Documentation for Black

Release 18.3a4

Łukasz Langa and contributors to Black

Mar 30, 2018

Contents

By using *Black*, you agree to cease control over minutiae of hand-formatting. In return, *Black* gives you speed, determinism, and freedom from *pycodestyle* nagging about formatting. You will save time and mental energy for more important matters.

Black makes code review faster by producing the smallest diffs possible. Blackened code looks the same regardless of the project you're reading. Formatting becomes transparent after a while and you can focus on the content instead.

Note: Black is an early pre-release.

CHAPTER 1

Testimonials

Dusty Phillips, writer:

> *Black is opinionated so you don't have to be.*

Hynek Schlawack, creator of attrs, core developer of Twisted and CPython:

> *An auto-formatter that doesn't suck is all I want for Xmas!*

Carl Meyer, Django core developer:

> *At least the name is good.*

Kenneth Reitz, creator of requests and pipenv:

> *This vastly improves the formatting of our code. Thanks a ton!*

CHAPTER 2

Contents

2.1 Installation and Usage

2.1.1 Installation

Black can be installed by running `pip install black`. It requires Python 3.6.0+ to run but you can reformat Python 2 code with it, too.

2.1.2 Usage

To get started right away with sensible defaults:

```
black {source_file_or_directory}
```

2.1.3 Command line options

Black doesn't provide many options. You can list them by running `black --help`:

```
black [OPTIONS] [SRC]...

Options:
  -l, --line-length INTEGER   Where to wrap around.  [default: 88]
  --check                     Don't write back the files, just return the
                              status.  Return code 0 means nothing would
                              change.  Return code 1 means some files would be
                              reformatted.  Return code 123 means there was an
                              internal error.
  --fast / --safe             If --fast given, skip temporary sanity checks.
                              [default: --safe]
  --version                   Show the version and exit.
  --help                      Show this message and exit.
```

Black is a well-behaved Unix-style command-line tool:

- it does nothing if no sources are passed to it;
- it will read from standard input and write to standard output if `-` is used as the filename;
- it only outputs messages to users on standard error;
- exits with code 0 unless an internal error occured (or `--check` was used).

2.1.4 NOTE: This is an early pre-release

Black can already successfully format itself and the standard library. It also sports a decent test suite. However, it is still very new. Things will probably be wonky for a while. This is made explicit by the "Alpha" trove classifier, as well as by the "a" in the version number. What this means for you is that **until the formatter becomes stable, you should expect some formatting to change in the future**.

Also, as a temporary safety measure, *Black* will check that the reformatted code still produces a valid AST that is equivalent to the original. This slows it down. If you're feeling confident, use `--fast`.

2.2 The *Black* code style

Black reformats entire files in place. It is not configurable. It doesn't take previous formatting into account. It doesn't reformat blocks that start with `# fmt: off` and end with `# fmt: on`. It also recognizes YAPF's block comments to the same effect, as a courtesy for straddling code.

2.2.1 How *Black* wraps lines

Black ignores previous formatting and applies uniform horizontal and vertical whitespace to your code. The rules for horizontal whitespace are pretty obvious and can be summarized as: do whatever makes `pycodestyle` happy. The coding style used by *Black* can be viewed as a strict subset of PEP 8.

As for vertical whitespace, *Black* tries to render one full expression or simple statement per line. If this fits the allotted line length, great.

```
# in:

l = [1,
     2,
     3,
]

# out:

l = [1, 2, 3]
```

If not, *Black* will look at the contents of the first outer matching brackets and put that in a separate indented line.

```
# in:

l = [[n for n in list_bosses()], [n for n in list_employees()]]

# out:

```

(continues on next page)

(continued from previous page)

```
l = [
    [n for n in list_bosses()], [n for n in list_employees()]
]
```

If that still doesn't fit the bill, it will decompose the internal expression further using the same rule, indenting matching brackets every time. If the contents of the matching brackets pair are comma-separated (like an argument list, or a dict literal, and so on) then *Black* will first try to keep them on the same line with the matching brackets. If that doesn't work, it will put all of them in separate lines.

```
# in:

def very_important_function(template: str, *variables, file: os.PathLike, debug: bool␣
↪= False):
    """Applies `variables` to the `template` and writes to `file`."""
    with open(file, 'w') as f:
        ...

# out:

def very_important_function(
    template: str,
    *variables,
    file: os.PathLike,
    debug: bool = False,
):
    """Applies `variables` to the `template` and writes to `file`."""
    with open(file, 'w') as f:
        ...
```

You might have noticed that closing brackets are always dedented and that a trailing comma is always added. Such formatting produces smaller diffs; when you add or remove an element, it's always just one line. Also, having the closing bracket dedented provides a clear delimiter between two distinct sections of the code that otherwise share the same indentation level (like the arguments list and the docstring in the example above).

2.2.2 Line length

You probably noticed the peculiar default line length. *Black* defaults to 88 characters per line, which happens to be 10% over 80. This number was found to produce significantly shorter files than sticking with 80 (the most popular), or even 79 (used by the standard library). In general, 90-ish seems like the wise choice.

If you're paid by the line of code you write, you can pass `--line-length` with a lower number. *Black* will try to respect that. However, sometimes it won't be able to without breaking other rules. In those rare cases, auto-formatted code will exceed your allotted limit.

You can also increase it, but remember that people with sight disabilities find it harder to work with line lengths exceeding 100 characters. It also adversely affects side-by-side diff review on typical screen resolutions. Long lines also make it harder to present code neatly in documentation or talk slides.

If you're using Flake8, you can bump `max-line-length` to 88 and forget about it. Alternatively, use Bugbear's B950 warning instead of E501 and keep the max line length at 80 which you are probably already using. You'd do it like this:

```
[flake8]
max-line-length = 80
...
```

(continues on next page)

(continued from previous page)

```
select = C,E,F,W,B,B950
ignore = E501
```

You'll find *Black*'s own .flake8 config file is configured like this. If you're curious about the reasoning behind B950, Bugbear's documentation explains it. The tl;dr is "it's like highway speed limits, we won't bother you if you overdo it by a few km/h".

2.2.3 Empty lines

Black avoids spurious vertical whitespace. This is in the spirit of PEP 8 which says that in-function vertical whitespace should only be used sparingly. One exception is control flow statements: *Black* will always emit an extra empty line after `return`, `raise`, `break`, `continue`, and `yield`. This is to make changes in control flow more prominent to readers of your code.

Black will allow single empty lines inside functions, and single and double empty lines on module level left by the original editors, except when they're within parenthesized expressions. Since such expressions are always reformatted to fit minimal space, this whitespace is lost.

It will also insert proper spacing before and after function definitions. It's one line before and after inner functions and two lines before and after module-level functions. *Black* will put those empty lines also between the function definition and any standalone comments that immediately precede the given function. If you want to comment on the entire function, use a docstring or put a leading comment in the function body.

2.2.4 Trailing commas

Black will add trailing commas to expressions that are split by comma where each element is on its own line. This includes function signatures.

Unnecessary trailing commas are removed if an expression fits in one line. This makes it 1% more likely that your line won't exceed the allotted line length limit. Moreover, in this scenario, if you added another argument to your call, you'd probably fit it in the same line anyway. That doesn't make diffs any larger.

One exception to removing trailing commas is tuple expressions with just one element. In this case *Black* won't touch the single trailing comma as this would unexpectedly change the underlying data type. Note that this is also the case when commas are used while indexing. This is a tuple in disguise: `numpy_array[3, ]`.

One exception to adding trailing commas is function signatures containing `*`, `*args`, or `**kwargs`. In this case a trailing comma is only safe to use on Python 3.6. *Black* will detect if your file is already 3.6+ only and use trailing commas in this situation. If you wonder how it knows, it looks for f-strings and existing use of trailing commas in function signatures that have stars in them. In other words, if you'd like a trailing comma in this situation and *Black* didn't recognize it was safe to do so, put it there manually and *Black* will keep it.

2.3 Editor integration

2.3.1 Emacs

Use proofit404/blacken.

2.3.2 Vim

Commands and shortcuts:

- `,=` or `:Black` to format the entire file (ranges not supported);
- `:BlackUpgrade` to upgrade *Black* inside the virtualenv;
- `:BlackVersion` to get the current version of *Black* inside the virtualenv.

Configuration:

- `g:black_fast` (defaults to `0`)
- `g:black_linelength` (defaults to `88`)
- `g:black_virtualenv` (defaults to `~/.vim/black`)

To install, copy the plugin from vim/plugin/black.vim. Let me know if this requires any changes to work with Vim 8's builtin `packadd`, or Pathogen, or Vundle, and so on.

This plugin **requires Vim 7.0+ built with Python 3.6+ support**. It needs Python 3.6 to be able to run *Black* inside the Vim process which is much faster than calling an external command.

On first run, the plugin creates its own virtualenv using the right Python version and automatically installs *Black*. You can upgrade it later by calling `:BlackUpgrade` and restarting Vim.

If you need to do anything special to make your virtualenv work and install *Black* (for example you want to run a version from master), just create a virtualenv manually and point `g:black_virtualenv` to it. The plugin will use it.

How to get Vim with Python 3.6? On Ubuntu 17.10 Vim comes with Python 3.6 by default. On macOS with HomeBrew run: `brew install vim --with-python3`. When building Vim from source, use: `./configure --enable-python3interp=yes`. There's many guides online how to do this.

2.3.3 Visual Studio Code

Use joslarson.black-vscode.

2.3.4 Other editors

Atom/Nuclide integration is planned by the author, others will require external contributions.

Patches welcome!

Any tool that can pipe code through *Black* using its stdio mode (just use `-` as the file name). The formatted code will be returned on stdout (unless `--check` was passed). *Black* will still emit messages on stderr but that shouldn't affect your use case.

This can be used for example with PyCharm's File Watchers.

2.4 Contributing to Black

Welcome! Happy to see you willing to make the project better. Have you read the entire user documentation yet?

2.4.1 Bird's eye view

In terms of inspiration, *Black* is about as configurable as *gofmt* and *rustfmt* are. This is deliberate.

Bug reports and fixes are always welcome! Please follow the issue template on GitHub for best results.

Before you suggest a new feature or configuration knob, ask yourself why you want it. If it enables better integration with some workflow, fixes an inconsistency, speeds things up, and so on - go for it! On the other hand, if your answer is "because I don't like a particular formatting" then you're not ready to embrace *Black* yet. Such changes are unlikely to get accepted. You can still try but prepare to be disappointed.

2.4.2 Technicalities

Development on the latest version of Python is preferred. As of this writing it's 3.6.4. You can use any operating system. I am using macOS myself and CentOS at work.

Install all development dependencies using:

```
$ pipenv install --dev
```

If you haven't used `pipenv` before but are comfortable with virtualenvs, just run `pip install pipenv` in the virtualenv you're already using and invoke the command above from the cloned Black repo. It will do the correct thing.

Before submitting pull requests, run tests with:

```
$ python setup.py test
```

Also run mypy and flake8 on `black.py` and `test_black.py`. Travis will run all that for you but if you left any errors here, it will be quicker and less embarrassing to fix them locally ;-)

2.4.3 Hygiene

If you're fixing a bug, add a test. Run it first to confirm it fails, then fix the bug, run it again to confirm it's really fixed.

If adding a new feature, add a test. In fact, always add a test. But wait, before adding any large feature, first open an issue for us to discuss the idea first.

2.4.4 Finally

Thanks again for your interest in improving the project! You're taking action when most people decide to sit and watch.

2.5 Change Log

2.5.1 18.3a5 (unreleased)

- fixed handling of standalone comments within nested bracketed expressions; Black will no longer produce super long lines or put all standalone comments at the end of the expression (#22)
- fixed 18.3a4 regression: don't crash and burn on empty lines with trailing whitespace (#80)
- only allow up to two empty lines on module level and only single empty lines within functions (#74)

2.5.2 18.3a4

- `# fmt: off` and `# fmt: on` are implemented (#5)
- automatic detection of deprecated Python 2 forms of print statements and exec statements in the formatted file (#49)
- use proper spaces for complex expressions in default values of typed function arguments (#60)
- only return exit code 1 when –check is used (#50)
- don't remove single trailing commas from square bracket indexing (#59)
- don't omit whitespace if the previous factor leaf wasn't a math operator (#55)
- omit extra space in kwarg unpacking if it's the first argument (#46)
- omit extra space in Sphinx auto-attribute comments (#68)

2.5.3 18.3a3

- don't remove single empty lines outside of bracketed expressions (#19)
- added ability to pipe formatting from stdin to stdin (#25)
- restored ability to format code with legacy usage of `async` as a name (#20, #42)
- even better handling of numpy-style array indexing (#33, again)

2.5.4 18.3a2

- changed positioning of binary operators to occur at beginning of lines instead of at the end, following a recent change to PEP8 (#21)
- ignore empty bracket pairs while splitting. This avoids very weirdly looking formattings (#34, #35)
- remove a trailing comma if there is a single argument to a call
- if top level functions were separated by a comment, don't put four empty lines after the upper function
- fixed unstable formatting of newlines with imports
- fixed unintentional folding of post scriptum standalone comments into last statement if it was a simple statement (#18, #28)
- fixed missing space in numpy-style array indexing (#33)
- fixed spurious space after star-based unary expressions (#31)

2.5.5 18.3a1

- added `--check`
- only put trailing commas in function signatures and calls if it's safe to do so. If the file is Python 3.6+ it's always safe, otherwise only safe if there are no `*args` or `**kwargs` used in the signature or call. (#8)
- fixed invalid spacing of dots in relative imports (#6, #13)
- fixed invalid splitting after comma on unpacked variables in for-loops (#23)
- fixed spurious space in parenthesized set expressions (#7)

- fixed spurious space after opening parentheses and in default arguments (#14, #17)
- fixed spurious space after unary operators when the operand was a complex expression (#15)

2.5.6 18.3a0

- first published version, Happy Day 2018!
- alpha quality
- date-versioned (see: https://calver.org/)

2.6 Style reference and PEP 8

Black reformats entire files in place. It is not configurable. It doesn't take previous formatting into account.

This section orders topics in the same sequence as Python's PEP 8 (HTML | Source).

2.6.1 Code layout

- horizontal_linespace
- vertical_linespace
- empty_lines

2.6.2 String Quotes

2.6.3 Whitespace in Expressions and Statements

2.6.4 When to use trailing commas

- trailing_commas

2.6.5 Comments

2.6.6 Naming Conventions

2.6.7 Programming Recommendations

- guards

2.7 Developer reference

Contents are subject to change.

2.7.1 *Black* classes

Contents are subject to change.

BracketTracker

class black.**BracketTracker**(*depth=0*, *bracket_match=NOTHING*, *delimiters=NOTHING*, *previous=None*)

Keeps track of brackets on a line.

mark(*leaf: blib2to3.pytree.Leaf*) → None

Mark *leaf* with bracket-related metadata. Keep track of delimiters.

All leaves receive an int *bracket_depth* field that stores how deep within brackets a given leaf is. 0 means there are no enclosing brackets that started on this line.

If a leaf is itself a closing bracket, it receives an *opening_bracket* field that it forms a pair with. This is a one-directional link to avoid reference cycles.

If a leaf is a delimiter (a token on which Black can split the line if needed) and it's on depth 0, its *id()* is stored in the tracker's *delimiters* field.

any_open_brackets() → bool

Return True if there is an yet unmatched open bracket on the line.

max_delimiter_priority(*exclude: Iterable[int] = ()*) → int

Return the highest priority of a delimiter found on the line.

Values are consistent with what *is_delimiter()* returns.

EmptyLineTracker

class black.**EmptyLineTracker**(*previous_line=None*, *previous_after=0*, *previous_defs=NOTHING*)

Provides a stateful method that returns the number of potential extra empty lines needed before and after the currently processed line.

Note: this tracker works on lines that haven't been split yet. It assumes the prefix of the first leaf consists of optional newlines. Those newlines are consumed by *maybe_empty_lines()* and included in the computation.

maybe_empty_lines(*current_line: black.Line*) → Tuple[int, int]

Return the number of extra empty lines before and after the *current_line*.

This is for separating *def*, *async def* and *class* with extra empty lines (two on module-level), as well as providing an extra empty line after flow control keywords to make them more prominent.

Line

class black.**Line**(*depth=0*, *leaves=NOTHING*, *comments=NOTHING*, *bracket_tracker=NOTHING*, *inside_brackets=False*, *has_for=False*, *for_loop_variable=False*)

Holds leaves and comments. Can be printed with *str(line)*.

append(*leaf: blib2to3.pytree.Leaf*, *preformatted: bool = False*) → None

Add a new *leaf* to the end of the line.

Unless *preformatted* is True, the *leaf* will receive a new consistent whitespace prefix and metadata applied by *BracketTracker*. Trailing commas are maybe removed, unpacked for loop variables are demoted from being delimiters.

Inline comments are put aside.

append_safe(*leaf: blib2to3.pytree.Leaf*, *preformatted: bool = False*) → None

Like *append()* but disallow invalid standalone comment structure.

Raises ValueError when any *leaf* is appended after a standalone comment or when a standalone comment is not the first leaf on the line.

is_comment
Is this line a standalone comment?

is_decorator
Is this line a decorator?

is_import
Is this an import line?

is_class
Is this line a class definition?

is_def
Is this a function definition? (Also returns True for async defs.)

is_flow_control
Is this line a flow control statement?

Those are *return*, *raise*, *break*, and *continue*.

is_yield
Is this line a yield statement?

contains_standalone_comments
If so, needs to be split before emitting.

maybe_remove_trailing_comma(*closing: blib2to3.pytree.Leaf*) → bool
Remove trailing comma if there is one and it's safe.

maybe_increment_for_loop_variable(*leaf: blib2to3.pytree.Leaf*) → bool
In a for loop, or comprehension, the variables are often unpacks.

To avoid splitting on the comma in this situation, increase the depth of tokens between *for* and *in*.

maybe_decrement_after_for_loop_variable(*leaf: blib2to3.pytree.Leaf*) → bool
See *maybe_increment_for_loop_variable* above for explanation.

append_comment(*comment: blib2to3.pytree.Leaf*) → bool
Add an inline or standalone comment to the line.

comments_after(*leaf: blib2to3.pytree.Leaf*) → Iterator[blib2to3.pytree.Leaf]
Generate comments that should appear directly after *leaf*.

remove_trailing_comma() → None
Remove the trailing comma and moves the comments attached to it.

__str__() → str
Render the line.

__bool__() → bool
Return True if the line has leaves or comments.

LineGenerator

class black.**LineGenerator**(*current_line=NOTHING*)
Bases: *black.Visitor*

Generates reformatted Line objects. Empty lines are not emitted.

Note: destroys the tree it's visiting by mutating prefixes of its leaves in ways that will no longer stringify to valid Python code on the tree.

line(*indent: int = 0*, *type: Type[black.Line] = <class 'black.Line'>*) → Iterator[black.Line]
Generate a line.

If the line is empty, only emit if it makes sense. If the line is too long, split it first and then generate.

If any lines were generated, set up a new current_line.

visit(*node: Union[blib2to3.pytree.Leaf, blib2to3.pytree.Node]*) → Iterator[black.Line]
Main method to visit *node* and its children.

Yields `Line` objects.

visit_default(*node: Union[blib2to3.pytree.Leaf, blib2to3.pytree.Node]*) → Iterator[black.Line]
Default *visit_*()* implementation. Recurses to children of *node*.

visit_INDENT(*node: blib2to3.pytree.Node*) → Iterator[black.Line]
Increase indentation level, maybe yield a line.

visit_DEDENT(*node: blib2to3.pytree.Node*) → Iterator[black.Line]
Decrease indentation level, maybe yield a line.

visit_stmt(*node: blib2to3.pytree.Node, keywords: Set[str]*) → Iterator[black.Line]
Visit a statement.

This implementation is shared for *if*, *while*, *for*, *try*, *except*, *def*, *with*, and *class*.

The relevant Python language *keywords* for a given statement will be NAME leaves within it. This methods puts those on a separate line.

visit_simple_stmt(*node: blib2to3.pytree.Node*) → Iterator[black.Line]
Visit a statement without nested statements.

visit_async_stmt(*node: blib2to3.pytree.Node*) → Iterator[black.Line]
Visit *async def*, *async for*, *async with*.

visit_decorators(*node: blib2to3.pytree.Node*) → Iterator[black.Line]
Visit decorators.

visit_SEMI(*leaf: blib2to3.pytree.Leaf*) → Iterator[black.Line]
Remove a semicolon and put the other statement on a separate line.

visit_ENDMARKER(*leaf: blib2to3.pytree.Leaf*) → Iterator[black.Line]
End of file. Process outstanding comments and end with a newline.

visit_unformatted(*node: Union[blib2to3.pytree.Leaf, blib2to3.pytree.Node]*) → Iterator[black.Line]
Used when file contained a *# fmt: off*.

Report

class black.**Report**(*check=False*, *change_count=0*, *same_count=0*, *failure_count=0*)
Provides a reformatting counter. Can be rendered with *str(report)*.

done(*src: pathlib.Path*, *changed: bool*) → None
Increment the counter for successful reformatting. Write out a message.

failed(*src: pathlib.Path*, *message: str*) → None
Increment the counter for failed reformatting. Write out a message.

return_code
Return the exit code that the app should use.

This considers the current state of changed files and failures: - if there were any failures, return 123; - if any files were changed and –check is being used, return 1; - otherwise return 0.

__str__() → str
Render a color report of the current state.

Use *click.unstyle* to remove colors.

UnformattedLines

class black.**UnformattedLines**(*depth=0*, *leaves=NOTHING*, *comments=NOTHING*, *bracket_tracker=NOTHING*, *inside_brackets=False*, *has_for=False*, *for_loop_variable=False*)
Bases: *black.Line*

Just like *Line* but stores lines which aren't reformatted.

append(*leaf: blib2to3.pytree.Leaf*, *preformatted: bool = True*) → None
Just add a new *leaf* to the end of the lines.

The *preformatted* argument is ignored.

Keeps track of indentation *depth*, which is useful when the user says *# fmt: on*. Otherwise, doesn't do anything with the *leaf*.

__str__() → str
Render unformatted lines from leaves which were added with *append()*.

depth is not used for indentation in this case.

append_comment(*comment: blib2to3.pytree.Leaf*) → bool
Not implemented in this class. Raises *NotImplementedError*.

maybe_remove_trailing_comma(*closing: blib2to3.pytree.Leaf*) → bool
Does nothing and returns False.

maybe_increment_for_loop_variable(*leaf: blib2to3.pytree.Leaf*) → bool
Does nothing and returns False.

Visitor

class black.**Visitor**
Bases: typing.Generic

Basic lib2to3 visitor that yields things of type *T* on *visit()*.

visit(*node: Union[blib2to3.pytree.Leaf, blib2to3.pytree.Node]*) → Iterator[T]
Main method to visit *node* and its children.

It tries to find a *visit_*()* method for the given *node.type*, like *visit_simple_stmt* for Node objects or *visit_INDENT* for Leaf objects. If no dedicated *visit_*()* method is found, chooses *visit_default()* instead.

Then yields objects of type *T* from the selected visitor.

visit_default(*node: Union[blib2to3.pytree.Leaf, blib2to3.pytree.Node]*) → Iterator[T]
Default *visit_*()* implementation. Recurses to children of *node*.

2.7.2 *Black* functions

Contents are subject to change.

Assertions and checks

black.**assert_equivalent**(*src: str*, *dst: str*) → None
Raise AssertionError if *src* and *dst* aren't equivalent.

black.**assert_stable**(*src: str*, *dst: str*, *line_length: int*) → None
Raise AssertionError if *dst* reformats differently the second time.

black.**is_delimiter**(*leaf: blib2to3.pytree.Leaf*) → int
Return the priority of the *leaf* delimiter. Return 0 if not delimiter.

Higher numbers are higher priority.

black.**is_import**(*leaf: blib2to3.pytree.Leaf*) → bool
Return True if the given leaf starts an import statement.

black.**is_python36**(*node: blib2to3.pytree.Node*) → bool
Return True if the current file is using Python 3.6+ features.

Currently looking for: - f-strings; and - trailing commas after * or ** in function signatures.

Formatting

black.**format_file_contents**(*src_contents: str*, *line_length: int*, *fast: bool*) → str
Reformat contents a file and return new contents.

If *fast* is False, additionally confirm that the reformatted code is valid by calling *assert_equivalent()* and *assert_stable()* on it. *line_length* is passed to *format_str()*.

black.**format_file_in_place**(*src: pathlib.Path*, *line_length: int*, *fast: bool*, *write_back: bool = False*) → bool
Format file under *src* path. Return True if changed.

If *write_back* is True, write reformatted code back to stdout. *line_length* and *fast* options are passed to *format_file_contents()*.

black.**format_stdin_to_stdout**(*line_length: int*, *fast: bool*, *write_back: bool = False*) → bool
Format file on stdin. Return True if changed.

If *write_back* is True, write reformatted code back to stdout. *line_length* and *fast* arguments are passed to *format_file_contents()*.

black.**format_str**(*src_contents: str*, *line_length: int*) → str
Reformat a string and return new contents.

line_length determines how many characters per line are allowed.

black.**schedule_formatting**(*sources: List[pathlib.Path]*, *line_length: int*, *write_back: bool*, *fast: bool*, *loop: asyncio.base_events.BaseEventLoop*, *executor: concurrent.futures._base.Executor*) → int
Run formatting of *sources* in parallel using the provided *executor*.

(Use ProcessPoolExecutors for actual parallelism.)

line_length, *write_back*, and *fast* options are passed to *format_file_in_place()*.

File operations

black.**dump_to_file**(**output*) → str
Dump *output* to a temporary file. Return path to the file.

black.**gen_python_files_in_dir**(*path: pathlib.Path*) → Iterator[pathlib.Path]
Generate all files under *path* which aren't under BLACKLISTED_DIRECTORIES and have one of the PYTHON_EXTENSIONS.

Parsing

black.**lib2to3_parse**(*src_txt: str*) → blib2to3.pytree.Node
Given a string with source, return the lib2to3 Node.

black.**lib2to3_unparse**(*node: blib2to3.pytree.Node*) → str
Given a lib2to3 node, return its string representation.

Split functions

black.**delimiter_split**(*line: black.Line*, *py36: bool = False*) → Iterator[black.Line]
Split according to delimiters of the highest priority.

If *py36* is True, the split will add trailing commas also in function signatures that contain * and **.

black.**left_hand_split**(*line: black.Line*, *py36: bool = False*) → Iterator[black.Line]
Split line into many lines, starting with the first matching bracket pair.

Note: this usually looks weird, only use this for function definitions. Prefer RHS otherwise.

black.**right_hand_split**(*line: black.Line*, *py36: bool = False*) → Iterator[black.Line]
Split line into many lines, starting with the last matching bracket pair.

black.**split_line**(*line: black.Line*, *line_length: int*, *inner: bool = False*, *py36: bool = False*) → Iterator[black.Line]
Split a *line* into potentially many lines.

They should fit in the allotted *line_length* but might not be able to. *inner* signifies that there were a pair of brackets somewhere around the current *line*, possibly transitively. This means we can fallback to splitting by delimiters if the LHS/RHS don't yield any results.

If *py36* is True, splitting may generate syntax that is only compatible with Python 3.6 and later.

black.**bracket_split_succeeded_or_raise**(*head: black.Line*, *body: black.Line*, *tail: black.Line*) → None
Raise *CannotSplit* if the last left- or right-hand split failed.

Do nothing otherwise.

A left- or right-hand split is based on a pair of brackets. Content before (and including) the opening bracket is left on one line, content inside the brackets is put on a separate line, and finally content starting with and following the closing bracket is put on a separate line.

Those are called *head*, *body*, and *tail*, respectively. If the split produced the same line (all content in *head*) or ended up with an empty *body* and the *tail* is just the closing bracket, then it's considered failed.

Utilities

black.DebugVisitor.**show**(*code: str*) → None
Pretty-print the lib2to3 AST of a given string of *code*.

black.**diff**(*a: str*, *b: str*, *a_name: str*, *b_name: str*) → str
Return a unified diff string between strings *a* and *b*.

black.**generate_comments**(*leaf: blib2to3.pytree.Leaf*) → Iterator[blib2to3.pytree.Leaf]
Clean the prefix of the *leaf* and generate comments from it, if any.

Comments in lib2to3 are shoved into the whitespace prefix. This happens in *pgen2/driver.py:Driver.parse_tokens()*. This was a brilliant implementation move because it does away with modifying the grammar to include all the possible places in which comments can be placed.

The sad consequence for us though is that comments don't "belong" anywhere. This is why this function generates simple parentless Leaf objects for comments. We simply don't know what the correct parent should be.

No matter though, we can live without this. We really only need to differentiate between inline and standalone comments. The latter don't share the line with any code.

Inline comments are emitted as regular token.COMMENT leaves. Standalone are emitted with a fake STANDALONE_COMMENT token identifier.

black.**make_comment**(*content: str*) → str
Return a consistently formatted comment from the given *content* string.

All comments (except for "##", "#!", "#:") should have a single space between the hash sign and the content.

If *content* didn't start with a hash sign, one is provided.

black.**normalize_prefix**(*leaf: blib2to3.pytree.Leaf*, *, *inside_brackets: bool*) → None
Leave existing extra newlines if not *inside_brackets*. Remove everything else.

Note: don't use backslashes for formatting or you'll lose your voting rights.

black.**preceding_leaf**(*node: Union[blib2to3.pytree.Leaf, blib2to3.pytree.Node, NoneType]*) → Union[blib2to3.pytree.Leaf, NoneType]
Return the first leaf that precedes *node*, if any.

black.**whitespace**(*leaf: blib2to3.pytree.Leaf*) → str
Return whitespace prefix if needed for the given *leaf*.

2.7.3 *Black* exceptions

Contents are subject to change.

exception black.**CannotSplit**
A readable split that fits the allotted line length is impossible.

Raised by *left_hand_split()*, *right_hand_split()*, and *delimiter_split()*.

exception black.**FormatError**(*consumed: int*) → None
Base exception for *# fmt: on* and *# fmt: off* handling.

It holds the number of bytes of the prefix consumed before the format control comment appeared.

exception black.**FormatOn**(*consumed: int*) → None
Found a comment like *# fmt: on* in the file.

exception black.**FormatOff**(*consumed: int*) → None
Found a comment like *# fmt: off* in the file.

exception black.**NothingChanged**
Raised by `format_file()` when reformatted code is the same as source.

2.8 Authors

Glued together by Łukasz Langa.

Maintained with Carol Willing and Carl Meyer.

Multiple contributions by:

- Artem Malyshev
- Daniel M. Capella
- Eli Treuherz
- Hugo van Kemenade
- Mika
- Osaetin Daniel

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Index

Symbols

A

B

C

D

E

F

G

I

L

M

N

P

R

S

U

V

W