Powerline

Release beta

May 01, 2017

Contents

1 Overview **1**
- 1.1 Features . . . 1
- 1.2 Screenshots . . . 2
  - 1.2.1 Vim statusline . . . 2

2 Installation **3**
- 2.1 Generic requirements . . . 3
- 2.2 Pip installation . . . 4
- 2.3 Fonts installation . . . 4
  - 2.3.1 Patched fonts . . . 4
- 2.4 Installation on various platforms . . . 5
  - 2.4.1 Installation on Linux . . . 5
  - 2.4.2 Installation on OS X . . . 6

3 Usage **9**
- 3.1 Application-specific requirements . . . 9
  - 3.1.1 Vim plugin requirements . . . 9
  - 3.1.2 Shell prompts requirements . . . 9
  - 3.1.3 WM widgets requirements . . . 9
  - 3.1.4 Terminal emulator requirements . . . 9
- 3.2 Plugins . . . 10
  - 3.2.1 Shell prompts . . . 10
  - 3.2.2 Window manager widgets . . . 12
  - 3.2.3 Other plugins . . . 14

4 Configuration and customization **19**
- 4.1 Quick setup guide . . . 20
- 4.2 References . . . 21
  - 4.2.1 Configuration reference . . . 21
  - 4.2.2 Segment reference . . . 27
  - 4.2.3 Lister reference . . . 49
  - 4.2.4 Selector functions . . . 51
  - 4.2.5 Local configuration overrides . . . 51

5 Developer guide **55**
- 5.1 Writing segments . . . 55
  - 5.1.1 Segment dictionary . . . 58

5.1.2 Segments layout . . . 59
5.1.3 Segment information used in various extensions . . . 59
5.1.4 Segment class . . . 61
5.1.5 PowerlineLogger class . . . 61
5.2 Writing listers . . . 62
5.3 Local themes . . . 62
5.3.1 Vim local themes . . . 63
5.3.2 Other local themes . . . 63
5.4 Creating new powerline extension . . . 63
5.4.1 Powerline class . . . 64
5.4.2 Renderer class . . . 67
5.5 Tips and tricks for powerline developers . . . 69
5.5.1 Profiling powerline in Vim . . . 69

6 Troubleshooting **71**
6.1 System-specific issues . . . 71
6.1.1 Troubleshooting on Linux . . . 71
6.1.2 Troubleshooting on OS X . . . 72
6.2 Common issues . . . 73
6.2.1 After an update something stopped working . . . 73
6.3 Tmux/screen-related issues . . . 75
6.3.1 I'm using tmux and Powerline looks like crap, what's wrong? . . . 75
6.3.2 I'm using tmux/screen and Powerline is colorless . . . 75
6.3.3 In tmux there is a green bar in place of powerline . . . 75
6.4 Shell issues . . . 75
6.4.1 When sourcing shell bindings it complains about missing command or file . . . 75
6.4.2 I am suffering bad lags before displaying shell prompt . . . 76
6.4.3 Prompt is spoiled after completing files in ksh . . . 76
6.4.4 When using z powerline shows wrong number of jobs . . . 76
6.4.5 When using shell I do not see powerline fancy characters . . . 76
6.4.6 Urxvt unicode3 and frills . . . 76
6.5 Vim issues . . . 76
6.5.1 My vim statusline has strange characters like `^B` in it! . . . 76
6.5.2 My vim statusline has a lot of `^` or underline characters in it! . . . 77
6.5.3 My vim statusline is hidden/only appears in split windows! . . . 77
6.5.4 My vim statusline is not displayed completely and has too much spaces . . . 77
6.5.5 Powerline loses color after editing vimrc . . . 77
6.5.6 Powerline loses color after saving any file . . . 77

7 Tips and tricks **79**
7.1 Vim . . . 79
7.1.1 Useful settings . . . 79
7.2 Rxvt-unicode . . . 79
7.2.1 Terminus font and urxvt . . . 79
7.2.2 Source Code Pro font and urxvt . . . 79
7.3 Reloading powerline after update . . . 80

8 License and credits **81**
8.1 Authors . . . 81
8.2 Contributors . . . 81

9 Powerline shell commands' manual pages **83**
9.1 powerline-config manual page . . . 83
9.1.1 Synopsis . . . 83
9.1.2 Description . . . 83

9.1.3 Author . . . 84
9.1.4 Reporting bugs . . . 84
9.1.5 See also . . . 84
9.2 powerline-daemon manual page . . . 84
9.2.1 Synopsis . . . 84
9.2.2 Description . . . 84
9.2.3 Author . . . 85
9.2.4 Reporting bugs . . . 85
9.2.5 See also . . . 85
9.3 powerline-lint manual page . . . 85
9.3.1 Synopsis . . . 85
9.3.2 Description . . . 85
9.3.3 Author . . . 85
9.3.4 Reporting bugs . . . 85
9.3.5 See also . . . 85
9.4 powerline manual page . . . 86
9.4.1 Synopsis . . . 86
9.4.2 Description . . . 86
9.4.3 Author . . . 87
9.4.4 Reporting bugs . . . 87
9.4.5 See also . . . 87

10 Indices and tables **89**

Python Module Index **91**

CHAPTER 1

Overview

Powerline is a statusline plugin for vim, and provides statuslines and prompts for several other applications, including zsh, bash, tmux, IPython, Awesome, i3 and Qtile.

Features

- **Extensible and feature rich, written in Python.** Powerline was completely rewritten in Python to get rid of as much vimscript as possible. This has allowed much better extensibility, leaner and better config files, and a structured, object-oriented codebase with no mandatory third-party dependencies other than a Python interpreter.
- **Stable and testable code base.** Using Python has allowed unit testing of all the project code. The code is tested to work in Python 2.6+ and Python 3.
- **Support for prompts and statuslines in many applications.** Originally created exclusively for vim statuslines, the project has evolved to provide statuslines in tmux and several WMs, and prompts for shells like bash/zsh and other applications. It's simple to write renderers for any other applications that Powerline doesn't yet support.
- **Configuration and colorschemes written in JSON.** JSON is a standardized, simple and easy to use file format that allows for easy user configuration across all of Powerline's supported applications.
- **Fast and lightweight, with daemon support for even better performance.** Although the code base spans a couple of thousand lines of code with no goal of "less than X lines of code", the main focus is on good performance and as little code as possible while still providing a rich set of features. The new daemon also ensures that only one Python instance is launched for prompts and statuslines, which provides excellent performance.

But I hate Python / I don't need shell prompts / this is just too much hassle for me / what happened to the original vim-powerline project / . . .

You should check out some of the Powerline derivatives. The most lightweight and feature-rich alternative is currently Bailey Ling's vim-airline[1] project.

[1] https://github.com/bling/vim-airline

Screenshots

Vim statusline

Mode-dependent highlighting

- NORMAL develop ./setup.py unix utf-8 python 2% 1:1
- INSERT develop ./setup.py unix utf-8 python 2% 1:1
- VISUAL develop ./setup.py unix utf-8 python 2% 1:1
- REPLACE develop ./setup.py unix utf-8 python 2% 1:1

Automatic truncation of segments in small windows

- NORMAL develop ./setup.py unix utf-8 python 2% 1:1
- NORMAL ./setup.py unix utf-8 python 2% 1:1
- NORMAL setup.py 1:1

CHAPTER 2

Installation

Generic requirements

- Python 2.6 or later, 3.2 or later, PyPy 2.0 or later, PyPy3 2.3 or later. It is the only non-optional requirement.
- C compiler. Required to build powerline client on linux. If it is not present then powerline will fall back to shell script or python client.
- `socat` program. Required for shell variant of client which runs a bit faster than python version of the client, but still slower than C version.
- `psutil` python package. Required for some segments like cpu_percent. Some segments have linux-only fallbacks for `psutil` functionality.
- `hglib` python package *and* mercurial executable. Required to work with mercurial repositories.
- `pygit2` python package or `git` executable. Required to work with `git` repositories.
- `bzr` python package (note: *not* standalone executable). Required to work with bazaar repositories.
- `pyuv` python package. Required for *libuv-based watcher* to work.
- `i3ipc` python package. Required for i3wm bindings and segments.
- `xrandr` program. Required for the multi-monitor lemonbar binding and the `powerline.listers.i3wm.output_lister()`.

Note: Until bazaar supports Python-3 or PyPy powerline will not support repository information when running in these interpreters.

Note: When using `pip {repository_root}` directory referenced in documentation may be found using `pip show powerline-status`. In the output of `pip show` there is a line like `Location: {path}`, that `{path}` is `{repository_root}`. Unless it is `--editable` installation this is only applicable for `{repository_root}/powerline/...` paths: something like `{repository_root}/scripts/powerline-render` is not present.

When using other packages referenced paths may not exist, in this case refer to package documentation.

Pip installation

Due to a naming conflict with an unrelated project powerline is available on PyPI under the `powerline-status` name:

```
pip install powerline-status
```

is the preferred method because this will get the latest release. To get current development version

```
pip install --user git+git://github.com/powerline/powerline
```

may be used. If powerline was already checked out into some directory

```
pip install --user --editable={path_to_powerline}
```

is useful, but note that in this case `pip` will not install `powerline` executable and something like

```
ln -s {path_to_powerline}/scripts/powerline ~/.local/bin
```

will have to be done (`~/.local/bin` should be replaced with some path present in `$PATH`).

Note: If ISP blocks git protocol for some reason github also provides `ssh` (`git+ssh://git@github.com/powerline/powerline`) and `https` (`git+https://github.com/powerline/powerline`) protocols. `git` protocol should be the fastest, but least secure one though.

Fonts installation

Powerline uses several special glyphs to get the arrow effect and some custom symbols for developers. This requires having either a symbol font or a patched font installed in the system. The used application (e.g. terminal emulator) must also either be configured to use patched fonts (in some cases even support it because custom glyphs live in private use area which some applications reserve for themselves) or support fontconfig for powerline to work properly with powerline-specific glyphs.

24-bit color support may be enabled if used terminal emulator supports it (see *the terminal emulator support matrix*).

There are basically two ways to get powerline glyphs displayed: use `PowerlineSymbols.otf` font as a fallback for one of the existing fonts or install a patched font.

Patched fonts

This method is the fallback method and works for every terminal.

Download the font from powerline-fonts[2]. If preferred font can't be found in the powerline-fonts[3] repo, then patching the preferred font is needed instead.

After downloading this font refer to platform-specific instructions.

[2] https://github.com/powerline/fonts
[3] https://github.com/powerline/fonts

Installation on various platforms

Installation on Linux

The following distribution-specific packages are officially supported, and they provide an easy way of installing and upgrading Powerline. The packages will automatically do most of the configuration.

- Arch Linux (AUR), Python 2 version[4]
- Arch Linux (AUR), Python 3 version[5]
- Gentoo Live ebuild in raiagent[6] overlay
- Powerline package is available for Debian starting from Wheezy (via backports[7]). Use search[8] to get more information.

If used distribution does not have an official package installation guide below should be followed:

1. Install Python 3.2+, Python 2.6+ or PyPy and `pip` with `setuptools`. This step is distribution-specific, so no commands provided.
2. Install Powerline using one of the following commands:

   ```
   pip install --user powerline-status
   ```

   will get the latest release version and

   ```
   pip install --user git+git://github.com/powerline/powerline
   ```

   will get the latest development version.

   Note: Due to the naming conflict with an unrelated project powerline is named `powerline-status` in PyPI.

   Note: Powerline developers should be aware that‘‘pip install –editable‘‘ does not currently fully work. Installation performed this way are missing `powerline` executable that needs to be symlinked. It will be located in `scripts/powerline`.

Fonts installation

Fontconfig

This method only works on Linux. It’s the second recommended method if terminal emulator supports it as patching fonts is not needed, and it generally works with any coding font.

1. Download the latest version of the symbol font and fontconfig file:

   ```
   wget https://github.com/powerline/powerline/raw/develop/font/PowerlineSymbols.otf
   wget https://github.com/powerline/powerline/raw/develop/font/10-powerline-symbols.
   ↪conf
   ```

[4] https://aur.archlinux.org/packages/python2-powerline-git/
[5] https://aur.archlinux.org/packages/python-powerline-git/
[6] https://github.com/leycec/raiagent
[7] https://packages.debian.org/wheezy-backports/powerline
[8] https://packages.debian.org/search?keywords=powerline&searchon=names&suite=all&section=all

2. Move the symbol font to a valid X font path. Valid font paths can be listed with `xset q`:

```
mv PowerlineSymbols.otf ~/.fonts/
```

3. Update font cache for the path the font was moved to (root priveleges may be needed to update cache for the system-wide paths):

```
fc-cache -vf ~/.fonts/
```

4. Install the fontconfig file. For newer versions of fontconfig the config path is `~/.config/fontconfig/conf.d/`, for older versions it's `~/.fonts.conf.d/`:

```
mv 10-powerline-symbols.conf ~/.config/fontconfig/conf.d/
```

If custom symbols still cannot be seen then try closing all instances of the terminal emulator. Restarting X may be needed for the changes to take effect.

If custom symbols *still* can't be seen, double-check that the font have been installed to a valid X font path, and that the fontconfig file was installed to a valid fontconfig path. Alternatively try to install a *patched font*.

Patched font installation

This is the preferred method, but it is not always available because not all fonts were patched and not all fonts *can* be patched due to licensing issues.

After downloading font the following should be done:

1. Move the patched font to a valid X font path. Valid font paths can be listed with `xset q`:

```
mv 'SomeFont for Powerline.otf' ~/.fonts/
```

2. Update font cache for the path the font was moved to (root priveleges may be needed for updating font cache for some paths):

```
fc-cache -vf ~/.fonts/
```

After installing patched font terminal emulator, GVim or whatever application powerline should work with must be configured to use the patched font. The correct font usually ends with *for Powerline*.

If custom symbols cannot be seen then try closing all instances of the terminal emulator. X server may need to be restarted for the changes to take effect.

If custom symbols *still* can't be seen then double-check that the font have been installed to a valid X font path.

Installation on OS X

Python package

1. Install a proper Python version (see issue #39[9] for a discussion regarding the required Python version on OS X):

```
sudo port select python python27-apple
```

Homebrew may be used here:

[9] https://github.com/powerline/powerline/issues/39

```
brew install python
```

Note: In case `powerline.sh` as a client `socat` and `coreutils` need to be installed. `coreutils` may be installed using `brew install coreutils`.

2. Install Powerline using one of the following commands:

```
pip install --user powerline-status
```

will get current release version and

```
pip install --user git+git://github.com/powerline/powerline
```

will get latest development version.

Warning: When using `brew install` to install Python one must not supply `--user` flag to `pip`.

Note: Due to the naming conflict with an unrelated project powerline is named `powerline-status` in PyPI.

Note: Powerline developers should be aware that `pip install --editable` does not currently fully work. Installation performed this way are missing `powerline` executable that needs to be symlinked. It will be located in `scripts/powerline`.

Vim installation

Any terminal vim version with Python 3.2+ or Python 2.6+ support should work, but MacVim users need to install it using the following command:

```
brew install macvim --env-std --with-override-system-vim
```

Fonts installation

To install patched font double-click the font file in Finder, then click *Install this font* in the preview window.

After installing the patched font MacVim or terminal emulator (whatever application powerline should work with) need to be configured to use the patched font. The correct font usually ends with *for Powerline*.

CHAPTER 3

Usage

Application-specific requirements

Vim plugin requirements

The vim plugin requires a vim version with Python support compiled in. Presense of Python support in Vim can be checked by running `vim --version | grep +python`.

If Python support is absent then Vim needs to be compiled with it. To do this use `--enable-pythoninterp` `./configure` flag (Python 3 uses `--enable-python3interp` flag instead). Note that this also requires the related Python headers to be installed. Please consult distribution's documentation for details on how to compile and install packages.

Vim version 7.4 or newer is recommended for performance reasons, but Powerline supports Vim 7.0.112 and higher.

Shell prompts requirements

Due to fish having incorrect code for prompt width calculations up to version 2.1 and no way to tell that certain sequence of characters has no width (`%{...%}` in zsh and `\[...\]` in bash prompts serve exactly this purpose) users that have fish versions below 2.1 are not supported..

WM widgets requirements

Awesome is supported starting from version 3.5.1, inclusive. QTile is supported from version 0.6, inclusive.

Terminal emulator requirements

Powerline uses several special glyphs to get the arrow effect and some custom symbols for developers. This requires either a symbol font or a patched font installed. Used terminal emulator must also support either patched fonts or fontconfig for Powerline to work properly.

24-bit color support can also be enabled if terminal emulator supports it.

Table 3.1: Application/terminal emulator feature support matrix

Name	OS	Patched font support	Fontconfig support	24-bit color support
Gvim	Linux	✔	✘	✔
iTerm2	OS X	✔	✘	✘
Konsole	Linux	✔	✔	✔
lxterminal	Linux	✔	✔	✘
MacVim	OS X	✔	✘	✔
rxvt-unicode	Linux	⚠[10]	✘	✘
st	Linux	✔	✔	✔[11]
Terminal.app	OS X	✔	✘	✘
libvte-based[12]	Linux	✔	✔	✔[13]
xterm	Linux	✔	✘	⚠[14]
fbterm	Linux	✔	✔	✘

Plugins

Shell prompts

Note: Powerline daemon is not run automatically by any of my bindings. It is advised to add

```
powerline-daemon -q
```

before any other powerline-related code in the shell configuration file.

Bash prompt

Add the following line to the `bashrc`, where `{repository_root}` is the absolute path to the Powerline installation directory (see *repository root*):

```
. {repository_root}/powerline/bindings/bash/powerline.sh
```

Note: Since without powerline daemon bash bindings are very slow PS2 (continuation) and PS3 (select) prompts are not set up. Thus it is advised to use

[10] Must be compiled with `--enable-unicode3` for the patched font to work.
[11] Since version 0.5.
[12] Including XFCE terminal and GNOME terminal.
[13] Since version 0.36.
[14] Uses nearest color from 8-bit palette.

```
powerline-daemon -q
POWERLINE_BASH_CONTINUATION=1
POWERLINE_BASH_SELECT=1
. {repository_root}/powerline/bindings/bash/powerline.sh
```

in the bash configuration file. Without `POWERLINE_BASH_*` variables PS2 and PS3 prompts are computed exactly once at bash startup.

Warning: At maximum bash continuation PS2 and select PS3 prompts are computed each time main PS1 prompt is computed. Thus putting e.g. current time into PS2 or PS3 prompt will not work as expected.

At minimum they are computed once on startup.

Zsh prompt

Add the following line to the `zshrc`, where `{repository_root}` is the absolute path to the Powerline installation directory (see *repository root*):

```
. {repository_root}/powerline/bindings/zsh/powerline.zsh
```

Fish prompt

Add the following line to `config.fish`, where `{repository_root}` is the absolute path to the Powerline installation directory (see *repository root*):

```
set fish_function_path $fish_function_path "{repository_root}/powerline/bindings/fish"
powerline-setup
```

Warning: Fish is supported only starting from version 2.1.

Rcsh prompt

Powerline supports Plan9 rc reimplementation *by Byron Rakitzis* packaged by many *nix distributions. To use it add

```
. {repository_root}/powerline/bindings/rc/powerline.rc
```

(`{repository_root}` is the absolute path to the Powerline installation directory, see *repository root*) to `rcrc` file (usually `~/.rcrc`) and make sure `rc` is started as a login shell (with `-l` argument): otherwise this configuration file is not read.

Warning: Original Plan9 shell and its *nix port are not supported because they are missing `prompt` special function (it is being called once before each non-continuation prompt). Since powerline could not support shell without this or equivalent feature some other not-so-critical features of that port were used.

Busybox (ash), mksh and dash prompt

After launching busybox run the following command:

```
. {repository_root}/powerline/bindings/shell/powerline.sh
```

where `{repository_root}` is the absolute path to the Powerline installation directory (see *repository root*).

Mksh users may put this line into `~/.mkshrc` file. Dash users may use the following in `~/.profile`:

```
if test "$0" != "${0#dash}" ; then
    export ENV={repository_root}/powerline/bindings/shell/powerline.sh
fi
```

Note: Dash users that already have `$ENV` defined should either put the `. .../shell/powerline.sh` line in the `$ENV` file or create a new file which will source (using `.` command) both former `$ENV` file and `powerline.sh` files and set `$ENV` to the path of this new file.

Warning: Mksh users have to set `$POWERLINE_SHELL_CONTINUATION` and `$POWERLINE_SHELL_SELECT` to 1 to get PS2 and PS3 (continuation and select) prompts support respectively: as command substitution is not performed in these shells for these prompts they are updated once each time PS1 prompt is displayed which may be slow.

It is also known that while PS2 and PS3 update is triggered at PS1 update it is *actually performed* only *next* time PS1 is displayed which means that PS2 and PS3 prompts will be outdated and may be incorrect for this reason.

Without these variables PS2 and PS3 prompts will be set once at startup. This only touches mksh users: busybox and dash both have no such problem.

Warning: Job count is using some weird hack that uses signals and temporary files for interprocess communication. It may be wrong sometimes. Not the case in mksh.

Warning: Busybox has two shells: `ash` and `hush`. Second is known to segfault in busybox 1.22.1 when using `powerline.sh` script.

Window manager widgets

Awesome widget

Note: Powerline currently only supports awesome 3.5.

Note: The Powerline widget will spawn a shell script that runs in the background and updates the statusline with `awesome-client`.

Add the following to `rc.lua`, where `{repository_root}` is the absolute path to Powerline installation directory (see *repository root*):

```
package.path = package.path .. ';{repository_root}/powerline/bindings/awesome/?.lua'
require('powerline')
```

Then add the `powerline_widget` to `wibox`:

```
right_layout:add(powerline_widget)
```

Qtile widget

Add the following to `~/.config/qtile/config.py`:

```
from libqtile.bar import Bar
from libqtile.config import Screen
from libqtile.widget import Spacer

from powerline.bindings.qtile.widget import PowerlineTextBox

screens = [
    Screen(
        top=Bar([
                PowerlineTextBox(update_interval=2, side='left'),
                Spacer(),
                PowerlineTextBox(update_interval=2, side='right'),
            ],
            35 # width
        ),
    ),
]
```

lemonbar (formerly bar-aint-recursive)

To run the bar simply start the binding script:

> python /path/to/powerline/bindings/lemonbar/powerline-lemonbar.py

You can specify options to be passed to `lemonbar` after `--`, like so:

> python /path/to/powerline/bindings/lemonbar/powerline-lemonbar.py –height 16 – -f "Source Code Pro for Powerline-9"

to run with i3, simply `exec` this in the i3 config file and set the `--i3` switch:

> exec python /path/to/powerline/bindings/lemonbar/powerline-lemonbar.py –i3

Running the binding in i3-mode will require i3ipc[15] (or the outdated i3-py[16]).

See the lemonbar documentation[17] for more information and options.

All `powerline-lemonbar.py` arguments:

```
powerline-lemonbar.py [--i3] [--height=PIXELS] [--interval=SECONDS]
                      [--bar-command=CMD] [--] [ARGS]...
```

[15] https://github.com/acrisci/i3ipc-python
[16] https://github.com/ziberna/i3-py
[17] https://github.com/LemonBoy/bar

–i3 Subscribe for i3 events.

–height *PIXELS* Bar height.

–interval, -i *SECONDS* Refresh interval.

–bar-command, -C *CMD* Name of the lemonbar executable to use.

ARGS Extra arguments for lemonbar. Should be preceded with `--` argument in order not to be confused with script own arguments.

-h, –help Display help and exit.

I3 bar

Note: As the patch to include background-colors in i3bar is likely not to be merged, it is recommended to instead run `bar` (see above). The source for i3bgbar is however still available here[18].

Add the following to `~/.i3/config`:

```
bar {
    i3bar_command i3bgbar

    status_command python /path/to/powerline/bindings/i3/powerline-i3.py
    font pango:PowerlineFont 12
}
```

where `i3bgbar` may be replaced with the path to the custom i3bar binary and `PowerlineFont` is any system font with powerline support.

Other plugins

Vim statusline

If installed using pip just add

```
python from powerline.vim import setup as powerline_setup
python powerline_setup()
python del powerline_setup
```

(replace `python` with `python3` if appropriate) to the `vimrc`.

Note: Status line will not appear by default when there is only a single window displayed. Run `:h 'laststatus'` in Vim for more information.

If the repository was just cloned the following line needs to be added to the `vimrc`:

```
set rtp+={repository_root}/powerline/bindings/vim
```

where `{repository_root}` is the absolute path to the Powerline installation directory (see *repository root*).

[18] https://github.com/S0lll0s/i3bgbar

If pathogen is used and Powerline functionality is not needed outside of Vim then it is possible to simply add Powerline as a bundle and point the path above to the Powerline bundle directory, e.g. `~/.vim/bundle/powerline/powerline/bindings/vim`.

Vundle and NeoBundle users may instead use

```
Bundle 'powerline/powerline', {'rtp': 'powerline/bindings/vim/'}
```

(NeoBundle users need `NeoBundle` in place of `Bundle`, otherwise setup is the same).

Vim-addon-manager setup is even easier because it is not needed to write this big path or install anything by hand: `powerline` can be installed and activated just like any other plugin using

```
call vam#ActivateAddons(['powerline'])
```

Warning: *Never* install powerline with pathogen/VAM/Vundle/NeoBundle *and* with pip. If powerline functionality is needed in applications other then Vim then system-wide installation (in case used OS distribution has powerline package), pip-only or `pip install --editable` kind of installation performed on the repository installed by Vim plugin manager should be used.

No issues are accepted in powerline issue tracker for double pip/non-pip installations, especially if these issues occur after update.

Note: If supplied `powerline.vim` file is used to load powerline there are additional configuration variables available: `g:powerline_pycmd` and `g:powerline_pyeval`. First sets command used to load powerline: expected values are `"py"` and `"py3"`. Second sets function used in statusline, expected values are `"pyeval"` and `"py3eval"`.

If `g:powerline_pycmd` is set to the one of the expected values then `g:powerline_pyeval` will be set accordingly. If it is set to some other value then `g:powerline_pyeval` must also be set. Powerline will not check that Vim is compiled with Python support if `g:powerline_pycmd` is set to an unexpected value.

These values are to be used to specify the only Python that is to be loaded if both versions are present: Vim may disable loading one python version if other was already loaded. They should also be used if two python versions are able to load simultaneously, but powerline was installed only for python-3 version.

Tmux statusline

Add the following lines to `.tmux.conf`, where `{repository_root}` is the absolute path to the Powerline installation directory (see *repository root*):

```
source "{repository_root}/powerline/bindings/tmux/powerline.conf"
```

Note: The availability of the `powerline-config` command is required for powerline support. The location of this script may be specified via the `$POWERLINE_CONFIG_COMMAND` environment variable.

Note: It is advised to run `powerline-daemon` before adding the above line to tmux.conf. To do so add:

```
run-shell "powerline-daemon -q"
```

to `.tmux.conf`.

Warning: Segments which depend on current working directory (e.g. *powerline.segments.common.vcs.branch()*) require also setting up *shell bindings*. It is not required to use powerline shell prompt, *components setting* allows to set up only powerline bindings for tmux without altering your prompt. Without setting up shell bindings powerline will use current working directory of *tmux server* which is probably not what you need.

Segments which depend on environment like *powerline.segments.common.env.virtualenv()* will not work at all (i.e. they will use environment of the tmux server), tracking environment changes is going to slow down shell a lot.

In any case it is suggested to avoid both kinds of segments in tmux *themes* because even support for tracking current directory is very limited:

1. It works only in shell. Should you e.g. run Vim and run `:cd` there you will get current working directory from shell.
2. It works only in local shell and requires configuring it.
3. Some shells are not supported at all.

IPython prompt

For IPython<0.11 add the following lines to `.ipython/ipy_user_conf.py`:

```
# top
from powerline.bindings.ipython.pre_0_11 import setup as powerline_setup

# main() function (assuming ipython was launched without configuration to
# create skeleton ipy_user_conf.py file):
powerline_setup()
```

For IPython>=0.11 add the following line to `~/.ipython/profile_default/ipython_config.py` file in the used profile:

```
c = get_config()
c.InteractiveShellApp.extensions = [
    'powerline.bindings.ipython.post_0_11'
]
```

For IPython>=5.0 you may use the above set up, but it is deprecated. It is suggested to use

```
from powerline.bindings.ipython.since_5 import PowerlinePrompts
c = get_config()
c.TerminalInteractiveShell.simple_prompt = False
c.TerminalInteractiveShell.prompts_class = PowerlinePrompts
```

Note: Setting `simple_prompt` to False after IPython-5.0 is required regardless of whether you use `c.InteractiveShellApp.extensions` setting or `c.TerminalInteractiveShell.prompts_class`. But you probably already have this line because IPython is not very useful without it.

IPython=0.11* is not supported and does not work. IPython<0.10 was not tested (not installable by pip).

PDB prompt

To use Powerline with PDB prompt you need to use custom class. Inherit your class from `pdb.Pdb` and decorate it with `powerline.bindings.pdb.use_powerline_prompt()`:

```
import pdb

from powerline.bindings.pdb import use_powerline_prompt

@use_powerline_prompt
class MyPdb(pdb.Pdb):
    pass

MyPdb.run('some.code.to.debug()')
```

. Alternatively you may use

```
python -mpowerline.bindings.pdb path/to/script.py
```

just like you used `python -m pdb`.

CHAPTER 4

Configuration and customization

Note: Forking the main GitHub repo is not needed to personalize Powerline configuration! Please read through the *Quick setup guide* for a quick introduction to user configuration.

Powerline is configured with one main configuration file, and with separate configuration files for themes and colorschemes. All configuration files are written in JSON, with the exception of segment definitions, which are written in Python.

Powerline provides default configurations in the following locations:

Main configuration `powerline/config.json`

Colorschemes `powerline/colorschemes/name.json`, `powerline/colorschemes/extension/__main__.json`, `powerline/colorschemes/extension/name.json`

Themes `powerline/themes/top_theme.json`, `powerline/themes/extension/__main__.json`, `powerline/themes/extension/default.json`

Here *{powerline}* is one of the following:

1. The default configuration directory located in the main package: `powerline_root/powerline/config_files`. May be absent in some packages (e.g. when installing via Gentoo ebuilds).
2. If variable `$XDG_CONFIG_DIRS` is set and non-empty then to any `directory/powerline` where *{directory}* is a directory listed in a colon-separated `$XDG_CONFIG_DIRS` list. Directories are checked in reverse order.
3. User configuration directory located in `$XDG_CONFIG_HOME/powerline`. This usually corresponds to `~/.config/powerline` on all platforms.

If per-instance configuration is needed please refer to *Local configuration overrides*.

Note: Existing multiple configuration files that have the same name, but are placed in different directories, will be merged. Merging happens in the order given in the above list of possible *{powerline}* meanings.

When merging configuration only dictionaries are merged and they are merged recursively: keys from next file overrule those from the previous unless corresponding values are both dictionaries in which case these dictionaries are merged and key is assigned the result of the merge.

Note: Some configuration files (i.e. themes and colorschemes) have two level of merging: first happens merging described above, second theme- or colorscheme-specific merging happens.

Quick setup guide

This guide will help you with the initial configuration of Powerline.

Look at configuration in *powerline_root*`/powerline/config_files`. If you want to modify some file you can create `~/.config/powerline` directory and put modifications there: all configuration files are *merged* with each other.

Each extension (vim, tmux, etc.) has its own theme, and they are located in *config directory*`/themes/`*extension*`/default.json`. Best way to modify it is to copy this theme as a whole, remove `segment_data` key with corresponding value if present (unless you need to modify it, in which case only modifications must be left) and do necessary modifications in the list of segments (lists are not subject to merging: this is why you need a copy).

If you want to move, remove or customize any of the provided segments in the copy, you can do that by updating the segment dictionary in the theme you want to customize. A segment dictionary looks like this:

```
{
    "name": "segment_name"
    ...
}
```

You can move the segment dictionaries around to change the segment positions, or remove the entire dictionary to remove the segment from the prompt or statusline.

Note: It's essential that the contents of all your configuration files is valid JSON! It's strongly recommended that you run your configuration files through `jsonlint` after changing them.

Note: If your modifications appear not to work, run *powerline-lint script*. This script should show you the location of the error.

Some segments need a user configuration to work properly. Here's a couple of segments that you may want to customize right away:

E-mail alert segment You have to set your username and password (and possibly server/port) for the e-mail alert segment. If you're using GMail it's recommended that you generate an application-specific password[19] for this purpose.

Open a theme file, scroll down to the `email_imap_alert` segment and set your `username` and `password`. The server defaults to GMail's IMAP server, but you can set the server/port by adding a `server` and a `port` argument.

[19] https://accounts.google.com/IssuedAuthSubTokens

Weather segment The weather segment will try to find your location using a GeoIP lookup, so unless you're on a VPN you probably won't have to change the location query.

If you want to change the location query or the temperature unit you'll have to update the segment arguments. Open a theme file, scroll down to the weather segment and update it to include unit/location query arguments:

```
{
    "name": "weather",
    "priority": 50,
    "args": {
        "unit": "F",
        "location_query": "oslo, norway"
    }
},
```

References

Configuration reference

Main configuration

Location `powerline/config.json`

The main configuration file defines some common options that applies to all extensions, as well as some extension-specific options like themes and colorschemes.

Common configuration

Common configuration is a subdictionary that is a value of `common` key in `powerline/config.json` file.

`term_truecolor` Defines whether to output cterm indices (8-bit) or RGB colors (24-bit) to the terminal emulator. See the *Application/terminal emulator feature support matrix* for information on whether used terminal emulator supports 24-bit colors.

This variable is forced to be `false` if *term_escape_style* option is set to `"fbterm"` or if it is set to `"auto"` and powerline detected fbterm.

`term_escape_style` Defines what escapes sequences should be used. Accepts three variants:

Variant	Description
auto	`xterm` or `fbterm` depending on `$TERM` variable value: `TERM=fbterm` implies `fbterm` escaping style, all other values select `xterm` escaping.
xterm	Uses `\e[{fb};5;{color}m` for colors (`{fb}` is either `38` (foreground) or `48` (background)). Should be used for most terminals.
fbterm	Uses `\e[{fb};{color}}` for colors (`{fb}` is either `1` (foreground) or `2` (background)). Should be used for fbterm: framebuffer terminal.

`ambiwidth` Tells powerline what to do with characters with East Asian Width Class Ambigious (such as Euro, Registered Sign, Copyright Sign, Greek letters, Cyrillic letters). Valid values: any positive integer; it is suggested that this option is only set it to 1 (default) or 2.

`watcher` Select filesystem watcher. Variants are

Variant	Description
auto	Selects most performant watcher.
inotify	Select inotify watcher. Linux only.
stat	Select stat-based polling watcher.
uv	Select libuv-based watcher.

Default is `auto`.

`additional_escapes` Valid for shell extensions, makes sense only if *term_truecolor* is enabled. Is to be set from command-line. Controls additional escaping that is needed for tmux/screen to work with terminal true color escape codes: normally tmux/screen prevent terminal emulator from receiving these control codes thus rendering powerline prompt colorless. Valid values: `"tmux"`, `"screen"`, `null` (default).

`paths` Defines additional paths which will be searched for modules when using *function segment option* or *Vim local_themes option*. Paths defined here have priority when searching for modules.

`log_file` Defines how logs will be handled. There are three variants here:

1. Absent. In this case logging will be done to stderr: equivalent to `[["logging.StreamHandler", []]]` or `[null]`.
2. Plain string. In this case logging will be done to the given file: `"/file/name"` is equivalent to `[["logging.FileHandler", [["/file/name"]]]]` or `["/file/name"]`. Leading `~/` is expanded in the file name, so using `"~/.log/foo"` is permitted. If directory pointed by the option is absent, it will be created, but not its parent.
3. List of handler definitions. Handler definition may either be `null`, a string or a list with two or three elements:
   (a) Logging class name and module. If module name is absent, it is equivalent to `logging.handlers`.
   (b) Class constructor arguments in a form `[[args[, kwargs]]]`: accepted variants are `[]` (no arguments), `[args]` (e.g. `[["/file/name"]]`: only positional arguments) or `[args, kwargs]` (e.g. `[[], {"host": "localhost", "port": 6666}]`: positional and keyword arguments, but no positional arguments in the example).
   (c) Optional logging level. Overrides *log_level key* and has the same format.
   (d) Optional format string. Partially overrides *log_format key* and has the same format. "Partially" here means that it may only specify more critical level.

`log_level` String, determines logging level. Defaults to `WARNING`.

`log_format` String, determines format of the log messages. Defaults to `'%(asctime)s:%(level)s:%(message)s'`.

`interval` Number, determines time (in seconds) between checks for changed configuration. Checks are done in a seprate thread. Use `null` to check for configuration changes on `.render()` call in main thread. Defaults to `None`.

`reload_config` Boolean, determines whether configuration should be reloaded at all. Defaults to `True`.

`default_top_theme` String, determines which top-level theme will be used as the default. Defaults to `powerline_terminus` in unicode locales and `ascii` in non-unicode locales. See *Themes* section for more details.

Extension-specific configuration

Common configuration is a subdictionary that is a value of `ext` key in `powerline/config.json` file.

`colorscheme` Defines the colorscheme used for this extension.

theme Defines the theme used for this extension.

top_theme Defines the top-level theme used for this extension. See *Themes* section for more details.

local_themes Defines themes used when certain conditions are met, e.g. for buffer-specific statuslines in vim. Value depends on extension used. For vim it is a dictionary `{matcher_name : theme_name}`, where `matcher_name` is either `matcher_module.module_attribute` or `module_attribute` (`matcher_module` defaults to `powerline.matchers.vim`) and `module_attribute` should point to a function that returns boolean value indicating that current buffer has (not) matched conditions. There is an exception for `matcher_name` though: if it is `__tabline__` no functions are loaded. This special theme is used for `tabline` Vim option.

For shell and ipython it is a simple `{prompt_type : theme_name}`, where `prompt_type` is a string with no special meaning (specifically it does not refer to any Python function). Shell has `continuation`, and `select` prompts with rather self-explanatory names, IPython has `in2`, `out` and `rewrite` prompts (refer to IPython documentation for more details) while `in` prompt is the default.

For wm (*lemonbar* only) it is a dictionary `{output : theme_name}` that maps the `xrandr` output names to the local themes to use on that output.

components Determines which extension components should be enabled. This key is highly extension-specific, here is the table of extensions and corresponding components:

Extension	Component	Description
vim	statusline	Makes Vim use powerline statusline.
	tabline	Makes Vim use powerline tabline.
shell	prompt	Makes shell display powerline prompt.
	tmux	Makes shell report its current working directory and screen width to tmux for tmux powerline bindings.

All components are enabled by default.

update_interval Determines how often WM status bars need to be updated, in seconds. Only valid for WM extensions which use `powerline-daemon`. Defaults to 2 seconds.

Color definitions

Location `powerline/colors.json`

colors Color definitions, consisting of a dict where the key is the name of the color, and the value is one of the following:

- A cterm color index.
- A list with a cterm color index and a hex color string (e.g. `[123, "aabbcc"]`). This is useful for colorschemes that use colors that aren't available in color terminals.

gradients Gradient definitions, consisting of a dict where the key is the name of the gradient, and the value is a list containing one or two items, second item is optional:

- A list of cterm color indicies.
- A list of hex color strings.

It is expected that gradients are defined from least alert color to most alert or non-alert colors are used.

Colorschemes

Location `powerline/colorschemes/name.json`, `powerline/colorschemes/__main__.json`, `powerline/colorschemes/extension/name.json`

Colorscheme files are processed in order given: definitions from each next file override those from each previous file. It is required that either `powerline/colorschemes/name.json`, or `powerline/colorschemes/extension/name.json` exists.

name Name of the colorscheme.

groups Segment highlighting groups, consisting of a dict where the key is the name of the highlighting group (usually the function name for function segments), and the value is either

1. a dict that defines the foreground color, background color and attributes:

   fg Foreground color. Must be defined in *colors*.

   bg Background color. Must be defined in *colors*.

   attrs List of attributes. Valid values are one or more of `bold`, `italic` and `underline`. Note that some attributes may be unavailable in some applications or terminal emulators. If no attributes are needed this list should be left empty.

2. a string (an alias): a name of existing group. This group's definition will be used when this color is requested.

mode_translations Mode-specific highlighting for extensions that support it (e.g. the vim extension). It's an easy way of changing a color in a specific mode. Consists of a dict where the key is the mode and the value is a dict with the following options:

colors A dict where the key is the color to be translated in this mode, and the value is the new color. Both the key and the value must be defined in *colors*.

groups Segment highlighting groups for this mode. Same syntax as the main *groups* option.

Themes

Location `powerline/themes/top_theme.json`, `powerline/themes/extension/__main__.json`, `powerline/themes/extension/name.json`

Theme files are processed in order given: definitions from each next file override those from each previous file. It is required that file `powerline/themes/extension/name.json` exists.

{top_theme} component of the file name is obtained either from *top_theme extension-specific key* or from *default_top_theme common configuration key*. Powerline ships with the following top themes:

Theme	Description
powerline	Default powerline theme with fancy powerline symbols
powerline_unicode7	Theme with powerline dividers and unicode-7 symbols
unicode	Theme without any symbols from private use area
unicode_terminus	Theme containing only symbols from terminus PCF font
unicode_terminus_condensed	Like above, but occupies as less space as possible
powerline_terminus	Like unicode_terminus, but with powerline symbols
ascii	Theme without any unicode characters at all

name Name of the theme.

default_module Python module where segments will be looked by default. Defaults to `powerline.segments.{ext}`.

spaces Defines number of spaces just before the divider (on the right side) or just after it (on the left side). These spaces will not be added if divider is not drawn.

use_non_breaking_spaces Determines whether non-breaking spaces should be used in place of the regular ones. This option is needed because regular spaces are not displayed properly when using powerline with some font configuration. Defaults to `True`.

> **Note:** Unlike all other options this one is only checked once at startup using whatever theme is *the default*. If this option is set in the local themes it will be ignored. This option may also be ignored in some bindings.

outer_padding Defines number of spaces at the end of output (on the right side) or at the start of output (on the left side). Defaults to `1`.

dividers Defines the dividers used in all Powerline extensions.

The `hard` dividers are used to divide segments with different background colors, while the `soft` dividers are used to divide segments with the same background color.

cursor_space Space reserved for user input in shell bindings. It is measured in per cents.

cursor_columns Space reserved for user input in shell bindings. Unlike *cursor_space* it is measured in absolute amout of columns.

segment_data A dict where keys are segment names or strings `{module}.{function}`. Used to specify default values for various keys: *after*, *before*, *contents* (only for string segments if *name* is defined), *display*.

Key *args* (only for function and segment_list segments) is handled specially: unlike other values it is merged with all other values, except that a single `{module}.{function}` key if found prevents merging all `{function}` values.

When using *local themes* values of these keys are first searched in the segment description, then in `segment_data` key of a local theme, then in `segment_data` key of a *default theme*. For the *default theme* itself step 2 is obviously avoided.

> **Note:** Top-level themes are out of equation here: they are merged before the above merging process happens.

segments A dict with a `left` and a `right` lists, consisting of segment dictionaries. Shell themes may also contain `above` list of dictionaries. Each item in `above` list may have `left` and `right` keys like this dictionary, but no `above` key. `above` list is used for multiline shell configurations.

`left` and `right` lists are used for segments that should be put on the left or right side in the output. Actual mechanizm of putting segments on the left or the right depends on used renderer, but most renderers require one to specify segment with *width* `auto` on either side to make generated line fill all of the available width.

Each segment dictionary has the following options:

type The segment type. Can be one of `function` (default), `string` or `segment_list`:

function The segment contents is the return value of the function defined in the *function option*.

List of function segments is available in *Segment reference* section.

string A static string segment where the contents is defined in the *contents option*, and the highlighting group is defined in the *highlight_groups option*.

segment_list Sub-list of segments. This list only allows *function*, *segments* and *args* options.

List of lister segments is available in *Lister reference* section.

name Segment name. If present allows referring to this segment in *segment_data* dictionary by this name. If not `string` segments may not be referred there at all and `function` and `segment_list` segments may be referred there using either `{module}.{function_name}` or `{function_name}`, whichever will be found first. Function name is taken from *function key*.

> **Note:** If present prevents `function` key from acting as a segment name.

function Function used to get segment contents, in format `{module}.{function}` or `{function}`. If `{module}` is omitted *default_module option* is used.

highlight_groups Highlighting group for this segment. Consists of a prioritized list of highlighting groups, where the first highlighting group that is available in the colorscheme is used.

Ignored for segments that have `function` type.

before A string which will be prepended to the segment contents.

after A string which will be appended to the segment contents.

contents Segment contents, only required for `string` segments.

args A dict of arguments to be passed to a `function` segment.

align Aligns the segments contents to the left (`l`), center (`c`) or right (`r`). Has no sense if `width` key was not specified or if segment provides its own function for `auto width` handling and does not care about this option.

width Enforces a specific width for this segment.

This segment will work as a spacer if the width is set to `auto`. Several spacers may be used, and the space will be distributed equally among all the spacer segments. Spacers may have contents, either returned by a function or a static string, and the contents can be aligned with the `align` property.

priority Optional segment priority. Segments with priority `None` (the default priority, represented by `null` in json) will always be included, regardless of the width of the prompt/statusline.

If the priority is any number, the segment may be removed if the prompt/statusline width is too small for all the segments to be rendered. A lower number means that the segment has a higher priority.

Segments are removed according to their priority, with low priority segments (i.e. with a greater priority number) being removed first.

draw_hard_divider, draw_soft_divider Whether to draw a divider between this and the adjacent segment. The adjacent segment is to the *right* for segments on the *left* side, and vice versa. Hard dividers are used between segments with different background colors, soft ones are used between segments with same background. Both options default to `True`.

draw_inner_divider Determines whether inner soft dividers are to be drawn for function segments. Only applicable for functions returning multiple segments. Defaults to `False`.

exclude_modes, include_modes A list of modes where this segment will be excluded: the segment is not included or is included in all modes, *except* for the modes in one of these lists respectively. If `exclude_modes` is not present then it acts like an empty list (segment is not excluded from any modes). Without `include_modes` it acts like a list with all possible modes (segment is included in all modes). When there are both `exclude_modes` overrides `include_modes`.

exclude_function, include_function Function name in a form `{name}` or `{module}.{name}` (in the first form `{module}` defaults to `powerline.selectors.{ext}`). Determines under which condition specific segment will be included or excluded. By default segment is always included and never excluded. `exclude_function` overrides `include_function`.

> **Note:** Options *exclude_/include_modes* complement `exclude_/include_functions`: segment will be included if it is included by either `include_mode` or `include_function` and will be excluded if it is excluded by either `exclude_mode` or `exclude_function`.

display Boolean. If false disables displaying of the segment. Defaults to `True`.

segments A list of subsegments.

Segment reference

Segments

Segments are written in Python, and the default segments provided with Powerline are located in `powerline/segments/extension.py`. User-defined segments can be defined in any module in `sys.path` or *paths common configuration option*, import is always absolute.

Segments are regular Python functions, and they may accept arguments. All arguments should have a default value which will be used for themes that don't provide an `args` dict.

More information is available in *Writing segments* section.

Available segments

Common segments

VCS submodule

`powerline.segments.common.vcs.`**`branch`**(*ignore_statuses=()*, *status_colors=False*)

Return the current VCS branch.

Parameters

- **status_colors** (*bool*) – Determines whether repository status will be used to determine highlighting. Default: False.
- **ignore_statuses** (*bool*) – List of statuses which will not result in repo being marked as dirty. Most useful is setting this option to `["U"]`: this will ignore repository which has just untracked files (i.e. repository with modified, deleted or removed files will be marked as dirty, while just untracked files will make segment show clean repository). Only applicable if `status_colors` option is True.

Highlight groups used: `branch_clean`, `branch_dirty`, `branch`.

`powerline.segments.common.vcs.`**`stash`**()

Return the number of current VCS stash entries, if any.

Highlight groups used: `stash`.

System properties

`powerline.segments.common.sys.`**`cpu_load_percent`**(*interval=1*, *format=u'{0:.0f}%'*, *update_first=True*)

Return the average CPU load as a percentage.

Requires the `psutil` module.

Parameters **format** (*str*) – Output format. Accepts measured CPU load as the first argument.

Highlight groups used: `cpu_load_percent_gradient` (gradient) or `cpu_load_percent`.

`powerline.segments.common.sys.`**`system_load`**(*short=False*, *track_cpu_count=False*, *threshold_bad=2*, *threshold_good=1*, *format=u'{avg:.1f}'*)

Return system load average.

Highlights using `system_load_good`, `system_load_bad` and `system_load_ugly` highlighting groups, depending on the thresholds passed to the function.

Parameters

- **format** (*str*) – format string, receives `avg` as an argument
- **threshold_good** (*float*) – threshold for gradient level 0: any normalized load average below this value will have this gradient level.
- **threshold_bad** (*float*) – threshold for gradient level 100: any normalized load average above this value will have this gradient level. Load averages between `threshold_good` and `threshold_bad` receive gradient level that indicates relative position in this interval: (`100 * (cur-good) / (bad-good)`). Note: both parameters are checked against normalized load averages.
- **track_cpu_count** (*bool*) – if True powerline will continuously poll the system to detect changes in the number of CPUs.
- **short** (*bool*) – if True only the sys load over last 1 minute will be displayed.

Divider highlight group used: `background:divider`.

Highlight groups used: `system_load_gradient` (gradient) or `system_load`.

`powerline.segments.common.sys.`**`uptime`**(*shorten_len=3*, *seconds_format=u' {seconds:d}s'*, *minutes_format=u' {minutes:d}m'*, *hours_format=u' {hours:d}h'*, *days_format=u'{days:d}d'*)

Return system uptime.

Parameters

- **days_format** (*str*) – day format string, will be passed `days` as the argument
- **hours_format** (*str*) – hour format string, will be passed `hours` as the argument
- **minutes_format** (*str*) – minute format string, will be passed `minutes` as the argument
- **seconds_format** (*str*) – second format string, will be passed `seconds` as the argument
- **shorten_len** (*int*) – shorten the amount of units (days, hours, etc.) displayed

Divider highlight group used: `background:divider`.

Network

`powerline.segments.common.net.`**`external_ip`**(*interval=300*, *query_url=u'http://ipv4.icanhazip.com/'*)

Return external IP address.

Parameters **query_url** (*str*) – URI to query for IP address, should return only the IP address as a text string

Suggested URIs:

- http://ipv4.icanhazip.com/
- http://ipv6.icanhazip.com/
- http://icanhazip.com/ (returns IPv6 address if available, else IPv4)

Divider highlight group used: `background:divider`.

`powerline.segments.common.net.`**`hostname`**(*exclude_domain=False*, *only_if_ssh=False*)

Return the current hostname.

Parameters

- **only_if_ssh** (*bool*) – only return the hostname if currently in an SSH session
- **exclude_domain** (*bool*) – return the hostname without domain if there is one

`powerline.segments.common.net.`**`internal_ip`**(*ipv=4*, *interface=u'auto'*)

Return internal IP address

Requires `netifaces` module to work properly.

Parameters

- **interface** (*str*) – Interface on which IP will be checked. Use `auto` to automatically detect interface. In this case interfaces with lower numbers will be preferred over interfaces with similar names. Order of preference based on names:

  1. `eth` and `enp` followed by number or the end of string.
  2. `ath`, `wlan` and `wlp` followed by number or the end of string.
  3. `teredo` followed by number or the end of string.
  4. Any other interface that is not `lo*`.
  5. `lo` followed by number or the end of string.

  Use `default_gateway` to detect the interface based on the machine's default gateway[20] (i.e., the router to which it is connected).

- **ipv** (*int*) – 4 or 6 for ipv4 and ipv6 respectively, depending on which IP address you need exactly.

`powerline.segments.common.net.`**`network_load`**(*interval=1*, *update_first=True*, *interface=u'auto'*, *si_prefix=False*, *suffix=u'B/s'*, *sent_format=u'UL {value:>8}'*, *recv_format=u'DL {value:>8}'*)

Return the network load.

Uses the `psutil` module if available for multi-platform compatibility, falls back to reading `/sys/class/net/`*`interface`*`/statistics/`*`rx,tx`*`_bytes`.

Parameters

- **interface** (*str*) – Network interface to measure (use the special value "auto" to have powerline try to auto-detect the network interface).
- **suffix** (*str*) – String appended to each load string.
- **si_prefix** (*bool*) – Use SI prefix, e.g. MB instead of MiB.

[20] https://en.wikipedia.org/wiki/Default_gateway

- **recv_format** (*str*) – Format string that determines how download speed should look like. Receives `value` as argument.
- **sent_format** (*str*) – Format string that determines how upload speed should look like. Receives `value` as argument.
- **recv_max** (*float*) – Maximum number of received bytes per second. Is only used to compute gradient level.
- **sent_max** (*float*) – Maximum number of sent bytes per second. Is only used to compute gradient level.

Divider highlight group used: `network_load:divider`.

Highlight groups used: `network_load_sent_gradient` (gradient) or `network_load_recv_gradient` (gradient) or `network_load_gradient` (gradient), `network_load_sent` or `network_load_recv` or `network_load`.

Current environment

`powerline.segments.common.env.`**`cwd`**(*ellipsis=u'...'*, *use_path_separator=False*, *dir_limit_depth=None*, *dir_shorten_len=None*, *shorten_home=True*)

Return the current working directory.

Returns a segment list to create a breadcrumb-like effect.

Parameters

- **dir_shorten_len** (*int*) – shorten parent directory names to this length (e.g. `/long/path/to/powerline` → `/l/p/t/powerline`)
- **dir_limit_depth** (*int*) – limit directory depth to this number (e.g. `/long/path/to/powerline` → `⋯/to/powerline`)
- **use_path_separator** (*bool*) – Use path separator in place of soft divider.
- **shorten_home** (*bool*) – Shorten home directory to `~`.
- **ellipsis** (*str*) – Specifies what to use in place of omitted directories. Use None to not show this subsegment at all.

Divider highlight group used: `cwd:divider`.

Highlight groups used: `cwd:current_folder` or `cwd`. It is recommended to define all highlight groups.

`powerline.segments.common.env.`**`environment`**(*variable=None*)

Return the value of any defined environment variable

Parameters **variable** (*string*) – The environment variable to return if found

`powerline.segments.common.env.`**`user`**(*hide_domain=False*, *hide_user=None*)

Return the current user.

Parameters

- **hide_user** (*str*) – Omit showing segment for users with names equal to this string.
- **hide_domain** (*bool*) – Drop domain component if it exists in a username (delimited by '@').

Highlights the user with the `superuser` if the effective user ID is 0.

Highlight groups used: `superuser` or `user`. It is recommended to define all highlight groups.

`powerline.segments.common.env.virtualenv`(*ignore_conda=False*, *ignore_venv=False*)

Return the name of the current Python or conda virtualenv.

Parameters

- **ignore_venv** (*bool*) – Whether to ignore virtual environments. Default is False.
- **ignore_conda** (*bool*) – Whether to ignore conda environments. Default is False.

Battery

`powerline.segments.common.bat.battery`(*offline=u' '*, *online=u'C'*, *empty_heart=u'O'*, *full_heart=u'O'*, *gamify=False*, *steps=5*, *format=u'{ac_state} {capacity:3.0%}'*)

Return battery charge status.

Parameters

- **format** (*str*) – Percent format in case gamify is False. Format arguments: `ac_state` which is equal to either `online` or `offline` string arguments and `capacity` which is equal to current battery capacity in interval [0, 100].
- **steps** (*int*) – Number of discrete steps to show between 0% and 100% capacity if gamify is True.
- **gamify** (*bool*) – Measure in hearts (♥) instead of percentages. For full hearts `battery_full` highlighting group is preferred, for empty hearts there is `battery_empty`. `battery_online` or `battery_offline` group will be used for leading segment containing `online` or `offline` argument contents.
- **full_heart** (*str*) – Heart displayed for "full" part of battery.
- **empty_heart** (*str*) – Heart displayed for "used" part of battery. It is also displayed using another gradient level and highlighting group, so it is OK for it to be the same as full_heart as long as necessary highlighting groups are defined.
- **online** (*str*) – Symbol used if computer is connected to a power supply.
- **offline** (*str*) – Symbol used if computer is not connected to a power supply.

`battery_gradient` and `battery` groups are used in any case, first is preferred.

Highlight groups used: `battery_full` or `battery_gradient` (gradient) or `battery`, `battery_empty` or `battery_gradient` (gradient) or `battery`, `battery_online` or `battery_ac_state` or `battery_gradient` (gradient) or `battery`, `battery_offline` or `battery_ac_state` or `battery_gradient` (gradient) or `battery`.

Weather

`powerline.segments.common.wthr.weather`(*interval=600*, *update_first=True*, *location_query=None*, *temp_hottest=40*, *temp_coldest=-30*, *temp_format=None*, *unit=u'C'*, *icons=None*)

Return weather from Yahoo! Weather.

Uses GeoIP lookup from http://geoip.nekudo.com to automatically determine your current location. This should be changed if you're in a VPN or if your IP address is registered at another location.

Returns a list of colorized icon and temperature segments depending on weather conditions.

Parameters

- **unit** (*str*) – temperature unit, can be one of `F`, `C` or `K`
- **location_query** (*str*) – location query for your current location, e.g. `oslo, norway`
- **icons** (*dict*) – dict for overriding default icons, e.g. `{'heavy_snow' : u'❄'}`
- **temp_format** (*str*) – format string, receives `temp` as an argument. Should also hold unit.
- **temp_coldest** (*float*) – coldest temperature. Any temperature below it will have gradient level equal to zero.
- **temp_hottest** (*float*) – hottest temperature. Any temperature above it will have gradient level equal to 100. Temperatures between `temp_coldest` and `temp_hottest` receive gradient level that indicates relative position in this interval (`100 * (cur-coldest) / (hottest-coldest)`).

Divider highlight group used: `background:divider`.

Highlight groups used: `weather_conditions` or `weather`, `weather_temp_gradient` (gradient) or `weather`. Also uses `weather_conditions_{condition}` for all weather conditions supported by Yahoo.

Date and time

`powerline.segments.common.time.`**`date`**(*istime=False*, *format=u'%Y-%m-%d'*)

Return the current date.

Parameters

- **format** (*str*) – strftime-style date format string
- **istime** (*bool*) – If true then segment uses `time` highlight group.

Divider highlight group used: `time:divider`.

Highlight groups used: `time` or `date`.

`powerline.segments.common.time.`**`fuzzy_time`**(*unicode_text=False*)

Display the current time as fuzzy time, e.g. "quarter past six".

Parameters **unicode_text** (*bool*) – If true then hyphenminuses (regular ASCII -) and single quotes are replaced with unicode dashes and apostrophes.

Mail

`powerline.segments.common.mail.`**`email_imap_alert`**(*username*, *password*, *interval=60*, *update_first=True*, *use_ssl=None*, *folder=u'INBOX'*, *port=993*, *server=u'imap.gmail.com'*, *max_msgs=None*)

Return unread e-mail count for IMAP servers.

Parameters

- **username** (*str*) – login username
- **password** (*str*) – login password

- **server** (*str*) – e-mail server
- **port** (*int*) – e-mail server port
- **folder** (*str*) – folder to check for e-mails
- **max_msgs** (*int*) – Maximum number of messages. If there are more messages then max_msgs then it will use gradient level equal to 100, otherwise gradient level is equal to `100 * msgs_num / max_msgs`. If not present gradient is not computed.
- **use_ssl** (*bool*) – If `True` then use SSL connection. If `False` then do not use it. Default is `True` if port is equal to 993 and `False` otherwise.

Highlight groups used: `email_alert_gradient` (gradient), `email_alert`.

Media players

`powerline.segments.common.players.`**`clementine`**(*state_symbols={u'fallback': u'‘, u'pause': u'~', u'stop': u'X', u'play': u'>'}, format=u'{state_symbol} {artist} - {title} ({total})'*)

Return clementine player information

Requires `dbus` python module.

This player segment should be added like this:

```
{
        "function": "powerline.segments.common.players.clementine",
        "name": "player"
}
```

(with additional `"args": {...}` if needed).

Highlight groups used: `player_fallback` or `player`, `player_play` or `player`, `player_pause` or `player`, `player_stop` or `player`.

Parameters

- **format** (*str*) – Format used for displaying data from player. Should be a str.format-like string with the following keyword parameters:

Parameter	Description
state_symbol	Symbol displayed for play/pause/stop states. There is also “fallback” state used in case function failed to get player state. For this state symbol is by default empty. All symbols are defined in `state_symbols` argument.
album	Album that is currently played.
artist	Artist whose song is currently played
title	Currently played composition.
elapsed	Composition duration in format M:SS (minutes:seconds).
total	Composition length in format M:SS.

- **state_symbols** (*dict*) – Symbols used for displaying state. Must contain all of the following keys:

Key	Description
play	Displayed when player is playing.
pause	Displayed when player is paused.
stop	Displayed when player is not playing anything.
fallback	Displayed if state is not one of the above or not known.

`powerline.segments.common.players.`**`cmus`**(*state_symbols={u'fallback': u'‘, u'pause': u'~', u'stop': u'X', u'play': u'>'}, format=u'{state_symbol} {artist} - {title} ({total})'*)

Return CMUS player information

Requires cmus-remote command be acessible from $PATH.

This player segment should be added like this:

```
{
        "function": "powerline.segments.common.players.cmus",
        "name": "player"
}
```

(with additional `"args": {...}` if needed).

Highlight groups used: `player_fallback` or `player`, `player_play` or `player`, `player_pause` or `player`, `player_stop` or `player`.

Parameters

- **`format`** (*`str`*) – Format used for displaying data from player. Should be a str.format-like string with the following keyword parameters:

Parameter	Description
state_symbol	Symbol displayed for play/pause/stop states. There is also “fallback” state used in case function failed to get player state. For this state symbol is by default empty. All symbols are defined in `state_symbols` argument.
album	Album that is currently played.
artist	Artist whose song is currently played
title	Currently played composition.
elapsed	Composition duration in format M:SS (minutes:seconds).
total	Composition length in format M:SS.

- **`state_symbols`** (*`dict`*) – Symbols used for displaying state. Must contain all of the following keys:

Key	Description
play	Displayed when player is playing.
pause	Displayed when player is paused.
stop	Displayed when player is not playing anything.
fallback	Displayed if state is not one of the above or not known.

`powerline.segments.common.players.`**`dbus_player`**(*player_name, state_symbols={u'fallback': u'‘, u'pause': u'~', u'stop': u'X', u'play': u'>'}, format=u'{state_symbol} {artist} - {title} ({total})'*)

Return generic dbus player state

Requires `dbus` python module. Only for players that support specific protocol (e.g. like *`spotify()`* and *`clementine()`*).

This player segment should be added like this:

```
{
        "function": "powerline.segments.common.players.dbus_player",
        "name": "player"
}
```

(with additional `"args": {...}` if needed).

Highlight groups used: `player_fallback` or `player`, `player_play` or `player`, `player_pause` or `player`, `player_stop` or `player`.

Parameters

- **`format`** (*`str`*) – Format used for displaying data from player. Should be a str.format-like string with the following keyword parameters:

Parameter	Description
state_symbol	Symbol displayed for play/pause/stop states. There is also "fallback" state used in case function failed to get player state. For this state symbol is by default empty. All symbols are defined in `state_symbols` argument.
album	Album that is currently played.
artist	Artist whose song is currently played
title	Currently played composition.
elapsed	Composition duration in format M:SS (minutes:seconds).
total	Composition length in format M:SS.

- **`state_symbols`** (*`dict`*) – Symbols used for displaying state. Must contain all of the following keys:

Key	Description
play	Displayed when player is playing.
pause	Displayed when player is paused.
stop	Displayed when player is not playing anything.
fallback	Displayed if state is not one of the above or not known.

- **`player_name`** (*`str`*) – Player name. Used in error messages only.
- **`bus_name`** (*`str`*) – Dbus bus name.
- **`player_path`** (*`str`*) – Path to the player on the given bus.
- **`iface_prop`** (*`str`*) – Interface properties name for use with dbus.Interface.
- **`iface_player`** (*`str`*) – Player name.

`powerline.segments.common.players.`**`itunes`**(*state_symbols={u'fallback': u'‘', u'pause': u'~', u'stop': u'X', u'play': u'>'}, format=u'{state_symbol} {artist} - {title} ({total})'*)

Return iTunes now playing information

Requires `osascript`.

This player segment should be added like this:

```
{
        "function": "powerline.segments.common.players.itunes",
        "name": "player"
}
```

(with additional `"args": {...}` if needed).

Highlight groups used: `player_fallback` or `player`, `player_play` or `player`, `player_pause` or `player`, `player_stop` or `player`.

Parameters

- **format** (*str*) – Format used for displaying data from player. Should be a str.format-like string with the following keyword parameters:

Parameter	Description
state_symbol	Symbol displayed for play/pause/stop states. There is also "fallback" state used in case function failed to get player state. For this state symbol is by default empty. All symbols are defined in `state_symbols` argument.
album	Album that is currently played.
artist	Artist whose song is currently played
title	Currently played composition.
elapsed	Composition duration in format M:SS (minutes:seconds).
total	Composition length in format M:SS.

- **state_symbols** (*dict*) – Symbols used for displaying state. Must contain all of the following keys:

Key	Description
play	Displayed when player is playing.
pause	Displayed when player is paused.
stop	Displayed when player is not playing anything.
fallback	Displayed if state is not one of the above or not known.

`powerline.segments.common.players.`**mpd**(*state_symbols={u'fallback': u'‘', u'pause': u'~', u'stop': u'X', u'play': u'>'}, format=u'{state_symbol} {artist} - {title} ({total})', port=6600, password=None, host=u'localhost'*)

Return Music Player Daemon information

Requires `mpd` Python module (e.g. `python-mpd2`[21] or `python-mpd`[22] Python package) or alternatively the `mpc` command to be acessible from $PATH.

This player segment should be added like this:

```
{
        "function": "powerline.segments.common.players.mpd",
        "name": "player"
}
```

(with additional `"args": {...}` if needed).

Highlight groups used: `player_fallback` or `player`, `player_play` or `player`, `player_pause` or `player`, `player_stop` or `player`.

[21] https://pypi.python.org/pypi/python-mpd2
[22] https://pypi.python.org/pypi/python-mpd

Parameters

- **format** (*str*) – Format used for displaying data from player. Should be a str.format-like string with the following keyword parameters:

Parameter	Description
state_symbol	Symbol displayed for play/pause/stop states. There is also "fallback" state used in case function failed to get player state. For this state symbol is by default empty. All symbols are defined in `state_symbols` argument.
album	Album that is currently played.
artist	Artist whose song is currently played
title	Currently played composition.
elapsed	Composition duration in format M:SS (minutes:seconds).
total	Composition length in format M:SS.

- **state_symbols** (*dict*) – Symbols used for displaying state. Must contain all of the following keys:

Key	Description
play	Displayed when player is playing.
pause	Displayed when player is paused.
stop	Displayed when player is not playing anything.
fallback	Displayed if state is not one of the above or not known.

- **host** (*str*) – Host on which mpd runs.
- **password** (*str*) – Password used for connecting to daemon.
- **port** (*int*) – Port which should be connected to.

`powerline.segments.common.players.`**`rdio`**(*state_symbols={u'fallback': u'‘', u'pause': u'~', u'stop': u'X', u'play': u'>'}, format=u'{state_symbol} {artist} - {title} ({total})'*)

Return rdio player information

Requires `osascript` available in $PATH.

This player segment should be added like this:

```
{
        "function": "powerline.segments.common.players.rdio",
        "name": "player"
}
```

(with additional `"args": {...}` if needed).

Highlight groups used: `player_fallback` or `player`, `player_play` or `player`, `player_pause` or `player`, `player_stop` or `player`.

Parameters

- **format** (*str*) – Format used for displaying data from player. Should be a str.format-like string with the following keyword parameters:

Parameter	Description
state_symbol	Symbol displayed for play/pause/stop states. There is also "fallback" state used in case function failed to get player state. For this state symbol is by default empty. All symbols are defined in `state_symbols` argument.
album	Album that is currently played.
artist	Artist whose song is currently played
title	Currently played composition.
elapsed	Composition duration in format M:SS (minutes:seconds).
total	Composition length in format M:SS.

- **`state_symbols`** (*`dict`*) – Symbols used for displaying state. Must contain all of the following keys:

Key	Description
play	Displayed when player is playing.
pause	Displayed when player is paused.
stop	Displayed when player is not playing anything.
fallback	Displayed if state is not one of the above or not known.

`powerline.segments.common.players.`**`rhythmbox`**(*state_symbols={u'fallback': u'‘', u'pause': u'~', u'stop': u'X', u'play': u'>'}, format=u'{state_symbol} {artist} - {title} ({total})'*)

Return rhythmbox player information

Requires `rhythmbox-client` available in $PATH.

This player segment should be added like this:

```
{
        "function": "powerline.segments.common.players.rhythmbox",
        "name": "player"
}
```

(with additional `"args": {...}` if needed).

Highlight groups used: `player_fallback` or `player`, `player_play` or `player`, `player_pause` or `player`, `player_stop` or `player`.

Parameters

- **`format`** (*`str`*) – Format used for displaying data from player. Should be a str.format-like string with the following keyword parameters:

Parameter	Description
state_symbol	Symbol displayed for play/pause/stop states. There is also "fallback" state used in case function failed to get player state. For this state symbol is by default empty. All symbols are defined in `state_symbols` argument.
album	Album that is currently played.
artist	Artist whose song is currently played
title	Currently played composition.
elapsed	Composition duration in format M:SS (minutes:seconds).
total	Composition length in format M:SS.

- **state_symbols** (*dict*) – Symbols used for displaying state. Must contain all of the following keys:

Key	Description
play	Displayed when player is playing.
pause	Displayed when player is paused.
stop	Displayed when player is not playing anything.
fallback	Displayed if state is not one of the above or not known.

`powerline.segments.common.players.`**`spotify`**(*state_symbols={u'fallback': u'‘', u'pause': u'~', u'stop': u'X', u'play': u'>'}, format=u'{state_symbol} {artist} - {title} ({total})'*)

Return spotify player information

Requires `dbus` python module.

This player segment should be added like this:

```
{
        "function": "powerline.segments.common.players.spotify",
        "name": "player"
}
```

(with additional `"args": {...}` if needed).

Highlight groups used: `player_fallback` or `player`, `player_play` or `player`, `player_pause` or `player`, `player_stop` or `player`.

Parameters

- **format** (*str*) – Format used for displaying data from player. Should be a str.format-like string with the following keyword parameters:

Parameter	Description
state_symbol	Symbol displayed for play/pause/stop states. There is also “fallback” state used in case function failed to get player state. For this state symbol is by default empty. All symbols are defined in `state_symbols` argument.
album	Album that is currently played.
artist	Artist whose song is currently played
title	Currently played composition.
elapsed	Composition duration in format M:SS (minutes:seconds).
total	Composition length in format M:SS.

- **state_symbols** (*dict*) – Symbols used for displaying state. Must contain all of the following keys:

Key	Description
play	Displayed when player is playing.
pause	Displayed when player is paused.
stop	Displayed when player is not playing anything.
fallback	Displayed if state is not one of the above or not known.

`powerline.segments.common.players.`**`spotify_apple_script`**(*state_symbols={u'fallback': u'‘', u'pause': u'~', u'stop': u'X', u'play': u'>'}, format=u'{state_symbol} {artist} - {title} ({total})'*)

Return spotify player information

Requires `osascript` available in $PATH.

This player segment should be added like this:

```
{
        "function": "powerline.segments.common.players.spotify_apple_script",
        "name": "player"
}
```

(with additional `"args": {...}` if needed).

Highlight groups used: `player_fallback` or `player`, `player_play` or `player`, `player_pause` or `player`, `player_stop` or `player`.

Parameters

- **`format`** (*`str`*) – Format used for displaying data from player. Should be a str.format-like string with the following keyword parameters:

Parameter	Description
state_symbol	Symbol displayed for play/pause/stop states. There is also “fallback” state used in case function failed to get player state. For this state symbol is by default empty. All symbols are defined in `state_symbols` argument.
album	Album that is currently played.
artist	Artist whose song is currently played
title	Currently played composition.
elapsed	Composition duration in format M:SS (minutes:seconds).
total	Composition length in format M:SS.

- **`state_symbols`** (*`dict`*) – Symbols used for displaying state. Must contain all of the following keys:

Key	Description
play	Displayed when player is playing.
pause	Displayed when player is paused.
stop	Displayed when player is not playing anything.
fallback	Displayed if state is not one of the above or not known.

`powerline.segments.common.players.`**`spotify_dbus`**(*state_symbols={u'fallback': u'‘', u'pause': u'~', u'stop': u'X', u'play': u'>'}, format=u'{state_symbol} {artist} - {title} ({total})'*)

Return spotify player information

Requires `dbus` python module.

This player segment should be added like this:

```
{
        "function": "powerline.segments.common.players.spotify",
```

```
        "name": "player"
}
```

(with additional `"args": {...}` if needed).

Highlight groups used: `player_fallback` or `player`, `player_play` or `player`, `player_pause` or `player`, `player_stop` or `player`.

Parameters

- **`format`** (*`str`*) – Format used for displaying data from player. Should be a str.format-like string with the following keyword parameters:

Parameter	Description
state_symbol	Symbol displayed for play/pause/stop states. There is also "fallback" state used in case function failed to get player state. For this state symbol is by default empty. All symbols are defined in `state_symbols` argument.
album	Album that is currently played.
artist	Artist whose song is currently played
title	Currently played composition.
elapsed	Composition duration in format M:SS (minutes:seconds).
total	Composition length in format M:SS.

- **`state_symbols`** (*`dict`*) – Symbols used for displaying state. Must contain all of the following keys:

Key	Description
play	Displayed when player is playing.
pause	Displayed when player is paused.
stop	Displayed when player is not playing anything.
fallback	Displayed if state is not one of the above or not known.

i3wm segments

`powerline.segments.i3wm.`**`mode`**(*names={u'default': None}*)

Returns current i3 mode

Parameters **`names`** (*`dict`*) – Specifies the string to show for various modes. Use `null` to hide a mode (`default` is hidden by default).

Highligh groups used: `mode`

`powerline.segments.i3wm.`**`scratchpad`**(*icons={u'fresh': u'O', u'changed': u'X'}*)

Returns the windows currently on the scratchpad

Parameters **`icons`** (*`dict`*) – Specifies the strings to show for the different scratchpad window states. Must contain the keys `fresh` and `changed`.

Highlight groups used: `scratchpad` or `scratchpad:visible`, `scratchpad` or `scratchpad:focused`, `scratchpad` or `scratchpad:urgent`.

`powerline.segments.i3wm.`**`workspace`**(*strip=False, workspace=None*)

Return the specified workspace name

Parameters

- **workspace** (*str*) – Specifies which workspace to show. If unspecified, may be set by the `list_workspaces` lister if used, otherwise falls back to currently focused workspace.
- **strip** (*bool*) – Specifies whether workspace numbers (in the `1: name` format) should be stripped from workspace names before being displayed. Defaults to false.

Highlight groups used: `workspace` or `w_visible`, `workspace` or `w_focused`, `workspace` or `w_urgent`.

`powerline.segments.i3wm.`**`workspaces`**(*strip=0*, *output=None*, *only_show=None*)

Return list of used workspaces

Parameters

- **only_show** (*list*) – Specifies which workspaces to show. Valid entries are `"visible"`, `"urgent"` and `"focused"`. If omitted or `null` all workspaces are shown.
- **output** (*str*) – May be set to the name of an X output. If specified, only workspaces on that output are shown. Overrides automatic output detection by the lemonbar renderer and bindings.
- **strip** (*int*) – Specifies how many characters from the front of each workspace name should be stripped (e.g. to remove workspace numbers). Defaults to zero.

Highlight groups used: `workspace` or `w_visible`, `workspace` or `w_focused`, `workspace` or `w_urgent`.

PDB segments

`powerline.segments.pdb.`**`current_code_name`**()

Displays name of the code object of the current frame

`powerline.segments.pdb.`**`current_context`**()

Displays currently executed context name

This is similar to *`current_code_name()`*, but gives more details.

Currently it only gives module file name if code_name happens to be `<module>`.

`powerline.segments.pdb.`**`current_file`**(*basename=True*)

Displays current file name

Parameters **basename** (*bool*) – If true only basename is displayed.

`powerline.segments.pdb.`**`current_line`**()

Displays line number that is next to be run

`powerline.segments.pdb.`**`stack_depth`**(*full_stack=False*)

Displays current stack depth

Result is relative to the stack depth at the time prompt was first run.

Parameters **full_stack** (*bool*) – If true then absolute depth is used.

Shell segments

`powerline.segments.shell.`**`continuation`**(*renames={}*, *right_align=False*, *omit_cmdsubst=True*)

Display parser state.

Parameters

- **omit_cmdsubst** (*bool*) – Do not display cmdsubst parser state if it is the last one.
- **right_align** (*bool*) – Align to the right.
- **renames** (*dict*) – Rename states: `{old_name : new_name}`. If `new_name` is `None` then given state is not displayed.

Highlight groups used: `continuation`, `continuation:current`.

`powerline.segments.shell.`**`cwd`**(*ellipsis=u'...'*, *use_path_separator=False*, *dir_limit_depth=None*, *dir_shorten_len=None*, *use_shortened_path=True*)

Return the current working directory.

Returns a segment list to create a breadcrumb-like effect.

Parameters

- **dir_shorten_len** (*int*) – shorten parent directory names to this length (e.g. `/long/path/to/powerline` → `/l/p/t/powerline`)
- **dir_limit_depth** (*int*) – limit directory depth to this number (e.g. `/long/path/to/powerline` → `⋯/to/powerline`)
- **use_path_separator** (*bool*) – Use path separator in place of soft divider.
- **use_shortened_path** (*bool*) – Use path from shortened_path `--renderer-arg` argument. If this argument is present `shorten_home` argument is ignored.
- **shorten_home** (*bool*) – Shorten home directory to `~`.
- **ellipsis** (*str*) – Specifies what to use in place of omitted directories. Use None to not show this subsegment at all.

Divider highlight group used: `cwd:divider`.

Highlight groups used: `cwd:current_folder` or `cwd`. It is recommended to define all highlight groups.

`powerline.segments.shell.`**`jobnum`**(*show_zero=False*)

Return the number of jobs.

Parameters **show_zero** (*bool*) – If False (default) shows nothing if there are no jobs. Otherwise shows zero for no jobs.

`powerline.segments.shell.`**`last_pipe_status`**()

Return last pipe status.

Highlight groups used: `exit_fail`, `exit_success`

`powerline.segments.shell.`**`last_status`**()

Return last exit code.

Highlight groups used: `exit_fail`

`powerline.segments.shell.`**`mode`**(*default=None*, *override={u'vicmd': u'COMMND', u'viins': u'INSERT'}*)

Return the current mode.

Parameters

- **override** (*dict*) – dict for overriding mode strings.
- **default** (*str*) – If current mode is equal to this string then this segment will not get displayed. If not specified the value is taken from `$POWERLINE_DEFAULT_MODE` variable. This variable is set by zsh bindings for any mode that does not start from `vi`.

Tmux segments

`powerline.segments.tmux.`**`attached_clients`**(*minimum=1*)

Return the number of tmux clients attached to the currently active session

Parameters **`minimum`** (*`int`*) – The minimum number of attached clients that must be present for this segment to be visible.

Vim segments

`powerline.segments.vim.`**`branch`**(*ignore_statuses=()*, *status_colors=False*)

Return the current working branch.

Parameters

- **`status_colors`** (*`bool`*) – Determines whether repository status will be used to determine highlighting. Default: False.
- **`ignore_statuses`** (*`bool`*) – List of statuses which will not result in repo being marked as dirty. Most useful is setting this option to `["U"]`: this will ignore repository which has just untracked files (i.e. repository with modified, deleted or removed files will be marked as dirty, while just untracked files will make segment show clean repository). Only applicable if `status_colors` option is True.

Highlight groups used: `branch_clean`, `branch_dirty`, `branch`.

Divider highlight group used: `branch:divider`.

`powerline.segments.vim.`**`bufnr`**(*show_current=True*)

Show buffer number

Parameters **`show_current`** (*`bool`*) – If False do not show current window number.

`powerline.segments.vim.`**`col_current`**()

Return the current cursor column.

`powerline.segments.vim.`**`csv_col_current`**(*name_format=u' ({column_name:.15})'*, *display_name=u'auto'*)

Display CSV column number and column name

Requires filetype to be set to `csv`.

Parameters

- **`or str name`** (*`bool`*) – May be `True`, `False` and `"auto"`. In the first case value from the first raw will always be displayed. In the second case it will never be displayed. In thi last case `csv.Sniffer().has_header()` will be used to detect whether current file contains header in the first column.
- **`name_format`** (*`str`*) – String used to format column name (in case `display_name` is set to `True` or `"auto"`). Accepts `column_name` keyword argument.

Highlight groups used: `csv:column_number` or `csv`, `csv:column_name` or `csv`.

`powerline.segments.vim.`**`file_directory`**(*shorten_home=False*, *shorten_cwd=True*, *shorten_user=True*, *remove_scheme=True*)

Return file directory (head component of the file path).

Parameters

- **remove_scheme** (*bool*) – Remove scheme part from the segment name, if present. See documentation of file_scheme segment for the description of what scheme is. Also removes the colon.
- **shorten_user** (*bool*) – Shorten `$HOME` directory to `~/`. Does not work for files with scheme.
- **shorten_cwd** (*bool*) – Shorten current directory to `./`. Does not work for files with scheme present.
- **shorten_home** (*bool*) – Shorten all directories in `/home/` to `~user/` instead of `/home/user/`. Does not work for files with scheme present.

`powerline.segments.vim.`**`file_encoding`**(*segment_info*)

Return file encoding/character set.

Returns file encoding/character set or None if unknown or missing file encoding

Divider highlight group used: `background:divider`.

`powerline.segments.vim.`**`file_format`**(*segment_info*)

Return file format (i.e. line ending type).

Returns file format or None if unknown or missing file format

Divider highlight group used: `background:divider`.

`powerline.segments.vim.`**`file_name`**(*no_file_text=u'[No file]'*, *display_no_file=False*)

Return file name (tail component of the file path).

Parameters

- **display_no_file** (*bool*) – display a string if the buffer is missing a file name
- **no_file_text** (*str*) – the string to display if the buffer is missing a file name

Highlight groups used: `file_name_no_file` or `file_name`, `file_name`.

`powerline.segments.vim.`**`file_scheme`**()

Return the protocol part of the file.

Protocol is the part of the full filename just before the colon which starts with a latin letter and contains only latin letters, digits, plus, period or hyphen (refer to RFC3986[23] for the description of URI scheme). If there is no such a thing `None` is returned, effectively removing segment.

Note: Segment will not check whether there is `//` just after the colon or if there is at least one slash after the scheme. Reason: it is not always present. E.g. when opening file inside a zip archive file name will look like `zipfile:/path/to/archive.zip::file.txt`. `file_scheme` segment will catch `zipfile` part here.

`powerline.segments.vim.`**`file_size`**(*si_prefix=False*, *suffix=u'B'*)

Return file size in &encoding.

Parameters

- **suffix** (*str*) – string appended to the file size
- **si_prefix** (*bool*) – use SI prefix, e.g. MB instead of MiB

Returns file size or None if the file isn't saved or if the size is too big to fit in a number

[23] http://tools.ietf.org/html/rfc3986#section-3.1

`powerline.segments.vim.`**`file_type`**(*segment_info*)
Return file type.

Returns file type or None if unknown file type

Divider highlight group used: `background:divider`.

`powerline.segments.vim.`**`file_vcs_status`**()
Return the VCS status for this buffer.

Highlight groups used: `file_vcs_status`.

`powerline.segments.vim.`**`line_count`**()
Return the line count of the current buffer.

`powerline.segments.vim.`**`line_current`**()
Return the current cursor line.

`powerline.segments.vim.`**`line_percent`**(*gradient=False*)
Return the cursor position in the file as a percentage.

Parameters **`gradient`** (*`bool`*) – highlight the percentage with a color gradient (by default a green to red gradient)

Highlight groups used: `line_percent_gradient` (gradient), `line_percent`.

`powerline.segments.vim.`**`mode`**(*override=None*)
Return the current vim mode.

If mode (returned by `mode()` VimL function, see `:h mode()` in Vim) consists of multiple characters and necessary mode is not known to powerline then it will fall back to mode with last character(s) ignored.

Parameters **`override`** (*`dict`*) – dict for overriding default mode strings, e.g. `{ 'n': 'NORM' }`

`powerline.segments.vim.`**`modified_buffers`**(*join_str=u', ', text=u'+ '*)
Return a comma-separated list of modified buffers.

Parameters

- **`text`** (*`str`*) – text to display before the modified buffer list
- **`join_str`** (*`str`*) – string to use for joining the modified buffer list

`powerline.segments.vim.`**`modified_indicator`**(*text=u'+'*)
Return a file modified indicator.

Parameters **`text`** (*`string`*) – text to display if the current buffer is modified

`powerline.segments.vim.`**`paste_indicator`**(*text=u'PASTE'*)
Return a paste mode indicator.

Parameters **`text`** (*`string`*) – text to display if paste mode is enabled

`powerline.segments.vim.`**`position`**(*gradient=False, position_strings={u'top': u'Top', u'all': u'All', u'bottom': u'Bot'}*)
Return the position of the current view in the file as a percentage.

Parameters

- **`position_strings`** (*`dict`*) – dict for translation of the position strings, e.g. `{"top":"Oben", "bottom":"Unten", "all":"Alles"}`
- **`gradient`** (*`bool`*) – highlight the percentage with a color gradient (by default a green to red gradient)

Highlight groups used: `position_gradient` (gradient), `position`.

`powerline.segments.vim.readonly_indicator(text=u'RO')`
Return a read-only indicator.

Parameters **text** (*string*) – text to display if the current buffer is read-only

`powerline.segments.vim.stash()`
Return the number of stashes in the current working branch.

Highlight groups used: `stash`.

`powerline.segments.vim.tab(end=False)`
Mark start of the clickable region for tabpage

Parameters **end** (*bool*) – In place of starting region for the current tab end it.

No highlight groups are used (literal segment).

`powerline.segments.vim.tab_modified_indicator(text=u'+')`
Return a file modified indicator for tabpages.

Parameters **text** (*string*) – text to display if any buffer in the current tab is modified

Highlight groups used: `tab_modified_indicator` or `modified_indicator`.

`powerline.segments.vim.tabnr(show_current=True)`
Show tabpage number

Parameters **show_current** (*bool*) – If False do not show current tabpage number. This is default because tabnr is by default only present in tabline.

`powerline.segments.vim.trailing_whitespace()`
Return the line number for trailing whitespaces

It is advised not to use this segment in insert mode: in Insert mode it will iterate over all lines in buffer each time you happen to type a character which may cause lags. It will also show you whitespace warning each time you happen to type space.

Highlight groups used: `trailing_whitespace` or `warning`.

`powerline.segments.vim.virtcol_current(gradient=True)`
Return current visual column with concealed characters ingored

Parameters **gradient** (*bool*) – Determines whether it should show textwidth-based gradient (gradient level is `virtcol * 100 / textwidth`).

Highlight groups used: `virtcol_current_gradient` (gradient), `virtcol_current` or `col_current`.

`powerline.segments.vim.visual_range(V_text=u'L:{rows}', v_text_multiline=u'L:{rows}', v_text_oneline=u'C:{vcols}', CTRL_V_text=u'{rows} x {vcols}')`
Return the current visual selection range.

Parameters

- **CTRL_V_text** (*str*) – Text to display when in block visual or select mode.
- **v_text_oneline** (*str*) – Text to display when in charaterwise visual or select mode, assuming selection occupies only one line.
- **v_text_multiline** (*str*) – Text to display when in charaterwise visual or select mode, assuming selection occupies more then one line.
- **V_text** (*str*) – Text to display when in linewise visual or select mode.

All texts are format strings which are passed the following parameters:

Parameter	Description
sline	Line number of the first line of the selection
eline	Line number of the last line of the selection
scol	Column number of the first character of the selection
ecol	Column number of the last character of the selection
svcol	Virtual column number of the first character of the selection
secol	Virtual column number of the last character of the selection
rows	Number of lines in the selection
cols	Number of columns in the selection
vcols	Number of virtual columns in the selection

`powerline.segments.vim.`**`window_title`**`()`

Return the window title.

This currently looks at the `quickfix_title` window variable, which is used by Syntastic and Vim itself.

It is used in the quickfix theme.

`powerline.segments.vim.`**`winnr`**`(`*show_current=True*`)`

Show window number

Parameters **`show_current`** (*`bool`*) – If False do not show current window number.

Plugin-specific segments

Syntastic segments

`powerline.segments.vim.plugin.syntastic.`**`syntastic`**`(`*warn_format=u'WARN: \ue0a1 {first_line} ({num}) ', err_format=u'ERR: \ue0a1 {first_line} ({num}) '*`)`

Show whether syntastic has found any errors or warnings

Parameters

- **`err_format`** (*`str`*) – Format string for errors.
- **`warn_format`** (*`str`*) – Format string for warnings.

Highlight groups used: `syntastic:warning` or `warning`, `syntastic:error` or `error`.

Command-T segments

`powerline.segments.vim.plugin.commandt.`**`finder`**`()`

Display Command-T finder name

Requires $command_t.active_finder and methods (code above may monkey-patch $command_t to add them). All Command-T finders have `CommandT::` module prefix, but it is stripped out (actually, any `CommandT::` substring will be stripped out).

Highlight groups used: `commandt:finder`.

`powerline.segments.vim.plugin.commandt.`**`path`**`()`

Display path used by Command-T

Requires $command_t.active_finder and .path methods (code above may monkey-patch $command_t to add them).

$command_t.active_finder is required in order to omit displaying path for finders `MRUBufferFinder`, `BufferFinder`, `TagFinder` and `JumpFinder` (pretty much any finder, except `FileFinder`).

Highlight groups used: `commandt:path`.

Tagbar segments

`powerline.segments.vim.plugin.tagbar.`**`current_tag`**(*flags=u's'*)

Return tag that is near the cursor.

Parameters **`flags`** (*`str`*) – Specifies additional properties of the displayed tag. Supported values:

- s - display complete signature
- f - display the full hierarchy of the tag
- p - display the raw prototype

More info in the official documentation[24] (search for "tagbar#currenttag").

NERDTree segments

`powerline.segments.vim.plugin.nerdtree.`**`nerdtree`**()

Return directory that is shown by the current buffer.

Highlight groups used: `nerdtree:path` or `file_name`.

Capslock segments

`powerline.segments.vim.plugin.capslock.`**`capslock_indicator`**(*text=u'CAPS'*)

Shows the indicator if tpope/vim-capslock plugin is enabled

Note: In the current state plugin automatically disables itself when leaving insert mode. So trying to use this segment not in insert or replace modes is useless.

Parameters **`text`** (*`str`*) – String to show when software capslock presented by this plugin is active.

Lister reference

Listers are special segment collections which allow to show some list of segments for each entity in the list of entities (multiply their segments list by a list of entities). E.g. `powerline.listers.vim.tablister` presented with `powerline.segments.vim.tabnr` and `....file_name` as segments will emit segments with buffer names and tabpage numbers for each tabpage shown by vim.

Listers appear in configuration as irregular segments having `segment_list` as their type and `segments` key with a list of segments (a bit more details in *Themes section of configuration reference*).

More information in *Writing listers* section.

[24] https://github.com/majutsushi/tagbar/blob/master/doc/tagbar.txt

Vim listers

`powerline.listers.vim.`**`bufferlister`**(*show_unlisted=False*)

List all buffers in segment_info format

Specifically generates a list of segment info dictionaries with `buffer` and `bufnr` keys set to buffer-specific ones, `window`, `winnr` and `window_id` keys set to None.

Adds one of `buf:`, `buf_nc:`, `buf_mod:`, or `buf_nc_mod` prefix to all segment highlight groups.

Parameters **`show_unlisted`** (*`bool`*) – True if unlisted buffers should be shown as well. Current buffer is always shown.

`powerline.listers.vim.`**`tablister`**()

List all tab pages in segment_info format

Specifically generates a list of segment info dictionaries with `window`, `winnr`, `window_id`, `buffer` and `bufnr` keys set to tab-local ones and additional `tabpage` and `tabnr` keys.

Adds either `tab:` or `tab_nc:` prefix to all segment highlight groups.

Works best with vim-7.4 or later: earlier versions miss tabpage object and thus window objects are not available as well.

Pdb listers

`powerline.listers.pdb.`**`frame_lister`**(*maxframes=3*, *full_stack=False*)

List all frames in segment_info format

Parameters

- **`full_stack`** (*`bool`*) – If true, then all frames in the stack are listed. Normally N first frames are discarded where N is a number of frames present at the first invocation of the prompt minus one.
- **`maxframes`** (*`int`*) – Maximum number of frames to display.

i3wm listers

`powerline.listers.i3wm.`**`output_lister`**()

List all outputs in segment_info format

`powerline.listers.i3wm.`**`workspace_lister`**(*output=None*, *only_show=None*)

List all workspaces in segment_info format

Sets the segment info values of `workspace` and `output` to the name of the i3 workspace and the `xrandr` output respectively and the keys `"visible"`, `"urgent"` and `"focused"` to a boolean indicating these states.

Parameters

- **`only_show`** (*`list`*) – Specifies which workspaces to list. Valid entries are `"visible"`, `"urgent"` and `"focused"`. If omitted or `null` all workspaces are listed.
- **`output`** (*`str`*) – May be set to the name of an X output. If specified, only workspaces on that output are listed. Overrides automatic output detection by the lemonbar renderer and bindings. Set to `false` to force all workspaces to be shown.

Selector functions

Selector functions are functions that return `True` or `False` depending on application state. They are used for *exclude_function and include_function segment options*.

Available selectors

Vim selectors

`powerline.selectors.vim.`**`single_tab`**(*segment_info*, *mode*)

Returns True if Vim has only one tab opened

Local configuration overrides

Depending on the application used it is possible to override configuration. Here is the list:

Vim overrides

Vim configuration can be overridden using the following options:

`g:powerline_config_overrides` Dictionary, recursively merged with contents of `powerline/config.json`.

`g:powerline_theme_overrides` Dictionary mapping theme names to theme overrides, recursively merged with contents of `powerline/themes/vim/key.json`. Note that this way some value (e.g. segment) in a list cannot be redefined, only the whole list itself: only dictionaries are merged recursively.

`g:powerline_config_paths` Paths list (each path must be expanded, ~ shortcut is not supported). Points to the list of directories which will be searched for configuration. When this option is present, none of the other locations are searched.

`g:powerline_no_python_error` If this variable is set to a true value it will prevent Powerline from reporting an error when loaded in a copy of vim without the necessary Python support.

`g:powerline_use_var_handler` This variable may be set to either 0 or 1. If it is set to 1 then Vim will save log in `g:powerline_log_messages` variable in addition to whatever was configured in *log_* options*. Level is always *log_level*, same for format.

> **Warning:** This variable is deprecated. Use *log_file option* in conjunction with `powerline.vim.VimVarHandler` class and *Vim config overrides variable*. Using this is also the only variant to make saving into the environment variable the *only* place where log is saved or save into different variable.

class `powerline.vim.`**`VimVarHandler`**(*varname*)

Vim-specific handler which emits messages to Vim global variables

Parameters **`varname`** (*`str`*) – Variable where

Powerline script overrides

Powerline script has a number of options controlling powerline behavior. Here `VALUE` always means "some JSON object".

-c KEY.NESTED_KEY=VALUE or --config-override=KEY.NESTED_KEY=VALUE Overrides options from `powerline/config.json`. `KEY.KEY2.KEY3=VALUE` is a shortcut for `KEY={"KEY2": {"KEY3": VALUE}}`. Multiple options (i.e. `-c K1=V1 -c K2=V2`) are allowed, result (in the example: `{"K1": V1, "K2": V2}`) is recursively merged with the contents of the file.

If `VALUE` is omitted then corresponding key will be removed from the configuration (if it was present).

-t THEME_NAME.KEY.NESTED_KEY=VALUE or --theme-override=THEME_NAME.KEY.NESTED_KEY=VALUE Overrides options from `powerline/themes/`*`ext`*`/`*`THEME_NAME`*`.json`. `KEY.NESTED_KEY=VALUE` is processed like described above, `{ext}` is the first argument to powerline script. May be passed multiple times.

If `VALUE` is omitted then corresponding key will be removed from the configuration (if it was present).

-p PATH or --config-path=PATH Sets directory where configuration should be read from. If present, no default locations are searched for configuration. No expansions are performed by powerline script itself, but `-p ~/.powerline` will likely be expanded by the shell to something like `-p /home/user/.powerline`.

> **Warning:** Such overrides are suggested for testing purposes only. Use *Environment variables overrides* for other purposes.

Environment variables overrides

All bindings that use `POWERLINE_COMMAND` environment variable support taking overrides from environment variables. In this case overrides should look like the following:

```
OVERRIDE='key1.key2.key3=value;key4.key5={"value":1};key6=true;key1.key7=10'
```

. This will be parsed into

```
{
    "key1": {
        "key2": {
            "key3": "value"
        },
        "key7": 10,
    },
    "key4": {
        "key5": {
            "value": 1,
        },
    },
    "key6": True,
}
```

. Rules:

1. Environment variable must form a semicolon-separated list of key-value pairs: `key=value;key2=value2`.
2. Keys are always dot-separated strings that must not contain equals sign (as well as semicolon) or start with an underscore. They are interpreted literally and create a nested set of dictionaries: `k1.k2.k3` creates `{"k1":{"k2":{}}}` and inside the innermost dictionary last key (`k3` in the example) is contained with its value.
3. Value may be empty in which case they are interpreted as an order to remove some value: `k1.k2=` will form `{"k1":{"k2":REMOVE_THIS_KEY}}` nested dictionary where `k2` value is a special value that tells dictionary-merging function to remove `k2` rather then replace it with something.

4. Value may be a JSON strings like `{"a":1}` (JSON dictionary), `["a",1]` (JSON list), `1` or `-1` (JSON number), `"abc"` (JSON string) or `true`, `false` and `null` (JSON boolean objects and `Null` object from JSON). General rule is that anything starting with a digit (U+0030 till U+0039, inclusive), a hyphenminus (U+002D), a quotation mark (U+0022), a left curly bracket (U+007B) or a left square bracket (U+005B) is considered to be some JSON object, same for *exact* values `true`, `false` and `null`.
5. Any other value is considered to be literal string: `k1=foo:bar` parses to `{"k1": "foo:bar"}`.

The following environment variables may be used for overrides according to the above rules:

POWERLINE_CONFIG_OVERRIDES Overrides values from `powerline/config.json`.

POWERLINE_THEME_OVERRIDES Overrides values from `powerline/themes/ext/key.json`. Top-level key is treated as a name of the theme for which overrides are used: e.g. to disable cwd segment defined in `powerline/themes/shell/default.json` one needs to use:

```
POWERLINE_THEME_OVERRIDES=default.segment_data.cwd.display=false
```

Additionally one environment variable is a usual *colon*-separated list of directories: `POWERLINE_CONFIG_PATHS`. This one defines paths which will be searched for configuration. Empty paths in `POWERLINE_CONFIG_PATHS` are ignored.

Note: Overrides from environment variables have lower priority then *Powerline script overrides*. Latter are suggested for tests only.

Zsh/zpython overrides

Here overrides are controlled by similarly to the powerline script, but values are taken from zsh variables. *Environment variable overrides* are also supported: if variable is a string this variant is used.

POWERLINE_CONFIG_OVERRIDES Overrides options from `powerline/config.json`. Should be a zsh associative array with keys equal to `KEY.NESTED_KEY` and values being JSON strings. Pair `KEY.KEY1 VALUE` is equivalent to `{"KEY": {"KEY1": VALUE}}`. All pairs are then recursively merged into one dictionary and this dictionary is recursively merged with the contents of the file.

POWERLINE_THEME_OVERRIDES Overrides options from `powerline/themes/shell/*.json`. Should be a zsh associative array with keys equal to `THEME_NAME.KEY.NESTED_KEY` and values being JSON strings. Is processed like the above `POWERLINE_CONFIG_OVERRIDES`, but only subdictionaries for `THEME_NAME` key are merged with theme configuration when theme with given name is requested.

POWERLINE_CONFIG_PATHS Sets directories where configuration should be read from. If present, no default locations are searched for configuration. No expansions are performed by powerline script itself, but zsh usually performs them on its own if variable without is set without quotes: `POWERLINE_CONFIG_PATHS=( ~/example )`. In addition to arrays usual colon-separated "array" string can be used: `POWERLINE_CONFIG_PATHS=$HOME/path1:$HOME/path2`.

Ipython overrides

Ipython overrides depend on ipython version. Before ipython-0.11 additional keyword arguments should be passed to setup() function. After ipython-0.11 `c.Powerline.KEY` should be used. Supported `KEY` strings or keyword argument names:

config_overrides Overrides options from `powerline/config.json`. Should be a dictionary that will be recursively merged with the contents of the file.

theme_overrides Overrides options from `powerline/themes/ipython/*.json`. Should be a dictionary where keys are theme names and values are dictionaries which will be recursively merged with the contents of the given theme.

config_paths Sets directories where configuration should be read from. If present, no default locations are searched for configuration. No expansions are performed thus paths starting with `~/` cannot be used: use `os.path.expanduser()`.

Prompt command

In addition to the above configuration options `$POWERLINE_COMMAND` environment variable can be used to tell shell or tmux to use specific powerline implementation and `$POWERLINE_CONFIG_COMMAND` to tell zsh or tmux where `powerline-config` script is located. This is mostly useful for putting powerline into different directory.

Note: `$POWERLINE_COMMAND` is always treated as one path in shell bindings, so path with spaces in it may be used. To specify additional arguments one may use `$POWERLINE_COMMAND_ARGS`, but note that this variable exists for testing purposes only and may be removed. One should use *Environment variable overrides* instead.

To disable prompt in shell, but still have tmux support or to disable tmux support environment variables `$POWERLINE_NO_{SHELL}_PROMPT` and `$POWERLINE_NO_{SHELL}_TMUX_SUPPORT` can be used (substitute `{SHELL}` with the name of the shell (all-caps) that should be affected (e.g. `BASH`) or use all-inclusive `SHELL` that will disable support for all shells). These variables have no effect after configuration script was sourced (in fish case: after `powerline-setup` function was run). To disable specific feature support set one of these variables to some non-empty value.

In order to keep shell prompt, but avoid launching Python twice to get unused *above* lines in tcsh `$POWERLINE_NO_TCSH_ABOVE` or `$POWERLINE_NO_SHELL_ABOVE` variable should be set.

In order to remove additional space from the end of the right prompt in fish that was added in order to support multiline prompt `$POWERLINE_NO_FISH_ABOVE` or `$POWERLINE_NO_SHELL_ABOVE` variable should be set.

PDB overrides

Like shell bindings *PDB bindings* take overrides from *environment variables*.

CHAPTER 5

Developer guide

Writing segments

Each powerline segment is a callable object. It is supposed to be either a Python function or *powerline.segments.Segment* class. As a callable object it should receive the following arguments:

Note: All received arguments are keyword arguments.

pl A *powerline.PowerlineLogger* instance. It must be used every time something needs to be logged.

segment_info A dictionary. It is only received if callable has `powerline_requires_segment_info` attribute.

Refer to *segment_info detailed description* for further details.

create_watcher Function that will create filesystem watcher once called. Which watcher will be created exactly is controlled by *watcher configuration option*.

And also any other argument(s) specified by user in *args key* (no additional arguments by default).

Note: For powerline-lint to work properly the following things may be needed:

1. If segment is a *powerline.segments.Segment* instance and used arguments are scattered over multiple methods *powerline.segments.Segment.argspecobjs()* should be overridden in subclass to tell powerline-lint which objects should be inspected for arguments.
2. If segment takes some arguments that are never listed, but accessed via `kwargs.get()` or previous function cannot be used for whatever reason *powerline.segments.Segment.additional_args()* should be overridden in subclass.
3. If user is expected to use one *name* for multiple segments which cannot be linked to the segment function automatically by powerline-lint (e.g. because there are no instances of the segments in question in the default configuration) `powerline.lint.checks.register_common_name()` function should be used.

Object representing segment may have the following attributes used by powerline:

powerline_requires_segment_info This attribute controls whether segment will receive `segment_info` argument: if it is present argument will be received.

powerline_requires_filesystem_watcher This attribute controls whether segment will receive `create_watcher` argument: if it is present argument will be received.

powerline_segment_datas This attribute must be a dictionary containing `top_theme: segment_data` mapping where `top_theme` is any theme name (it is expected that all of the names from *top-level themes list* are present) and `segment_data` is a dictionary like the one that is contained inside *segment_data dictionary in configuration*. This attribute should be used to specify default theme-specific values for *third-party* segments: powerline theme-specific values go directly to *top-level themes*.

startup This attribute must be a callable which accepts the following keyword arguments:

- `pl`: *`powerline.PowerlineLogger`* instance which is to be used for logging.
- `shutdown_event`: `Event` object which will be set when powerline will be shut down.
- Any arguments found in user configuration for the given segment (i.e. *args key*).

This function is called at powerline startup when using long-running processes (e.g. powerline in vim, in zsh with libzpython, in ipython or in powerline daemon) and not called when `powerline-render` executable is used (more specific: when *`powerline.Powerline`* constructor received true `run_once` argument).

shutdown This attribute must be a callable that accepts no arguments and shuts down threads and frees any other resources allocated in `startup` method of the segment in question.

This function is not called when `startup` method is not called.

expand This attribute must be a callable that accepts the following keyword arguments:

- `pl`: *`powerline.PowerlineLogger`* instance which is to be used for logging.
- `amount`: integer number representing amount of display cells result must occupy.

> **Warning:** "Amount of display cells" is *not* number of Unicode codepoints, string length, or byte count. It is suggested that this function should look something like `return (' ' * amount) + segment['contents']` where `' '` may be replaced with anything that is known to occupy exactly one display cell.

- `segment`: *segment dictionary*.
- Any arguments found in user configuration for the given segment (i.e. *args key*).

It must return new value of *contents* key.

truncate Like *expand function*, but for truncating segments. Here `amount` means the number of display cells which must be freed.

This function is called for all segments before powerline starts purging them to free space.

This callable object should may return either a string (`unicode` in Python2 or `str` in Python3, *not* `str` in Python2 or `bytes` in Python3) object or a list of dictionaries. String object is a short form of the following return value:

```
[{
    'contents': original_return,
    'highlight_groups': [segment_name],
}]
```

Returned list is a list of segments treated independently, except for *draw_inner_divider key*.

All keys in segments returned by the function override those obtained from *configuration* and have the same meaning.

Detailed description of used dictionary keys:

contents Text displayed by segment. Should be a `unicode` (Python2) or `str` (Python3) instance.

literal_contents Text that needs to be output literally (i.e. without passing through `powerline.renderer.strwidth()` to determine length, through `powerline.renderer.escape()` to escape special characters and through `powerline.renderer.hl()` to highlight it). Should be a tuple `(contents_length, contents)` where `contents_length` is an integer and `contents` is a `unicode` (Python2) or `str` (Python3) instance.

If this key is present and its second value is true then other contents keys (*contents*, *after*, *before*) will be ignored.

Note: If target is inclusion of the segment in powerline upstream all segment functions that output *only* subsegments with `literal_contents` key must contain the following string in documentation:

```
No highlight groups are used (literal segment).
```

String must be present on the separate line.

draw_hard_divider, draw_soft_divider, draw_inner_divider Determines whether given divider should be drawn. All have the same meaning as *the similar keys in configuration* (*draw_inner_divider*).

highlight_groups Determines segment highlighting. Refer to *themes documentation* for more details.

Defaults to the name of the segment.

Note: If target is inclusion of the segment in powerline upstream all used highlighting groups must be specified in the segment documentation in the form:

```
Highlight groups used: ``g1``[ or ``g2``]*[, ``g3`` (gradient)[ or ``g4``]*]*.
```

I.e. use:

```
Highlight groups used: ``foo_gradient`` (gradient) or ``foo``, ``bar``.
```

to specify that the segment uses *either* `foo_gradient` group or `foo` group *and* `bar` group meaning that `powerline-lint` will check that at least one of the first two groups is defined (and if `foo_gradient` is defined it must use at least one gradient color) and third group is defined as well.

All groups must be specified on one line.

divider_highlight_group Determines segment divider highlight group. Only applicable for soft dividers: colors for hard dividers are determined by colors of adjacent segments.

Note: If target is inclusion of the segment in powerline upstream used divider highlight group must be specified in the segment documentation in the form:

```
    Divider highlight group used: ``group``.

This text must not wrap and all divider highlight group names are
supposed to end with ``:divider``: e.g. ``cwd:divider``.
```

gradient_level First and the only key that may not be specified in user configuration. It determines which color should be used for this segment when one of the highlighting groups specified by *highlight_groups* was defined to use the color gradient.

This key may have any value from 0 to 100 inclusive, value is supposed to be an `int` or `float` instance.

No error occurs if segment has this key, but no used highlight groups use gradient color.

_* Keys starting with underscore are reserved for powerline and must not be returned.

__* Keys starting with two underscores are reserved for the segment functions, specifically for *expand function*.

Segment dictionary

Segment dictionary contains the following keys:

- All keys returned by segment function (if it was used).
- All of the following keys:

name Segment name: value of the *name key* or function name (last component of the *function key*). May be `None`.

type *Segment type*. Always represents actual type and is never `None`.

highlight_groups, divider_highlight_group Used highlight groups. May be `None`.

highlight_group_prefix If this key is present then given prefix will be prepended to each highlight group (both regular and divider) used by this segment in a form `{prefix}:{group}` (note the colon). This key is mostly useful for *segment listers*.

before, after Value of *before* or *after* configuration options. May be `None` as well as an empty string.

contents_func Function used to get segment contents. May be `None`.

contents Actual segment contents, excluding dividers and *before/after*. May be `None`.

priority *Segment priority*. May be `None` for no priority (such segments are always shown).

draw_soft_divider, draw_hard_divider, draw_inner_divider *Divider control flags*.

side Segment side: `right` or `left`.

display_condition Contains function that takes three position parameters: `powerline.PowerlineLogger` instance, *segment_info* dictionary and current mode and returns either `True` or `False` to indicate whether particular segment should be processed.

This key is constructed based on *exclude_/include_modes keys* and *exclude_/include_function keys*.

width, align *Width and align options*. May be `None`.

expand, truncate Partially applied *expand* or *truncate* function. Accepts `pl`, `amount` and `segment` positional parameters, keyword parameters from *args* key were applied.

startup Partially applied *startup function*. Accepts `pl` and `shutdown_event` positional parameters, keyword parameters from *args* key were applied.

shutdown *Shutdown function*. Accepts no argument.

Segments layout

Powerline segments are all located in one of the `powerline.segments` submodules. For extension-specific segments `powerline.segments.{ext}` module should be used (e.g. `powerline.segments.shell`), for extension-agnostic there is `powerline.segments.common`.

Plugin-specific segments (currently only those that are specific to vim plugins) should live in `powerline.segments.{ext}.plugin.{plugin_name}`: e.g. `powerline.segments.vim.plugin.gundo`.

Segment information used in various extensions

Each `segment_info` value should be a dictionary with at least the following keys:

environ Current environment, may be an alias to `os.environ`. Is guaranteed to have `__getitem__` and `get` methods and nothing more.

> **Warning:** `os.environ` must not ever be used:
>
> - If segment is run in the daemon this way it will get daemon's environment which is not correct.
> - If segment is run in Vim or in zsh with libzpython `os.environ` will contain Vim or zsh environ *at the moment Python interpreter was loaded*.

getcwd Function that returns current working directory being called with no arguments. `os.getcwd` must not be used for the same reasons the use of `os.environ` is forbidden, except that current working directory is valid in Vim and zsh (but not in daemon).

home Current home directory. May be false.

Vim

Vim `segment_info` argument is a dictionary with the following keys:

window `vim.Window` object. `vim.current.window` or `vim.windows[number - 1]` may be used to obtain such object. May be a false object, in which case any of this object's properties must not be used.

winnr Window number. Same as `segment_info['window'].number` *assuming* Vim is new enough for `vim.Window` object to have `number` attribute.

window_id Internal powerline window id, unique for each newly created window. It is safe to assume that this ID is hashable and supports equality comparison, but no other assumptions about it should be used. Currently uses integer numbers incremented each time window is created.

buffer `vim.Buffer` object. One may be obtained using `vim.current.buffer`, `segment_info['window'].buffer` or `vim.buffers[some_number]`. Note that in the latter case depending on vim version `some_number` may be `bufnr` or the internal Vim buffer index which is *not* buffer number. For this reason to get `vim.Buffer` object other then stored in `segment_info` dictionary iteration over `vim.buffers` and checking their `number` attributes should be performed.

bufnr Buffer number.

tabpage `vim.Tabpage` object. One may be obtained using `vim.current.tabpage` or `vim.tabpages[number - 1]`. May be a false object, in which case no object's properties can be used.

tabnr Tabpage number.

mode Current mode.

encoding Value of `&encoding` from the time when powerline was initialized. It should be used to convert return values.

Note: Segment generally should not assume that it is run for the current window, current buffer or current tabpage. "Current window" and "current buffer" restrictions may be ignored if `window_cached` decorator is used, "current tabpage" restriction may be safely ignored if segment is not supposed to be used in tabline.

Warning: Powerline is being tested with vim-7.0.112 (some minor sanity check) and latest Vim. This means that most of the functionality like `vim.Window.number`, `vim.*.vars`, `vim.*.options` or even `dir(vim object)` should be avoided in segments that want to be included in the upstream.

Shell

args Parsed shell arguments: a `argparse.Namespace` object. Check out `powerline-render --help` for the list of all available arguments. Currently it is expected to contain at least the following attributes:

- **last_exit_code** Exit code returned by last shell command. Is either one integer, `sig{name}` or `sig{name}+core` (latter two are only seen in `rc` shell).
- **last_pipe_status** List of exit codes returned by last programs in the pipe or some false object. Only available in `zsh` and `rc`. Is a list of either integers, `sig{name}` or `sig{name}+core` (latter two are only seen in `rc` shell).
- **jobnum** Number of background jobs.
- **renderer_arg** Dictionary containing some keys that are additional arguments used by shell bindings. *This attribute must not be used directly*: all arguments from this dictionary are merged with `segment_info` dictionary. Known to have at least the following keys:
  - **client_id** Identifier unique to one shell instance. Is used to record instance state by powerline daemon. In tmux this is the same as *pane_id*.

    It is not guaranteed that existing client ID will not be retaken when old shell with this ID quit: usually process PID is used as a client ID.

    It is also not guaranteed that client ID will be process PID, number or something else at all. It is guaranteed though that client ID will be some hashable object which supports equality comparison.
  - **local_theme** Local theme that will be used by shell. One should not rely on the existence of this key.
  - **pane_id** Identifier unique to each tmux pane. Is always an integer, optional. Obtained by using `tmux display -p '#D'`, then all leading spaces and per cent signs are stripped and the result is converted into an integer.

  Other keys, if any, are specific to segments.

Ipython

ipython Some object which has `prompt_count` attribute. Currently it is guaranteed to have only this attribute.

Attribute `prompt_count` contains the so-called "history count" (equivalent to `\N` in `in_template`).

Pdb

pdb Currently active `pdb.Pdb` instance.

curframe Frame which will be run next. Note: due to the existence of *powerline.listers.pdb.frame_lister()* one must not use `segment_info['pdb'].curframe`.

initial_stack_length Equal to the length of `pdb.Pdb.stack` at the first invocation of the prompt decremented by one.

i3wm

mode Currently active i3 mode (as a string).

output `xrandr` output name currently drawing to. Currently only available in lemonbar bindings.

workspace dictionary containing the workspace name under the key `"name"` and boolean values for the `"visible"`, `"urgent"` and `"focused"` keys, indicating the state of the workspace. Currently only provided by the *powerline.listers.i3wm.workspace_lister()* lister.

Segment class

class `powerline.segments.`**`Segment`**

Base class for any segment that is not a function

Required for powerline.lint.inspect to work properly: it defines methods for omitting existing or adding new arguments.

Note: Until python-3.4 `inspect.getargspec` does not support querying callable classes for arguments of their `__call__` method, requiring to use this method directly (i.e. before 3.4 you should write `getargspec(obj.__call__)` in place of `getargspec(obj)`).

---

static **`additional_args`**`()`

Returns a list of (additional argument name[, default value]) tuples.

`argspecobjs``()`

Return a list of valid arguments for inspect.getargspec

Used to determine function arguments.

`omitted_args`(*name*, *method*)

List arguments which should be omitted

Returns a tuple with indexes of omitted arguments.

PowerlineLogger class

class `powerline.`**`PowerlineLogger`**(*use_daemon_threads*, *logger*, *ext*)

Proxy class for logging.Logger instance

It emits messages in format `{ext}:{prefix}:{message}` where

`{ext}` is a used powerline extension (e.g. "vim", "shell", "ipython").

`{prefix}` is a local prefix, usually a segment name.

`{message}` is the original message passed to one of the logging methods.

Each of the methods (`critical`, `exception`, `info`, `error`, `warn`, `debug`) expects to receive message in an `str.format` format, not in printf-like format.

Log is saved to the location *specified by user*.

critical (*msg*, **args*, ***kwargs*)

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

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

exception (*msg*, **args*, ***kwargs*)

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

warn (*msg*, **args*, ***kwargs*)

Writing listers

Listers provide a way to show some segments multiple times: once per each entity (buffer, tabpage, etc) lister knows. They are functions which receive the following arguments:

pl A `powerline.PowerlineLogger` class instance. It must be used for logging.

segment_info Base segment info dictionary. Lister function or class must have `powerline_requires_segment_info` to receive this argument.

> **Warning:** Listers are close to useless if they do not have access to this argument.

Refer to *segment_info detailed description* for further details.

draw_inner_divider If False (default) soft dividers between segments in the listed group will not be drawn regardless of actual segment settings. If True they will be drawn, again regardless of actual segment settings. Set it to `None` in order to respect segment settings.

And also any other argument(s) specified by user in *args key* (no additional arguments by default).

Listers must return a sequence of pairs. First item in the pair must contain a `segment_info` dictionary specific to one of the listed entities.

Second item must contain another dictionary: it will be used to modify the resulting segment. In addition to *usual keys that describe segment* the following keys may be present (it is advised that *only* the following keys will be used):

priority_multiplier Value (usually a `float`) used to multiply segment priority. It is useful for finer-grained controlling which segments disappear first: e.g. when listing tab pages make first disappear directory names of the tabpages which are most far away from current tabpage, then (when all directory names disappeared) buffer names. Check out existing listers implementation in `powerline/listers/vim.py`.

Local themes

From the user point of view local themes are the regular themes with a specific scope where they are applied (i.e. specific vim window or specific kind of prompt). Used themes are defined in *local_themes key*.

Vim local themes

Vim is the only available extension that has a wide variaty of options for local themes. It is the only extension where local theme key refers to a function as described in *local_themes value documentation*.

This function always takes a single value named `matcher_info` which is the same dictionary as *segment_info dictionary*. Unlike segments it takes this single argument as a *positional* argument, not as a keyword one.

Matcher function should return a boolean value: `True` if theme applies for the given `matcher_info` dictionary or `False` if it is not. When one of the matcher functions returns `True` powerline takes the corresponding theme at uses it for the given window. Matchers are not tested in any particular order.

In addition to *local_themes configuration key* developer of some plugin which wishes to support powerline without including his code in powerline tree may use `powerline.vim.VimPowerline.add_local_theme()` method. It accepts two arguments: matcher name (same as in *local_themes*) and dictionary with theme. This dictionary is merged with *top theme* and `powerline/themes/vim/__main__.json`. Note that if user already specified the matcher in his configuration file `KeyError` is raised.

Other local themes

Except for Vim only IPython and shells have local themes. Unlike Vim these themes are names with no special meaning (they do not refer to or cause loading of any Python functions):

Extension	Theme name	Description
Shell	continuation	Shown for unfinished command (unclosed quote, unfinished cycle).
	select	Shown for `select` command available in some shells.
IPython	in2	Continuation prompt: shown for unfinished (multiline) expression, unfinished class or function definition.
	out	Displayed before the result.
	rewrite	Displayed before the actually executed code when `autorewrite` IPython feature is enabled.

Creating new powerline extension

Powerline extension is a code that tells powerline how to highlight and display segments in some set of applications. Specifically this means

1. Creating a *`powerline.Powerline`* subclass that knows how to obtain *local configuration overrides*. It also knows how to load local themes, but not when to apply them.

   Instance of this class is the only instance that interacts directly with bindings code, so it has a proxy *`powerline.Powerline.render()`* and *`powerline.Powerline.shutdown()`* methods and other methods which may be useful for bindings.

   This subclass must be placed directly in `powerline` directory (e.g. in `powerline/vim.py`) and named like `VimPowerline` (version of the file name without directory and extension and first capital letter + `Powerline`). There is no technical reason for naming classes like this.

2. Creating a *`powerline.renderer.Renderer`* subclass that knows how to highlight a segment or reset highlighting to the default value (only makes sense in prompts). It is also responsible for selecting local themes and computing text width.

   This subclass must be placed directly in `powerline/renderers` directory (for powerline extensions developed for a set of applications use `powerline/renderers/ext/*.py`) and named like `ExtRenderer`

or `AppPromptRenderer`. For technical reasons the class itself must be referenced in `renderer` module attribute thus allowing only one renderer per one module.

3. Creating an extension bindings. These are to be placed in `powerline/bindings/ext` and may contain virtually anything which may be required for powerline to work inside given applications, assuming it does not fit in other places.

Powerline class

class `powerline.`**`Powerline`**(**args*, ***kwargs*)

Main powerline class, entrance point for all powerline uses. Sets powerline up and loads the configuration.

Parameters

- **`ext`** (*`str`*) – extension used. Determines where configuration files will searched and what renderer module will be used. Affected: used `ext` dictionary from `powerline/config.json`, location of themes and colorschemes, render module (`powerline.renders.{ext}`).
- **`renderer_module`** (*`str`*) – Overrides renderer module (defaults to `ext`). Should be the name of the package imported like this: `powerline.renderers.{render_module}`. If this parameter contains a dot `powerline.renderers.` is not prepended. There is also a special case for renderers defined in toplevel modules: `foo.` (note: dot at the end) tries to get renderer from module `foo` (because `foo` (without dot) tries to get renderer from module `powerline.renderers.foo`). When `.foo` (with leading dot) variant is used `renderer_module` will be `powerline.renderers.{ext}{renderer_module}`.
- **`run_once`** (*`bool`*) – Determines whether `render()` method will be run only once during python session.
- **`logger`** (*`Logger`*) – If present no new logger will be created and the provided logger will be used.
- **`use_daemon_threads`** (*`bool`*) – When creating threads make them daemon ones.
- **`shutdown_event`** (*`Event`*) – Use this Event as shutdown_event instead of creating new event.
- **`config_loader`** (*`ConfigLoader`*) – Instance of the class that manages (re)loading of the configuration.

`create_logger`()

Create logger

This function is used to create logger unless it was already specified at initialization.

Returns

Three objects:

1. `logging.Logger` instance.
2. `PowerlineLogger` instance.
3. Function, output of `gen_module_attr_getter()`.

`create_renderer`(*load_main=False*, *load_colors=False*, *load_colorscheme=False*, *load_theme=False*)

(Re)create renderer object. Can be used after Powerline object was successfully initialized. If any of the below parameters except `load_main` is True renderer object will be recreated.

Parameters

- **load_main** (*bool*) – Determines whether main configuration file (`config.json`) should be loaded. If appropriate configuration changes implies `load_colorscheme` and `load_theme` and recreation of renderer object. Won't trigger recreation if only unrelated configuration changed.
- **load_colors** (*bool*) – Determines whether colors configuration from `colors.json` should be (re)loaded.
- **load_colorscheme** (*bool*) – Determines whether colorscheme configuration should be (re)loaded.
- **load_theme** (*bool*) – Determines whether theme configuration should be reloaded.

static do_setup()

Function that does initialization

Should be overridden by subclasses. May accept any number of regular or keyword arguments.

static get_config_paths()

Get configuration paths.

Should be overridden in subclasses in order to provide a way to override used paths.

Returns list of paths

static get_encoding()

Get encoding used by the current application

Usually returns encoding of the current locale.

static get_local_themes(*local_themes*)

Get local themes. No-op here, to be overridden in subclasses if required.

Parameters **local_themes** (*dict*) – Usually accepts `{matcher_name : theme_name}`. May also receive None in case there is no local_themes configuration.

Returns anything accepted by `self.renderer.get_theme` and processable by `self.renderer.add_local_theme`. Renderer module is determined by `__init__` arguments, refer to its documentation.

init(*ext*, *renderer_module=None*, *run_once=False*, *logger=None*, *use_daemon_threads=True*, *shutdown_event=None*, *config_loader=None*)

Do actual initialization.

__init__ function only stores the arguments and runs this function. This function exists for powerline to be able to reload itself: it is easier to make `__init__` store arguments and call overriddable `init` than tell developers that each time they override Powerline.__init__ in subclasses they must store actual arguments.

load_colors_config()

Get colorscheme.

Returns dictionary with *colors configuration*.

load_colorscheme_config(*name*)

Get colorscheme.

Parameters **name** (*str*) – Name of the colorscheme to load.

Returns dictionary with *colorscheme configuration*.

load_config(*cfg_path*, *cfg_type*)

Load configuration and setup watches

Parameters

- **cfg_path** (*str*) – Path to the configuration file without any powerline configuration directory or `.json` suffix.
- **cfg_type** (*str*) – Configuration type. May be one of `main` (for `config.json` file), `colors`, `colorscheme`, `theme`.

Returns dictionary with loaded configuration.

load_main_config()

Get top-level configuration.

Returns dictionary with *top-level configuration*.

load_theme_config(*name*)

Get theme configuration.

Parameters **name** (*str*) – Name of the theme to load.

Returns dictionary with *theme configuration*

reload()

Reload powerline after update.

Should handle most (but not all) powerline updates.

Purges out all powerline modules and modules imported by powerline for segment and matcher functions. Requires defining `setup` function that updates reference to main powerline object.

> **Warning:** Not guaranteed to work properly, use it at your own risk. It may break your python code.

render(**args*, ***kwargs*)

Update/create renderer if needed and pass all arguments further to `self.renderer.render()`.

render_above_lines(**args*, ***kwargs*)

Like .render(), but for `self.renderer.render_above_lines()`

setup(**args*, ***kwargs*)

Setup the environment to use powerline.

Must not be overridden by subclasses. This one only saves setup arguments for `reload()` method and calls `do_setup()`.

setup_components(*components*)

Run component-specific setup

Parameters **components** (*set*) – Set of the enabled componets or None.

Should be overridden by subclasses.

shutdown(*set_event=True*)

Shut down all background threads.

Parameters **set_event** (*bool*) – Set `shutdown_event` and call `renderer.shutdown` which should shut down all threads. Set it to False unless you are exiting an application.

If set to False this does nothing more then resolving reference cycle `powerline` → `config_loader` → `bound methods` → `powerline` by unsubscribing from config_loader events.

update_renderer()

Updates/creates a renderer if needed.

Renderer class

class `powerline.renderer.`**`Renderer`**(*theme_config*, *local_themes*, *theme_kwargs*, *pl*, *ambiwidth=1*, ***options*)

Object that is responsible for generating the highlighted string.

Parameters

- **`theme_config`** (*`dict`*) – Main theme configuration.
- **`local_themes`** – Local themes. Is to be used by subclasses from `.get_theme()` method, base class only records this parameter to a `.local_themes` attribute.
- **`theme_kwargs`** (*`dict`*) – Keyword arguments for `Theme` class constructor.
- **`pl`** (`PowerlineLogger`) – Object used for logging.
- **`ambiwidth`** (*`int`*) – Width of the characters with east asian width unicode attribute equal to `A` (Ambigious).
- **`options`** (*`dict`*) – Various options. Are normally not used by base renderer, but all options are recorded as attributes.

`do_render`(*mode*, *width*, *side*, *line*, *output_raw*, *output_width*, *segment_info*, *theme*)

Like Renderer.render(), but accept theme in place of matcher_info

`escape`(*string*)

Method that escapes segment contents.

`get_segment_info`(*segment_info*, *mode*)

Get segment information.

Must return a dictionary containing at least `home`, `environ` and `getcwd` keys (see documentation for `segment_info` attribute). This implementation merges `segment_info` dictionary passed to `.render()` method with `.segment_info` attribute, preferring keys from the former. It also replaces `getcwd` key with function returning `segment_info['environ']['PWD']` in case `PWD` variable is available.

Parameters **`segment_info`** (*`dict`*) – Segment information that was passed to `.render()` method.

Returns dict with segment information.

`get_theme`(*matcher_info*)

Get Theme object.

Is to be overridden by subclasses to support local themes, this variant only returns `.theme` attribute.

Parameters **`matcher_info`** – Parameter `matcher_info` that `.render()` method received. Unused.

`hl`(*contents*, *fg=None*, *bg=None*, *attrs=None*)

Output highlighted chunk.

This implementation just outputs `hlstyle()` joined with `contents`.

static **`hl_join`**()

Join a list of rendered segments into a resulting string

This method exists to deal with non-string render outputs, so *segments* may actually be not an iterable with strings.

Parameters **`segments`** (*`list`*) – Iterable containing rendered segments. By "rendered segments" `Renderer.hl()` output is meant.

Returns Results of joining these segments.

hlstyle (*fg=None*, *bg=None*, *attrs=None*)

Output highlight style string.

Assuming highlighted string looks like `{style}{contents}` this method should output `{style}`. If it is called without arguments this method is supposed to reset style to its default.

render (*mode=None*, *width=None*, *side=None*, *line=0*, *output_raw=False*, *output_width=False*, *segment_info=None*, *matcher_info=None*)

Render all segments.

When a width is provided, low-priority segments are dropped one at a time until the line is shorter than the width, or only segments with a negative priority are left. If one or more segments with `"width": "auto"` are provided they will fill the remaining space until the desired width is reached.

Parameters

- **mode** (*str*) – Mode string. Affects contents (colors and the set of segments) of rendered string.
- **width** (*int*) – Maximum width text can occupy. May be exceeded if there are too much non-removable segments.
- **side** (*str*) – One of `left`, `right`. Determines which side will be rendered. If not present all sides are rendered.
- **line** (*int*) – Line number for which segments should be obtained. Is counted from zero (botmost line).
- **output_raw** (*bool*) – Changes the output: if this parameter is `True` then in place of one string this method outputs a pair `(colored_string, colorless_string)`.
- **output_width** (*bool*) – Changes the output: if this parameter is `True` then in place of one string this method outputs a pair `(colored_string, string_width)`. Returns a three-tuple if `output_raw` is also `True`: `(colored_string, colorless_string, string_width)`.
- **segment_info** (*dict*) – Segment information. See also `get_segment_info()` method.
- **matcher_info** – Matcher information. Is processed in `get_segment_info()` method.

render_above_lines (***kwargs*)

Render all segments in the {theme}/segments/above list

Rendering happens in the reversed order. Parameters are the same as in .render() method.

Yield rendered line.

segment_info = {u'environ': <environ dictionary>, u'getcwd': <built-in function getcwdu>, u'home': <home directory

Basic segment info

Is merged with local segment information by `get_segment_info()` method. Keys:

environ Object containing environment variables. Must define at least the following methods: `.__getitem__(var)` that raises `KeyError` in case requested environment variable is not present, `.get(var, default=None)` that works like `dict.get` and be able to be passed to `Popen`.

getcwd Function that returns current working directory. Will be called without any arguments, should return `unicode` or (in python-2) regular string.

home String containing path to home directory. Should be `unicode` or (in python-2) regular string or `None`.

shutdown()

Prepare for interpreter shutdown. The only job it is supposed to do is calling `.shutdown()` method for all theme objects. Should be overridden by subclasses in case they support local themes.

strwidth(*s*)

Function that returns string width.

Is used to calculate the place given string occupies when handling `width` argument to `.render()` method. Must take east asian width into account.

Parameters **string** (*unicode*) – String whose width will be calculated.

Returns unsigned integer.

Tips and tricks for powerline developers

Profiling powerline in Vim

Given that current directory is the root of the powerline repository the following command may be used:

```
vim --cmd 'let g:powerline_pyeval="powerline#debug#profile_pyeval"' \
    --cmd 'set rtp=powerline/bindings/vim' \
    -c 'runtime! plugin/powerline.vim' \
    {other arguments if needed}
```

After some time run `:WriteProfiling {filename}` Vim command. Currently this only works with recent Vim and python-2*. It should be easy to modify `powerline/bindings/vim/autoload/powerline/debug.vim` to suit other needs.

CHAPTER 6

Troubleshooting

System-specific issues

Troubleshooting on Linux

I can't see any fancy symbols, what's wrong?

- Make sure that you've configured gvim or your terminal emulator to use a patched font.
- You need to set your `LANG` and `LC_*` environment variables to a UTF-8 locale (e.g. `LANG=en_US.utf8`). Consult your Linux distro's documentation for information about setting these variables correctly.
- Make sure that vim is compiled with the `--with-features=big` flag.
- If you're using rxvt-unicode make sure that it's compiled with the `--enable-unicode3` flag.
- If you're using xterm make sure you have told it to work with unicode. You may need `-u8` command-line argument, `uxterm` shell wrapper that is usually shipped with xterm for this or `xterm*utf8` property set to `1` or `2` in `~/.Xresources` (applied with `xrdb`). Note that in case `uxterm` is used configuration is done via `uxterm*...` properties and not `xterm*...`.

  In any case the only absolute requirement is launching xterm with UTF-8 locale.
- If you are using bitmap font make sure that `/etc/fonts/conf.d/70-no-bitmaps.conf` does not exist. If it does check out your distribution documentation to find a proper way to remove it (so that it won't reappear after update). E.g. in Gentoo this is:

  ```
  eselect fontconfig disable 70-no-bitmaps.conf
  ```

  (currently this only removes the symlink from `/etc/fonts/conf.d`). Also check out that no other fontconfig file does not have `rejectfont` tag that tells fontconfig to disable bitmap fonts (they are referenced as not scalable).

The fancy symbols look a bit blurry or "off"!

- Make sure that you have patched all variants of your font (i.e. both the regular and the bold font files).

I am seeing strange blocks in place of playing/paused/stopped signs

If you are using `powerline_unicode7` *top-level theme* then symbols for player segments are taken from U+23F4–U+23FA range which is missing from most fonts. You may fix the issue by using Symbola[25] font (or any other font which contains these glyphs).

If your terminal emulator is using fontconfig library then you can create a fontconfig configuration file with the following contents:

```
<?xml version="1.0"?>
<!DOCTYPE fontconfig SYSTEM "fonts.dtd">

<fontconfig>
    <alias>
            <family>Terminus</family>
            <prefer><family>Symbola</family></prefer>
    </alias>
</fontconfig>
```

(replace `Terminus` with the name of the font you are using). Exact sequence of actions you need to perform is different across distributions, most likely it will work if you put the above xml into `/etc/fonts/conf.d/99-prefer-symbola.conf`. On Gentoo you need to put it into `/etc/fonts/conf.d/99-prefer-symbola.conf` and run:

```
eselect fontconfig enable 99-prefer-symbola
```

.

> **Warning:** This answer is only applicable if you use `powerline_unicode7` theme or if you configured powerline to use the same characters yourself.

Troubleshooting on OS X

I can't see any fancy symbols, what's wrong?

- If you're using iTerm2, please update to this revision[26] or newer. Also make sure that Preferences>Profiles>Text>Non-ASCII Font is the same as your main Font.
- You need to set your `LANG` and `LC_*` environment variables to a UTF-8 locale (e.g. `LANG=en_US.utf8`). Consult your Linux distro's documentation for information about setting these variables correctly.

The colors look weird in the default OS X Terminal app!

- The arrows may have the wrong colors if you have changed the "minimum contrast" slider in the color tab of your OS X settings.

[25] http://users.teilar.gr/~g1951d/
[26] https://github.com/gnachman/iTerm2/commit/8e3ad6dabf83c60b8cf4a3e3327c596401744af6

- The default OS X Terminal app is known to have some issues with the Powerline colors. Please use another terminal emulator. iTerm2 should work fine.

The colors look weird in iTerm2!

- The arrows may have the wrong colors if you have changed the "minimum contrast" slider in the color tab of your OS X settings.
- Please disable background transparency to resolve this issue.

Statusline is getting wrapped to the next line in iTerm2

- Turn off "Treat ambigious-width characters as double width" in *Preferences –> Text*.
- Alternative: remove fancy dividers (they suck in this case), set *ambiwidth* to 2.

I receive a `NameError` when trying to use Powerline with MacVim!

- Please install MacVim using this command:

  ```
  brew install macvim --env-std --override-system-vim
  ```

  Then install Powerline locally with `pip install --user`, or by running these commands in the `powerline` directory:

  ```
  ./setup.py build
  ./setup.py install --user
  ```

I receive an `ImportError` when trying to use Powerline on OS X!

- This is caused by an invalid `sys.path` when using system vim and system Python. Please try to select another Python distribution:

  ```
  sudo port select python python27-apple
  ```

- See issue #39[27] for a discussion and other possible solutions for this issue.

I receive "FSEventStreamStart: register_with_server: ERROR" with status_colors

This is a known[28] libuv issue that happens if one is trying to watch too many files. It should be fixed in libuv-0.12. Until then it is suggested to either disable `status_colors` (from *`powerline.segments.common.vcs.branch()`*) or choose stat-based watcher (will have effectively the same effect as disabling `status_colors`).

Common issues

After an update something stopped working

Assuming powerline was working before update and stopped only after there are two possible explanations:

[27] https://github.com/powerline/powerline/issues/39
[28] https://github.com/joyent/node/issues/5463

- You have more then one powerline installation (e.g. `pip` and `Vundle` installations) and you have updated only one.
- Update brought some bug to powerline.

In the second case you, of course, should report the bug to powerline bug tracker[29]. In the first you should make sure you either have only one powerline installation or you update all of them simultaneously (beware that in the second case you are not supported). To diagnose this problem you may do the following:

1. If this problem is observed within the shell make sure that

   ```
   python -c 'import powerline; print (powerline.__file__)'
   ```

   which should report something like `/usr/lib64/python2.7/site-packages/powerline/__init__.pyc` (if powerline is installed system-wide) or `/home/USER/.../powerline/__init__.pyc` (if powerline was cloned somewhere, e.g. in `/home/USER/.vim/bundle/powerline`) reports the same location you use to source in your shell configuration: in first case it should be some location in `/usr` (e.g. `/usr/share/zsh/site-contrib/powerline.zsh`), in the second it should be something like `/home/USER/.../powerline/bindings/zsh/powerline.zsh`. If this is true it may be a powerline bug, but if locations do not match you should not report the bug until you observe it on configuration where locations do match.

2. If this problem is observed specifically within bash make sure that you clean `$POWERLINE_COMMAND` and `$PROMPT_COMMAND` environment variables on startup or, at least, that it was cleaned after update. While different `$POWERLINE_COMMAND` variable should not cause any troubles most of time (and when it will cause troubles are rather trivial) spoiled `$PROMPT_COMMAND` may lead to strange error messages or absense of exit code reporting.

   These are the sources which may keep outdated environment variables:

   - Any command launched from any application inherits its environment unless callee explicitly requests to use specific environment. So if you did `exec bash` after update it is rather unlikely to fix the problem.
   - More interesting: *tmux* is a client-server application, it keeps one server instance per one user. You probably already knew that, but there is an interesting consequence: once *tmux* server was started it inherits its environment from the callee and keeps it *forever* (i.e. until server is killed). This environment is then inherited by applications you start with `tmux new-session`. Easiest solution is to kill tmux with `tmux kill-server`, but you may also use `tmux set-environment -u` to unset offending variables.
   - Also check *When using z powerline shows wrong number of jobs*: though this problem should not be seen after update only, it contains another example of `$PROMPT_COMMAND` spoiling results.

3. If this problem is observed within the vim instance you should check out the output of the following Ex mode commands

   ```
   python import powerline as pl ; print (pl.__file__)
   python3 import powerline as pl ; print (pl.__file__)
   ```

   One (but not both) of them will most likely error out, this is OK. The same rules apply as in the 1), but in place of sourcing you should seek for the place where you modify *runtimepath* vim option. If you install powerline using VAM[30] then no explicit modifications of runtimpath were performed in your vimrc (runtimepath is modified by VAM in this case), but powerline will be placed in *plugin_root_dir*`/powerline` where *{plugin_root_dir}* is stored in VAM settings dictionary: do *echo g:vim_addon_manager.plugin_root_dir*.

There is a hint if you want to place powerline repository somewhere, but still make powerline package importable anywhere: use

[29] https://github.com/powerline/powerline
[30] https://github.com/MarcWeber/vim-addon-manager

```
pip install --user --editable path/to/powerline
```

Tmux/screen-related issues

I'm using tmux and Powerline looks like crap, what's wrong?

- You need to tell tmux that it has 256-color capabilities. Add this to your `.tmux.conf` to solve this issue:

```
set -g default-terminal "screen-256color"
```

- If you're using iTerm2, make sure that you have enabled the setting *Set locale variables automatically* in *Profiles → Terminal → Environment*.
- Make sure tmux knows that terminal it is running in support 256 colors. You may tell it tmux by using `-2` option when launching it.

I'm using tmux/screen and Powerline is colorless

- If the above advices do not help, then you need to disable *term_truecolor*.
- Alternative: set *additional_escapes* to `"tmux"` or `"screen"`. Note that it is known to work perfectly in screen, but in tmux it may produce ugly spaces.

> **Warning:** Both tmux and screen are not resending sequences escaped in such a way. Thus even though additional escaping will work for the last shown prompt, highlighting will eventually go away when tmux or screen will redraw screen for some reason.
>
> E.g. in screen it will go away when you used copy mode and prompt got out of screen and in tmux it will go away immediately after you press `<Enter>`.

In tmux there is a green bar in place of powerline

In order for tmux bindings to work `powerline-config` script is required to be present in `$PATH`. Alternatively one may define `$POWERLINE_CONFIG_COMMAND` environment variable pointing to the location of the script. *This variable must be defined prior to launching tmux server and in the environment where server is started from.*

Shell issues

When sourcing shell bindings it complains about missing command or file

If you are using `pip` based installation do not forget to add pip-specific executable path to `$PATH` environment variable. This path usually looks something like `$HOME/.local/bin` (linux) or `$HOME/Library/Python/{python_version}/bin` (OS X). One may check out where `powerline-config` script was installed by using `pip show -f powerline-status | grep powerline-config` (does not always work).

I am suffering bad lags before displaying shell prompt

To get rid of these lags there currently are two options:

- Run `powerline-daemon`. Powerline does not automatically start it for you.
- Compile and install `libzpython` module that lives in https://bitbucket.org/ZyX_I/zpython. This variant is zsh-specific.

Prompt is spoiled after completing files in ksh

This is exactly why powerline has official mksh support, but not official ksh support. If you know the solution feel free to share it in powerline bug tracker[31].

When using z powerline shows wrong number of jobs

This happens because z[32] is launching some jobs in the background from `$POWERLINE_COMMAND` and these jobs fail to finish before powerline prompt is run.

Solution to this problem is simple: be sure that `z.sh` is sourced strictly after `powerline/bindings/bash/powerline.sh`. This way background jobs are spawned by z[33] after powerline has done its job.

When using shell I do not see powerline fancy characters

If your locale encoding is not unicode (any encoding that starts with "utf" or "ucs" will work, case is ignored) powerline falls back to ascii-only theme. You should set up your system to use unicode locale or forget about powerline fancy characters.

Urxvt unicode3 and frills

Make sure that, whatever urxvt package you're installing, both the *unicode3* and *frills* features are enabled at compile time. Run `urxvt --help 2>&1 | grep options:` to get a list of enabled options. This should contain at least *frills*, *unicode3* and optionally *iso14755* if you want to input Unicode characters as well.

Compiler flags example:

> –enable-frills –enable-unicode3

As long as your terminal emulator is compiled without unicode rendering, no amount of configuration will make it display unicode characters. They're being considered 'unnecessary features', but they add negligible overhead to the size of the installed package (~100KB).

Vim issues

My vim statusline has strange characters like `^B` in it!

- Please add `set encoding=utf-8` to your `vimrc`.

[31] https://github.com/powerline/powerline
[32] https://github.com/rupa/z
[33] https://github.com/rupa/z

My vim statusline has a lot of ^ or underline characters in it!

- You need to configure the `fillchars` setting to disable statusline fillchars (see `:h 'fillchars'` for details). Add this to your `vimrc` to solve this issue:

```
set fillchars+=stl:\ ,stlnc:\
```

My vim statusline is hidden/only appears in split windows!

- Make sure that you have `set laststatus=2` in your `vimrc`.

My vim statusline is not displayed completely and has too much spaces

- Be sure you have `ambiwidth` option set to `single`.
- Alternative: set *ambiwidth* to 2, remove fancy dividers (they suck when `ambiwidth` is set to double).

Powerline loses color after editing vimrc

If your vimrc has something like

```
autocmd! BufWritePost ~/.vimrc :source ~/.vimrc
```

used to automatically source vimrc after saving it then you must add `nested` after pattern (`vimrc` in this case):

```
autocmd! BufWritePost ~/.vimrc nested :source ~/.vimrc
```

. Alternatively move `:colorscheme` command out of the vimrc to the file which will not be automatically resourced.

Observed problem is that when you use `:colorscheme` command existing highlighting groups are usually cleared, including those defined by powerline. To workaround this issue powerline hooks `Colorscheme` event, but when you source vimrc with `BufWritePost` (or any other) event, but without `nested` this event is not launched. See also autocmd-nested[34] Vim documentation.

Powerline loses color after saving any file

It may be one of the incarnations of the above issue: specifically minibufexpl is known to trigger it. If you are using minibufexplorer you should set

```
let g:miniBufExplForceSyntaxEnable = 1
```

variable so that this issue is not triggered. Complete explanation:

1. When MBE autocommand is executed it launches `:syntax enable` Vim command...
2. ... which makes Vim source `syntax/syntax.vim` file ...
3. ... which in turn sources `syntax/synload.vim` ...
4. ... which executes `:colorscheme` command. Normally this command triggers `Colorscheme` event, but in the first point minibufexplorer did set up autocommands that miss `nested` attribute meaning that no events will be triggered when processing MBE events.

[34] http://vimcommunity.bitbucket.org/doc/autocmd.txt.html#autocmd-nested

Note: This setting was introduced in version 6.3.1 of minibufexpl[35] and removed in version 6.5.0 of its successor minibufexplorer[36]. It is highly advised to use the latter because minibufexpl[37] was last updated late in 2004.

[35] http://www.vim.org/scripts/script.php?script_id=159
[36] http://www.vim.org/scripts/script.php?script_id=3239
[37] http://www.vim.org/scripts/script.php?script_id=159

CHAPTER 7

Tips and tricks

Vim

Useful settings

You may find the following vim settings useful when using the Powerline statusline:

```
set laststatus=2 " Always display the statusline in all windows
set showtabline=2 " Always display the tabline, even if there is only one tab
set noshowmode " Hide the default mode text (e.g. -- INSERT -- below the statusline)
```

Rxvt-unicode

Terminus font and urxvt

The Terminus fonts does not have the powerline glyphs and unless someone submits a patch to the font author, it is unlikely to happen. However, Andre Klärner came up with this work around: In your `~/.Xdefault` file add the following:

```
urxvt*font: xft:Terminus:pixelsize=12,xft:Inconsolata\ for\ Powerline:pixelsize=12
```

This will allow urxvt to fallback onto the Inconsolata fonts in case it does not find the right glyphs within the terminus font.

Source Code Pro font and urxvt

Much like the terminus font that was mentioned above, a similar fix can be applied to the Source Code Pro fonts.

In the `~/.Xdefaults` add the following:

```
URxvt*font: xft:Source\ Code\ Pro\ Medium:pixelsize=13:antialias=true:hinting=true,
↪xft:Source\ Code\ Pro\ Medium:pixelsize=13:antialias=true:hinting=true
```

I noticed that Source Code Pro has the glyphs there already, but the pixel size of the fonts play a role in whether or not the > or the < separators showing up or not. Using font size 12, glyphs on the right hand side of the powerline are present, but the ones on the left don't. Pixel size 14, brings the reverse problem. Font size 13 seems to work just fine.

Reloading powerline after update

Once you have updated powerline you generally have the following options:

1. Restart the application you are using it in. This is the safest one. Will not work if the application uses `powerline-daemon`.
2. For shell and tmux bindings (except for zsh with libzpython): do not do anything if you do not use `powerline-daemon`, run `powerline-daemon --replace` if you do.
3. Use powerline reloading feature.

> **Warning:** This feature is an unsafe one. It is not guaranteed to work always, it may render your Python constantly error out in place of displaying powerline and sometimes may render your application useless, forcing you to restart.
>
> *Do not report any bugs occurred when using this feature unless you know both what caused it and how this can be fixed.*

- When using zsh with libzpython use

```
powerline-reload
```

> **Note:** This shell function is only defined when using libzpython.

- When using IPython use

```
%powerline reload
```

- When using Vim use

```
py powerline.reload()
" or (depending on Python version you are using)
py3 powerline.reload()
```

CHAPTER 8

License and credits

Powerline is licensed under the MIT license[38].

Authors

- Kim Silkebækken[39]
- Nikolay Pavlov[40]
- Kovid Goyal[41]

Contributors

- List of contributors[42]
- The glyphs in the font patcher are created by Fabrizio Schiavi, creator of the excellent coding font Pragmata Pro[43].

[38] https://raw.github.com/powerline/powerline/develop/LICENSE
[39] https://github.com/Lokaltog
[40] https://github.com/ZyX-I
[41] https://github.com/kovidgoyal
[42] https://github.com/powerline/powerline/contributors
[43] http://www.fsd.it/fonts/pragmatapro.htm

CHAPTER 9

Powerline shell commands' manual pages

powerline-config manual page

Synopsis

```
powerline-config [-pPATH]... tmux ACTION ( [-s |n )]
powerline-config [-pPATH]... shell ACTION [COMPONENT] [-sSHELL]
```

Description

-p, –config-path *PATH* Path to configuration directory. If it is present then configuration files will only be seeked in the provided path. May be provided multiple times to search in a list of directories.

-h, –help Display help and exit.

Arguments specific to tmux subcommand

ACTION If action is `source` then version-specific tmux configuration files are sourced, if it is `setenv` then special (prefixed with `_POWERLINE`) tmux global environment variables are filled with data from powerline configuration. Action `setup` is just doing `setenv` then `source`.

-s, –source When using `setup`: always use configuration file sourcing. By default this is determined automatically based on tmux version: this is the default for tmux 1.8 and below.

-n, –no-source When using `setup`: in place of sourcing directly execute configuration files. That is, read each needed powerline-specific configuration file, substitute `$_POWERLINE_...` variables with appropriate values and run `tmux config line`. This is the default behaviour for tmux 1.9 and above.

-h, –help Display help and exit.

Arguments specific to shell subcommand

ACTION If action is `command` then preferred powerline command is output, if it is `uses` then powerline-config script will exit with 1 if specified component is disabled and 0 otherwise.

COMPONENT Only applicable for `uses` subcommand: makes `powerline-config` exit with 0 if specific component is enabled and with 1 otherwise. `tmux` component stands for tmux bindings (e.g. those that notify tmux about current directory changes), `prompt` component stands for shell prompt.

-s, –shell *SHELL* Shell for which query is run

-h, –help Display help and exit.

Author

Written by Kim Silkebækken, Nikolay Pavlov, Kovid Goyal and contributors. The glyphs in the font patcher are created by Fabrizio Schiavi.

Reporting bugs

Report powerline-config bugs to https://github.com/powerline/powerline/issues.

See also

`powerline(1)`

powerline-daemon manual page

Synopsis

```
powerline-daemon [--quiet] [--socket=S] ( [--kill] | (
                 [--foreground] | [--replace] ) )
```

Description

–quiet, -q Without other options: do not complain about already running powerline-daemon instance. Will still exit with 1. With `--kill` and `--replace`: do not show any messages. With `--foreground`: ignored. Does not silence exceptions in any case.

–socket, -s *S* Specify socket which will be used for connecting to daemon.

–kill, -k Kill an already running instance.

–foreground, -f Run in the foreground (don't daemonize).

–replace, -r Replace an already running instance.

-h, –help Display help and exit.

Author

Written by Kim Silkebækken, Nikolay Pavlov, Kovid Goyal and contributors. The glyphs in the font patcher are created by Fabrizio Schiavi.

Reporting bugs

Report powerline-daemon bugs to https://github.com/powerline/powerline/issues.

See also

powerline(1)

powerline-lint manual page

Synopsis

```
powerline-lint [-pPATH]... [-d]
```

Description

-p, –config-path *PATH* Paths where configuration should be checked, in order. You must supply all paths necessary for powerline to work, checking partial (e.g. only user overrides) configuration is not supported.

-d, –debug Display additional information. Used for debugging `powerline-lint` itself, not for debugging configuration.

-h, –help Display help and exit.

Author

Written by Kim Silkebækken, Nikolay Pavlov, Kovid Goyal and contributors. The glyphs in the font patcher are created by Fabrizio Schiavi.

Reporting bugs

Report powerline-lint bugs to https://github.com/powerline/powerline/issues.

See also

powerline(1), *powerline-config(1)*

powerline manual page

Synopsis

```
powerline EXT [SIDE] [-rMODULE] [-wWIDTH] [--last-exit-code=INT]
          [--last-pipe-status=LIST] [--jobnum=INT]
          [-cKEY.KEY=VALUE]... [-tTHEME.KEY.KEY=VALUE]... [-RKEY=VAL]...
          [-pPATH]... [--socket=ADDRESS]
```

Description

EXT Extension: application for which powerline command is launched (usually `shell` or `tmux`). Also supports `wm.` extensions: `wm.awesome`.

SIDE Side: `left` and `right` represent left and right side respectively, `above` emits lines that are supposed to be printed just above the prompt and `aboveleft` is like concatenating `above` with `left` with the exception that only one Python instance is used in this case. May be omitted for `wm.*` extensions.

-r, –renderer-module *MODULE* Renderer module. Usually something like `.bash` or `.zsh` (with leading dot) which is `powerline.renderers.{ext}{MODULE}`, may also be full module name (must contain at least one dot or end with a dot in case it is top-level module) or `powerline.renderers` submodule (in case there are no dots).

-w, –width *WIDTH* Maximum prompt with. Triggers truncation of some segments.

–last-exit-code *INT* Last exit code.

–last-pipe-status *LIST* Like above, but is supposed to contain space-separated array of statuses, representing exit statuses of commands in one pipe.

–jobnum *INT* Number of jobs.

-c, –config-override *KEY.KEY=VALUE* Configuration overrides for `config.json`. Is translated to a dictionary and merged with the dictionary obtained from actual JSON configuration: KEY.KEY=VALUE is translated to `{"KEY": {"KEY": VALUE}}` and then merged recursively. VALUE may be any JSON value, values that are not `null`, `true`, `false`, start with digit, `{`, `[` are treated like strings. If VALUE is omitted then corresponding key is removed.

-t, –theme-override *THEME.KEY.KEY=VALUE* Like above, but theme-specific. THEME should point to an existing and used theme to have any effect, but it is fine to use any theme here.

-R, –renderer-arg *KEY=VAL* Like above, but provides argument for renderer. Is supposed to be used only by shell bindings to provide various data like last-exit-code or last-pipe-status (they are not using `--renderer-arg` for historical resons: `--renderer-arg` was added later).

-p, –config-path *PATH* Path to configuration directory. If it is present then configuration files will only be seeked in the provided path. May be provided multiple times to search in a list of directories.

–socket *ADDRESS* Socket address to use in daemon clients. Is always UNIX domain socket on linux and file socket on Mac OS X. Not used here, present only for compatibility with other powerline clients. This argument must always be the first one and be in a form `--socket ADDRESS`: no = or short form allowed (in other powerline clients, not here).

-h, –help Display help and exit.

Author

Written by Kim Silkebækken, Nikolay Pavlov, Kovid Goyal and contributors. The glyphs in the font patcher are created by Fabrizio Schiavi.

Reporting bugs

Report powerline bugs to https://github.com/powerline/powerline/issues.

See also

powerline-daemon(1), *powerline-config(1)*

CHAPTER 10

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

powerline.listers.i3wm, 50
powerline.listers.pdb, 50
powerline.listers.vim, 50
powerline.segments.common.bat, 31
powerline.segments.common.env, 30
powerline.segments.common.mail, 32
powerline.segments.common.net, 28
powerline.segments.common.players, 33
powerline.segments.common.sys, 27
powerline.segments.common.time, 32
powerline.segments.common.vcs, 27
powerline.segments.common.wthr, 31
powerline.segments.i3wm, 41
powerline.segments.pdb, 42
powerline.segments.shell, 42
powerline.segments.tmux, 44
powerline.segments.vim, 44
powerline.segments.vim.plugin.capslock, 49
powerline.segments.vim.plugin.commandt, 48
powerline.segments.vim.plugin.nerdtree, 49
powerline.segments.vim.plugin.syntastic, 48
powerline.segments.vim.plugin.tagbar, 49
powerline.selectors.vim, 51

Index

A

additional_args() (powerline.segments.Segment static method), 61
argspecobjs() (powerline.segments.Segment method), 61
attached_clients() (in module powerline.segments.tmux), 44

B

battery() (in module powerline.segments.common.bat), 31
branch() (in module powerline.segments.common.vcs), 27
branch() (in module powerline.segments.vim), 44
bufferlister() (in module powerline.listers.vim), 50
bufnr() (in module powerline.segments.vim), 44

C

capslock_indicator() (in module powerline.segments.vim.plugin.capslock), 49
clementine() (in module powerline.segments.common.players), 33
cmus() (in module powerline.segments.common.players), 34
col_current() (in module powerline.segments.vim), 44
continuation() (in module powerline.segments.shell), 42
cpu_load_percent() (in module powerline.segments.common.sys), 27
create_logger() (powerline.Powerline method), 64
create_renderer() (powerline.Powerline method), 64
critical() (powerline.PowerlineLogger method), 62
csv_col_current() (in module powerline.segments.vim), 44
current_code_name() (in module powerline.segments.pdb), 42
current_context() (in module powerline.segments.pdb), 42
current_file() (in module powerline.segments.pdb), 42
current_line() (in module powerline.segments.pdb), 42
current_tag() (in module powerline.segments.vim.plugin.tagbar), 49
cwd() (in module powerline.segments.common.env), 30
cwd() (in module powerline.segments.shell), 43

D

date() (in module powerline.segments.common.time), 32
dbus_player() (in module powerline.segments.common.players), 34
debug() (powerline.PowerlineLogger method), 62
do_render() (powerline.renderer.Renderer method), 67
do_setup() (powerline.Powerline static method), 65

E

email_imap_alert() (in module powerline.segments.common.mail), 32
environment() (in module powerline.segments.common.env), 30
error() (powerline.PowerlineLogger method), 62
escape() (powerline.renderer.Renderer method), 67
exception() (powerline.PowerlineLogger method), 62
external_ip() (in module powerline.segments.common.net), 28

F

file_directory() (in module powerline.segments.vim), 44
file_encoding() (in module powerline.segments.vim), 45
file_format() (in module powerline.segments.vim), 45
file_name() (in module powerline.segments.vim), 45
file_scheme() (in module powerline.segments.vim), 45
file_size() (in module powerline.segments.vim), 45
file_type() (in module powerline.segments.vim), 45
file_vcs_status() (in module powerline.segments.vim), 46
finder() (in module powerline.segments.vim.plugin.commandt), 48
frame_lister() (in module powerline.listers.pdb), 50
fuzzy_time() (in module powerline.segments.common.time), 32

G

get_config_paths() (powerline.Powerline static method), 65

get_encoding() (powerline.Powerline static method), 65
get_local_themes() (powerline.Powerline static method), 65
get_segment_info() (powerline.renderer.Renderer method), 67
get_theme() (powerline.renderer.Renderer method), 67

H

hl() (powerline.renderer.Renderer method), 67
hl_join() (powerline.renderer.Renderer static method), 67
hlstyle() (powerline.renderer.Renderer method), 68
hostname() (in module powerline.segments.common.net), 29

I

info() (powerline.PowerlineLogger method), 62
init() (powerline.Powerline method), 65
internal_ip() (in module powerline.segments.common.net), 29
itunes() (in module powerline.segments.common.players), 35

J

jobnum() (in module powerline.segments.shell), 43

L

last_pipe_status() (in module powerline.segments.shell), 43
last_status() (in module powerline.segments.shell), 43
line_count() (in module powerline.segments.vim), 46
line_current() (in module powerline.segments.vim), 46
line_percent() (in module powerline.segments.vim), 46
load_colors_config() (powerline.Powerline method), 65
load_colorscheme_config() (powerline.Powerline method), 65
load_config() (powerline.Powerline method), 65
load_main_config() (powerline.Powerline method), 66
load_theme_config() (powerline.Powerline method), 66

M

mode() (in module powerline.segments.i3wm), 41
mode() (in module powerline.segments.shell), 43
mode() (in module powerline.segments.vim), 46
modified_buffers() (in module powerline.segments.vim), 46
modified_indicator() (in module powerline.segments.vim), 46
mpd() (in module powerline.segments.common.players), 36

N

nerdtree() (in module powerline.segments.vim.plugin.nerdtree), 49
network_load() (in module powerline.segments.common.net), 29

O

omitted_args() (powerline.segments.Segment method), 61
output_lister() (in module powerline.listers.i3wm), 50

P

paste_indicator() (in module powerline.segments.vim), 46
path() (in module powerline.segments.vim.plugin.commandt), 48
position() (in module powerline.segments.vim), 46
Powerline (class in powerline), 64
powerline.listers.i3wm (module), 50
powerline.listers.pdb (module), 50
powerline.listers.vim (module), 50
powerline.segments.common.bat (module), 31
powerline.segments.common.env (module), 30
powerline.segments.common.mail (module), 32
powerline.segments.common.net (module), 28
powerline.segments.common.players (module), 33
powerline.segments.common.sys (module), 27
powerline.segments.common.time (module), 32
powerline.segments.common.vcs (module), 27
powerline.segments.common.wthr (module), 31
powerline.segments.i3wm (module), 41
powerline.segments.pdb (module), 42
powerline.segments.shell (module), 42
powerline.segments.tmux (module), 44
powerline.segments.vim (module), 44
powerline.segments.vim.plugin.capslock (module), 49
powerline.segments.vim.plugin.commandt (module), 48
powerline.segments.vim.plugin.nerdtree (module), 49
powerline.segments.vim.plugin.syntastic (module), 48
powerline.segments.vim.plugin.tagbar (module), 49
powerline.selectors.vim (module), 51
PowerlineLogger (class in powerline), 61

R

rdio() (in module powerline.segments.common.players), 37
readonly_indicator() (in module powerline.segments.vim), 47
reload() (powerline.Powerline method), 66
render() (powerline.Powerline method), 66
render() (powerline.renderer.Renderer method), 68
render_above_lines() (powerline.Powerline method), 66
render_above_lines() (powerline.renderer.Renderer method), 68
Renderer (class in powerline.renderer), 67
rhythmbox() (in module powerline.segments.common.players), 38

S

scratchpad() (in module powerline.segments.i3wm), 41
Segment (class in powerline.segments), 61
segment_info (powerline.renderer.Renderer attribute), 68
setup() (powerline.Powerline method), 66
setup_components() (powerline.Powerline method), 66
shutdown() (powerline.Powerline method), 66
shutdown() (powerline.renderer.Renderer method), 68
single_tab() (in module powerline.selectors.vim), 51
spotify() (in module powerline.segments.common.players), 39
spotify_apple_script() (in module powerline.segments.common.players), 39
spotify_dbus() (in module powerline.segments.common.players), 40
stack_depth() (in module powerline.segments.pdb), 42
stash() (in module powerline.segments.common.vcs), 27
stash() (in module powerline.segments.vim), 47
strwidth() (powerline.renderer.Renderer method), 69
syntastic() (in module powerline.segments.vim.plugin.syntastic), 48
system_load() (in module powerline.segments.common.sys), 28

T

tab() (in module powerline.segments.vim), 47
tab_modified_indicator() (in module powerline.segments.vim), 47
tablister() (in module powerline.listers.vim), 50
tabnr() (in module powerline.segments.vim), 47
trailing_whitespace() (in module powerline.segments.vim), 47

U

update_renderer() (powerline.Powerline method), 66
uptime() (in module powerline.segments.common.sys), 28
user() (in module powerline.segments.common.env), 30

V

VimVarHandler (class in powerline.vim), 51
virtcol_current() (in module powerline.segments.vim), 47
virtualenv() (in module powerline.segments.common.env), 30
visual_range() (in module powerline.segments.vim), 47

W

warn() (powerline.PowerlineLogger method), 62
weather() (in module powerline.segments.common.wthr), 31
window_title() (in module powerline.segments.vim), 48
winnr() (in module powerline.segments.vim), 48
workspace() (in module powerline.segments.i3wm), 41
workspace_lister() (in module powerline.listers.i3wm), 50
workspaces() (in module powerline.segments.i3wm), 42