Powerline

Release beta

September 22, 2014

Contents

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 and Qtile.

1.1 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 project.

1.2 Screenshots

1.2.1 Vim statusline

Mode-dependent highlighting

-

- 

- 

- **Automatic truncation of segments in small windows**

- 

- 

-

CHAPTER 2

Installation

2.1 Generic requirements

- Python 2.6 or later, 3.2 or later, PyPy 2.0 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.
- `mercurial` python package (note: *not* standalone 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.
- `i3-py`, available on github. Required for i3wm bindings and segments.

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

2.2 Pip installation

This project is currently unavailable from PyPI due to a naming conflict with an unrelated project, thus you will have to use the following command to install powerline with `pip`:

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

. You may also choose to clone powerline repository somewhere and use

```
pip install -e --user {path_to_powerline}
```

, but note that in this case `pip` will not install `powerline` executable and you will have to do something like

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

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

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

To install release version uploaded to PyPI use

```
pip install powerline-status
```

2.3 Fonts installation

Powerline uses several special glyphs to get the arrow effect and some custom symbols for developers. This requires that you either have a symbol font or a patched font on your system. Your terminal emulator must also support either patched fonts or fontconfig for Powerline to work properly.

You can also enable *24-bit color support* if your 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.

2.3.1 Patched fonts

This method is the fallback method and works for every terminal, with the exception of *rxvt-unicode*.

Download the font of your choice from powerline-fonts. If you can't find your preferred font in the powerline-fonts repo, you'll have to patch your own font instead.

After downloading this font refer to platform-specific instructions.

2.4 Installation on various platforms

2.4.1 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 for you.

- Arch Linux (AUR), Python 2 version
- Arch Linux (AUR), Python 3 version
- Gentoo Live ebuild in raiagent overlay

If you're running a distribution without an official package you'll have to follow the installation guide below:

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

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

Note: You need to use the GitHub URI when installing Powerline! This project is currently unavailable on the PyPI due to a naming conflict with an unrelated project.

Note: If you are powerline developer you should be aware that `pip install --editable` does not currently fully work. If you install powerline this way you will be missing `powerline` executable and need to symlink it. It will be located in `scripts/powerline`.

Fonts installation

Fontconfig

This method only works on Linux. It's the recommended method if your terminal emulator supports it as you don't have to patch any fonts, and it generally works well with any coding font.

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

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

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 you moved the font to (you may need to be root to update the cache for 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 you can't see the custom symbols, please close all instances of your terminal emulator. You may need to restart X for the changes to take effect.

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

Patched font installation

After downloading font you should do the following:

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

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

2. Update font cache for the path you moved the font to (you may need to be root to update the cache for system-wide paths):

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

After installing the patched font you need to update Gvim or your terminal emulator to use the patched font. The correct font usually ends with *for Powerline*.

If you can't see the custom symbols, please close all instances of your terminal emulator. You may need to restart X for the changes to take effect.

If you *still* can't see the custom symbols, double-check that you have installed the font to a valid X font path.

2.4.2 Installation on OS X

Python package

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

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

   . You may use homebrew for this:

   ```
   brew install python
   ```

   .

   Note: In case you want or have to use `powerline.sh` socat-based client you should also install GNU env named `genv`. This may be achieved by running `brew install coreutils`.

2. Install Powerline using the following command:

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

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

   Note: You need to use the GitHub URI when installing Powerline! This project is currently unavailable on the PyPI due to a naming conflict with an unrelated project.

   Note: If you are powerline developer you should be aware that `pip install --editable` does not currently fully work. If you install powerline this way you will be missing `powerline` executable and need to symlink it. 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 if you're using MacVim you need to install it using the following command:

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

Fonts installation

Install downloaded patched font by double-clicking the font file in Finder, then clicking *Install this font* in the preview window.

After installing the patched font you need to update MacVim or your terminal emulator to use the patched font. The correct font usually ends with *for Powerline*.

CHAPTER 3

Usage

3.1 Application-specific requirements

3.1.1 Vim plugin requirements

The vim plugin requires a vim version with Python support compiled in. You can check if your vim supports Python by running `vim --version | grep +python`.

If your vim version doesn't have support for Python, you'll have to compile it with the `--enable-pythoninterp` flag (`--enable-python3interp` if you want Python 3 support instead). Note that this also requires the related Python headers to be installed on your system. Please consult your 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 is known to work on vim-7.0.112 (some segments may not work though as it was not actually tested).

3.1.2 Terminal emulator requirements

Powerline uses several special glyphs to get the arrow effect and some custom symbols for developers. This requires that you either have a symbol font or a patched font on your system. Your terminal emulator must also support either patched fonts or fontconfig for Powerline to work properly.

You can also enable *24-bit color support* if your 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	⚠ [6]	✘	✘
st	Linux	✔	✔	✔ [7]
Terminal.app	OS X	✔	✘	✘
libvte-based [8]	Linux	✔	✔	✔ [9]
xterm	Linux	✔	✘	⚠ [10]

3.2 Plugins

3.2.1 Shell prompts

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

```
powerline-daemon -q
```

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

Bash prompt

Add the following line to your `bashrc`, where `{repository_root}` is the absolute path to your Powerline installation directory:

```
. {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

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

in your 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. Do not expect it to work properly if you e.g. put current time there.
At minimum they are computed once on startup.

Zsh prompt

Add the following line to your `zshrc`, where `{repository_root}` is the absolute path to your Powerline installation directory:

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

Fish prompt

Add the following line to your `config.fish`, where `{repository_root}` is the absolute path to your Powerline installation directory:

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

Busybox (ash), mksh and dash prompt

After launching busybox run the following command:

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

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

```
if test "x$0" != "x${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.

3.2.2 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 your `rc.lua`, where `{repository_root}` is the absolute path to your Powerline installation directory:

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

Then add the `powerline_widget` to your `wibox`:

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

Qtile widget

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

```
from powerline.bindings.qtile.widget import Powerline

screens = [
    Screen(
        top=bar.Bar([
                # ...
                Powerline(timeout=2),
                # ...
            ],
        ),
    ),
]
```

I3 bar

Note: Until the patch is done in i3, you will need a custom `i3bar` build called `i3bgbar`. The source is available here.

Add the following to your `~/.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.

3.2.3 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 your `vimrc`.

If you just cloned the repository add the following line to your `vimrc`, where `{repository_root}` is the absolute path to your Powerline installation directory:

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

If you're using pathogen and don't want Powerline functionality in any other applications, simply add Powerline as a bundle and point the path above to the Powerline bundle directory, e.g. `~/.vim/bundle/powerline/powerline/bindings/vim`.

With Vundle you may instead use

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

(replace `Bundle` with `NeoBundle` for NeoBundle).

For vim-addon-manager it is even easier since you don't need to write this big path or install anything by hand: `powerline` is installed and run just like any other plugin using

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

> **Warning:** *Never* install powerline with pathogen/VAM/Vundle/NeoBundle *and* with pip. If you want powerline functionality in vim and other applications use system-wide installation if your system has powerline package, pip-only or `pip install --editable` kind of installation performed on the repository installed by Vim plugin manager.
> If you have installed powerline with pip and with some of Vim package managers do never report any errors to powerline bug tracker, especially errors occurring after updates.

Note: If you use supplied `powerline.vim` file 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 you must also set `g:powerline_pyeval`. Powerline will not check that Vim is compiled with Python support if you set `g:powerline_pycmd` to an unexpected value.

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

Tmux statusline

Add the following lines to your `.tmux.conf`, where `{repository_root}` is the absolute path to your Powerline installation directory:

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

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

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

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

to `.tmux.conf`.

IPython prompt

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

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

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

For IPython>=0.11 add the following line to your `ipython_config.py` file in the profile you are using:

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

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

CHAPTER 4

Configuration and customization

Note: You DO NOT have to fork the main GitHub repo to personalize your 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/colorscheme/`*`extension`*`/__main__.json`, `powerline/colorschemes/`*`extension`*`/`*`name`*`.json`

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

The default configuration files are stored in the main package. User configuration files are stored in `$XDG_CONFIG_HOME/powerline` for Linux users, and in `~/.config/powerline` for OS X users. This usually corresponds to `~/.config/powerline` on both platforms.

If you need per-instance configuration please refer to *Local configuration overrides*.

Note: If you have multiple configuration files with the same name in different directories then these files will be merged. Merging happens in the following order:

- *`powerline_root`*`/powerline/config_files` is checked for configuration first. Configuration from this source has least priority.
- `$XDG_CONFIG_DIRS/powerline` directories are the next ones to check. Checking happens in the reversed order: directories mentioned last are checked before directories mentioned first. Each new found file is merged with the result of previous merge.
- `$XDG_CONFIG_HOME/powerline` directory is the last to check. Configuration from there has top priority.

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.

4.1 Quick setup guide

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

Start by copying the entire set of default configuration files to the corresponding path in your user config directory:

```
mkdir ~/.config/powerline
cp -R /path/to/powerline/config_files/* ~/.config/powerline
```

Each extension (vim, tmux, etc.) has its own theme, and they are located in *config directory*/themes/*extension*/default.json.

If you want to move, remove or customize any of the provided segments, 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.

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 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.

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"
    }
},
```

4.2 References

4.2.1 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 your terminal emulator supports 24-bit colors.

`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 you 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 (unless you are sure you always need it). 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 path which will hold powerline logs. If not present, logging will be done to stderr.

`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` 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.

`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.

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 you define gradients from least alert color to most alert or use non-alert colors.

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*.

   attr 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 you do not need any attributes leave this 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
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
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.

---

dividers Defines the dividers used in all Powerline extensions. This option should usually only be changed if you don't have a patched font, or if you use a font patched with the legacy font patcher.

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 segments_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 `segments_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_group option*.

segments_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 `segments_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_group 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 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.

4.2.2 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`**(*status_colors=False*)

Return the current VCS branch.

Parameters **status_colors** (*bool*) – determines whether repository status will be used to determine highlighting. Default: False.

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

System properties

`powerline.segments.common.sys.`**`cpu_load_percent`**(*interval=1*, *format=u'{0:.0f}%'*, *shutdown_event=None*, *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`**(*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.

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'detect'*)

Return internal IP address

Requires `netifaces` module to work properly.

Parameters

- **interface** (*str*) – Interface on which IP will be checked. Use `detect` 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.

- **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*, *after_update=False*, *key=None*, *shutdown_event=None*, *update_first=True*, *interface=u'detect'*, *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 "detect" 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
- **recv_format** (*str*) – format string, receives `value` as argument
- **sent_format** (*str*) – format string, 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: `background: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_user=None*)

Return the current user.

Parameters **hide_user** (*str*) – Omit showing segment for users with names equal to this string.

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`**()

Return the name of the current Python virtualenv.

Battery

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

Return battery charge status.

Parameters

- **format** (*str*) – Percent format in case gamify is False.
- **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`.
- **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.

`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`.

Weather

`powerline.segments.common.wthr.`**`weather`**(*interval=600*, *after_update=False*, *key=None*, *shutdown_event=None*, *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://freegeoip.net/ 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*, *after_update=False*, *key=None*, *shutdown_event=None*, *update_first=True*, *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.

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

Media players

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**(*status_colors=False*)
Return the current working branch.

Parameters **status_colors** (*bool*) – determines whether repository status will be used to determine highlighting. Default: False.

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.**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 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

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.

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.**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`.

Ctrl-P segments

powerline.segments.vim.plugin.ctrlp.**ctrlp**(*side*)

Highlight groups used: `ctrlp.regex` or `background`, `ctrlp.prev` or `background`, `ctrlp.item` or `file_name`, `ctrlp.next` or `background`, `ctrlp.marked` or `background`, `ctrlp.focus` or `background`, `ctrlp.byfname` or `background`, `ctrlp.progress` or `file_name`, `ctrlp.progress` or `file_name`.

powerline.segments.vim.plugin.ctrlp.**ctrlp_stl_left_main**(*focus*, *byfname*, *regex*, *prev*, *item*, *next*, *marked*)

Highlight groups used: `ctrlp.regex` or `background`, `ctrlp.prev` or `background`, `ctrlp.item` or `file_name`, `ctrlp.next` or `background`, `ctrlp.marked` or `background`.

powerline.segments.vim.plugin.ctrlp.**ctrlp_stl_left_prog**(*progress*)

Highlight groups used: `ctrlp.progress` or `file_name`.

powerline.segments.vim.plugin.ctrlp.**ctrlp_stl_right_main**(*focus*, *byfname*, *regex*, *prev*, *item*, *next*, *marked*)

Highlight groups used: `ctrlp.focus` or `background`, `ctrlp.byfname` or `background`.

powerline.segments.vim.plugin.ctrlp.**ctrlp_stl_right_prog**(*progress*)

Highlight groups used: `ctrlp.progress` or `file_name`.

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 (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`.

4.2.3 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.

Currently only Vim listers are available.

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 either `buf:` or `buf_nc:` 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.

4.2.4 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

4.2.5 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__{theme_name} Dictionary, recursively merged with contents of `powerline/themes/vim/`*`theme_name`*`.json`. Note that this way you can't redefine some value (e.g. segment) in list, 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.

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=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_option=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`.

Zsh/zpython overrides

Here overrides are controlled by similarly to the powerline script, but values are taken from zsh variables.

POWERLINE_CONFIG 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_CONFIG 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`, 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 you set variable without quotes:

`POWERLINE_CONFIG_PATHS=( ~/example )`. You should use array parameter or the usual colon-separated `POWERLINE_CONFIG_PATHS=$HOME/path1:$HOME/path2`.

Ipython overrides

Ipython overrides depend on ipython version. Before ipython-0.11 you should pass additional keyword arguments to setup() function. After ipython-0.11 you should use `c.Powerline.KEY`. 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.

`paths` Sets directories where configuration should be read from. If present, no default locations are searched for configuration. No expansions are performed thus you cannot use paths starting with `~/`.

Prompt command

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

Note: `$POWERLINE_COMMAND` appears in shell scripts without quotes thus you can specify additional parameters in bash. In tmux it is passed to `eval` and depends on the shell used. POSIX-compatible shells, zsh, bash and fish will split this variable in this case.

If you want to disable prompt in shell, but still have tmux support or if you want to disable tmux support you can use variables `$POWERLINE_NO_{SHELL}_PROMPT`/`$POWERLINE_NO_SHELL_PROMPT` and `$POWERLINE_NO_{SHELL}_TMUX_SUPPORT`/`$POWERLINE_NO_SHELL_TMUX_SUPPORT` (substitute `{SHELL}` with the name of the shell (all-caps) you want to disable support for (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.

If you do not want to disable prompt in shell, but yet do not want to launch python twice to get *above* lines you do not use in tcsh you should set `$POWERLINE_NO_TCSH_ABOVE` or `$POWERLINE_NO_SHELL_ABOVE` variable.

If you do not want to see additional space which is added to the right prompt in fish in order to support multiline prompt you should set `$POWERLINE_NO_FISH_ABOVE` or `$POWERLINE_NO_SHELL_ABOVE` variables.

CHAPTER 5

Developer guide

5.1 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 you need to log something.

`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).

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 your 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_group': [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.

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

Defaults to the name of the segment.

Note: If you want to include your segment in powerline you must specify all highlighting groups used 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 your 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.

You must specify all groups 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 you want to include your segment in powerline you must specify used groups in the segment documentation in the form:

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

This text must not wrap and you are supposed to end all divider highlight group names 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_group* 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*.

5.1.1 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_group, 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.

5.1.2 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`.

5.1.3 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:** You must not ever use `os.environ`. If your segment is run in daemon you will get daemon's environment which is not correct. If your segment is run in Vim or in zsh with libzpython you will get Vim or zsh environment at python startup.

getcwd Function that returns current working directory being called with no arguments. You must not use `os.getcwd` for the same reasons you must not use `os.environ`, 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. You may obtain one using `vim.current.window` or `vim.windows[number - 1]`. May be a false object, in which case you should not use any of this objects' properties.

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. You should assume that this ID is hashable and supports equality comparison, but you must not use any other assumptions about it. Currently uses integer numbers incremented each time window is created.

buffer `vim.Buffer` object. You may obtain one 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 you must iterate over `vim.buffers` and check their `number` attributes.

bufnr Buffer number.

tabpage `vim.Tabpage` object. You may obtain one using `vim.current.tabpage` or `vim.tabpages[number - 1]`. May be a false object, in which case you should not use any of this objects' properties.

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: Your 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 you use `window_cached` decorator, "current tabpage" restriction may be safely ignored if you do not plan to ever see your segment in the tabline.

Warning: Powerline is being tested with vim-7.2 and will be tested with it until travis changes used vim version. This means that you may not use most of the functionality like `vim.Window.number`, `vim.*.vars`, `vim.*.options` or even `dir(vim object)` if you want your segment to be included in powerline.

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.

last_pipe_status List of exit codes returned by last programs in the pipe or some false object. Only available in `zsh`.

jobnum Number of background jobs.

renderer_arg Dictionary containing some keys that are additional arguments used by shell bindings. *You must not use this attribute 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.

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.

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`).

5.1.4 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.

5.1.5 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*)

5.2 Writing listers

Listers allow you 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`.

5.3 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*.

5.3.1 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 your matcher in his configuration file `KeyError` is raised.

5.3.2 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.

5.4 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 (if you are creating powerline extension 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.

5.4.1 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_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.

default_log_stream = <open file '<stdout>', mode 'w' at 0x7f3a89426150>

Default stream for default log handler

Usually it is `sys.stderr`, but there is sometimes a reason to prefer `sys.stdout` or a custom file-like object. It is not supposed to be used to write to some file.

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.

5.4.2 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.

character_translations = {}
Character translations for use in escape() function.

See documentation of `unicode.translate` for details.

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*, *attr=None*)
Output highlighted chunk.

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

hlstyle (*fg=None*, *bg=None*, *attr=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.

np_character_translations = {0: u'^@', 1: u'^A', 2: u'^B', 3: u'^C', 4: u'^D', 5: u'^E', 6: u'^F', 7: u'^G', 8: u'
Non-printable character translations

These are used to transform characters in range 0x00—0x1F into `^@`, `^A` and so on. Unilke with `.escape()` method (and `character_translations`) result is passed to `.strwidth()` method.

Note: transforms tab into `^I`.

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_theme()` 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'getcwd': <built-in function getcwdu>, u'environ': {'CELERY_LOG_REDIRECT_LEVEL': 'WAR
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(*string*)
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.

5.5 Tips and tricks for powerline developers

5.5.1 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

6.1 System-specific issues

6.1.1 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.

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).

6.1.2 Troubleshooting on OS X

I can't see any fancy symbols, what's wrong?

- If you're using iTerm2, please update to this revision or newer.
- 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.
- 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 for a discussion and other possible solutions for this issue.

6.2 Common issues

6.2.1 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.

6.2.2 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.

6.2.3 After an update something stopped working

Assuming powerline was working before update and stopped only after there are two possible explanations:

- 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. 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 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

```
pip install --user --editable path/to/powerline
```

6.3 Shell issues

6.3.1 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.

6.3.2 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.

6.3.3 When using z powerline shows wrong number of jobs

This happens because `z` 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` after powerline has done its job.

6.3.4 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.

6.4 Vim issues

6.4.1 My vim statusline has strange characters like ^B in it!

- Please add `set encoding=utf-8` to your `vimrc`.

6.4.2 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:\
```

6.4.3 My vim statusline is hidden/only appears in split windows!

- Make sure that you have `set laststatus=2` in your `vimrc`.

6.4.4 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).

6.4.5 Powerline loses color after editing vimrc

If your vimrc has something like

```
autocmd! BufWritePost vimrc :source ~/.vimrc
```

to automatically source vimrc after saving it you must then 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` event, but without `nested` this event is not launched. See also autocmd-nested Vim documentation.

6.4.6 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.

Note: This setting was introduced in version 6.3.1 of minibufexpl and removed in version 6.5.0 of its successor minibufexplorer. It is highly advised to use the latter because minibufexpl was last updated late in 2004.

CHAPTER 7

Tips and tricks

7.1 Vim

7.1.1 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)
```

7.2 Rxvt-unicode

7.2.1 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.

7.2.2 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\
```

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.

7.3 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.

8.1 Authors

- Kim Silkebækken
- Nikolay Pavlov
- Kovid Goyal

8.2 Contributors

- List of contributors
- The glyphs in the font patcher are created by Fabrizio Schiavi, creator of the excellent coding font Pragmata Pro.

CHAPTER 9

Indices and tables

- *genindex*
- *modindex*
- *search*

p