blackbench

Release 21.8a2

Richard Si

Aug 15, 2021

CONTENTS

CHAPTER

ONE

QUICKSTART

1.1 Installation

Prerequisite: Python 3.8 or higher[1]

Blackbench is available through PyPI so it's pretty easy to install.

```
python -m pip install blackbench
```

Your other option is installing via a VSC URL with pip. This will pull down the newest revision of blackbench (in other words, whatever is in `main`). Unreleased versions of blackbench should work just fine, but may have bugs and other half-implemented features.

```
python -m pip install git+git://github.com/ichard26/blackbench
```

Tip: You can enable tab completion following these instructions:

Add this to `~/.bashrc`:

```
eval "$(_BLACKBENCH_COMPLETE=bash_source blackbench)"
```

Add this to `~/.zshrc`:

```
eval "$(_BLACKBENCH_COMPLETE=zsh_source blackbench)"
```

Add this to `~/.config/fish/completions/blackbench.fish`:

```
eval (env _BLACKBENCH_COMPLETE=fish_source blackbench)
```

Oh and my apologies if your shell isn't supported, I'm just using Click's built-in framework so it's out of my control.

---

[1] I know Black's Python runtime requirement is probably lower (it's 3.6.2+ as of writing) but given this is a developer-focused tool, I consider it fine (especially since I don't see blackbench's requirement being bumped anytime soon).

1.2 Tuning your system

To get reliable results - which is needed so comparisons still make sense - the system should be tuned to avoid system jitter. While there's a lot of OS-specific / low-level tuning (eg. `isolcpu=1` on Linux) you can do to cut down on the benchmark noise, they can take a fair bit of effort to setup correctly (not to mention if they're even compatible with your system). So to avoid making this quick start more like a "long guide" here's some quick and easy things you can do to increase benchmark stability:

Warning: Note the suggested modifications may not be supported for your specific environment and also can be annoying to undo (a simple reboot should clear them though).

- Run `pyperf system tune`[2] - personally this command feels like magic, but in essence it tries to configure your system automatically according to this list of operations
- Use the pyperf `--affinity` option to pin worker processes to a specific CPU core (since it's a pyperf option, you'll need to precede it with `--` after your blackbench arguments)
- Try to leave your system alone while it's benchmarking as much as reasonably possible
- Oh and the classic advice of closing down unnecessary background processes still applies :)

If you're curious for more, don't worry! Further in the User Guide will be more detailed information on this topic.

1.3 Running benchmarks

1. Install one of the revisions of Black you'd like to compare.
2. Call the `run` command with a filepath to dump results as the argument.

```
dev@example:~/blackbench$ blackbench run normal.json -- --affinity 1
[*] Versions: blackbench: 21.7.dev2, pyperf: 2.2.0, black: 21.6b0
[*] Created temporary workdir at `/tmp/blackbench-workdir-ly0edox8`.
[*] Alright, let's start!
[*] Running `fmt-black/__init__` benchmark (1/17)
.....................
fmt-black/__init__: Mean +- std dev: 1.50 sec +- 0.05 sec
[*] Took 41.291 seconds.
[*] Running `fmt-black/brackets` benchmark (2/17)

[snipped ...]

[*] Cleaning up.
[*] Results dumped.
[*] Blackbench run finished in 373.794 seconds.
```

3. Repeat steps 1-2 until you have tested all of the revisions.

Note: The default configuration uses the `fmt` task and all targets (ie. tests standard Black behaviour with safety checks against all available code files). This can be pretty slow depending on how fast your machine is. A quick fix would be to use `--fast` to collect less values.

[2] Don't worry, pyperf should've been installed alongside blackbench since the latter depends on it.

1.4 Comparing runs

1. Run `pyperf compare_to` passing in the result files from the previous step.[3]

```
dev@example:~/blackbench$ pyperf compare_to normal.json with-esp.json
fmt-black/__init__: Mean +- std dev: [normal] 1.50 sec +- 0.05 sec -> [with-esp] 1.
↪68 sec +- 0.03 sec: 1.12x slower
fmt-black/brackets: Mean +- std dev: [normal] 479 ms +- 13 ms -> [with-esp] 515 ms␣
↪+- 11 ms: 1.07x slower
fmt-black/comments: Mean +- std dev: [normal] 382 ms +- 6 ms -> [with-esp] 400 ms +-
↪ 11 ms: 1.05x slower
fmt-black/mode: Mean +- std dev: [normal] 167 ms +- 3 ms -> [with-esp] 175 ms +- 5␣
↪ms: 1.05x slower
fmt-black/strings: Mean +- std dev: [normal] 282 ms +- 8 ms -> [with-esp] 298 ms +-␣
↪6 ms: 1.06x slower
fmt-dict-literal: Mean +- std dev: [normal] 227 ms +- 8 ms -> [with-esp] 244 ms +-␣
↪6 ms: 1.08x slower
fmt-list-literal: Mean +- std dev: [normal] 134 ms +- 4 ms -> [with-esp] 156 ms +-␣
↪19 ms: 1.17x slower
fmt-strings-list: Mean +- std dev: [normal] 43.2 ms +- 1.7 ms -> [with-esp] 184 ms␣
↪+- 4 ms: 4.25x slower

Benchmark hidden because not significant (9): fmt-black/linegen, fmt-black/lines,␣
↪fmt-black/nodes, fmt-black/output, fmt-comments, fmt-flit/install, fmt-flit/sdist,
↪ fmt-flit_core/config, fmt-nested

Geometric mean: 1.12x slower
```

2. Celebrate because you've successfully used blackbench to benchmark and compare different revisions of Black!

1.5 What's next?

See the rest of the User Guide for an indepth explanation and guide to using blackbench effectively, to learn blackbench's more advanced features, and overall get better at benchmarking. You can also take a look at the pyperf documentation since blackbench heavily depends on it for many parts of its functionality.

Thank you for using blackbench and I hope you make good use out of it!

---

[3] These numbers aren't actually representative of the slowdown experimental string processing causes, I got these numbers without taking any care to get stable results (so I could write these docs quickly).

CHAPTER

TWO

TASKS & TARGETS

Blackbench comes with tasks and targets. Tasks represent a specific task Black has do during formatting, and come with a specific template. Targets are the actual files Black (or more specifically the task's template / code which calls the relevant Black APIs) is run against to gauge performance. Together they create benchmarks that are ran using pyperf.

Why not just ship ready-to-go benchmark scripts instead? Well, there's a few issues with that: running static benchmarks limits the available configuration. Want to benchmark Black with the magic trailing comma disabled? You'd have to edit each benchmark which is annoying. And even if providing scripts for all of the different possible configurations was possible, they would be annoying to maintain.

Benchmarks ran by blackbench are named using the task and target they are based off. For example a benchmark using the `parse` task and the `strings-list` target would be named `parse-strings-list`. Oh and benchmarks based off micro targets are called also microbenchmarks (since the goal of checking the performance of a specific area transfers).

Below are details about the tasks and targets that ship with blackbench:

2.1 Tasks

- `fmt`: standard Black formatting behaviour although safety checks will **always** be run
- `fmt-fast`: like `fmt` but using `--fast` so safety checks are disabled
- `parse`: only do blib2to3 parsing

Important: The `fmt` task forces safety checks to run (by adding trailing newlines) or else it could skew the data in a nasty way. Normally safety checks only run if changes are made so there's a possibility one version of Black will have to do more work over the other one you're comparing against, totally throwing off the results for any sort of comparisons.

2.2 Targets

Targets are a bit more complex since there's two types: normal and micro. Normal targets represent real-world code (and aim to do a decent job to representing real-world use cases and performance). Micro targets are typically smaller and are focused on one area of black formatting (and mostly exist to measure performance in a specific area, like string processing).

Normal targets:

- `black/*`: source code files from Black 21.6b0 (9 targets)
- `flit/*` & `filt_core/*`: source code files from Flit 3.2.0 (3 targets)

Micro targets:

- `dict-literal`: a long dictionary literal
- `comments`: code that uses a lot of (maybe special) comments
- `list-literal`: a long list literal
- `nested`: nested functions, literals, if statements ... all the nested!
- `strings-list`: a list containing 100s of sometimes comma separated strings

2.3 Compatibility

While it would be great if blackbench could support all versions of Black, this is difficult if not impossible since tasks use internal components of Black's implementation. This provides a benchmarking speedup and is also unavoidable for lower level tasks like `parse`. Due to this, each task imposes restrictions to what version of Black their benchmarks can be run under:

- `fmt`, `fmt-fast`, and `parse`: >= 19.3b0

2.4 Useful commands

Blackbench does have a few commands that interact directly with tasks and targets:

`blackbench dump ${name}` Dumps the source code for a specific task or target.

```
dev@example:~$ blackbench dump fmt-fast
from pathlib import Path

import pyperf

import black

runner = pyperf.Runner()
code = Path(r"{target}").read_text(encoding="utf8")


def format_func(code):
    try:
        black.format_file_contents(code, fast=True, mode=black.FileMode({mode}))
    except black.NothingChanged:
        pass


runner.bench_func("{name}", format_func, code)
```

`blackbench info` Lists all of the built-in tasks and targets.

```
dev@example:~$ blackbench info
Tasks:
  1. fmt - Standard Black run although safety checks will *always* run
  2. fmt-fast - Standard Black run but safety checks are *disabled*
  3. parse - Only do blib2to3 parsing
```

(continues on next page)

(continued from previous page)

```
Normal targets:
  1. black/__init__ [1132 lines] - Black source code from 21.6b0
  2. black/brackets [334 lines] - Black source code from 21.6b0
  3. black/comments [272 lines] - Black source code from 21.6b0
  4. black/linegen [985 lines] - Black source code from 21.6b0
  5. black/lines [734 lines] - Black source code from 21.6b0
  6. black/mode [123 lines] - Black source code from 21.6b0
  7. black/nodes [843 lines] - Black source code from 21.6b0
  8. black/output [84 lines] - Black source code from 21.6b0
   9. black/strings [216 lines] - Black source code from 21.6b0
  10. flit/install [415 lines] - Flit source code from 3.2.0
  11. flit/sdist [216 lines] - Flit source code from 3.2.0
  12. flit_core/config [630 lines] - Flit source code from 3.2.0

Micro targets:
  1. dict-literal [150 lines] - A long dictionary literal
  2. comments [97 lines] - Code that uses a lot of (maybe special) comments
  3. list-literal [150 lines] - A long list literal
  4. nested [41 lines] - Nested functions, literals, if statements ... all the␣
↪nested!
  5. strings-list [52 lines] - A list containing 100s of sometimes comma separated␣
↪strings
```

CHAPTER

THREE

RUNNING BENCHMARKS

Pre-requisite: an installation of Black that's importable in the current environment (please make sure the task you're using *supports your installed version of Black*).

The simplest way of running benchmarks is to call the run command providing a filepath to dump results to:

```
dev@example:~/blackbench$ blackbench run example.json
[*] Versions: blackbench: 21.7.dev2, pyperf: 2.2.0, black: 21.7b0
[*] Created temporary workdir at `/tmp/blackbench-workdir-67vki43p`.
[*] Alright, let's start!
[*] Running `fmt-black/__init__` benchmark (1/17)
.....................
WARNING: the benchmark result may be unstable
* the standard deviation (546 ms) is 30% of the mean (1.84 sec)
* the maximum (4.64 sec) is 153% greater than the mean (1.84 sec)

Try to rerun the benchmark with more runs, values and/or loops.
Run 'python -m pyperf system tune' command to reduce the system jitter.
Use pyperf stats, pyperf dump and pyperf hist to analyze results.
Use --quiet option to hide these warnings.

fmt-black/__init__: Mean +- std dev: 1.84 sec +- 0.55 sec
[*] Took 166.059 seconds.

[snipped ...]

[*] Cleaning up.
[*] Results dumped.
[*] Blackbench run finished in 818.794 seconds.
```

Note how there's a "WARNING: the benchmark result may be unstable" line in the output. This leads perfectly into the next topic when running benchmarks: stability and reliability.

3.1 Benchmark stability

While blackbench is supposed to be rather easy to use, one must understand the basics of *stable* benchmarks and their importance. Stable as in the data doesn't vary all of the place for absolutely no good reason (it's sorta like avoiding flakey tests). Instable benchmarks don't produce quality data or allow for accurate comparisons.

For a good backgrounder on stable benchmarks I'd recommend Victor Stinner's "My journey to stable benchmark" series. In particular the My journey to stable benchmark, part 1 (system) and My journey to stable benchmark, part 3 (average) articles. But in general, pyperf has good documentation on tuning your system to increase benchmark stability.

> **Warning:** Note the suggested modifications may not be supported for your specific environment and also can be annoying to undo (a simple reboot should clear them though).

Some concrete advice is to 1) use pyperf's great system tuning features. Not only does it have the automagicial `pyperf system tune` command, there's the lovely `pyperf system show` command which emits revelant system information and even some advice to further tweak your system! 2) Even if you can't isolate a CPU core, always use CPU pinning via pyperf's `--affinity`. This avoids the noise caused by the worker process being constantly assigned and reassigned to different CPU cores over time. 3) If you feel like it, try learning pyperf's benchmark parameters and see what works well for you (eg. maybe two warmups is better than one for you!).

If you're curious what I, Richard aka @ichard26, do to tune my system in preparation, here's a summary:

System notes: it's a dual-core running Ubuntu 20.04 LTS :P

- Reboot
- At the boot menu, add the following Linux kernel parameters: `isolcpus=1`, `nohz_full=1`, and `rcu_nocbs=1`
- Once booted, run `pyperf system tune`
- Then run `my-custom-script.bash`:

  ```
  # This has to be run under a root shell
  echo userspace > /sys/devices/system/cpu/cpu0/cpufreq/scaling_governor
  echo userspace > /sys/devices/system/cpu/cpu1/cpufreq/scaling_governor
  echo 2100000 > /sys/devices/system/cpu/cpu0/cpufreq/scaling_setspeed
  echo 2100000 > /sys/devices/system/cpu/cpu1/cpufreq/scaling_setspeed
  echo 2100000 > /sys/devices/system/cpu/cpu0/cpufreq/scaling_min_freq
  echo 2100000 > /sys/devices/system/cpu/cpu1/cpufreq/scaling_min_freq
  echo 2100000 > /sys/devices/system/cpu/cpu0/cpufreq/scaling_max_freq
  echo 2100000 > /sys/devices/system/cpu/cpu1/cpufreq/scaling_max_freq
  echo 1 > /proc/sys/kernel/perf_event_max_sample_rate
  echo "System tuned :D"
  ```

  This exists because my laptop's cooling capacity isn't good enough to handle the `performance` scaling governor that `pyperf system tune` sets. Eventually the CPU frequency would gradually go down, killing any hope at reliable results. So instead I lock the CPU frequency at 2.1 GHz. There's also a perf event config but that's less cool :)

- Run `pyperf system show` to verify I haven't missed anything dumb

3.2 Task & target selection

By default, all targets will selected (i.e. `--targets all`) with the `fmt` task. If you'd like to use a different task and/or use a specific kind of targets, there's options for that:

`--task` Choices are `parse`, `fmt-fast`, and `fmt`.

`--targets` Choices are `micro`, `normal`, and `all`.

See also:

Tasks & targets

3.3 Blackbench's slowness

Blackbench can be quite slow, this is because pyperf favours rigourness over speed. Many data points are collected over a series of processes which while does increase the accuracy it also does increase total benchmark duration.

You can pass `--fast` (which is actually an alias for `-- --fast`) to ask pyperf to collect less values for faster result turnaround at the price of result quality. Although with a well tuned system, the reduction in benchmarking time is well worth the (not too bad) drop in result quality.

3.4 pyperf configuration

pyperf is the library handling the benchmarking work and while its defaults are excellent (and blackbench just leaves everything on default) sometimes you'll need to modify the benchmark settings for stability or time requirement reasons. It's possible to pass all[1] `pyperf.Runner` CLI options and flags.

Just call `blackbench run` with your usual arguments PLUS `--` and then any pyperf arguments. The `--` is strongly recommended since anything that comes after will be left unprocessed and won't be treated as options to blackbench.

Examples include:

```
$ blackbench run example.json -- --fast
```

```
$ blackbench run example2.json --task parse -- --affinity 3
```

```
$ blackbench run example3.json --targets micro --format-config "experimental_string_
↪processing=True" -- --values 3 --warmups 2
```

3.5 Benchmark customization

If you're using a format type task, you can use `--format-config` to pass custom formatting options to Black during benchmarking. The value is substituted into `black.FileMode({VALUE})` so it must be valid argument Python code. The substitution context has the Black package imported. For example, passing a custom line length can be done with `--format-config "line_length=79"`. The generated benchmark script will look something like this:

[1] Although note that not all options will play nicely with blackbench's integration with pyperf. Examples include `--help`, `--output`, and `--append`.

```
# In reality, there's some more supporting code, but it's irrelevant here.

import black

def format_func(code):
    try:
        black.format_file_contents(code, fast=True, mode=black.FileMode(line_length=79))
    except black.NothingChanged:
        pass

runner.bench_func("example-task-example-target", format_func, code)
```

CHAPTER

FOUR

ANALYZING RESULTS

Blackbench's output is actually JSON output directly from pyperf. The JSON file can be loaded as an instance of `pyperf.BenchmarkSuite`. Anyway, this means that all analyzing of the results will happen with pyperf directly. Don't worry, pyperf should have been installed alongside blackbench.

See also:

pyperf's excellent analyzing benchmark results docs.

4.1 Analyzing a single run

4.1.1 Summary

For a short summary you can use pyperf show and pass the JSON file.

```
dev@example:~/blackbench$ pyperf show normal.json
fmt-black/__init__: Mean +- std dev: 1.50 sec +- 0.05 sec
fmt-black/brackets: Mean +- std dev: 479 ms +- 13 ms
fmt-black/comments: Mean +- std dev: 382 ms +- 6 ms
fmt-black/linegen: Mean +- std dev: 1.52 sec +- 0.04 sec
fmt-black/lines: Mean +- std dev: 1.09 sec +- 0.05 sec
fmt-black/mode: Mean +- std dev: 167 ms +- 3 ms
fmt-black/nodes: Mean +- std dev: 1.23 sec +- 0.04 sec
fmt-black/output: Mean +- std dev: 161 ms +- 6 ms
fmt-black/strings: Mean +- std dev: 282 ms +- 8 ms
fmt-comments: Mean +- std dev: 163 ms +- 7 ms
fmt-dict-literal: Mean +- std dev: 227 ms +- 8 ms
fmt-flit/install: Mean +- std dev: 740 ms +- 75 ms
fmt-flit/sdist: Mean +- std dev: 381 ms +- 17 ms
fmt-flit_core/config: Mean +- std dev: 993 ms +- 48 ms
fmt-list-literal: Mean +- std dev: 134 ms +- 4 ms
fmt-nested: Mean +- std dev: 141 ms +- 29 ms
fmt-strings-list: Mean +- std dev: 43.2 ms +- 1.7 ms
```

4.1.2 Indepth statistics

For more indepth information pyperf stats works wonders[1]:

```
ichard26@acer-ubuntu:~/programming/oss/blackbench$ pyperf stats example.json
normal
======

Number of benchmarks: 17
Total duration: 2 min 6.1 sec
Start date: 2021-07-26 16:49:33
End date: 2021-07-26 16:52:24

fmt-black/__init__
------------------

Total duration: 18.6 sec
Start date: 2021-07-26 16:49:33
End date: 2021-07-26 16:49:55
Raw value minimum: 1.45 sec
Raw value maximum: 1.56 sec

Number of calibration run: 1
Number of run with values: 5
Total number of run: 6

Number of warmup per run: 1
Number of value per run: 1
Loop iterations per value: 1
Total number of values: 5

Minimum:         1.45 sec
Median +- MAD:   1.49 sec +- 0.04 sec
Mean +- std dev: 1.50 sec +- 0.05 sec
Maximum:         1.56 sec

  0th percentile: 1.45 sec (-4% of the mean) -- minimum
  5th percentile: 1.45 sec (-3% of the mean)
 25th percentile: 1.48 sec (-2% of the mean) -- Q1
 50th percentile: 1.49 sec (-1% of the mean) -- median
 75th percentile: 1.55 sec (+3% of the mean) -- Q3
 95th percentile: 1.55 sec (+3% of the mean)
100th percentile: 1.56 sec (+4% of the mean) -- maximum

Number of outlier (out of 1.38 sec..1.65 sec): 0

fmt-black/brackets
------------------

[snipped ...]
```

[1] I gave up trying to make my hastily gathered (I asked pyperf to collect like only five values per benchmark!) data look normal, please don't @ me if your data doesn't look like mine :P

4.1.3 Histogram

pyperf hist is rather useful if you're curious to how instable the data is:

```
ichard26@acer-ubuntu:~/programming/oss/blackbench$ pyperf hist example.json
fmt-black/__init__
==================

1.52 sec:  3 ######################
1.56 sec:  4 ############################
1.60 sec: 11 ################################################################################
↪###
1.64 sec: 11 ################################################################################
↪###
1.67 sec:  9 ##################################################################
1.71 sec:  8 ############################################################
1.75 sec:  5 ####################################
1.79 sec:  3 ######################
1.83 sec:  1 #######
1.87 sec:  0 |
1.91 sec:  1 #######
1.95 sec:  1 #######
1.99 sec:  0 |
2.03 sec:  1 #######
2.06 sec:  0 |
2.10 sec:  0 |
2.14 sec:  0 |
2.18 sec:  0 |
2.22 sec:  0 |
2.26 sec:  0 |
2.30 sec:  0 |
2.34 sec:  0 |
2.38 sec:  1 #######
2.41 sec:  0 |
2.45 sec:  0 |
2.49 sec:  1 #######

fmt-black/brackets
==================

[snipped ...]
```

Tip: You can extract the results for a single benchmark via [`pyperf convert in.json --include-benchmark "${BENCHMARK}" -o out.json`][pyperf-convert] . Also, most pyperf commands should support `--benchmark` to select only one or a few benchmarks when passed in a BenchmarkSuite JSON file.

Tip: If you need the source for either a task or a target for even deeper analysis, you can call `blackbench dump ${name}`.

4.2 Comparing multiple runs

Comparisons between different runs can be done via pyperf compare_to. You can pass as many files as you'd wish, although note the order to ensure the deltas make sense.

```
dev@example:~/blackbench$ pyperf compare_to normal.json with-esp.json
fmt-black/__init__: Mean +- std dev: [normal] 1.50 sec +- 0.05 sec -> [with-esp] 1.68␣
↪sec +- 0.03 sec: 1.12x slower
fmt-black/brackets: Mean +- std dev: [normal] 479 ms +- 13 ms -> [with-esp] 515 ms +- 11␣
↪ms: 1.07x slower
fmt-black/comments: Mean +- std dev: [normal] 382 ms +- 6 ms -> [with-esp] 400 ms +- 11␣
↪ms: 1.05x slower
fmt-black/mode: Mean +- std dev: [normal] 167 ms +- 3 ms -> [with-esp] 175 ms +- 5 ms: 1.
↪05x slower
fmt-black/strings: Mean +- std dev: [normal] 282 ms +- 8 ms -> [with-esp] 298 ms +- 6␣
↪ms: 1.06x slower
fmt-dict-literal: Mean +- std dev: [normal] 227 ms +- 8 ms -> [with-esp] 244 ms +- 6 ms:␣
↪1.08x slower
fmt-list-literal: Mean +- std dev: [normal] 134 ms +- 4 ms -> [with-esp] 156 ms +- 19␣
↪ms: 1.17x slower
fmt-strings-list: Mean +- std dev: [normal] 43.2 ms +- 1.7 ms -> [with-esp] 184 ms +- 4␣
↪ms: 4.25x slower

Benchmark hidden because not significant (9): fmt-black/linegen, fmt-black/lines, fmt-
↪black/nodes, fmt-black/output, fmt-comments, fmt-flit/install, fmt-flit/sdist, fmt-
↪flit_core/config, fmt-nested

Geometric mean: 1.12x slower
```

Note how pyperf determines whether two samples differ significantly (using a Student's two-sample, two-tailed t-test with alpha equals to 0.95). This helps out a lot by ignoring non-meaningful differences, but there's more to know! Getting stable numbers is really hard, so there's a possibility "significant" results are still just noise (or are actual results, but are so small to be meaningless). In this case applying a cutoff might be a good idea (you can ask pyperf to do this for you via `--min-speed`). What cutoff to use depends on what benchmarks you ran - a 5% perf improvement on a microbenchmark most likely isn't as meaningful as one on a (normal) benchmark, AND how stable your data was (if you system was very noisy then maybe the great results you're seeing aren't actually real ...). One final tip is to use the "Geometric mean" value, if you see a general speedup by 10%, then it seems likely you got a nice win on your hands!

4.2.1 Table view

While's compare_to's default format is neatly compact, it can be a bit hard to parse. Using `--table` fixes that:

```
dev@example:~/blackbench$ pyperf compare_to normal.json with-esp.json --table
+--------------------+----------+------------------------+
| Benchmark          | normal   | with-esp               |
+====================+==========+========================+
| fmt-black/__init__ | 1.50 sec | 1.68 sec: 1.12x slower |
+--------------------+----------+------------------------+
| fmt-black/brackets | 479 ms   | 515 ms: 1.07x slower   |
+--------------------+----------+------------------------+
| fmt-black/comments | 382 ms   | 400 ms: 1.05x slower   |
```

(continues on next page)

(continued from previous page)

```
+--------------------+----------+------------------------+
| fmt-black/mode     | 167 ms   | 175 ms: 1.05x slower   |
+--------------------+----------+------------------------+
| fmt-black/strings  | 282 ms   | 298 ms: 1.06x slower   |
+--------------------+----------+------------------------+
| fmt-dict-literal   | 227 ms   | 244 ms: 1.08x slower   |
+--------------------+----------+------------------------+
| fmt-list-literal   | 134 ms   | 156 ms: 1.17x slower   |
+--------------------+----------+------------------------+
| fmt-strings-list   | 43.2 ms  | 184 ms: 4.25x slower   |
+--------------------+----------+------------------------+
| Geometric mean     | (ref)    | 1.12x slower           |
+--------------------+----------+------------------------+

Benchmark hidden because not significant (9): fmt-black/linegen, fmt-black/lines, fmt-
↪black/nodes, fmt-black/output, fmt-comments, fmt-flit/install, fmt-flit/sdist, fmt-
↪flit_core/config, fmt-nested
```

Tip: Passing the `-G` flag causes compare_to's output to be organized in groups of faster/slower/not-significant. This usually makes the output more readable.

Todo: Provide more examples and also improve their quality. Perhaps also add some more prose and discussion on then using this data to make inferences and conclusions (as much as that makes this ever closer to some sort of statistics 101 primer).

Todo: Provide an example demonstrating `pyperf metadata` once blackbench injects useful metadata.

CHAPTER

FIVE

CONTRIBUTING

Hey thanks for considering contributing to blackbench! It's awesome to see you here. Anyway, there's a lot of ways you can contribute to blackbench, whether that's opening a constructive bug report or feature request, writing docs, to actually writing some code.

5.1 Setting up development environment

5.1.1 Requirements

- **CPython 3.8 or higher**
- **Nox**

  All development commands are managed by Nox which provides automated environment provisioning. For us, it's basically a task runner. I strongly recommend using pipx.

- **pre-commit** *[optional]*

  pre-commit runs Git commit hooks that assert a baseline of quality. Once again, pipx is encouraged.

5.1.2 Steps

1. Fork the blackbench project on GitHub if you haven't already done so.
2. Clone your fork and `cd` into the resulting directory.

```
dev@example:~$ git clone https://github.com/${USERNAME}/blackbench.git
Cloning into 'blackbench'...
remote: Enumerating objects: 244, done.
remote: Counting objects: 100% (244/244), done.
remote: Compressing objects: 100% (141/141), done.
remote: Total 244 (delta 85), reused 212 (delta 58), pack-reused 0
Receiving objects: 100% (244/244), 154.29 KiB | 360.00 KiB/s, done.
Resolving deltas: 100% (85/85), done.
dev@example:~$ cd blackbench
dev@example:~/blackbench$
```

3. Add the fork's parent (ie. upstream) as an additional remote.

```
dev@example~/blackbench$ git remote add upstream https://github.com/ichard26/
↪blackbench.git
```

4. If pre-commit is available, install the Git pre commit hooks.

```
dev@example:~/blackbench$ pre-commit install
pre-commit installed at .git/hooks/pre-commit
```

5. Run the `setup-env` session with Nox.

   You'll use this virtual environment to run blackbench from source (ie, during manual testing). Flit's editable install feature will be used so don't worry about recreating the environment after changing something, they should be instantly reflected!

```
dev@example:~/blackbench$ nox -s setup-env
nox > Running session setup-env
nox > /home/dev/.local/pipx/venvs/nox/bin/python -m virtualenv /home/dev/blackbench/
↪venv
nox > /home/dev/blackbench/venv/bin/python -m pip install flit
nox > /home/dev/blackbench/venv/bin/python -m flit install --deps production --
↪symlink
nox > Virtual environment at project root named `venv` ready to go!
nox > Session setup-env was successful.
```

   Important: The environment created by `setup-env` is **only** for manual testing. Automated testing or other development commands should be done via the sessions configured in `noxfile.py`. This is why the created virtual environment only has the dependencies needed to run blackbench.

6. Activate the created virtual environment.

```
dev@example:~/blackbench$ . venv/bin/activate
(venv) dev@example:~/blackbench$
```

```
dev@example:~/blackbench$ . venv/bin/activate.zsh
(venv) dev@example:~/blackbench$
```

```
dev@example:~/blackbench$ source venv/bin/activate.fish
(venv) dev@example:~/blackbench$
```

```
C:\Users\dev\blackbench> venv\Scripts\activate
(venv) C:\Users\dev\blackbench>
```

7. Celebrate! … and then get to work on that change you've been thinking about :)

5.2 Development commands

As already mentioned, Nox is basically a task runner here. Most development commands are already configured in `noxfile.py` so running them correctly is easy as can be.

All the Nox sessions should support both `-r` and `-R`, so if the sessions are too slow (especially for rapid iteration during development), try of them. The only major exception is `setup-env` but that one doesn't use a Nox-provisioned environment anyway.

Also, it's possible to pass extra arguments to any the sessions' main command. Want to add `-k "not provided"` to the pytest run in `tests`? That's possible via `nox -s tests -- -k "not provided"`.

See also:

[Nox: Command-line usage][nox-usage].

5.2.1 Testing

To run the test suite, just run the `tests` session:

```
$ nox -s tests
```

If you want to collect coverage too, there's the `tests-cov` sessions for that:

```
$ nox -s tests-cov
```

Nox has one more awesome feature and that's making it really easy to run the test suite against multiple versions of Python at once. Matter of fact, the commands above will make Nox run the test session for every supported Python version it can find. You can select a single version by prepending the version (eg. `nox -s tests-3.9`).

Tip: If you need to run the test suite with a specific version of Black you can use the `--black-req` option. Eg. `nox -s tests-3.8 -- --black-req "black==21.5b2"`. Note that the `--` is important since the option was implemented at the session level and is 100% custom.

5.2.2 Linting

Calling pre-commit to run linters is as simple as:

```
$ nox -s lint
```

5.2.3 Docs

There's two sessions for documentation, which one you choose depends on what your goal is. If you're looking to do a complete and clean (re)build of the documentation, just run the aptly named `docs` session:

```
$ nox -s docs
```

BUT, if you're actively making changes to the documentation, having it automatically rebuild and refresh on changes will make your life easier. That's available using:

```
$ nox -s docs-live
```

Once the first build has been completed, there should be a link that serves the built documentation. As mentioned, the page will automagically refresh on changes!

5.3 PR guidelines

To make it easier for all of us to collaborate and get your PR merged, there's a few guidelines to be noted:

- If your PR has user-facing changes (eg. new target, bugfix), please add a changelog entry.
- Your PR should try to maintain excellent project coverage, although this isn't a blocker.
- Please include an explanation for the changes (and maybe a summary too if complicated enough) in the commit message.
- If CI fails, please address it, especially if the test suite failed since compatibility with muliple systems and environments must be maintained.
- You should expect somesort of response within three days. If not, feel free to ping @ichard26.

5.4 Getting help

If you get stuck, don't hesistate to ask for help on the relevant issue or PR. Alternatively, we can talk in the #black-formatter[1] text channel on Python Discord. Here's an invite link!

5.5 Appendix A: area-specific notes

5.5.1 Adding a new target

I'm looking for two kinds of targets: "normal" and "micro" . Normal targets should represent real-world code (so the benchmarking data actually represents real-world performance). Micro targets should be small and are focused on one area of Black formatting (and mostly exist to measure performance in a specific area, like string processing).

In terms of guidelines: normal targets shouldn't be bigger than ~2000 lines (this is to keep time requirements to run the benchmark based off the target manageable), and micro targets shouldn't be bigger than ~400 lines. Oh and for micro targets, make sure their focus hasn't been already covered by another target.

5.5.2 Release process

Before you fear what lies in front of you please know that the release process was designed to be simple and lightweight. The fact you're doing one in the first place is awesome and your time should be treated well! So in pursuit of that, here's the blackbench release process:

Note: You don't have to follow these steps carefully, they're more like guidelines that aim to make the 99% case easy. I'm sure there's situations this release process won't work and in that case, just use your best judgement.

---

1. Once you've decided that a release is due, please verify the following things:
    - the changelog has at least one entry (unless you're doing a post-release or something like that)
    - CI for the main branch is all green

[1] I know it's specifically for Black, but blackbench is a development tool for Black so I consider it acceptable - although I never asked … but then again, I am a maintainer of Black so yeah :p

2. If you don't have a local development environment, either setup one up or just make sure you have flit[2] ready to go
3. Checkout main and/or cleanup your local repository
4. Run `flit -s do-release -- <version>` with this release's version. The Nox session will handle the rest by:
    - checking the local repository is reasonably clean
    - updating both the version string and changelog to include the new version and today's date
    - commiting those changes and then tagging the commit
    - checking out the repository in a temporary isolated directory
    - running the `flit publish` command
    - updating the version string and changelog again for development
    - and finally commiting those changes
5. Push the newly created commits and tag to the GitHub repository
6. Go get a coffee or something, you just did a release! Congrats!

---

[2] I'd strongly recommend also setting up pre-commit so any dumb mistakes by the release automation is caught before release, but the release automation shouldn't be buggy so it's your call.

CHAPTER

SIX

CHANGELOG

6.1 21.8a2

Date of release: August 15th, 2021

Bugfixes & enhancements:

- Fixed broken (whoops!) documentation link and classifiers in project metadata.
- Fixed a bug with the `black.FileMode` configuration pre-check which caused it to crash with mypyc compiled Black.

6.2 21.8a1

Date of release: August 14th, 2021

Bugfixes & enhancements:

- Fixed the pyperf requirement which was originally pinning pyperf to 2.0.0 by accident.
- Added pre-checks for both pyperf arguments and –format-config configuration so issues are caught earlier.
- The parse task now supports installations of Black as old as 19.3b0!
- Additional metadata is now injected into the pyperf JSON files before being dumped.
- The `comments` and `nested` micro targets were tweaked to be more microbenchmark-y.

Project:

- The release process and automation is now ready for use. Which is great since this is the first release planned to be published to PyPI!

6.3 21.7.dev2

Date of release: July 27th, 2021

Bugfixes & enhancements:

- Use posix slashes for target and benchmark names to avoid accidental newlines leading to crashes on Windows
- Added `dump` command to easily query the source code for any built-in task or target
- Force safety checks to run under the `format` task to avoid *occasional but bad skewing of the data*

- Verify that Black is actually available in the current environment before running any benchmarks (and emit a human readable error instead of the original long tracebacky error)
- Arguments can now be passed to the underlying pyperf processes by adding `--` followed by such arguments. Useful for debugging stability or slowness issues.
- The command line interface UX has been improved by 1) much prettier help outputs thanks to cloup, and 2) official autocompletion support.
- The `--targets` (or now `-t`!) option can now also specify a specific target. You can also now repeat the option to additively choose your own custom collection of targets.
- The target name now doesn't include the suffix (eg. `black/cache.py` -> `black/cache`). Also the benchmark name format has been changed to `$task-name-$target-name` (eg. `[parse]-[black/cache.py]` -> `parse-black/cache`). Finally `format` and `format-fast` were renamed to `fmt` and `fmt-fast` respectively. These changes should make the names easier to use (even if less pretty!).
- Redid the targets collection to balance out normal vs micro and also add non-pre-black-formatted code! Run `blackbench info` for more!
- The output from the `info` command is now colorful and more informational. Line counts and descriptions are now also emitted.
- The output emitted during the `run` command is now colourful and a bit more human :)

Project:

- Created initial test suite to make sure blackbench isn't broken on all supported platforms
- Initial documentation including user guide and contributing docs have been written and are hosted on ReadTheDocs.

6.4 21.6.dev1

Date of release: June 14th, 2021

Initial development version of blackbench. Unsurprisingly it is functional but extremely limited. Windows support is actually straight up broken due to a bug :P

CHAPTER

SEVEN

ACKNOWLEDGEMENTS

Maintainers:

- Richard S. (@ichard26)

Blackbench also sees outside contributions whose contributors I greatly appreciate. A list of all contributors can be found on the repo's insights page.

Finally, this project wouldn't have existed if it wasn't for black which I help to maintain. Thank you goes to all maintainers and contributors to Black, for both the great tool and this fun project!

A benchmarking suite for Black, the Python code formatter. It's intended to help quantify changes in performance between versions of Black in a robust and repeatable manner. Especially useful for verifying a patch doesn't introduce performance regressions.

Reliable at its core

Under the hood, blackbench uses pyperf to handle the benchmarking heavylifting. The pyperf toolkit was designed with benchmark stability as its number one goal. This is transferred to blackbench, *so as long the system is properly tuned, the results can be safely considered reliable.*

Customizable benchmarks

Blackbench is really a collection of targets and task templates. Benchmarks are generated on the fly using the task's template as the base and the targets as the profiling data. Want to benchmark Black with experimental string processing on? A simple option and you're good to go!

Ready-to-go & complete

Blackbench comes with pre-curated tasks and targets, allowing for simplified yet complete benchmarking of Black. Notably, blackbench has both normal and micro targets to measure general and specific performance respectively.

Comparable results

Due to the pyperf base, the benchmarking results are in JSON. It's standard pyperf output and it's expected that the data analysis is performed using pyperf directly with its excellent analysis features.

CHAPTER

EIGHT

EXAMPLE RUN

```
dev@example:~/blackbench$ blackbench run mypyc-opt1.json --fast --task parse --targets␣
↪micro -- --affinity 1
[*] Versions: blackbench: 21.7.dev2, pyperf: 2.2.0, black: 21.7b0
[*] Created temporary workdir at `/tmp/blackbench-workdir-c5hoese9`.
[*] Alright, let's start!
[*] Running `parse-comments` microbenchmark (1/5)
.....
parse-comments: Mean +- std dev: 34.1 ms +- 1.0 ms
[*] Took 9.876 seconds.
[*] Running `parse-dict-literal` microbenchmark (2/5)
.....
parse-dict-literal: Mean +- std dev: 37.7 ms +- 2.4 ms
[*] Took 10.548 seconds.
[*] Running `parse-list-literal` microbenchmark (3/5)
.....
parse-list-literal: Mean +- std dev: 21.8 ms +- 2.4 ms
[*] Took 11.154 seconds.
[*] Running `parse-nested` microbenchmark (4/5)
.....
parse-nested: Mean +- std dev: 20.5 ms +- 1.2 ms
[*] Took 10.79 seconds.
[*] Running `parse-strings-list` microbenchmark (5/5)
.....
parse-strings-list: Mean +- std dev: 5.71 ms +- 1.09 ms
[*] Took 11.588 seconds.
[*] Cleaning up.
[*] Results dumped.
[*] Blackbench run finished in 54.139 seconds.
```

A breakdown of what's happening here:

- `mypyc-opt1.json`: the filepath to save the results
- `--task parse`: the timing workload is initial blib2to3 parsing
- `--targets micro`: only run microbenchmarks (that perform the selected task)
- `--fast`: collect less values so results are ready sooner
- everything after `--`: arguments passed to the underlying pyperf process

CHAPTER

NINE

LICENSE

Blackbench: MIT.

Targets based off real code maintain their original license. Please check the directory containing the target in question for a license file.