its parameters should be available in the scope where `method_caller` is called.

The `cProfile` module should be available, or else a `RuntimeError` is raised. If a `filename` is provided, the profiling results will be saved to that file.

asv_runner.benchmarks.time

Module Contents

Classes

TimeBenchmark	Represents a single benchmark for timing.

Data

wall_timer
export_as_benchmark

API

asv_runner.benchmarks.time.**wall_timer**

None

class asv_runner.benchmarks.time.**TimeBenchmark**(*name*, *func*, *attr_sources*)

Bases: *asv_runner.benchmarks._base.Benchmark*

Represents a single benchmark for timing.

This class inherits from Benchmark and is specialized for timing benchmarks.

Attributes

name_regex (re.Pattern)
Regular expression that matches the name of the timing benchmarks.

rounds (int)
Number of rounds to execute the benchmark.

repeat (int)
Number of times the code will be repeated during each round.

min_run_count (int)
Minimum number of runs required for the benchmark.

number (int)
The argument to timeit.timeit, specifying the number of executions of the setup statement.

sample_time (float)
The target time for each sample.

warmup_time (float)
The time spent warming up the benchmark.

timer (callable)
The timer to use, by default it uses timeit.default_timer.

Initialization

Initialize a new instance of `TimeBenchmark`.

Parameters

name (`str`)
The name of the benchmark.

func (`callable`)
The function to benchmark.

attr_sources (`list`)
A list of objects from which to draw attributes.

`name_regex`
None

`_load_vars`()
Loads benchmark variables from attribute sources.

`do_setup`()
Execute the setup method and load variables.

`_get_timer`(param*)**
Get a `timeit.Timer` for the current benchmark.

`run`(param*)**
Run the benchmark with the given parameters.

Parameters

param (`tuple`)
The parameters to pass to the benchmark function.

Returns

result (`dict`)
A dictionary with the benchmark results. It contains the samples taken and the number of times the function was called in each sample.

Notes

The benchmark timing method is designed to adaptively find an optimal `number` of function executions to time based on the estimated performance. This number is then used for the final timings.

The warmup time is determined based on the Python interpreter in use. PyPy and GraalPython need longer warmup times due to their JIT compilers. For CPython, a short warmup time is used to account for transient effects such as OS scheduling.

The `repeat` attribute specifies how many times to run the function for timing. It can be an integer, meaning the function is run that many times, or a tuple of three values, specifying the minimum number of runs, the maximum number of runs, and the maximum total time to spend on runs.

After obtaining the timing samples, each sample is divided by the `number` of function executions to get the average time per function call, and these values are returned as the "samples" in the result.

benchmark_timing(*timer*, *min_repeat*, *max_repeat*, *max_time*, *warmup_time*, *number*, *min_run_count*)

Benchmark the timing of the function execution.

Parameters

timer (**timeit.Timer**)
The timer to use for the benchmarking.

min_repeat (**int**)
The minimum number of times to repeat the function execution.

max_repeat (**int**)
The maximum number of times to repeat the function execution.

max_time (**float**)
The maximum total time to spend on the benchmarking.

warmup_time (**float**)
The time spent warming up the benchmark.

number (**int**)
The number of executions of the setup statement.

min_run_count (**int**)
The minimum number of runs required for the benchmark.

Returns

result (**tuple**)
A tuple with the samples taken and the number of times the function was called in each sample.

Notes

The `too_slow` internal function is used to stop taking samples when certain limits are exceeded. These limits are the minimum run count, the minimum repeat count, and the maximum time.

If `number` is zero, a suitable number of function executions is estimated, and the system is warmed up at the same time.

If the warmup time is greater than zero, a warmup phase is initiated where the function is called repeatedly until the warmup time has passed.

After these initial steps, the function execution times are sampled and added to the `samples` list, stopping when reaching the maximum repeat count or when the `too_slow` function indicates to stop.

asv_runner.benchmarks.time.**export_as_benchmark**

None

asv_runner.benchmarks._exceptions

Module Contents

API

exception asv_runner.benchmarks._exceptions.**NotRequired**(*message*)

Bases: ImportError

Exception raised when a requirement is not met.

This exception inherits from `ImportError`. It's typically used when a particular package, module or other dependency that is not essential for the overall function of the program is not found or doesn't meet specific requirements.

Attributes

message (**str**)
A string that provides a more detailed explanation of the error.

Example

This exception might be used in a scenario where an optional feature of a program relies on a specific package that is not installed:

```
try:
    import optional_package
except ImportError:
    raise NotRequired("optional_package is not installed.")
```

Initialization

Initialize a new instance of `NotRequired`.

Parameters

message (**str**)
A string that provides a more detailed explanation of the error.

asv_runner.benchmarks.timeraw

Module Contents

Classes

_SeparateProcessTimer	This class provides a timer that runs a given function in a separate Python process.

continues on next page

Table 6 – continued from previous page

TimerawBenchmark	Represents a benchmark for tracking timing benchmarks run once in a separate process.

Data

export_as_benchmark

API

class asv_runner.benchmarks.timeraw.**_SeparateProcessTimer**(*func*)

This class provides a timer that runs a given function in a separate Python process.

The function should return the statement to be timed. This statement is executed using the Python timeit module in a new Python process. The execution time is then returned.

Attributes

subprocess_tmpl (str)
The template Python code to be run in the subprocess. It imports necessary modules and prints the execution time of the statement.

func (callable)
The function to be timed. This function should return a string of Python code to be executed, or a tuple of two strings: the code to be executed and the setup code to be run before timing.

Methods

timeit(number)
Run the function's code `number` times in a separate Python process, and return the execution time.

Initialization

subprocess_tmpl
None

timeit(*number*)
Run the function's code `number` times in a separate Python process, and return the execution time.

Parameters

number (int)
The number of times to execute the function's code.

Returns

time (float)
The time it took to execute the function's code number times.

Notes

The function's code is executed in a separate Python process to avoid interference from the parent process. The function can return either a single string of code to be executed, or a tuple of two strings: the code to be executed and the setup code to be run before timing.

class asv_runner.benchmarks.timeraw.**TimerawBenchmark**(*name*, *func*, *attr_sources*)

Bases: *asv_runner.benchmarks.time.TimeBenchmark*

Represents a benchmark for tracking timing benchmarks run once in a separate process.

This class inherits from TimeBenchmark and modifies it to run the benchmark function in a separate process. This is useful for isolating the benchmark from any potential side effects caused by other Python code running in the same process.

Attributes

name_regex (re.Pattern)
The regular expression used to match the names of functions that should be considered as raw timing benchmarks.

number (int)
The number of times to execute the function's code. By default, the function's code is executed once.

Methods

_load_vars()
Loads variables for the benchmark from the function's attributes or from default values.

***_get_timer(*param)**
Returns a timer that runs the benchmark function in a separate process.

do_profile(filename=None)
Raises a ValueError. Raw timing benchmarks cannot be profiled.

Initialization

Initialize a new instance of TimeBenchmark.

Parameters

name (str)
The name of the benchmark.

func (callable)
The function to benchmark.

attr_sources (list)
A list of objects from which to draw attributes.

name_regex
None

_load_vars()
Loads variables for the benchmark from the function's attributes or from default values.

_get_timer(*param)
Returns a timer that runs the benchmark function in a separate process.

Parameters

param (tuple)
The parameters to pass to the benchmark function.

Returns

timer (_SeparateProcessTimer)
A timer that runs the function in a separate process.

do_profile(filename=None)
Raises a ValueError. Raw timing benchmarks cannot be profiled.

Parameters

filename (str, optional)
The name of the file to which to save the profile. Default is None.

Raises

ValueError
Always. Raw timing benchmarks cannot be profiled.

asv_runner.benchmarks.timeraw.**export_as_benchmark**
None

asv_runner.benchmarks.peakmem

Module Contents

Classes

PeakMemBenchmark	Represents a single benchmark for tracking the peak memory consumption of the whole program.

Data

export_as_benchmark

API

class asv_runner.benchmarks.peakmem.**PeakMemBenchmark**(*name*, *func*, *attr_sources*)

Bases: asv_runner.benchmarks._base.Benchmark

Represents a single benchmark for tracking the peak memory consumption of the whole program.

The PeakMemBenchmark class provides a benchmark type for tracking the peak memory consumption of the program while the benchmark function is running.

Attributes

name_regex (re.Pattern)
The regular expression used to match the names of functions that should be considered as peak memory benchmarks.

type (str)
The type of the benchmark. The default type is "peakmemory".

unit (str)
The unit of the value that's being tracked. By default, this is "bytes".

Methods

***run(param)**
Runs the benchmark function and returns its result.

Initialization

Initializes a new instance of the PeakMemBenchmark class.

Parameters

name (str)
The name of the benchmark.

func (callable)
The function to benchmark.

attr_sources (list)
A list of objects to search for attributes that might be used by the benchmark.

name_regex
None

run(**param*)
Runs the benchmark function and measures its peak memory consumption.

Parameters

param (tuple)
The parameters to pass to the benchmark function.

Returns

result (int)
The peak memory consumption in bytes of the program while the benchmark function was running.

asv_runner.benchmarks.peakmem.**export_as_benchmark**
None

asv_runner.benchmarks.mark

Module Contents

Functions

skip_for_params	Decorator to set skip parameters for a benchmark function.
skip_benchmark	Decorator to mark a function as skipped for benchmarking.
skip_benchmark_if	Decorator to skip benchmarking of a function if a condition is met.
skip_params_if	Decorator to set skip parameters for a benchmark function if a condition is met.
parameterize_class_with	Class Decorator to set benchmark parameters for a class.
parameterize_func_with	Function Decorator to set benchmark parameters for a function.
parameterize	Decorator to set benchmark parameters for a function or a class.
timeout_class_at	Class Decorator to set timeout for a class.
timeout_func_at	Function Decorator to set timeout for a function.

continues on next page

Table 10 – continued from previous page

timeout_at	Decorator to set a timeout for a function or a class.

Data

__all__

API

exception asv_runner.benchmarks.mark.**SkipNotImplemented**(*message=''*)

Bases: NotImplementedError

Exception raised to indicate a skipped benchmark.

This exception inherits from `NotImplementedError`. It's used within an ASV benchmark to skip the current benchmark for certain parameters or conditions that are not implemented or do not apply.

Attributes

message (str)
A string that provides a more detailed explanation of the skip reason.

Warning

Use of `SkipNotImplemented` is less efficient than the `@skip_for_params` decorator as the setup for the benchmarks and the benchmarks themselves are run before the error is raised, thus consuming unnecessary resources. Use `@skip_for_params` where possible to avoid running the benchmarks that should be skipped.

Notes

This is mainly provided for backwards compatibility with the behavior of asv before 0.5 wherein individual benchmarks could raise and be skipped. From 0.5 onwards, only the setup function is meant to raise `NotImplemented` for skipping parameter sets.

Example

This exception might be used in a scenario where a benchmark should be skipped for certain conditions or parameters:

```
class Simple:
    params = ([False, True])
    param_names = ["ok"]

    def time_failure(self, ok):
        if ok:
            x = 34.2**4.2
        else:
            raise SkipNotImplemented
```

Initialization

Initialize a new instance of `SkipNotImplemented`.

Parameters

message (`str`)
: A string that provides a more detailed explanation of the skip reason. Optional; if not provided, defaults to an empty string.

`asv_runner.benchmarks.mark.`**skip_for_params**(*skip_params_list*)

Decorator to set skip parameters for a benchmark function.

Parameters

skip_params_list (`list`): A list of tuples, each specifying a combination of parameter values that should cause the benchmark function to be skipped.

Returns

decorator (`function`): A decorator function that sets the skip parameters for the benchmark function.

Notes

The `skip_for_params` decorator can be used to specify conditions under which a benchmark function should be skipped. Each tuple in the list represents a combination of parameter values which, if received by the benchmark function, will cause that function to be skipped during the benchmarking process.

The decorated function's `skip_params` attribute will be set with the provided skip parameters, which will be used during the benchmarking process.

Using this decorator is always more efficient than raising a `SkipNotImplemented` exception within the benchmark function, as the function setup and execution can be avoided entirely for skipped parameters.

Example

```
class Simple:
    params = ([False, True])
    param_names = ["ok"]

    @skip_for_params([(False, )])
    def time_failure(self, ok):
        if ok:
            x = 34.2**4.2
```

`asv_runner.benchmarks.mark.`**skip_benchmark**(*func*)

Decorator to mark a function as skipped for benchmarking.

Parameters

func (function)
The function to be marked as skipped.

Returns

wrapper (function)
A wrapped function that is marked to be skipped for benchmarking.

Notes

The `skip_benchmark` decorator can be used to mark a specific function as skipped for benchmarking. When the decorated function is encountered during benchmarking, it will be skipped and not included in the benchmarking process.

`asv_runner.benchmarks.mark.`**`skip_benchmark_if`**`(`*condition*`)`

Decorator to skip benchmarking of a function if a condition is met.

Parameters

condition (`bool`)
A boolean that indicates whether to skip benchmarking. If `True`, the decorated function will be skipped for benchmarking. If `False`, the decorated function will be benchmarked as usual.

Returns

decorator (function)
A decorator function that sets the condition under which the decorated function will be skipped for benchmarking.

Notes

The `skip_if` decorator can be used to skip the benchmarking of a specific function if a condition is met. It is faster than raising `SkipNotImplemented` as it skips the `setup()` as well.

`asv_runner.benchmarks.mark.`**`skip_params_if`**`(`*skip_params_list*, *condition*`)`

Decorator to set skip parameters for a benchmark function if a condition is met.

Parameters

skip_params_list (`list`): A list specifying the skip parameters for the benchmark function.

condition (`bool`)
A boolean that indicates whether to set the skip parameters. If `True`, the skip parameters will be set for the decorated function. If `False`, no parameters will be skipped.

Returns

decorator (function): A decorator function that sets the skip parameters for the benchmark function if the condition is met.

Notes

The `skip_params_if` decorator can be used to specify skip parameters for a benchmark function if a condition is met.

`asv_runner.benchmarks.mark.`**`parameterize_class_with`**(*param_dict*)

Class Decorator to set benchmark parameters for a class.

Parameters

param_dict (`dict`): A dictionary specifying the parameters for the benchmark class. The keys represent the parameter names, and the values are lists of values for those parameters.

Returns

decorator (function): A class decorator that sets the parameters for the benchmark functions.

Notes

The `parameterize_class_with` decorator can be used to specify parameters for a benchmark class. The parameters are defined as a dictionary, where keys are the parameter names and values are lists of respective values. The decorated class's `params` and `param_names` attributes will be set with the provided parameters and names, which will be used during the benchmarking process. This decorator will overwrite any existing `params` and `param_names` attributes in the class.

`asv_runner.benchmarks.mark.`**`parameterize_func_with`**(*param_dict*)

Function Decorator to set benchmark parameters for a function.

Parameters

param_dict (`dict`): A dictionary specifying the parameters for the benchmark function. The keys represent the parameter names, and the values are lists of values for those parameters.

Returns

decorator (function): A function decorator that sets the parameters for the benchmark function.

Notes

The `parameterize_func_with` decorator can be used to specify parameters for a benchmark function. The parameters are defined as a dictionary, where keys are the parameter names and values are lists of respective values. The decorated function's `params` and `param_names` attributes will be set with the provided parameters and names, which will be used during the benchmarking process. This decorator will overwrite any existing `params` and `param_names` attributes in the function, and it should not be used with methods of a class.

`asv_runner.benchmarks.mark.`**`parameterize`**`(`*`param_dict`*`)`

Decorator to set benchmark parameters for a function or a class.

Parameters

param_dict (`dict`): A dictionary specifying the parameters for the benchmark. The keys represent the parameter names, and the values are lists of values for those parameters.

Returns

decorator (function): A function or class decorator that sets the parameters for the benchmark.

Notes

The `parameterize` decorator can be used to specify parameters for a benchmark function or class. The parameters are defined as a dictionary, where keys are the parameter names and values are lists of respective values. The decorated function or class's `params` and `param_names` attributes will be set with the provided parameters and names, which will be used during the benchmarking process.

`asv_runner.benchmarks.mark.`**`timeout_class_at`**`(`*`seconds`*`)`

Class Decorator to set timeout for a class.

Parameters

seconds (`float`)
: The number of seconds after which the class methods should be timed out.

Returns

decorator (function)
: A class decorator that sets the timeout for the class.

Notes

The `timeout_class_at` decorator can be used to specify a timeout for all methods in a class. The timeout is stored as an attribute on the class and applies to all its methods. Individual methods can override this timeout by using the `timeout_func_at` or `timeout_at` decorators.

`asv_runner.benchmarks.mark.`**`timeout_func_at`**`(`*`seconds`*`)`

Function Decorator to set timeout for a function.

Parameters

seconds (float)
The number of seconds after which the function should be timed out.

Returns

decorator (function)
A function decorator that sets the timeout for the function.

Notes

The `timeout_func_at` decorator can be used to specify a timeout for a specific function. This is particularly useful for benchmarking, where you might want to stop execution of functions that take too long. The timeout is stored as an attribute on the function.

`asv_runner.benchmarks.mark.timeout_at(`*seconds*`)`

Decorator to set a timeout for a function or a class.

Parameters

seconds (float)
The number of seconds after which the function or the class methods should be timed out.

Returns

decorator (function)
A decorator that sets the timeout for the function or the class.

Notes

The `timeout_at` decorator can be used to set a specific timeout for a function or all methods in a class. If applied to a class, the timeout is stored as an attribute on the class and applies to all its methods. Individual methods can override this timeout by using the `timeout_func_at` or `timeout_at` decorators. If applied to a function, the timeout is stored directly on the function.

`asv_runner.benchmarks.mark.__all__`

['parameterize', 'skip_benchmark', 'skip_benchmark_if', 'skip_for_params', 'skip_params_if', 'timeou…

asv_runner.benchmarks.track

Module Contents

Classes

TrackBenchmark	Represents a single benchmark for tracking an arbitrary value.

Data

export_as_benchmark

API

class asv_runner.benchmarks.track.**TrackBenchmark**(*name*, *func*, *attr_sources*)

Bases: *asv_runner.benchmarks._base.Benchmark*

Represents a single benchmark for tracking an arbitrary value.

The TrackBenchmark class provides a benchmark type for tracking any arbitrary value that your code produces. This can be useful when you need to track a value that isn't related to time or memory usage.

Attributes

name_regex (re.Pattern)
The regular expression used to match the names of functions that should be considered as track benchmarks.

type (str)
The type of the benchmark. The default type is "track".

unit (str)
The unit of the value that's being tracked. By default, this is "unit".

Methods

***run(param)**
Runs the benchmark function and returns its result.

Initialization

Initializes a new instance of the TrackBenchmark class.

Parameters

name (str)
The name of the benchmark.

func (callable)
The function to benchmark.

attr_sources (list)
A list of objects to search for attributes that might be used by the benchmark.

name_regex
None

run(**param*)
Runs the benchmark function and returns its result.

Parameters

param (tuple)
The parameters to pass to the benchmark function.

Returns

result
The result of the benchmark function.

asv_runner.benchmarks.track.**export_as_benchmark**

None

asv_runner.benchmarks.mem

Module Contents

Classes

MemBenchmark	Represents a single benchmark for tracking the memory consumption of an object.

Data

export_as_benchmark

API

class asv_runner.benchmarks.mem.**MemBenchmark**(*name*, *func*, *attr_sources*)

Bases: *asv_runner.benchmarks._base.Benchmark*

Represents a single benchmark for tracking the memory consumption of an object.

The MemBenchmark class provides a benchmark type for tracking the memory consumption of the object returned by the benchmark function.

Attributes

name_regex (re.Pattern)
The regular expression used to match the names of functions that should be considered as memory benchmarks.

type (str)
The type of the benchmark. The default type is "memory".

unit (str)
The unit of the value that's being tracked. By default, this is "bytes".

Methods

***run(*param)**
Runs the benchmark function and returns the memory consumption of the object returned by the function.

Initialization

Initializes a new instance of the MemBenchmark class.

Parameters

name (str)
The name of the benchmark.

func (callable)
The function to benchmark.

attr_sources (list)
A list of objects to search for attributes that might be used by the benchmark.

name_regex
None

run(**param*)
Runs the benchmark function and measures the memory consumption of the object returned by the function.

Parameters

param (tuple)
The parameters to pass to the benchmark function.

Returns

result (int)
The memory consumption in bytes of the object returned by the benchmark function.

asv_runner.benchmarks.mem.**export_as_benchmark**
None

Package Contents

Data

pkgname
pkgpath
submodule_names
asv_modules
benchmark_types

API

asv_runner.benchmarks.**pkgname**

None

asv_runner.benchmarks.**pkgpath**

None

asv_runner.benchmarks.**submodule_names**

None

asv_runner.benchmarks.**asv_modules**

None

asv_runner.benchmarks.**benchmark_types**

[]

1.1.2 Submodules

asv_runner.console

A set of utilities for writing output to the console.

Module Contents

Classes

Log

Functions

isatty	Determines if a file is a tty.
_color_text	Returns a string wrapped in ANSI color codes for coloring the text in a terminal.
_write_with_fallback	Writes the supplied string to the given file-like object, handling potential UnicodeEncodeErrors by falling back to the locale's preferred encoding.

continues on next page

Table 18 – continued from previous page

color_print	`Prints colored and styled text to the terminal using ANSI escape sequences.` `#### Parameters` `*args (`tuple` of `str`):` `The positional arguments should come in pairs (`msg`, `color`), where `msg` is the string to display and `color` is the color to display it in. `color` is an ANSI terminal color name. Must be one of: black, red, green, brown, blue, magenta, cyan, lightgrey, default, darkgrey, lightred, lightgreen, yellow, lightblue, lightmagenta, lightcyan, white, or '' (the empty string).` `file` (writable file-like object, optional):` `Where to write to. Defaults to `sys.stdout`. If `file` is not a tty (as determined by calling its `isatty` member, if one exists), no coloring will be included. It's passed as a keyword argument.`  `end` (`str`, optional):` `The ending of the message. Defaults to "`
get_answer_default	Prompts the user for input and returns the entered value or a default.
truncate_left	

Data

WIN
_unicode_translations

API

asv_runner.console.**WIN**

None

asv_runner.console.**isatty**(*file*)

Determines if a file is a tty.

Parameters

file (**file-like object**)
The file-like object to check.

Returns

isatty (**bool**)
Returns `True` if the file is a tty, `False` otherwise.

Notes

Most built-in Python file-like objects have an `isatty` member, but some user-defined types may not. In such cases, this function assumes those are not ttys.

asv_runner.console.**_color_text**(*text*, *color*)

Returns a string wrapped in ANSI color codes for coloring the text in a terminal.

Parameters

text (**str**)
The string to colorize.

color (**str**)
An ANSI terminal color name. Must be one of the following: 'black', 'red', 'green', 'brown', 'blue', 'magenta', 'cyan', 'lightgrey', 'default', 'darkgrey', 'lightred', 'lightgreen', 'yellow', 'lightblue', 'lightmagenta', 'lightcyan', 'white', or '' (the empty string).

Returns

colored_text (**str**)
The input string, bounded by the appropriate ANSI color codes.

Notes

This function wraps the input text with ANSI color codes based on the given color. It won't actually affect the text until it is printed to the terminal.

asv_runner.console.**_unicode_translations**

None

asv_runner.console.**_write_with_fallback**(*s*, *fileobj*)

Writes the supplied string to the given file-like object, handling potential UnicodeEncodeErrors by falling back to the locale's preferred encoding.

Parameters

`s` (`str`): The Unicode string to be written to the file-like object. Raises a `ValueError` if `s` is not a Unicode string.

`fileobj` (file-like object): The file-like object to which the string `s` is to be written. On Python 3, this must be a text stream. On Python 2, this must be a `file` byte stream.

Notes

This function first tries to write the input string `s` to the file object `fileobj`. If a `UnicodeError` occurs during this process (indicating that the string contains characters not representable in the file's encoding), the function falls back to encoding the string in the locale's preferred encoding before writing.

If the string `s` still cannot be encoded in the locale's preferred encoding, the function translates the string to replace problematic Unicode characters with ASCII ones using the `_unicode_translations` dictionary, and then encodes and writes the resulting string to `fileobj` using the "replace" error handling scheme (which replaces any non-encodable characters with a suitable replacement marker).

After the write operation, the function flushes the file object's output buffer to ensure that the written data is actually saved to the file.

asv_runner.console.**color_print**(**args*, ***kwargs*)

```
Prints colored and styled text to the terminal using ANSI escape sequences.

#### Parameters
*args (`tuple` of `str`):
The positional arguments should come in pairs (`msg`, `color`), where `msg`
is the string to display and `color` is the color to display it in. `color`
is an ANSI terminal color name. Must be one of: black, red, green, brown,
blue, magenta, cyan, lightgrey, default, darkgrey, lightred, lightgreen,
yellow, lightblue, lightmagenta, lightcyan, white, or '' (the empty string).

`file` (writable file-like object, optional):
Where to write to. Defaults to `sys.stdout`. If `file` is not a tty (as determined
by calling its `isatty` member, if one exists), no coloring will be included. It's
passed as a keyword argument.

`end` (`str`, optional):
The ending of the message. Defaults to "
```

". The `end` will be printed after resetting any color or font state. It's passed as a keyword argument.

```
#### Notes
This function allows you to print text in various colors to the console, which can
be helpful for distinguishing different kinds of output or for drawing attention to
particular messages.

It works by applying ANSI escape sequences to the input strings according to the
specified colors. These escape sequences are interpreted by the terminal emulator
to apply the specified colors and styles.

#### Example
```{code-block} python
color_print('This is the color ', 'default', 'GREEN', 'green')
```
```

asv_runner.console.**get_answer_default**(*prompt*, *default*, *use_defaults=False*)

Prompts the user for input and returns the entered value or a default.

Parameters

`prompt` (`str`): The string that is presented to the user.

`default` (any): The value returned if the user doesn't enter anything and just hits Enter. This value is also shown in the prompt to indicate to the user what the default is.

`use_defaults` (`bool`, optional): If True, the function will immediately return the default value without prompting the user for input. Defaults to False.

Returns

The user's input, or the provided default value if the user didn't enter anything.

Notes

This function enhances the built-in `input` function by allowing a default value to be specified, which is returned if the user doesn't enter anything.

asv_runner.console.**truncate_left**(*s*, *l*)

class asv_runner.console.**Log**

Initialization

_stream_formatter(*record*)

The formatter for standard output

indent()

A context manager to increase the indentation level.

dot()

set_nitems(*n*)

Set the number of remaining items to process. Each of these steps should be incremented through using `step`.

Can be called multiple times. The progress percentage is ensured to be non-decreasing, except if 100% was already reached in which case it is restarted from 0%.

step()

Write that a step has been completed. A percentage is displayed along with it.

If we are stepping beyond the number of items, stop counting.

enable(*verbose=False*)

set_level(*level*)

is_debug_enabled()

_message(*routine*, *message*, *reserve_space=False*, *color=None*, *continued=False*)

info(**args*, ***kwargs*)

warning(**args*, ***kwargs*)

debug(**args*, ***kwargs*)

error(**args*, ***kwargs*)

add(*msg*)

add_padded(*msg*)

Final part of two-part info message. Should be preceded by a call to info/warn/...(msg, reserve_space=True)

flush()

Flush any trailing newlines. Needs to be called before printing to stdout via other means, after using Log.

asv_runner.statistics

Module Contents

Classes

LaplacePosterior	Class to represent univariate Laplace posterior distribution.

Functions

get_err	Computes an 'error measure' based on the interquartile range of the measurement results.
binom_pmf	Computes the Probability Mass Function (PMF) for a binomial distribution.
quantile	Computes a quantile/percentile from a dataset.
quantile_ci	Compute a quantile and a confidence interval for a given dataset.

continues on next page

Table 21 – continued from previous page

compute_stats	Performs statistical analysis on the provided samples.

API

asv_runner.statistics.**get_err**(*result*, *stats*)

Computes an 'error measure' based on the interquartile range of the measurement results.

Parameters

result (**any**)
The measurement results. Currently unused.

stats (**dict**)
A dictionary of statistics computed from the measurement results. It should contain the keys "q_25" and "q_75" representing the 25th and 75th percentiles respectively.

Returns

error (**float**)
The error measure, defined as half the interquartile range (i.e., (Q3 - Q1) / 2).

asv_runner.statistics.**binom_pmf**(*n*, *k*, *p*)

Computes the Probability Mass Function (PMF) for a binomial distribution.

Parameters

n (**int**)
The number of trials in the binomial distribution.

k (**int**)
The number of successful trials.

p (**float**)
The probability of success on each trial.

Returns

pmf (**float**)
The binomial PMF computed as (n choose k) * p**k * (1 - p)**(n - k).

Notes

Handles edge cases where p equals 0 or 1.

asv_runner.statistics.**quantile**(*x*, *q*)

Computes a quantile/percentile from a dataset.

Parameters

x (**list of float**)
: The dataset for which the quantile is to be computed.

q (**float**)
: The quantile to compute. Must be in the range [0, 1].

Returns

m (**float**)
: The computed quantile from the dataset.

Raises

ValueError
: If the provided quantile q is not in the range [0, 1].

Notes

This function sorts the input data and calculates the quantile using a linear interpolation method if the desired quantile lies between two data points.

asv_runner.statistics.**quantile_ci**(*x*, *q*, *alpha_min=0.01*)

Compute a quantile and a confidence interval for a given dataset.

Parameters

x (**list of float**)
: The dataset from which the quantile and confidence interval are computed.

q (**float**)
: The quantile to compute. Must be in the range [0, 1].

alpha_min (**float, optional**)
: Limit for coverage. The result has coverage >= 1 - alpha_min. Defaults to 0.01.

Returns

m (float)
The computed quantile from the dataset.

ci (tuple of float)
Confidence interval (a, b), of coverage >= alpha_min.

Notes

This function assumes independence but is otherwise nonparametric. It sorts the input data and calculates the quantile using a linear interpolation method if the desired quantile lies between two data points. The confidence interval is computed using a known property of the cumulative distribution function (CDF) of a binomial distribution. This method calculates the smallest range (y[r-1], y[s-1]) for which the coverage is at least alpha_min.

class asv_runner.statistics.**LaplacePosterior**(*y*, *nu=None*)

Class to represent univariate Laplace posterior distribution.

Description

This class represents the univariate posterior distribution defined as `p(beta|y) = N [sum(|y_j - beta|)]**(-nu-1)` where N is the normalization factor.

Parameters

y (list of float)
A list of sample values from the distribution.

nu (float, optional)
Degrees of freedom. Default is `len(y) - 1`.

Attributes

mle (float)
The maximum likelihood estimate for beta which is the median of y.

Notes

This is the posterior distribution in the Bayesian model assuming Laplace distributed noise, where `p(y|beta, sigma) = N exp(- sum_j (1/sigma) |y_j - beta|)`, `p(sigma) ~ 1/sigma`, and `nu = len(y) - 1`. The MLE for beta is `median(y)`. Applying the same approach to a Gaussian model results to `p(beta|y) = N T(t, m-1)`, `t = (beta - mean(y)) / (sstd(y) / sqrt(m))` where `T(t, nu)` is the Student t-distribution pdf, which gives the standard textbook formulas for the mean.

Initialization

Initializes an instance of the LaplacePosterior class.

Parameters

y (`list of float`):
The samples from the distribution.

nu (`float, optional`):
The degrees of freedom. Default is `len(y) - 1`.

Raises

`ValueError`: If `y` is an empty list.

Notes

This constructor sorts the input data `y` and calculates the MLE (Maximum Likelihood Estimate). It computes a scale factor, `_y_scale`, to prevent overflows when computing unnormalized CDF integrals. The input data `y` is then shifted and scaled according to this computed scale. The method also initializes a memoization dictionary `_cdf_memo` for the unnormalized CDF, and a placeholder `_cdf_norm` for the normalization constant of the CDF.

`_cdf_unnorm`(*beta*)

Computes the unnormalized cumulative distribution function (CDF).

Parameters

beta (`float`):
The upper limit of the integration for the CDF.

Returns

Returns the unnormalized CDF evaluated at `beta`.

Notes

The method computes the unnormalized CDF as:

```
cdf_unnorm(b) = int_{-oo}^{b} 1/(sum_j |y - b'|)**(m+1) db'
```

The method integrates piecewise, resolving the absolute values separately for each section. The results of these calculations are memoized to speed up subsequent computations.

It also handles special cases, such as when `beta` is not a number (returns `beta` as is), or when `beta` is positive infinity (memoizes the integral value at the end of the list `y`).

`_ppf_unnorm`(*cdfx*)

Computes the inverse function of `_cdf_unnorm`.

Parameters

cdfx (`float`):
The value for which to compute the inverse cumulative distribution function (CDF).

Returns

Returns the unnormalized quantile function evaluated at `cdfx`.

Notes

This method computes the inverse of `_cdf_unnorm`. It first finds the interval within which `cdfx` lies, then performs the inversion on this interval.

Special cases are handled when the interval index `k` is 0 (the computation of `beta` involves a check for negative infinity), or when the calculated `c` is 0. The result `beta` is clipped at the upper bound of the interval, ensuring it does not exceed `self.y[k]`.

`pdf`(*beta*)

Computes the probability distribution function (PDF).

Parameters

beta (`float`)
The point at which to evaluate the PDF.

Returns

A `float` which is the probability density function evaluated at `beta`.

Notes

This function computes the PDF by exponentiating the result of `self.logpdf(beta)`. The `logpdf` method should therefore be implemented in the class that uses this method.

`logpdf`(*beta*)

Computes the logarithm of the probability distribution function (log-PDF).

Parameters

beta (`float`)
The point at which to evaluate the log-PDF.

Returns

A `float` which is the logarithm of the probability density function evaluated at `beta`.

Notes

This function computes the log-PDF by first checking if the scale of the distribution `_y_scale` is zero. If so, it returns `math.inf` if `beta` equals the maximum likelihood estimate `mle`, otherwise it returns `-math.inf`.

The `beta` value is then transformed by subtracting the maximum likelihood estimate `mle` and dividing by `_y_scale`.

If the unnormalized cumulative distribution function `_cdf_norm` has not been computed yet, it is computed by calling `_cdf_unnorm(math.inf)`.

The function then computes the sum of absolute differences between `beta` and all points in `y`, applies the log-PDF formula and returns the result.

cdf(*beta*)

Computes the cumulative distribution function (CDF).

Parameters

beta (**float**)
: The point at which to evaluate the CDF.

Returns

A `float` which is the value of the cumulative distribution function evaluated at `beta`.

Notes

This function computes the CDF by first checking if the scale of the distribution `_y_scale` is zero. If so, it returns 1 if `beta` is greater than the maximum likelihood estimate `mle`, and 0 otherwise.

The `beta` value is then transformed by subtracting the maximum likelihood estimate `mle` and dividing by `_y_scale`.

If the unnormalized cumulative distribution function `_cdf_norm` has not been computed yet, it is computed by calling `_cdf_unnorm(math.inf)`.

The function then computes the unnormalized CDF at `beta` and normalizes it by dividing with `_cdf_norm`.

ppf(*cdf*)

Computes the percent point function (PPF), also known as the inverse cumulative distribution function.

Parameters

cdf (float)
The cumulative probability for which to compute the inverse CDF. It must be between 0 and 1 (inclusive).

Returns

A `float` which is the value of the percent point function evaluated at `cdf`.

Notes

This function computes the PPF by first checking if `cdf` is not between 0 and 1. If it is not, it returns `math.nan`.

If the scale of the distribution `_y_scale` is zero, it returns the maximum likelihood estimate `mle`.

If the unnormalized cumulative distribution function `_cdf_norm` has not been computed yet, it is computed by calling `_cdf_unnorm(math.inf)`.

The function then scales `cdf` by `_cdf_norm` (making sure it does not exceed `_cdf_norm`), computes the unnormalized PPF at this scaled value, and transforms it back to the original scale.

`asv_runner.statistics.`**`compute_stats`**(*samples*, *number*)

Performs statistical analysis on the provided samples.

Parameters

samples (list of float)
A list of total times (in seconds) of benchmarks.

number (int)
The number of times each benchmark was repeated.

Returns

beta_hat (float)
The estimated time per iteration.

stats (dict)
A dictionary containing various statistical measures of the estimator. It includes:

- **"ci_99_a"**: The lower bound of the 99% confidence interval.
- **"ci_99_b"**: The upper bound of the 99% confidence interval.
- **"q_25"**: The 25th percentile of the sample times.
- **"q_75"**: The 75th percentile of the sample times.
- **"repeat"**: The total number of samples.
- **"number"**: The repeat number for each sample.

Notes

This function first checks if there are any samples. If there are none, it returns `None, None`.

It then calculates the median and the 25th and 75th percentiles of the samples. If the nonparametric confidence interval estimation did not provide an estimate, it computes the posterior distribution for the location, assuming exponential noise. The Maximum Likelihood Estimate (MLE) is equal to the median. The function uses the confidence interval from that distribution to extend beyond the sample bounds if necessary.

Finally, it produces the median as the result and a dictionary of the computed statistics.

asv_runner.run

Module Contents

Functions

_run	Runs a specified benchmark and writes the result to a file.

API

asv_runner.run.**_run**(*args*)

Runs a specified benchmark and writes the result to a file.

Parameters

args (**tuple**)
: A tuple containing benchmark directory, benchmark id, parameters string, profile path, and result file path.

Notes

This function first loads the extra parameters and sets the CPU affinity based on them. It then creates a benchmark from the `benchmark_id`. If the benchmark has a setup cache key, it loads the cache from a file and inserts it into the benchmark parameters.

Then, the function runs the setup for the benchmark. If the setup indicates that the benchmark should be skipped, it sets the result as `math.nan`. Otherwise, it runs the benchmark and profiles it if a `profile_path` is provided. After running the benchmark, it performs the teardown for the benchmark and writes the result to the `result_file`.

The `args` tuple contains:

- **benchmark_dir** (`str`) : The directory where the benchmarks are located.
- **benchmark_id** (`str`) : The id of the benchmark to run.
- **params_str** (`str`) : A string containing JSON-encoded extra parameters.
- **profile_path** (`str`) : The path for profile data. "None" implies no profiling.
- **result_file** (`str`) : The path to the file where the result should be written.

asv_runner.util

Various low-level utilities.

Module Contents

Functions

ceildiv	Calculate the ceiling division of two numbers.
human_float	Formats a float into a human-friendly string.
human_time	Formats a duration in seconds into a human-friendly time string.

Data

terminal_width
_human_time_units

API

asv_runner.util.**terminal_width**

None

asv_runner.util.**ceildiv**(*numerator*, *denominator*)

Calculate the ceiling division of two numbers.

Parameters

numerator (**int**)
The numerator in the division.

denominator (**int**)
The denominator in the division.

Returns

`int`: The result of the division rounded up to the nearest integer.

Notes

This function calculates the ceiling division of two numbers, i.e., division that rounds up. It is equivalent to `math.ceil(numerator/denominator)`, but avoids the conversion of numerator and denominator to float.

asv_runner.util.**human_float**(*value*, *significant=3*, *truncate_small=None*, *significant_zeros=False*)

Formats a float into a human-friendly string.

Parameters

value (float)
The float value to format.

significant (int)
Number of significant digits to include in the output. Default is 3.

truncate_small (int, optional)
If defined, leading zeros of numbers < 1 are counted as significant.

significant_zeros (bool)
If True, trailing unnecessary zeros are included. Default is False.

Returns

`str`: A string representing the float with human-friendly significant digits.

Notes

Switches to scientific notation for very large or very small numbers. The magnitude of the number is calculated using `math.log10(value)`.

asv_runner.util.**_human_time_units**

((‘ns’, 1e-09), (‘s’, 1e-06), (‘ms’, 0.001), (‘s’, 1), (‘m’, 60), (‘h’,), (‘d’,), (‘w’,), (‘y’,), (…

asv_runner.util.**human_time**(*seconds*, *err=None*)

Formats a duration in seconds into a human-friendly time string.

Depending on the number of seconds given, can be one of::

```
1w 3d
2d 4h
1h 5m
1m 4s
  15s
```

The representation is always exactly 6 characters long.

Parameters

seconds (int)
The number of seconds to represent.

err (float, optional)
If provided, formats the duration as "{value}±{err}", e.g., "1h±5m". It can be used to represent the uncertainty in the measurement.

Returns

str: A human-friendly representation of the given duration. If the duration is NaN, returns "n/a".

asv_runner._aux

Module Contents

Classes

SpecificImporter	Module importer that only allows loading a given module from the given path.

Functions

update_sys_path	Update sys.meta_path to include the SpecificImporter.
posix_redirect_output	Redirect stdout/stderr to a file, using posix dup2.
recvall	Receive data of given size from a socket connection.
set_cpu_affinity_from_params	Set CPU affinity based on the provided parameters.

API

class asv_runner._aux.**SpecificImporter**(*name*, *root*)
Module importer that only allows loading a given module from the given path.

Notes

Using this enables importing the asv benchmark suite without adding its parent directory to sys.path. The parent directory can in principle contain anything, including some version of the project module (common situation if asv.conf.json is on project repository top level).

Initialization

Initialize a new instance of `SpecificImporter`.

Parameters

name (`str`)
The name of the module to load.

root (`str`)
The path to the directory containing the module.

`find_spec`(*fullname*, *path*, *target*)
Find the module specification for the given module.

Parameters

fullname (`str`)
The fully qualified name of the module.

path (list or None)
The path for module search, or None if unavailable.

target (object)
The target object to import.

Returns

spec (`ModuleSpec` or None)
The module specification if the module is found, or None otherwise.

Notes

This method is called by the import system to find the module specification for the requested module. If the requested module matches the name of the SpecificImporter instance, it returns the module specification using the `importlib.machinery.PathFinder`.

`asv_runner._aux.update_sys_path`(*root*)
Update sys.meta_path to include the SpecificImporter.

Parameters

`root` (`str`): The path to the root directory.

Notes

This function inserts the SpecificImporter into the `sys.meta_path` at the beginning, allowing the module to be imported using the SpecificImporter when it is encountered during the import process.

`asv_runner._aux.`**`posix_redirect_output`**(*filename=None*, *permanent=True*)

Redirect stdout/stderr to a file, using posix `dup2`.

Parameters

filename (`str` or None, optional)
The name of the file to redirect the output to. If None, a temporary file will be created.

permanent (`bool`, optional)
Indicates whether the redirection is permanent or temporary. If False, the original stdout/stderr will be restored after the context is exited.

Yields

filename (`str`)
The name of the file where the output is redirected.

Notes

The function redirects the `stdout` and `stderr` streams to a file using the posix `dup2` function. It is typically used within a `with` statement to encapsulate the code block where the redirection is desired.

If `filename` is not provided, a temporary file will be created and used for redirection.

If `permanent` is `True`, the redirection will persist after the context is exited. If `False`, the original `stdout`/`stderr` will be restored.

`asv_runner._aux.`**`recvall`**(*sock*, *size*)

Receive data of given size from a socket connection.

Parameters

sock (socket object)
The socket connection to receive data from.

size (`int`)
The size of the data to receive, in bytes.

Returns

data (`bytes`)
The received data.

Raises

RuntimeError
If the data received from the socket is less than the specified size.

Notes

The function receives data from a socket connection in multiple chunks until the specified size is reached. It ensures that all the required data is received before returning.

If the received data size is less than the specified size, a `RuntimeError` is raised indicating the failure to receive the complete data.

`asv_runner._aux.`**`set_cpu_affinity_from_params`**`(`*extra_params*`)`

Set CPU affinity based on the provided parameters.

Parameters

extra_params (`dict` or `None`)
Additional parameters containing CPU affinity information.

Notes

This function attempts to set the CPU affinity for the current process based on the provided parameters. It uses the `set_cpu_affinity` function internally to perform the actual affinity setting.

If the `extra_params` dictionary contains a key "cpu_affinity" with a valid affinity list, the CPU affinity will be set accordingly.

Raises

BaseException
If setting the CPU affinity fails, an exception is raised and an error message is printed.

Example

```
extra_params = {"cpu_affinity": [0, 1]}
set_cpu_affinity_from_params(extra_params)
```

asv_runner.timing

Module Contents

Functions

_timing	Executes a timing benchmark.

API

asv_runner.timing.**_timing**(*argv*)

Executes a timing benchmark.

Parameters

argv (**list of str**)
Command line arguments.

Notes

This function parses the command line arguments, including options for setup, number of repeats, timing method, and output format (JSON or not). It selects the appropriate timing function based on the `--timer` argument.

It creates an instance of the `TimeBenchmark` class, with the provided statement to be executed, and runs it. The setup is provided from the `--setup` argument.

Once the benchmark is run, it computes the statistics of the results and formats the output. If the `--json` flag is not set, it prints the output in a human- readable format. Otherwise, it outputs the result, samples, and stats as a JSON.

asv_runner.check

Module Contents

Functions

_check	Checks all the discovered benchmarks in the provided benchmark directory.

API

asv_runner.check.**_check**(*args*)

Checks all the discovered benchmarks in the provided benchmark directory.

Parameters

args (**tuple**)
A tuple containing the benchmark directory.

Notes

This function updates the system path with the root directory of the benchmark suite. Then, it iterates over all benchmarks discovered in the root directory. For each benchmark, it calls the check method of the benchmark and updates the 'ok' flag.

If all benchmarks pass the check, it exits with a status code 0. If any benchmark fails, it exits with a status code 1.

asv_runner.setup_cache

Module Contents

Functions

_setup_cache	Sets up a cache for a benchmark and pickles it into a file.

API

asv_runner.setup_cache.**_setup_cache**(*args*)

Sets up a cache for a benchmark and pickles it into a file.

Parameters

args (**tuple**)
A tuple containing the benchmark directory, benchmark ID, and parameter string.

- benchmark_dir (str): The directory where the benchmarks are located.
- benchmark_id (str): The ID of the specific benchmark to be set up.
- params_str (str): A JSON string containing extra parameters for the benchmark.

Notes

This function sets up a cache for a specific benchmark and saves it into a pickle file.

First, it reads the extra parameters from the parameter string and sets the CPU affinity accordingly. Then, it retrieves the benchmark from the provided benchmark ID, and sets up its cache. The cache is then saved into a pickle file called 'cache.pickle'.

The function is used to prepare a cache for a benchmark in a separate process, before running the actual benchmark.

asv_runner.server

Module Contents

Functions

recvall	Receives data from a socket until the specified size of data has been received.
_run_server	Runs a server that executes benchmarks based on the received commands.

Data

wall_timer

API

asv_runner.server.**wall_timer**

None

asv_runner.server.**recvall**(*sock*, *size*)

Receives data from a socket until the specified size of data has been received.

Parameters

sock (**socket**)
The socket from which the data will be received. This socket should already be connected to the other end from which data is to be received.

size (**int**)
The total size of data to be received from the socket.

Returns

data (**bytes**)
The data received from the socket. The length of this data will be equal to the size specified.

Raises

RuntimeError
If the socket closed before the specified size of data could be received.

Notes

This function continuously receives data from the provided socket in a loop until the total length of the received data is equal to the specified size. If the socket closes before the specified size of data could be received, a `RuntimeError` is raised. The function returns the received data as a byte string.

`asv_runner.server._run_server(args)`

Runs a server that executes benchmarks based on the received commands.

Parameters

args (tuple)
A tuple containing the benchmark directory and socket name.

- `benchmark_dir` (`str`): The directory where the benchmarks are located.
- `socket_name` (`str`): The name of the UNIX socket to be used for
- communication.

Raises

RuntimeError
If the received command contains unknown data.

Notes

This function creates a server that listens on a UNIX socket for commands. It can perform two actions based on the received command: quit or preimport benchmarks.

If the command is "quit", the server stops running. If the command is "preimport", the function imports all the benchmarks in the specified directory, capturing all the I/O to a file during import. After the benchmarks are imported, the function sends the contents of the output file back through the socket.

If the action is not "quit" or "preimport", the function assumes it is a command to run a specific benchmark. It then runs the benchmark and waits for the results. It also handles a timeout for the benchmark execution and sends the results back through the socket.

The function continuously accepts new commands until it receives a "quit" command or a KeyboardInterrupt.

It uses UNIX domain sockets for inter-process communication. The name of the socket is passed as a parameter in `args`. The socket is created, bound to the socket name, and set to listen for connections. When a connection is accepted, the command is read from the socket, parsed, and executed accordingly. After executing the command, the server sends back the result through the socket and waits for the next command.

asv_runner.discovery

Module Contents

Functions

_get_benchmark	Retrieves benchmark function based on attribute name, module, class, and function.
disc_modules	Recursively imports a module and all sub-modules in the package.
disc_benchmarks	Discovers all benchmarks in a given directory tree, yielding Benchmark objects.
get_benchmark_from_name	Creates a benchmark from a fully-qualified benchmark name.
list_benchmarks	Lists all discovered benchmarks to a file pointer as JSON.
_discover	Discovers all benchmarks in the provided benchmark directory and lists them to a file.

API

asv_runner.discovery.**_get_benchmark**(*attr_name*, *module*, *klass*, *func*)

Retrieves benchmark function based on attribute name, module, class, and function.

Parameters

attr_name (str)
The attribute name of the function.

module (module)
The module where the function resides.

klass (class or None)
The class defining the function, or None if not applicable.

func (function)
The function to be benchmarked.

Returns

benchmark (Benchmark instance or None)
A benchmark instance with the name of the benchmark, the function to be benchmarked, and its sources. Returns None if no matching benchmark is found or the function is marked to be skipped.

Notes

The function tries to get the `benchmark_name` from `func`. If it fails, it uses `attr_name` to match with the name regex in the benchmark types. If a match is found, it creates a new benchmark instance and returns it. If no match is found or the function is marked to be skipped, it returns None.

asv_runner.discovery.**disc_modules**(*module_name*, *ignore_import_errors=False*)

Recursively imports a module and all sub-modules in the package.

Parameters

module_name (str)
The name of the module to import.

ignore_import_errors (bool, optional)
Whether to ignore import errors. Default is False.

Yields

module (module)
The imported module in the package tree.

Notes

This function imports the given module and yields it. If `ignore_import_errors` is set to True, the function will continue executing even if the import fails and will print the traceback. If `ignore_import_errors` is set to False and the import fails, the function will raise the error. After yielding the imported module, the function looks for sub-modules within the package of the imported module and recursively imports and yields them.

`asv_runner.discovery.disc_benchmarks(root, ignore_import_errors=False)`

Discovers all benchmarks in a given directory tree, yielding Benchmark objects.

Parameters

root (str)
The root of the directory tree where the function begins to search for benchmarks.

ignore_import_errors (bool, optional)
Specifies if import errors should be ignored. Default is False.

Yields

benchmark (Benchmark instance or None)
A benchmark instance containing the benchmark's name, the function to be benchmarked, and its sources if a matching benchmark is found.

Notes

For each class definition, the function searches for methods with a specific name. For each free function, it yields all functions with a specific name. The function initially imports all modules and submodules in the directory tree using the `disc_modules` function. Then, for each imported module, it searches for classes and functions that might be benchmarks. If it finds a class, it looks for methods within that class that could be benchmarks. If it finds a free function, it considers it as a potential benchmark. A potential benchmark is confirmed by the `_get_benchmark` function. If this function returns a benchmark instance, the instance is yielded.

`asv_runner.discovery.get_benchmark_from_name(root, name, extra_params=None)`

Creates a benchmark from a fully-qualified benchmark name.

Parameters

root (str)
Path to the root of a benchmark suite.

name (str)
Fully-qualified name of a specific benchmark.

extra_params (dict, optional)
Extra parameters to be added to the benchmark.

Returns

benchmark (Benchmark instance)
A benchmark instance created from the given fully-qualified benchmark name.

Raises

ValueError
If the provided benchmark ID is invalid or if the benchmark could not be found.

Notes

This function aims to create a benchmark from the given fully-qualified name. It splits the name using the "-" character. If "-" is present in the name, the string after the "-" is converted to an integer and is considered as the parameter index. If "-" is not present, the parameter index is set to None. The function then tries to directly import the benchmark function by guessing its import module name. If the benchmark is not found this way, the function searches for the benchmark in the directory tree root using `disc_benchmarks`. If the benchmark is still not found, it raises a ValueError. If extra parameters are provided, they are added to the benchmark.

`asv_runner.discovery.list_benchmarks(root, fp)`

Lists all discovered benchmarks to a file pointer as JSON.

Parameters

root (str)
Path to the root of a benchmark suite.

fp (file object)
File pointer where the JSON list of benchmarks should be written.

Notes

The function updates the system path with the root directory of the benchmark suite. Then, it iterates over all benchmarks discovered in the root directory. For each benchmark, it creates a dictionary containing all attributes of the benchmark that are of types `str`, `int`, `float`, `list`, `dict`, `bool` and don't start with an underscore `_`. These attribute dictionaries are then dumped as JSON into the file pointed by `fp`.

`asv_runner.discovery._discover(args)`

Discovers all benchmarks in the provided benchmark directory and lists them to a file.

Parameters

args (`tuple`)
A tuple containing benchmark directory and result file path.

Notes

The function takes a tuple as an argument. The first element of the tuple should be the path to the benchmark directory, and the second element should be the path to the result file. It opens the result file for writing and calls the `list_benchmarks` function with the benchmark directory and the file pointer of the result file.

CHAPTER

TWO

EXTERNAL PLUGIN LIST

Here are the existing external plugins which are supported by `asv` and `asv_runner` (pull requests welcome).

2.1 Benchmark Plugins

- `asv_bench_memray` enables `RayMyClass` or `ray_funcname` for peak memory as profiled by `memray`, which is able to handle native calls and traces every function call

CHAPTER

THREE

DEVELOPING BENCHMARKS

All benchmark plugins must follow a strict hierarchy:

- The package name must begin with `asv_bench`.
- Benchmark classes are defined in a `benchmarks` folder under the package module.
- Each exported new benchmark type has the `export_as_benchmark = [NAMEBenchmark]` attribute.

For more conventions, see the internally defined benchmark types within `asv_runner`.

CHAPTER

FOUR

CHANGELOG

All notable changes to this project will be documented in this file.

The format is based on Keep a Changelog, and this project adheres to Semantic Versioning.

This project uses *towncrier* and the changes for the upcoming release can be found in https://github.com/twisted/my-project/tree/main/changelog.d/.

4.1 0.2.0 - 11-02-2024

4.1.1 Other Changes and Additions

- `asv_runner` now uses `towncrier` to manage the changelog, also adds the changeglog to the generated documentation. (#38)
- The lowest supported version of `python` for building the `asv_runner` documentation is now `3.8`, since `3.7` has been EOL for many months now. (#39)

4.2 0.1.0 - 11-09-2023

4.2.1 Bug Fixes

- Default `max_time` is set to `60.0` seconds to fix `--quick`. (#29)
- `asv` will not try to access a missing `colorama` attribute. (#32)

4.2.2 Other Changes and Additions

- `pip-tools` and `pip-compile` are used to pin transitive dependencies for read the docs. (#31)

4.3 0.0.9 - 20-08-2023

4.3.1 New Features

- Adds a `skip_benchmark` decorator.

```
from asv_runner.benchmarks.helpers import skip_benchmark

@skip_benchmark
class TimeSuite:
    """
    An example benchmark that times the performance of various kinds
    of iterating over dictionaries in Python.
    """
    def setup(self):
        self.d = {}
        for x in range(500):
            self.d[x] = None

    def time_keys(self):
        for key in self.d.keys():
            pass

    def time_values(self):
        for value in self.d.values():
            pass

    def time_range(self):
        d = self.d
        for key in range(500):
            d[key]
```

Usage requires `asv 0.6.0`. (#13)

- Finely grained `skip_benchmark_if` and `skip_params_if` have been added.

```
from asv_runner.benchmarks.mark import skip_benchmark_if, skip_params_if
import datetime

class TimeSuite:
    """
    An example benchmark that times the performance of various kinds
    of iterating over dictionaries in Python.
    """
    params = [100, 200, 300, 400, 500]
    param_names = ["size"]

    def setup(self, size):
        self.d = {}
        for x in range(size):
            self.d[x] = None

    @skip_benchmark_if(datetime.datetime.now().hour >= 12)
```

(continues on next page)

(continued from previous page)

```
    def time_keys(self, size):
        for key in self.d.keys():
            pass

    @skip_benchmark_if(datetime.datetime.now().hour >= 12)
    def time_values(self, size):
        for value in self.d.values():
            pass

    @skip_benchmark_if(datetime.datetime.now().hour >= 12)
    def time_range(self, size):
        d = self.d
        for key in range(size):
            d[key]

    # Skip benchmarking when size is either 100 or 200 and the current hour is
12 or later.
    @skip_params_if([(100,), (200,)],
                    datetime.datetime.now().hour >= 12)
    def time_dict_update(self, size):
        d = self.d
        for i in range(size):
            d[i] = i
```

Usage requires `asv 0.6.0`. (#17)

- Benchmarks can now be parameterized using decorators.

```
import numpy as np
from asv_runner.benchmarks.mark import parameterize

@parameterize({"n":[10, 100]})
def time_sort(n):
    np.sort(np.random.rand(n))

@parameterize({'n': [10, 100], 'func_name': ['range', 'arange']})
def time_ranges_multi(n, func_name):
    f = {'range': range, 'arange': np.arange}[func_name]
    for i in f(n):
        pass

@parameterize({"size": [10, 100, 200]})
class TimeSuiteDecoratorSingle:
    def setup(self, size):
        self.d = {}
        for x in range(size):
            self.d[x] = None

    def time_keys(self, size):
        for key in self.d.keys():
            pass

    def time_values(self, size):
```

(continues on next page)

(continued from previous page)

```
        for value in self.d.values():
            pass

@parameterize({'n': [10, 100], 'func_name': ['range', 'arange']})
class TimeSuiteMultiDecorator:
    def time_ranges(self, n, func_name):
        f = {'range': range, 'arange': np.arange}[func_name]
        for i in f(n):
            pass
```

Usage requires `asv 0.6.0`. (#18)

- Benchmarks can now be skipped during execution.

```
from asv_runner.benchmarks.mark import skip_for_params, parameterize,
SkipNotImplemented

# Fast because no setup is called
class SimpleFast:
    params = ([False, True])
    param_names = ["ok"]

    @skip_for_params([(False, )])
    def time_failure(self, ok):
        if ok:
            x = 34.2**4.2

@parameterize({"ok": [False, True]})
class SimpleSlow:
    def time_failure(self, ok):
        if ok:
            x = 34.2**4.2
        else:
            raise SkipNotImplemented(f"{ok} is skipped")
```

Usage requires `asv 0.6.0`. (#20)

4.3.2 Bug Fixes

- It is possible to set a default timeout from `asv`. (#19)

4.3.3 Other Changes and Additions

- Documentation, both long-form and API level has been added. (#6)

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

a

INDEX

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

W