CONTENTS

1 Getting started 1
2 Advanced scripting 49
3 Getting involved 67
4 Miscellaneous 75
Index 77
1.1 Installing Qtile

1.1.1 Distro Guides

Below are the preferred installation methods for specific distros. If you are running something else, please see *Installing From Source*.

**Installing on Arch Linux**

Stable versions of Qtile are currently packaged for Arch Linux. To install this package, run:

```
pacman -S qtile
```

Please see the ArchWiki for more information on Qtile.

**Installing on Fedora**

Stable versions of Qtile are currently packaged for current versions of Fedora. To install this package, run:

```
dnf -y install qtile
```

**Installing on Funtoo**

Latest versions of Qtile are available on Funtoo. To install it, run:

```
emerge -av x11-wm/qtile
```

You can also install the development version from GitHub:

```
echo "x11-wm/qtile-9999 ***" >> /etc/portage/package.accept_keywords
emerge -av qtile
```
You can customize your installation with the following useflags:

- `dbus`
- `widget-khal-calendar`
- `widget-imap`
- `widget-keyboardkbdd`
- `widget-launchbar`
- `widget-mpd`
- `widget-mpris`
- `widget-wlan`

The `dbus` useflag is enabled by default. Disable it only if you know what it is and know you don’t use/need it.

All `widget-*` useflags are disabled by default because these widgets require additional dependencies while not everyone will use them. Enable only widgets you need to avoid extra dependencies thanks to these useflags.

Visit Funtoo Qtile documentation for more details on Qtile installation on Funtoo.

**Installing on Debian or Ubuntu**

Note: As of Ubuntu 20.04 (Focal Fossa), the package has been outdated and removed from the Ubuntu’s official package list. Users are advised to follow the instructions of Installing From Source.

On other recent Ubuntu (17.04 or greater) and Debian unstable versions, there are Qtile packages available via:

```bash
sudo apt-get install qtile
```

On older versions of Ubuntu (15.10 to 16.10) and Debian 9, the dependencies are available via:

```bash
sudo apt-get install python3-xcffib python3-cairocffi
```

**Installing on Slackware**

Qtile is available on the SlackBuilds.org as:

<table>
<thead>
<tr>
<th>Package Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>qtile</td>
<td>stable branch (release)</td>
</tr>
</tbody>
</table>

**Using slpkg (third party package manager)**

The easy way to install Qtile is with `slpkg`. For example:

```bash
slpkg -s sbo qtile
```
Manual installation

Download dependencies first and install them. The order in which you need to install is:

- pycparser
- cffi
- futures
- python-xcffib
- trollius
- cairocffi
- qtile

Please see the HOWTO for more information on SlackBuild Usage HOWTO.

Installing on FreeBSD

Qtile is available via FreeBSD Ports. It can be installed with

```
pkg install qtile
```

1.1.2 Installing From Source

First, you need to install all of Qtile’s dependencies (although some are optional/not needed depending on your Python version, as noted below).

Note that Python 3 versions 3.5 and newer are currently supported and tested, including corresponding PyPy3 versions.

xcffib

Qtile uses xcffib as an XCB binding, which has its own instructions for building from source. However, if you’d like to skip building it, you can install its dependencies, you will need libxcb and libffi with the associated headers (libxcb-render0-dev and libffi-dev on Ubuntu), and install it via PyPI:

```
pip install xcffib
```

cairocffi

Qtile uses cairocffi with XCB support via xcffib. You’ll need libcairo2, the underlying library used by the binding. You should be sure before you install cairocffi that xcffib has been installed, otherwise the needed cairo-xcb bindings will not be built. Once you’ve got the dependencies installed, you can use the latest version on PyPI:

```
pip install --no-cache-dir cairocffi
```
You'll also need `libpangocairo`, which on Ubuntu can be installed via `sudo apt-get install libpangocairo-1.0-0`. Qtile uses this to provide text rendering (and binds directly to it via cffi with a small in-tree binding).

**dbus/gobject**

Until someone comes along and writes an asyncio-based dbus library, qtile will depend on `python-dbus` to interact with dbus. This means that if you want to use things like notification daemon or mpris widgets, you'll need to install `python-gobject` and `python-dbus`. Qtile will run fine without these, although it will emit a warning that some things won't work.

**Qtile**

With the dependencies in place, you can now install qtile:

```bash
git clone git://github.com/qtile/qtile.git
cd qtile
pip install .
```

Stable versions of Qtile can be installed from PyPI:

```bash
pip install qtile
```

As long as the necessary libraries are in place, this can be done at any point, however, it is recommended that you first install xcffib to ensure the cairo-xcb bindings are built (see above).

### 1.2 Configuration

Qtile is configured in Python. A script (`~/.config/qtile/config.py` by default) is evaluated, and a small set of configuration variables are pulled from its global namespace.

#### 1.2.1 Configuration lookup order

Qtile looks in the following places for a configuration file, in order:

- The location specified by the `-c` argument.
- `$XDG_CONFIG_HOME/qtile/config.py`, if it is set
- `~/.config/qtile/config.py`
- It reads the module `libqtile.resources.default_config`, included by default with every Qtile installation.

Qtile will try to create the configuration file as a copy of the default config, if it doesn’t exist yet.
1.2.2 Default Configuration

The default configuration is invoked when qtile cannot find a configuration file. In addition, if qtile is restarted via qshell, qtile will load the default configuration if the config file it finds has some kind of error in it. The documentation below describes the configuration lookup process, as well as what the key bindings are in the default config.

The default config is not intended to be suitable for all users; it’s mostly just there so qtile does /something/ when fired up, and so that it doesn’t crash and cause you to lose all your work if you reload a bad config.

Key Bindings

The mod key for the default config is mod4, which is typically bound to the “Super” keys, which are things like the windows key and the mac command key. The basic operation is:

- mod + k or mod + j: switch windows on the current stack
- mod + <space>: put focus on the other pane of the stack (when in stack layout)
- mod + <tab>: switch layouts
- mod + w: close window
- mod + <ctrl> + r: restart qtile with new config
- mod + <group name>: switch to that group
- mod + <shift> + <group name>: send a window to that group
- mod + <enter>: start terminal guessed by libqtile.utils.guess_terminal
- mod + r: start a little prompt in the bar so users can run arbitrary commands

The default config defines one screen and 8 groups, one for each letter in asdfuiop. It has a basic bottom bar that includes a group box, the current window name, a little text reminder that you’re using the default config, a system tray, and a clock.

The default configuration has several more advanced key combinations, but the above should be enough for basic usage of qtile.

See Keybindings in images for visual keybindings in keyboard layout.

Mouse Bindings

By default, holding your mod key and clicking (and holding) a window will allow you to drag it around as a floating window.

1.2.3 Configuration variables

A Qtile configuration consists of a file with a bunch of variables in it, which qtile imports and then runs as a python file to derive its final configuration. The documentation below describes the most common configuration variables; more advanced configuration can be found in the qtile-examples repository, which includes a number of real-world configurations that demonstrate how you can tune Qtile to your liking. (Feel free to issue a pull request to add your own configuration to the mix!)
Lazy objects

The `lazy.lazy` object is a special helper object to specify a command for later execution. This object acts like the root of the object graph, which means that we can specify a key binding command with the same syntax used to call the command through a script or through `qshell`.

Example

```python
from libqtile.config import Key
from libqtile.command import lazy

keys = [
    Key(
        ["mod1"], "k",
        lazy.layout.down()
    ),
    Key(
        ["mod1"], "j",
        lazy.layout.up()
    )
]
```

Lazy functions

This is an overview of the commonly used functions for the key bindings. These functions can be called from commands on the `Qtile` object or on another object in the command tree.

Some examples are given below.

General functions

<table>
<thead>
<tr>
<th>function</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>lazy.spawn(&quot;app&quot;)</td>
<td>Run the application</td>
</tr>
<tr>
<td>lazy.spawncmd()</td>
<td>Open command prompt on the bar. See prompt widget.</td>
</tr>
<tr>
<td>lazy.restart()</td>
<td>Restart Qtile and reload its config. It won’t close your windows</td>
</tr>
<tr>
<td>lazy.shutdown()</td>
<td>Close the whole Qtile</td>
</tr>
</tbody>
</table>
### Group functions

<table>
<thead>
<tr>
<th>function</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>lazy. next_layout()</td>
<td>Use next layout on the actual group</td>
</tr>
<tr>
<td>lazy. prev_layout()</td>
<td>Use previous layout on the actual group</td>
</tr>
<tr>
<td>lazy.screen.next_group()</td>
<td>Move to the group on the right</td>
</tr>
<tr>
<td>lazy.screen.prev_group()</td>
<td>Move to the group on the left</td>
</tr>
<tr>
<td>lazy.screen.toggle_group()</td>
<td>Move to the last visited group</td>
</tr>
<tr>
<td>lazy.group(&quot;group_name&quot;)</td>
<td>Move to the group called group_name. Takes an optional toggle parameter (defaults to True). If this group is already on the screen, then the group is toggled with last used</td>
</tr>
<tr>
<td>lazy.layout.increase_ratio</td>
<td>Increase the space for master window at the expense of slave windows</td>
</tr>
<tr>
<td>lazy.layout.decrease_ratio</td>
<td>Decrease the space for master window in the advantage of slave windows</td>
</tr>
</tbody>
</table>

### Window functions

<table>
<thead>
<tr>
<th>function</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>lazy.window.kill()</td>
<td>Close the focused window</td>
</tr>
<tr>
<td>lazy.layout.next()</td>
<td>Switch window focus to other pane(s) of stack</td>
</tr>
<tr>
<td>lazy.window.to_group(&quot;group_name&quot;)</td>
<td>Move focused window to the group called group_name</td>
</tr>
<tr>
<td>lazy.window.toggle_floating()</td>
<td>Put the focused window to/from floating mode</td>
</tr>
<tr>
<td>lazy.window.toggle_fullscreen()</td>
<td>Put the focused window to/from fullscreen mode</td>
</tr>
</tbody>
</table>
ScratchPad DropDown functions

<table>
<thead>
<tr>
<th>function</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>lazy. group(&quot;group_name&quot;) dropdown_toggle(&quot;name&quot;)</td>
<td>Toggles the visibility of the specified DropDown window. On first use, the configured process is spawned.</td>
</tr>
</tbody>
</table>

Groups

A group is a container for a bunch of windows, analogous to workspaces in other window managers. Each client window managed by the window manager belongs to exactly one group. The `groups` config file variable should be initialized to a list of `DGroup` objects.

`DGroup` objects provide several options for group configuration. Groups can be configured to show and hide themselves when they’re not empty, spawn applications for them when they start, automatically acquire certain groups, and various other options.

Example

```python
from libqtile.config import Group, Match

groups = [
    Group("a"),
    Group("b"),
    Group("c", matches=[Match(wm_class=['Firefox'])]),
]

# allow mod3+1 through mod3+0 to bind to groups; if you bind your groups # by hand in your config, you don’t need to do this.
from libqtile.dgroups import simple_key_binder
dgroups_key_binder = simple_key_binder("mod3")
```

Reference

Group

```python
class libqtile.config.Group (name, matches=None, exclusive=False, spawn=None, layout=None, layouts=None, persist=True, init=True, layout_opts=None, screen_affinity=None, position=9223372036854775807, label=None)
```

Represents a “dynamic” group.

These groups can spawn apps, only allow certain Matched windows to be on them, hide when they’re not in use, etc. Groups are identified by their name.

Parameters

- `name` [string] the name of this group
- `matches` [default None] list of `Match` objects whose windows will be assigned to this group
- `exclusive` [boolean] when other apps are started in this group, should we allow them here or not?
spawn [string or list of strings] this will be `exec()` d when the group is created, you can pass either a program name or a list of programs to `exec()`

layout [string] the name of default layout for this group (e.g. ‘max’ or ‘stack’). This is the name specified for a particular layout in config.py or if not defined it defaults in general the class name in all lower case.

layouts [list] the group layouts list overriding global layouts. Use this to define a separate list of layouts for this particular group.

persist [boolean] should this group stay alive with no member windows?

init [boolean] is this group alive when qtile starts?

position [int] group position

label [string] the display name of the group. Use this to define a display name other than name of the group. If set to None, the display name is set to the name.

```
libqtile.dgroups.simple_key_binder (mod, keynames=None)
```

Bind keys to mod+group position or to the keys specified as second argument

**Group Matching**

**Match**

```python
class libqtile.config.Match (title=None, wm_class=None, role=None, wm_type=None,
                             wm_instance_class=None, net_wm_pid=None)
```

Match for dynamic groups

It can match by title, wm_class, role, wm_type, wm_instance_class or net_wm_pid.

Match supports both regular expression objects (i.e. the result of `re.compile()` ) or strings (match as an “include”-match). If a window matches all specified values, it is considered a match.

**Parameters**

- **title**: matches against the title (WM_NAME)
- **wm_class**: matches against the second string in WM_CLASS atom
- **role**: matches against the WM_ROLE atom
- **wm_type**: matches against the WM_TYPE atom
- **wm_instance_class**: matches against the first string in WM_CLASS atom
- **net_wm_pid**: matches against the _NET_WM_PID atom (only int allowed for this rule)

**Rule**

```python
class libqtile.config.Rule (match, group=None, float=False, intrusive=False,
                           break_on_match=True)
```

How to act on a match

A Rule contains a list of Match objects, and a specification about what to do when any of them is matched.

**Parameters**

- **match**: Match object or a list of such associated with this Rule
- **float**: auto float this window?
intrusive: override the group’s exclusive setting?
break_on_match: Should we stop applying rules if this rule is matched?

ScratchPad and DropDown

ScratchPad is a special - by default invisible - group which acts as a container for DropDown configurations. A DropDown can be configured to spawn a defined process and bind that process’ window to it. The associated window can then be shown and hidden by the lazy command dropdown_toggle() (see Lazy objects) from the ScratchPad group. Thus - for example - your favorite terminal emulator turns into a quake-like terminal by the control of qtile.

If the DropDown window turns visible it is placed as a floating window on top of the current group. If the DropDown is hidden, it is simply switched back to the ScratchPad group.

Example

```python
from libqtile.config import Group, ScratchPad, DropDown, Key
from libqtile.command import lazy

groups = [
    ScratchPad("scratchpad", [
        # define a drop down terminal.
        # it is placed in the upper third of screen by default.
        DropDown("term", "urxvt", opacity=0.8),

        # define another terminal exclusively for qshell at different position
        DropDown("qshell", "urxvt -hold -e qshell",
                   x=0.05, y=0.4, width=0.9, height=0.6, opacity=0.9,
                   on_focus_lost_hide=True),
    ]),

    Group("a"),
]

toggle visibility of above defined DropDown named "term"
Key([], 'F11', lazy.group['scratchpad'].dropdown_toggle('term'))

Key([], 'F12', lazy.group['scratchpad'].dropdown_toggle('qshell'))
```

There is only one DropDown visible in current group at a time. If a further DropDown is set visible the currently shown DropDown turns invisible immediately.

Note that if the window is set to not floating, it is detached from DropDown and ScratchPad, and a new process is spawned next time the DropDown is set visible.

Reference

ScratchPad

```python
class libqtile.config.ScratchPad(name, dropdowns=None, position=922372036854775807, label="")
```

Represents a “ScratchPad” group

ScratchPad adds a (by default) invisible group to qtile. That group is used as a place for currently not visible windows spawned by a DropDown configuration.

Parameters
name [string] the name of this group

dropdowns [default None] list of DropDown objects

position [int] group position

label [string] The display name of the ScratchPad group. Defaults to the empty string such that the group is hidden in GroupList widget.

**DropDown**

class libqtile.config.DropDown(name, cmd, **config)

Configure a specified command and its associated window for the ScratchPad. That window can be shown and hidden using a configurable keystroke or any other scripted trigger.

<table>
<thead>
<tr>
<th>key</th>
<th>default</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>height</td>
<td>0.35</td>
<td>'Height of window as fraction of current screen.'</td>
</tr>
<tr>
<td>on_focus_lost_hide</td>
<td>True</td>
<td>'Shall the window be hidden if focus is lost? If so, the DropDown is hidden if window focus or the group is changed.'</td>
</tr>
<tr>
<td>opacity</td>
<td>0.9</td>
<td>'Opacity of window as fraction. Zero is opaque.'</td>
</tr>
<tr>
<td>warp_pointer</td>
<td>True</td>
<td>'Shall pointer warp to center of window on activation? This has only effect if any of the on_focus_lost_xxx configurations is True'</td>
</tr>
<tr>
<td>width</td>
<td>0.8</td>
<td>'Width of window as fraction of current screen width'</td>
</tr>
<tr>
<td>x</td>
<td>0.1</td>
<td>'X position of window as fraction of current screen width. 0 is the left most position.'</td>
</tr>
<tr>
<td>y</td>
<td>0.0</td>
<td>'Y position of window as fraction of current screen height. 0 is the top most position. To show the window at bottom, you have to configure a value &lt; 1 and an appropriate height.'</td>
</tr>
</tbody>
</table>

**Keys**

The keys variable defines Qtile’s key bindings. Individual key bindings are defined with libqtile.config.Key as demonstrated in the following example. Note that you may specify more than one callback functions.

```
from libqtile.config import Key

keys = [
    # Pressing "Meta + Shift + a".
    Key(["mod4", "shift"], "a", callback, ...),

    # Pressing "Control + p".
    Key(["control"], "p", callback, ...),

    # Pressing "Meta + Tab".
    Key(["mod4", "mod1"], "Tab", callback, ...),
]
```

The above may also be written more concisely with the help of the libqtile.config.EzKey helper class. The following example is functionally equivalent to the above:

```
from libqtile.config import EzKey as Key

keys = [
    EzKey(["mod4", "shift"], "a", callback, ...),
    EzKey(["control"], "p", callback, ...),
    EzKey(["mod4", "mod1"], "Tab", callback, ...),
]
```

(continues on next page)
Key("M-S-a", callback, ...),
Key("C-p", callback, ...),
Key("M-A-<Tab>", callback, ...),
]

The EzKey modifier keys (i.e. MASC) can be overwritten through the EzKey.modifier_keys dictionary. The defaults are:

```python
modifier_keys = {
    'M': 'mod4',
    'A': 'mod1',
    'S': 'shift',
    'C': 'control',
}
```

**Modifiers**

On most systems mod1 is the Alt key - you can see which modifiers, which are enclosed in a list, map to which keys on your system by running the `xmodmap` command. This example binds Alt-k to the “down” command on the current layout. This command is standard on all the included layouts, and switches to the next window (where “next” is defined differently in different layouts). The matching “up” command switches to the previous window.

Modifiers include: “shift”, “lock”, “control”, “mod1”, “mod2”, “mod3”, “mod4”, and “mod5”. They can be used in combination by appending more than one modifier to the list:

```python
Key(
    ["mod1", "control"], "k",
    lazy.layout.shuffle_down()
)
```

**Special keys**

These are most commonly used special keys. For complete list please see the code. You can create bindings on them just like for the regular keys. For example `Key(["mod1"], "F4", lazy.window.kill())`.

---

<table>
<thead>
<tr>
<th>Return</th>
<th>BackSpace</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tab</td>
<td>space</td>
</tr>
<tr>
<td>Home, End</td>
<td>Left, Up, Right, Down</td>
</tr>
<tr>
<td>F1, F2, F3, ...</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>XF86AudioRaiseVolume</th>
</tr>
</thead>
<tbody>
<tr>
<td>XF86AudioLowerVolume</td>
</tr>
<tr>
<td>XF86AudioMute</td>
</tr>
<tr>
<td>XF86AudioNext</td>
</tr>
<tr>
<td>XF86AudioPrev</td>
</tr>
<tr>
<td>XF86MonBrightnessUp</td>
</tr>
<tr>
<td>XF86MonBrightnessDown</td>
</tr>
</tbody>
</table>
Reference

Key

class libqtile.config.Key(modifiers: List[str], key: str, *commands, desc: str = '')
    Defines a keybinding.

    Parameters
    
    modifiers: A list of modifier specifications. Modifier specifications are one of: “shift”, “lock”, “control”, “mod1”, “mod2”, “mod3”, “mod4”, “mod5”.
    
    key: A key specification, e.g. “a”, “Tab”, “Return”, “space”.
    
    commands: A list of lazy command objects generated with the lazy.lazy helper. If multiple Call objects are specified, they are run in sequence.
    
    desc: description to be added to the key binding

EzConfig

class libqtile.config.EzConfig
    Helper class for defining key and button bindings in an emacs-like format. Inspired by Xmonad’s XMonad.Util.EZConfig.

Layouts

A layout is an algorithm for laying out windows in a group on your screen. Since Qtile is a tiling window manager, this usually means that we try to use space as efficiently as possible, and give the user ample commands that can be bound to keys to interact with layouts.

The `layouts` variable defines the list of layouts you will use with Qtile. The first layout in the list is the default. If you define more than one layout, you will probably also want to define key bindings to let you switch to the next and previous layouts.

See Built-in Layouts for a listing of available layouts.

Example

```python
from libqtile import layout
layouts = [
    layout.Max(),
    layout.Stack(stacks=2)
]
```
Mouse

The `mouse` config file variable defines a set of global mouse actions, and is a list of `Click` and `Drag` objects, which define what to do when a window is clicked or dragged.

Example

```python
from libqtile.config import Click, Drag

mouse = [
    Drag([mod], "Button1", lazy.window.set_position_floating(),
         start=lazy.window.get_position()),
    Drag([mod], "Button3", lazy.window.set_size_floating(),
         start=lazy.window.get_size()),
    Click([mod], "Button2", lazy.window.bring_to_front())
]
```

The above example can also be written more concisely with the help of the `EzClick` and `EzDrag` helpers:

```python
from libqtile.config import EzClick as Click, EzDrag as Drag

mouse = [
    Drag("M-1", lazy.window.set_position_floating(),
         start=lazy.window.get_position()),
    Drag("M-3", lazy.window.set_size_floating(),
         start=lazy.window.get_size()),
    Click("M-2", lazy.window.bring_to_front())
]
```

Reference

Click

```python
class libqtile.config.Click(modifiers: List[str], button: str, *commands, **kwargs)
```

Defines binding of a mouse click

It focuses clicked window by default. If you want to prevent it, pass `focus=None` as an argument

Drag

```python
class libqtile.config.Drag(*args, start=False, **kwargs)
```

Defines binding of a mouse to some dragging action

On each motion event command is executed with two extra parameters added x and y offset from previous move

It focuses clicked window by default. If you want to prevent it pass, `focus=None` as an argument
The `screens` configuration variable is where the physical screens, their associated `bars`, and the `widgets` contained within the bars are defined.

See *Built-in Widgets* for a listing of available widgets.

### Example

Tying together screens, bars and widgets, we get something like this:

```python
from libqtile.config import Screen
from libqtile import bar, widget

screens = [
    Screen(
        bottom=bar.Bar([widget.GroupBox(), widget.WindowName()], 30),
    ),
    Screen(
        bottom=bar.Bar([widget.GroupBox(), widget.WindowName()], 30),
    )
]
```

Bars support both solid background colors and gradients by supplying a list of colors that make up a linear gradient. For example, `bar.Bar(..., background="#000000")` will give you a black back ground (the default), while `bar.Bar(..., background=["#000000", "#FFFFFF"])` will give you a background that fades from black to white.

### Fake Screens

Instead of using the variable `screens` the variable `fake_screens` can be used to set split a physical monitor into multiple screens. They can be used like this:

```python
from libqtile.config import Screen
from libqtile import bar, widget

# screens look like this
# 600 300
# |-------------|-----|
# | 480| |580
# | A | B |
# |----------|--| |
# | 400|--|-----|
# | C | |400
# |----------| D |
# 500 |--------|
# 400

# Notice there is a hole in the middle
```

(continues on next page)
fake_screens = [
    Screen(
        bottom=bar.Bar(
            [widget.Prompt(),
             widget.Sep(),
             widget.WindowName(),
             widget.Sep(),
             widget.SysTray(),
             widget.Sep(),
             widget.Clock(format='%H:%M:%S %d.%m.%Y')
            ],
            24,
            background='#555555'
        ),
        x=0,
        y=0,
        width=600,
        height=480
    ),
    Screen(
        top=bar.Bar(
            [widget.GroupBox(),
             widget.WindowName(),
             widget.Clock()]
        ),
        x=600,
        y=0,
        width=300,
        height=580
    ),
    Screen(
        top=bar.Bar(
            [widget.GroupBox(),
             widget.WindowName(),
             widget.Clock()]
        ),
        x=0,
        y=480,
        width=500,
        height=400
    ),
    Screen(
        top=bar.Bar(
            [widget.GroupBox(),
             widget.WindowName(),
             widget.Clock()]
        ),
        x=600,
        y=480,
        width=300,
        height=580
    )]
Third-party bars

There might be some reasons to use third-party bars. For instance you can come from another window manager and you have already configured dzen2, xmobar, or something else. They definitely can be used with Qtile too. In fact, any additional configurations aren’t needed. Just run the bar and Qtile will adapt.

Reference

Screen


A physical screen, and its associated paraphernalia.

Define a screen with a given set of Bars of a specific geometry. Note that bar.Bar objects can only be placed at the top or the bottom of the screen (bar.Gap objects can be placed anywhere). Also, x, y, width, and height aren’t specified usually unless you are using ‘fake screens’.

The wallpaper parameter, if given, should be a path to an image file. How this image is painted to the screen is specified by the wallpaper_mode parameter. By default, the image will be placed at the screen’s origin and retain its own dimensions. If the mode is ‘fill’, the image will be centred on the screen and resized to fill it. If the mode is ‘stretch’, the image is stretched to fit all of it into the screen.

Bar

class libqtile.bar.Bar(widgets, size, **config)

A bar, which can contain widgets

Parameters

- widgets: A list of widget objects.
- size: The “thickness” of the bar, i.e. the height of a horizontal bar, or the width of a vertical bar.
## Gap

**class** `libqtile.bar.Gap(size)`

A gap placed along one of the edges of the screen.

If a gap has been defined, Qtile will avoid covering it with windows. The most probable reason for configuring a gap is to make space for a third-party bar or other static window.

**Parameters**

- `size`: The “thickness” of the gap, i.e. the height of a horizontal gap, or the width of a vertical gap.

## Hooks

Qtile provides a mechanism for subscribing to certain events in `libqtile.hook`. To subscribe to a hook in your configuration, simply decorate a function with the hook you wish to subscribe to.

See *Built-in Hooks* for a listing of available hooks.

## Examples

### Automatic floating dialogs

Let’s say we wanted to automatically float all dialog windows (this code is not actually necessary; Qtile floats all dialogs by default). We would subscribe to the `client_new` hook to tell us when a new window has opened and, if the type is “dialog”, as can set the window to float. In our configuration file it would look something like this:

```python
from libqtile import hook

@hook.subscribe.client_new
def floating_dialogs(window):
    dialog = window.window.get_wm_type() == 'dialog'
    transient = window.window.get_wm_transient_for()
    if dialog or transient:
        window.floating = True
```

A list of available hooks can be found in the *Built-in Hooks* reference.
Autostart

If you want to run commands or spawn some applications when Qtile starts, you’ll want to look at the `startup` and `startup_once` hooks. `startup` is emitted every time Qtile starts (including restarts), whereas `startup_once` is only emitted on the very first startup.

Let’s create a file `~/.config/qtile/autostart.sh` that will set our desktop wallpaper and start a few programs when Qtile first runs.

```bash
#!/bin/sh
feh --bg-scale ~/images/wallpaper.jpg &
pidgin &
dropbox start &
```

We can then subscribe to `startup_once` to run this script:

```python
import os
import subprocess

@hook.subscribe.startup_once
def autostart():
    home = os.path.expanduser('~/config/qtile/autostart.sh')
    subprocess.call([home])
```

Accessing the `qtile` object

If you want to do something with the Qtile manager instance inside a hook, it can be imported into your config:

```python
from libqtile import qtile
```

In addition to the above variables, there are several other boolean configuration variables that control specific aspects of Qtile’s behavior:
<table>
<thead>
<tr>
<th>variable</th>
<th>default</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>auto_fullscreen</td>
<td>True</td>
<td>If a window requests to be fullscreend, it is automatically fullscreened. Set this to false if you only want windows to be fullscreened if you ask them to be.</td>
</tr>
<tr>
<td>bring_front</td>
<td>False</td>
<td>When clicked, should the window be brought to the front or not. (This sets the X Stack Mode to Above.)</td>
</tr>
<tr>
<td>cursor_warp</td>
<td>False</td>
<td>If true, the cursor follows the focus as directed by the keyboard, warping to the center of the focused window.</td>
</tr>
<tr>
<td>dgroups_key_binder</td>
<td>None</td>
<td>A function which generates group binding hotkeys. It takes a single argument, the DGroups object, and can use that to set up dynamic key bindings. A sample implementation is available in libqtile/dgroups.py called simple_key_binder(), which will bind groups to mod+shift+0-10 by default.</td>
</tr>
<tr>
<td>dgroups_app_rules</td>
<td>[]</td>
<td>A list of Rule objects which can send windows to various groups based on matching criteria.</td>
</tr>
<tr>
<td>extension</td>
<td>as_wid_get_defaults</td>
<td>Default settings for extensions.</td>
</tr>
<tr>
<td>floating_layout</td>
<td>layout.Floating(float_rules=[. . .])</td>
<td>The default floating layout to use. This allows you to set custom floating rules among other things if you wish. See the configuration file for the default float_rules.</td>
</tr>
<tr>
<td>focus_on_window_activation</td>
<td>smart</td>
<td>Behavior of the _NET_ACTIVATE_WINDOW message sent by applications • urgent: urgent flag is set for the window • focus: automatically focus the window • smart: automatically focus if the window is in the current group • never: never automatically focus any window that requests it</td>
</tr>
<tr>
<td>follow_mouse_focus</td>
<td>True</td>
<td>Controls whether or not focus follows the mouse around as it moves across windows in a layout.</td>
</tr>
<tr>
<td>widget_defaults</td>
<td>dict(font='sans', fontsize=12, padding=3)</td>
<td>Default settings for bar widgets.</td>
</tr>
<tr>
<td>wminame</td>
<td>“LG3D”</td>
<td>Gasp! We’re lying here. In fact, nobody really uses or cares about this string besides java UI toolkits; you can see several discussions on the mailing lists, GitHub issues, and other WM documentation that suggest setting this string if your java app doesn’t work correctly. We may as well just lie and say that we’re a working one by default. We choose LG3D to maximize irony: it is a 3D non-reparenting WM written in java that happens to be on java’s whitelist.</td>
</tr>
</tbody>
</table>

### 1.2.4 Testing your configuration

The best way to test changes to your configuration is with the provided Xephyr script. This will run Qtile with your config.py inside a nested X server and prevent your running instance of Qtile from crashing if something goes wrong.

See *Hacking Qtile* for more information on using Xephyr.
1.2.5 Starting Qtile

There are several ways to start Qtile. The most common way is via an entry in your X session manager’s menu. The default Qtile behavior can be invoked by creating a `qtile.desktop` file in `/usr/share/xsessions`.

A second way to start Qtile is a custom X session. This way allows you to invoke Qtile with custom arguments, and also allows you to do any setup you want (e.g. special keyboard bindings like mapping caps lock to control, setting your desktop background, etc.) before Qtile starts. If you’re using an X session manager, you still may need to create a `custom.desktop` file similar to the `qtile.desktop` file above, but with `Exec=/etc/X11/xsession`. Then, create your own `~/.xsession`. There are several examples of user defined `xsessions` in the `qtile-examples` repository.

If there is no display manager such as SDDM, LightDM or other and there is need to start Qtile directly from `~/.xinitrc`, do that by adding `exec qtile` at the end.

In very special cases, ex. Qtile crashing during session, then suggestion would be to start through a loop to save running applications:

```
while true; do
  qtile
done
```

Finally, if you’re a gnome user, you can start integrate Qtile into Gnome’s session manager and use gnome as usual.

Running from systemctl

This case will cover automatic login to Qtile after booting the system without using display manager. It logins in virtual console and init X by running through session.

Automatic login to virtual console

To get login into virtual console as an example edit `getty` service by running `systemctl edit getty@tty1` and add instructions to `/etc/systemd/system/getty@tty1.service.d/override.conf`:

```
[Service]
ExecStart=-/usr/bin/agetty --autologin username --noclear %I $TERM
```

*username* should be changed to current user name.

Check more for other examples.

Autostart X session

After login X session should be started. That can be done by `.bash_profile` if bash is used or `.zprofile` in case of zsh. Other shells can be adjusted by given examples.

```
if systemctl -q is-active graphical.target &;&; [[ ! $DISPLAY &;&; $XDG_VTNR = eq 1 ]];;
  then
    exec startx
fi
```

And to start Qtile itself `.xinitrc` should be fixed:
# some apps that should be started before Qtile, ex.
# 
# ```
# [[ -f ~/.Xresources ]] && xrdb -merge ~/.Xresources
# ~/.fehbg &
# nm-applet &
# blueman-applet &
# dunst &
# 
# or
# ```
# 
# ```
# source ~/.xsession
# exec qtile
# ```

## Running Inside Gnome

Add the following snippet to your Qtile configuration. As per this page, it registers Qtile with gnome-session. Without it, a “Something has gone wrong!” message shows up a short while after logging in. dbus-send must be on your $PATH.

```python
import subprocess
import os
from libqtile import hook

@hook.subscribe.startup
def dbus_register():
    id = os.environ.get('DESKTOP_AUTOSTART_ID')
    if not id:
        return
    subprocess.Popen(['dbus-send',
                      '--session',
                      '--print-reply',
                      '--dest=org.gnome.SessionManager',
                      '/org/gnome/SessionManager',
                      'org.gnome.SessionManager.RegisterClient',
                      'string:qtile',
                      'string: ' + id])
```

This adds a new entry “Qtile GNOME” to GDM’s login screen.

```
$ cat /usr/share/xsessions/qtile_gnome.desktop
[Desktop Entry]
Name=Qtile GNOME
Comment=Tiling window manager
TryExec=/usr/bin/gnome-session
Exec=gnome-session --session=qtile
Type=XSession
```

The custom session for gnome-session.

For Gnome >= 3.23.2 (Ubuntu >= 17.04, Fedora >= 26, etc.)

```
$ cat /usr/share/gnome-session/sessions/qtile.session
[GNOME Session]
Name=Qtile session
```
Or for older Gnome versions

```
$ cat /usr/share/gnome-session/sessions/qtile.session
[GNOME Session]
Name=Qtile session
RequiredComponents=qtile;gnome-settings-daemon;
```

So that Qtile starts automatically on login.

```
$ cat /usr/share/applications/qtile.desktop
[Desktop Entry]
Type=Application
Encoding=UTF-8
Name=Qtile
Exec=qtile
NoDisplay=true
X-GNOME-WMName=Qtile
X-GNOME-Autostart-Phase=WindowManager
X-GNOME-Provides=windowmanager
X-GNOME-Autostart-Notify=false
```

The above does not start gnome-panel. Getting gnome-panel to work requires some extra Qtile configuration, mainly making the top and bottom panels static on panel startup and leaving a gap at the top (and bottom) for the panel window.

You might want to add keybindings to log out of the GNOME session.

```
Key([mod, 'control'], 'l', lazy.spawn('gnome-screensaver-command -l')),
Key([mod, 'control'], 'q', lazy.spawn('gnome-session-quit --logout --no-prompt')),
Key([mod, 'shift', 'control'], 'q', lazy.spawn('gnome-session-quit --power-off'))
```

The above apps need to be in your path (though they are typically installed in /usr/bin, so they probably are if they're installed at all).

## 1.3 Troubleshooting

### 1.3.1 So something has gone wrong... what do you do?

When Qtile is running, it logs error messages (and other messages) to its log file. This is found at ~/.local/share/qtile/qtile.log. This is the first place to check to see what is going on. If you are getting unexpected errors from normal usage or your configuration (and you’re not doing something wacky) and believe you have found a bug, then please report a bug.

If you are hacking on Qtile and you want to debug your changes, this log is your best friend. You can send messages to the log from within libqtile by using the logger:

```
from libqtile.log_utils import logger

logger.warning("Your message here")
logger.warning(variable_you_want_to_print)
```

(continues on next page)
# some changes here that might error
raise Exception as e:
    logger.exception(e)

logger.warning is convenient because its messages will always be visible in the log. logger.exception is helpful because it will print the full traceback of an error to the log. By sticking these amongst your changes you can look more closely at the effects of any changes you made to Qtile’s internals.

## 1.3.2 Capturing an xtrace

Occasionally, a bug will be low level enough to require an xtrace of Qtile’s conversations with the X server. To capture one of these, create an xinitrc or similar file with:

```
exec xtrace qtile >> ~/qtile.log
```

This will put the xtrace output in Qtile’s logfile as well. You can then demonstrate the bug, and paste the contents of this file into the bug report.

Note that xtrace may be named x11trace on some platforms, for example, on Fedora.

## 1.4 Shell commands

### 1.4.1 qshell

The Qtile command shell is a command-line shell interface that provides access to the full complement of Qtile command functions. The shell features command name completion, and full command documentation can be accessed from the shell itself. The shell uses GNU Readline when it’s available, so the interface can be configured to, for example, obey VI keybindings with an appropriate .inputrc file. See the GNU Readline documentation for more information.

### Navigating the Object Graph

The shell presents a filesystem-like interface to the object graph - the built-in “cd” and “ls” commands act like their familiar shell counterparts:

```
> ls
layout/ widget/ screen/ bar/ window/ group/
> cd bar
bar> ls
bottom/
bar> cd bottom
bar['bottom']> ls
screen/
bar['bottom']> cd ../..
> ls
layout/ widget/ screen/ bar/ window/ group/
```
Note that the shell provides a “short-hand” for specifying node keys (as opposed to children). The following is a valid shell path:

```
> cd group/4/window/31457314
```

The command prompt will, however, always display the Python node path that should be used in scripts and key bindings:

```
group['4'].window[31457314]>
```

**Live Documentation**

The shell `help` command provides the canonical documentation for the Qtile API:

```
> cd layout/1
layout[1]>
layout[1]>
layout[1]>
layout[1]>
help
help command  -- Help for a specific command.

Builtins
========
cd  exit  help  ls  q  quit

Commands for this object
========================
add  commands  current  delete  doc
down  get info  items  next  previous
rotate  shuffle_down  shuffle_up  toggle_split  up

layout[1]>
layout[1]>
layout[1]>
layout[1]>
help
help
previous
previous()
Focus previous stack.
```

**Reference**

**Qsh**

```python
class libqtile.sh.QSh (client: libqtile.command_interface.CommandInterface, completerkey='tab')
Qtile shell instance

    do_cd (arg) → str
    Change to another path.
```

**Examples**

```python
cd layout/0
cd ./layout

do_exit (args) → None
    Exit qshell

do_ls (arg: str) → str
    List contained items on a node.
```
**Examples**

```
> ls > ls ../layout
```

**do_pwd** *(arg) → str*

Returns the current working location

This is the same information as presented in the qshell prompt, but is very useful when running iqshell.

**Examples**

```
> pwd / > cd bar/top bar['top'] > pwd bar['top']
```

**do_help** *(arg) → str*

Give help on commands and builtins

When invoked without arguments, provides an overview of all commands. When passed as an argument, also provides a detailed help on a specific command or builtin.

**Examples**

```
> help
> help command
```

### 1.4.2 dqtile-cmd

A Rofi/dmenu interface to qtile-cmd. Accepts all arguments of qtile-cmd.

**Examples:**

**Output of dqtile-cmd -o cmd**

```
dmenu:  -
Alt-l  Prompt for args and show function help (if -f is present)
..     Go back to menu.
C-u    Clear input
Esc    Exit
-o cmd -f add_rule * Add a dgroup rule, returns rule id needed to remove it
-o cmd -f addgroup * Add a group with the given name
-o cmd -f commands Returns a list of possible commands for this object
-o cmd -f critical Set log level to CRITICAL
-o cmd -f debug Set log level to DEBUG
-o cmd -f delgroup * Delete a group with the given name
-o cmd -f display_kb * Display table of key bindings
-o cmd -f doc     * Returns the documentation for a specified command name
-o cmd -f error   Set log level to ERROR
-o cmd -f eval    * Evaluates code in the same context as this function
-o cmd -f findwindow * Launch prompt widget to find a window of the given name
-o cmd -f focus_by_click * Bring a window to the front
-o cmd -f function * Call a function with current object as argument
-o cmd -f get_info Prints info for all groups
-o cmd -f get_state Get pickled state for restarting qtile
```
Output of `dqtile-cmd -h`

`dqtile-cmd`

A Rofi/dmenu interface to qtile-cmd. Expects all arguments of qtile-cmd (see below).

**usage:** `dqtile-cmd [-h] [--object OBJ_SPEC [OBJ_SPEC ...]] [--function FUNCTION] [--args ARGS [ARGS ...]] [--info]

Simple tool to expose qtile.command functionality to shell.

**optional arguments:**
- `-h`, `--help` show this help message and exit
- `--object OBJ_SPEC [OBJ_SPEC ...]`, `-o OBJ_SPEC [OBJ_SPEC ...]`
  Specify path to object (space separated). If no --function flag display available commands.
- `--function FUNCTION`, `-f FUNCTION`
  Select function to execute.
- `--args ARGS [ARGS ...]`, `-a ARGS [ARGS ...]`
  Set arguments supplied to function.
- `--info`, `-i`
  With both --object and --function args prints documentation for function.

**Examples:**
- `dqtile-cmd`
- `dqtile-cmd -o cmd`
- `dqtile-cmd -o cmd -f prev_layout -i`
- `dqtile-cmd -o cmd -f prev_layout -a 3 # prev_layout on group 3`
- `dqtile-cmd -o group 3 -f focus_back`

If both rofi and dmenu are present rofi will be selected as default, to change this use `--force-dmenu` as the first argument.

### 1.4.3 iqshell

In addition to the standard `qshell` shell interface, we provide a kernel capable of running through Jupyter that hooks into the qshell client. The command structure and syntax is the same as qshell, so it is recommended you read that for more information about that.

#### Dependencies

In order to run iqshell, you must have `ipykernel` and `jupyter_console`. You can install the dependencies when you are installing qtile by running:

```
$ pip install qtile[ipython]
```

Otherwise, you can just install these two packages separately, either through PyPI or through your distribution package manager.
Installing and Running the Kernel

Once you have the required dependencies, you can run the kernel right away by running:

```
$ python3 -m libqtile.interactive.iqshell_kernel
```

However, this will merely spawn a kernel instance, you will have to run a separate frontend that connects to this kernel. A more convenient way to run the kernel is by registering the kernel with Jupyter. To register the kernel itself, run:

```
$ python3 -m libqtile.interactive.iqshell_install
```

If you run this as a non-root user, or pass the `--user` flag, this will install to the user Jupyter kernel directory. You can now invoke the kernel directly when starting a Jupyter frontend, for example:

```
$ jupyter console --kernel qshell
```

The `iqshell` script will launch a Jupyter terminal console with the qshell kernel.

**iqshell vs qshell**

One of the main drawbacks of running through a Jupyter kernel is the frontend has no way to query the current node of the kernel, and as such, there is no way to set a custom prompt. In order to query your current node, you can call `pwd`.

This, however, enables many of the benefits of running in a Jupyter frontend, including being able to save, run, and re-run code cells in frontends such as the Jupyter notebook.

The Jupyter kernel also enables more advanced help, text completion, and introspection capabilities (however, these are currently not implemented at a level much beyond what is available in the standard qshell).

### 1.4.4 qtile-cmd

This is a simple tool to expose qtile.command functionality to shell. This can be used standalone or in other shell scripts.

**Examples:**

**Output of qtile-cmd -h**

```
usage: qtile-cmd [-h] [--object OBJ_SPEC [OBJ_SPEC ...]]
                 [--function FUNCTION] [--args ARGS [ARGS ...]] [--info]

Simple tool to expose qtile.command functionality to shell.

optional arguments:
  -h, --help             show this help message and exit
  --object OBJ_SPEC [OBJ_SPEC ...], -o OBJ_SPEC [OBJ_SPEC ...]
                         Specify path to object (space separated). If no
                         --function flag display available commands.
  --function FUNCTION, -f FUNCTION
                         Select function to execute.
  --args ARGS [ARGS ...], -a ARGS [ARGS ...]
                         Set arguments supplied to function.
```

(continues on next page)
---

`--info, -i` With both `--object` and `--function` args prints documentation for function.

Examples:
```
qtile-cmd
qtile-cmd -o cmd
qtile-cmd -o cmd -f prev_layout -i
qtile-cmd -o cmd -f prev_layout -a 3 # prev_layout on group 3
qtile-cmd -o group 3 -f focus_back
```

**Output of `qtile-cmd -o group 3`**

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-o group 3 -f commands</code></td>
<td>Returns a list of possible commands for this object</td>
</tr>
<tr>
<td><code>-o group 3 -f doc</code></td>
<td>Returns the documentation for a specified command</td>
</tr>
<tr>
<td><code>-o group 3 -f name</code></td>
<td></td>
</tr>
<tr>
<td><code>-o group 3 -f eval</code></td>
<td>Evaluates code in the same context as this function</td>
</tr>
<tr>
<td><code>-o group 3 -f focus_back</code></td>
<td>Focus the window that had focus before the current one got it.</td>
</tr>
<tr>
<td><code>-o group 3 -f focus_by_name</code></td>
<td>Focus the first window with the given name. Do nothing if the name is nothing if the name is</td>
</tr>
<tr>
<td><code>-o group 3 -f function</code></td>
<td>Call a function with current object as argument</td>
</tr>
<tr>
<td><code>-o group 3 -f info</code></td>
<td>Returns a dictionary of info for this group</td>
</tr>
<tr>
<td><code>-o group 3 -f info_by_name</code></td>
<td>Returns the info for the first window with the given name without giving it</td>
</tr>
<tr>
<td><code>-o group 3 -f items</code></td>
<td>Returns a list of contained items for the specified name</td>
</tr>
<tr>
<td><code>-o group 3 -f name</code></td>
<td></td>
</tr>
<tr>
<td><code>-o group 3 -f next_window</code></td>
<td>Focus the next window in group.</td>
</tr>
<tr>
<td><code>-o group 3 -f prev_window</code></td>
<td>Focus the previous window in group.</td>
</tr>
<tr>
<td><code>-o group 3 -f set_label</code></td>
<td>Set the display name of current group to be used in <code>GroupBox</code> widget.</td>
</tr>
<tr>
<td><code>-o group 3 -f setlayout</code></td>
<td></td>
</tr>
<tr>
<td><code>-o group 3 -f switch_groups</code></td>
<td>Switch position of current group with name</td>
</tr>
<tr>
<td><code>-o group 3 -f toscreen</code></td>
<td>Pull a group to a specified screen.</td>
</tr>
<tr>
<td><code>-o group 3 -f unminimize_all</code></td>
<td>Unminimise all windows in this group</td>
</tr>
</tbody>
</table>

**Output of `qtile-cmd -o cmd`**

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-o cmd -f add_rule</code></td>
<td>Add a dgroup rule, returns rule_id needed to remove it</td>
</tr>
<tr>
<td><code>-o cmd -f addgroup</code></td>
<td>Add a group with the given name</td>
</tr>
<tr>
<td><code>-o cmd -f commands</code></td>
<td>Returns a list of possible commands for this object</td>
</tr>
<tr>
<td><code>-o cmd -f critical</code></td>
<td>Set log level to CRITICAL</td>
</tr>
<tr>
<td><code>-o cmd -f debug</code></td>
<td>Set log level to DEBUG</td>
</tr>
<tr>
<td><code>-o cmd -f delgroup</code></td>
<td>Delete a group with the given name</td>
</tr>
<tr>
<td><code>-o cmd -f display_kb</code></td>
<td>Display table of key bindings</td>
</tr>
<tr>
<td><code>-o cmd -f doc</code></td>
<td>Returns the documentation for a specified command</td>
</tr>
<tr>
<td><code>-o cmd -f error</code></td>
<td>Set log level to ERROR</td>
</tr>
<tr>
<td><code>-o cmd -f eval</code></td>
<td>Evaluates code in the same context as this function</td>
</tr>
<tr>
<td><code>-o cmd -f findwindow</code></td>
<td>Launch prompt widget to find a window of the given name</td>
</tr>
<tr>
<td><code>-o cmd -f focus_by_click</code></td>
<td>Bring a window to the front</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
- `o cmd -f function`  * Call a function with current object as argument
- `o cmd -f get_info`  Prints info for all groups
- `o cmd -f get_state`  Get pickled state for restarting qtile
- `o cmd -f get_test_data`  Returns any content arbitrarily set in the self.
- `o cmd -f groups`  Return a dictionary containing information for all groups
- `o cmd -f hide_show_bar`  * Toggle visibility of a given bar
- `o cmd -f info`  Set log level to INFO
- `o cmd -f internal_windows`  Return info for each internal window (bars, for example)
- `o cmd -f items`  * Returns a list of contained items for the specified name
- `o cmd -f list_widgets`  List of all addressable widget names
- `o cmd -f next_layout`  * Switch to the next layout.
- `o cmd -f next_screen`  Move to next screen
- `o cmd -f next_urgent`  Focus next window with urgent hint
- `o cmd -f pause`  Drops into pdb
- `o cmd -f prev_layout`  * Switch to the previous layout.
- `o cmd -f prev_screen`  Move to the previous screen
- `o cmd -f qtile_info`  Returns a dictionary of info on the Qtile instance
- `o cmd -f qtilecmd`  * Execute a Qtile command using the client syntax
- `o cmd -f remove_rule`  * Remove a dgroup rule by rule_id
- `o cmd -f restart`  Restart qtile
- `o cmd -f run_extension`  * Run extensions
- `o cmd -f run_extention`  * Deprecated alias for cmd_run_extension()
- `o cmd -f run_external`  Run external Python script
- `o cmd -f screens`  Return a list of dictionaries providing information on all screens
- `o cmd -f shutdown`  Quit Qtile
- `o cmd -f simulate_keypress`  * Simulates a keypress on the focused window.
- `o cmd -f spawn`  Run cmd in a shell.
- `o cmd -f spawncmd`  Spawn a command using a prompt widget, with tab-completion.
- `o cmd -f status`  Return "OK" if Qtile is running
- `o cmd -f switch_groups`  * Switch position of groupa to groupb
- `o cmd -f switchgroup`  Launch prompt widget to switch to a given group to the current screen
- `o cmd -f sync`  Sync the X display. Should only be used for development.
- `o cmd -f to_layout_index`  * Switch to the layout with the given index in self.layouts.
- `o cmd -f to_screen`  * Warp focus to screen n, where n is a 0-based screen number
- `o cmd -f togroup`  * Launch prompt widget to move current window to a given group
- `o cmd -f tracemalloc_dump`  Dump tracemalloc snapshot
- `o cmd -f tracemalloc_toggle`  Toggle tracemalloc status
- `o cmd -f warning`  Set log level to WARNING
- `o cmd -f windows`  Return info for each client window
1.4.5 qtile-run

Run a command applying rules to the new windows, ie, you can start a window in a specific group, make it floating, intrusive, etc.

The Windows must have NET_WM_PID.

```
# run xterm floating on group "test-group"
qtile-run -g test-group -f xterm
```

1.4.6 qtile-top

Is a top like to measure memory usage of Qtile’s internals.

- Built-in Extensions
- Built-in Hooks
- Built-in Layouts
- Built-in Widgets

1.5 Reference

1.5.1 Built-in Hooks

subscribe. **addgroup**(func)
Called when group is added

Arguments
- name of new group

subscribe. **changegroup**(func)
Called whenever a group change occurs

Arguments
- None

subscribe. **client_focus**(func)
Called whenever focus changes

Arguments
- window.Window object of the new focus.

subscribe. **client_killed**(func)
Called after a client has been unmanaged

Arguments
- window.Window object of the killed window.

subscribe. **client_managed**(func)
Called after Qtile starts managing a new client

Called after a window is assigned to a group, or when a window is made static. This hook is not called for internal windows.

Arguments
- window.Window object of the managed window

subscribe.client_mouse_enter(func)
Called when the mouse enters a client

Arguments
- window.Window of window entered

subscribe.client_name_updated(func)
Called when the client name changes

Arguments
- window.Window of client with updated name

subscribe.client_new(func)
Called before Qtile starts managing a new client

Use this hook to declare windows static, or add them to a group on startup. This hook is not called for internal windows.

Arguments
- window.Window object

Examples

```python
@libqtile.hook.subscribe.client_new
def func(c):
    if c.name == "xterm":
        c.togroup("a")
    elif c.name == "dzen":
        c.cmd_static(0)
```

subscribe.client_urgent_hint_changed(func)
Called when the client urgent hint changes

Arguments
- window.Window of client with hint change

subscribe.current_screen_change(func)
Called when the current screen (i.e. the screen with focus) changes

Arguments
None

subscribe.delgroup(func)
Called when group is deleted

Arguments
- name of deleted group

subscribe.enter_chord(func)
Called when key chord begins

Arguments
- name of chord(mode)
subscribe.float_change(func)
   Called when a change in float state is made

   Arguments
   None

subscribe.focus_change(func)
   Called when focus is changed

   Arguments
   None

subscribe.group_window_add(func)
   Called when a new window is added to a group

   Arguments
   None

subscribe.layout_change(func)
   Called on layout change

   Arguments
   • layout object for new layout
   • group object on which layout is changed

subscribe.leave_chord(func)
   Called when key chord ends

   Arguments
   None

subscribe.net_wm_icon_change(func)
   Called on _NET_WM_ICON change

   Arguments
   • window.Window of client with changed icon

subscribe.restart(func)
   Called before qtile is restarted

   Arguments
   None

subscribe.screen_change(func)
   Called when a screen is added or screen configuration is changed (via xrandr)

   Common usage is simply to call qtile.cmd_restart() on each event (to restart qtile when there is a new monitor): 

   Arguments
   • xproto.randr.ScreenChangeNotify event
Examples

```python
@libqtile.hook.subscribe.screen_change
def restart_on_randr(ev):
    libqtile.qtile.cmd_restart()
```

`subscribe.selection_change(func)`
Called on selection change

**Arguments**

- name of the selection
- dictionary describing selection, containing `owner` and `selection` as keys

`subscribe.selection_notify(func)`
Called on selection notify

**Arguments**

- name of the selection
- dictionary describing selection, containing `owner` and `selection` as keys

`subscribe.setgroup(func)`
Called when group is changed

**Arguments**

None

`subscribe.shutdown(func)`
Called before qtile is shutdown

**Arguments**

None

`subscribe.startup(func)`
Called when qtile is started

**Arguments**

None

`subscribe.startup_complete(func)`
Called when qtile is started after all resources initialized

**Arguments**

None

`subscribe.startup_once(func)`
Called when Qtile has started on first start

This hook is called exactly once per session (i.e. not on each `lazy.restart()`).

**Arguments**

None

`subscribe.window_name_change(func)`
Called whenever a windows name changes

Deprecated: use `client_name_updated` `Arguments`

None
1.5.2 Built-in Layouts

Floating

class libqtile.layout.floating.Floating (float_rules=None, no_reposition_rules=None, **config)

Floating layout, which does nothing with windows but handles focus order

<table>
<thead>
<tr>
<th>key</th>
<th>default</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>border_focus</td>
<td>'#0000ff'</td>
<td>‘Border colour for the focused window.’</td>
</tr>
<tr>
<td>border_normal</td>
<td>'#000000'</td>
<td>‘Border colour for un-focused windows.’</td>
</tr>
<tr>
<td>border_width</td>
<td>1</td>
<td>‘Border width.’</td>
</tr>
<tr>
<td>fullscreen_border_width</td>
<td>0</td>
<td>‘Border width for fullscreen.’</td>
</tr>
<tr>
<td>max_border_width</td>
<td>0</td>
<td>‘Border width for maximize.’</td>
</tr>
<tr>
<td>name</td>
<td>'floating'</td>
<td>‘Name of this layout.’</td>
</tr>
</tbody>
</table>

Bsp

class libqtile.layout.bsp.Bsp(**config)

This layout is inspired by bspwm, but it does not try to copy its features.

The first client occupies the entire screen space. When a new client is created, the selected space is partitioned in 2 and the new client occupies one of those subspaces, leaving the old client with the other.

The partition can be either horizontal or vertical according to the dimensions of the current space: if its width/height ratio is above a pre-configured value, the subspaces are created side-by-side, otherwise, they are created on top of each other. The partition direction can be freely toggled. All subspaces can be resized and clients can be shuffled around.

All clients are organized at the leaves of a full binary tree.

An example key configuration is:

```python
Key([mod], "j", lazy.layout.down()),
Key([mod], "k", lazy.layout.up()),
Key([mod], "h", lazy.layout.left()),
Key([mod], "l", lazy.layout.right()),
Key([mod], "shift", "j", lazy.layout.shuffle_down()),
Key([mod], "shift", "k", lazy.layout.shuffle_up()),
Key([mod], "shift", "h", lazy.layout.shuffle_left()),
Key([mod], "shift", "l", lazy.layout.shuffle_right()),
Key([mod], "mod1", "j", lazy.layout.flip_down()),
Key([mod], "mod1", "k", lazy.layout.flip_up()),
Key([mod], "mod1", "h", lazy.layout.flip_left()),
Key([mod], "mod1", "l", lazy.layout.flip_right()),
Key([mod], "control", "j", lazy.layout.grow_down()),
Key([mod], "control", "k", lazy.layout.grow_up()),
Key([mod], "control", "h", lazy.layout.grow_left()),
Key([mod], "control", "l", lazy.layout.grow_right()),
Key([mod], "shift", "n", lazy.layout.normalize()),
Key([mod], "Return", lazy.layout.toggle_split()),
```
### Columns

**class** `libqtile.layout.columns.Columns(**config)`

Extension of the Stack layout.

The screen is split into columns, which can be dynamically added or removed. Each column can present its windows in 2 modes: split or stacked. In split mode, all windows are presented simultaneously, splitting the column space. In stacked mode, only a single window is presented from the stack of windows. Columns and windows can be resized and windows can be shuffled around.

This layout can also emulate wmii’s default layout via:

```
layout.Columns(num_columns=1, insert_position=1)
```

Or the “Vertical”, and “Max”, depending on the default parameters.

An example key configuration is:

```python
Key([mod], "j", lazy.layout.down()),
Key([mod], "k", lazy.layout.up()),
Key([mod], "h", lazy.layout.left()),
Key([mod], "l", lazy.layout.right()),
Key([mod], "shift", "j", lazy.layout.shuffle_down()),
Key([mod], "shift", "k", lazy.layout.shuffle_up()),
Key([mod], "shift", "h", lazy.layout.shuffle_left()),
Key([mod], "shift", "l", lazy.layout.shuffle_right()),
Key([mod], "control", "j", lazy.layout.grow_down()),
Key([mod], "control", "k", lazy.layout.grow_up()),
Key([mod], "control", "h", lazy.layout.grow_left()),
Key([mod], "control", "l", lazy.layout.grow_right()),
Key([mod], "Return", lazy.layout.toggle_split()),
Key([mod], "n", lazy.layout.normalize()),
```

---

<table>
<thead>
<tr>
<th><strong>key</strong></th>
<th><strong>default</strong></th>
<th><strong>description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>border_focus</td>
<td>'#881111'</td>
<td>‘Border colour for the focused window.’</td>
</tr>
<tr>
<td>border_normal</td>
<td>'#220000'</td>
<td>‘Border colour for un-focused windows.’</td>
</tr>
<tr>
<td>border_width</td>
<td>2</td>
<td>‘Border width.’</td>
</tr>
<tr>
<td>fair</td>
<td>True</td>
<td>‘New clients are inserted in the shortest branch.’</td>
</tr>
<tr>
<td>grow_amount</td>
<td>10</td>
<td>‘Amount by which to grow a window/column.’</td>
</tr>
<tr>
<td>lower_right</td>
<td>True</td>
<td>‘New client occupies lower or right subspace.’</td>
</tr>
<tr>
<td>margin</td>
<td>0</td>
<td>‘Margin of the layout.’</td>
</tr>
<tr>
<td>name</td>
<td>'bsp'</td>
<td>‘Name of this layout.’</td>
</tr>
<tr>
<td>ratio</td>
<td>1.6</td>
<td>‘Width/height ratio that defines the partition direction.’</td>
</tr>
</tbody>
</table>
### Matrix

`class libqtile.layout.matrix.Matrix(columns=2, **config)`

This layout divides the screen into a matrix of equally sized cells and places one window in each cell. The number of columns is configurable and can also be changed interactively.

<table>
<thead>
<tr>
<th>key</th>
<th>default</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>border_focus</td>
<td>'#881111'</td>
<td>'Border colour for the focused window.'</td>
</tr>
<tr>
<td>border_normal</td>
<td>'#220000'</td>
<td>'Border colour for un-focused windows.'</td>
</tr>
<tr>
<td>border_width</td>
<td>2</td>
<td>'Border width.'</td>
</tr>
<tr>
<td>fair</td>
<td>False</td>
<td>'Add new windows to the column with least windows.'</td>
</tr>
<tr>
<td>grow_amount</td>
<td>10</td>
<td>'Amount by which to grow a window/column.'</td>
</tr>
<tr>
<td>insert_position</td>
<td>0</td>
<td>'Position relative to the current window where new ones are inserted (0 means right above the current window, 1 means right after).'</td>
</tr>
<tr>
<td>margin</td>
<td>0</td>
<td>'Margin of the layout.'</td>
</tr>
<tr>
<td>name</td>
<td>'columns'</td>
<td>'Name of this layout.'</td>
</tr>
<tr>
<td>num_columns</td>
<td>2</td>
<td>'Preferred number of columns.'</td>
</tr>
<tr>
<td>split</td>
<td>True</td>
<td>'New columns presentation mode.'</td>
</tr>
<tr>
<td>wrap_focus_columns</td>
<td>True</td>
<td>'Wrap the screen when moving focus across columns.'</td>
</tr>
<tr>
<td>wrap_focus_rows</td>
<td>True</td>
<td>'Wrap the screen when moving focus across rows.'</td>
</tr>
<tr>
<td>wrap_focus_stacks</td>
<td>True</td>
<td>'Wrap the screen when moving focus across stacked.'</td>
</tr>
</tbody>
</table>

### Max

`class libqtile.layout.max.Max(**config)`

Maximized layout

A simple layout that only displays one window at a time, filling the `screen_rect`. This is suitable for use on laptops and other devices with small screens. Conceptually, the windows are managed as a stack, with commands to switch to next and previous windows in the stack.

<table>
<thead>
<tr>
<th>key</th>
<th>default</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>'max'</td>
<td>'Name of this layout.'</td>
</tr>
</tbody>
</table>
MonadTall

class libqtile.layout.xmonad.MonadTall(**config)

Emulate the behavior of XMonad's default tiling scheme.

Main-Pane:

A main pane that contains a single window takes up a vertical portion of the screen_rect based on the ratio setting. This ratio can be adjusted with the `cmd_grow_main` and `cmd_shrink_main` or, while the main pane is in focus, `cmd_grow` and `cmd_shrink`.

Using the `cmd_flip` method will switch which horizontal side the main pane will occupy. The main pane is considered the “top” of the stack.

Secondary-panes:

Occupying the rest of the screen_rect are one or more secondary panes. The secondary panes will share the vertical space of the screen_rect however they can be resized at will with the `cmd_grow` and `cmd_shrink` methods. The other secondary panes will adjust their sizes to smoothly fill all of the space.

Panes can be moved with the `cmd_shuffle_up` and `cmd_shuffle_down` methods. As mentioned the main pane is considered the top of the stack; moving up is counter-clockwise and moving down is clockwise. The opposite is true if the layout is “flipped”.

(continues on next page)
Normalizing/Resetting:

To restore all secondary client windows to their default size ratios use the `cmd_normalize` method.

To reset all client windows to their default sizes, including the primary window, use the `cmd_reset` method.

Maximizing:

To toggle a client window between its minimum and maximum sizes simply use the `cmd_maximize` on a focused client.

Suggested Bindings:

```python
Key([modkey], "h", lazy.layout.left()),
Key([modkey], "l", lazy.layout.right()),
Key([modkey], "j", lazy.layout.down()),
Key([modkey], "k", lazy.layout.up()),
Key([modkey, "shift"], "h", lazy.layout.swap_left()),
Key([modkey, "shift"], "l", lazy.layout.swap_right()),
Key([modkey, "shift"], "j", lazy.layout.shuffle_down()),
Key([modkey, "shift"], "k", lazy.layout.shuffle_up()),
Key([modkey], "i", lazy.layout.grow()),
Key([modkey], "m", lazy.layout.shrink()),
Key([modkey], "n", lazy.layout.normalize()),
Key([modkey], "o", lazy.layout.maximize()),
Key([modkey, "shift"], "space", lazy.layout.flip()),
```

<table>
<thead>
<tr>
<th>key</th>
<th>default</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>align</td>
<td>0</td>
<td>‘Which side master plane will be placed (one of MonadTall._left or MonadTall._right)’</td>
</tr>
<tr>
<td>border_focus</td>
<td>‘#ff0000’</td>
<td>‘Border colour for the focused window.’</td>
</tr>
<tr>
<td>border_normal</td>
<td>‘#000000’</td>
<td>‘Border colour for un-focused windows.’</td>
</tr>
<tr>
<td>border_width</td>
<td>2</td>
<td>‘Border width.’</td>
</tr>
<tr>
<td>change_ratio</td>
<td>0.05</td>
<td>‘Resize ratio’</td>
</tr>
<tr>
<td>change_size</td>
<td>20</td>
<td>‘Resize change in pixels’</td>
</tr>
<tr>
<td>margin</td>
<td>0</td>
<td>‘Margin of the layout’</td>
</tr>
<tr>
<td>max_ratio</td>
<td>0.75</td>
<td>‘The percent of the screen-space the master pane should occupy at maximum.’</td>
</tr>
<tr>
<td>min_ratio</td>
<td>0.25</td>
<td>‘The percent of the screen-space the master pane should occupy at minimum.’</td>
</tr>
<tr>
<td>min_secondary_size</td>
<td>85</td>
<td>‘minimum size in pixel for a secondary pane window’</td>
</tr>
<tr>
<td>name</td>
<td>‘xmonadtall’</td>
<td>‘Name of this layout.’</td>
</tr>
<tr>
<td>new_at_current</td>
<td>False</td>
<td>‘Place new windows at the position of the active window.’</td>
</tr>
<tr>
<td>ratio</td>
<td>0.5</td>
<td>‘The percent of the screen-space the master pane should occupy by default.’</td>
</tr>
<tr>
<td>single_border_width</td>
<td></td>
<td>‘Border width for single window’</td>
</tr>
<tr>
<td>single_margin</td>
<td>None</td>
<td>‘Margin size for single window’</td>
</tr>
</tbody>
</table>
MonadWide

class libqtile.layout.xmonadMonadWide(**config)

Emulate the behavior of XMonad’s horizontal tiling scheme.

This layout attempts to emulate the behavior of XMonad wide tiling scheme.

Main-Pane:

A main pane that contains a single window takes up a horizontal portion of the screen_rect based on the ratio setting. This ratio can be adjusted with the cmd_grow_main and cmd_shrink_main or, while the main pane is in focus, cmd_grow and cmd_shrink.

Using the cmd_flip method will switch which vertical side the main pane will occupy. The main pane is considered the “top” of the stack.

Secondary-panes:

Occupying the rest of the screen_rect are one or more secondary panes. The secondary panes will share the horizontal space of the screen_rect however they can be resized at will with the cmd_grow and cmd_shrink methods. The other secondary panes will adjust their sizes to smoothly fill all of the space.

Panes can be moved with the cmd_shuffle_up and cmd_shuffle_down methods. As mentioned the main pane is considered the top of the stack; moving up is counter-clockwise and moving down is clockwise. The opposite is true if the layout is “flipped”.

(continues on next page)
Normalizing/Resetting:
To restore all secondary client windows to their default size ratios use the `cmd_normalize` method.
To reset all client windows to their default sizes, including the primary window, use the `cmd_reset` method.

Maximizing:
To toggle a client window between its minimum and maximum sizes simply use the `cmd_maximize` on a focused client.

Suggested Bindings:

```python
Key([modkey], "h", lazy.layout.left()),
Key([modkey], "l", lazy.layout.right()),
Key([modkey], "j", lazy.layout.down()),
Key([modkey], "k", lazy.layout.up()),
Key([modkey, "shift"], "h", lazy.layout.swap_left()),
Key([modkey, "shift"], "l", lazy.layout.swap_right()),
Key([modkey, "shift"], "j", lazy.layout.shuffle_down()),
Key([modkey, "shift"], "k", lazy.layout.shuffle_up()),
Key([modkey], "i", lazy.layout.grow()),
Key([modkey], "m", lazy.layout.shrink()),
Key([modkey], "n", lazy.layout.normalize()),
Key([modkey], "o", lazy.layout.maximize()),
Key([modkey, "shift"], "space", lazy.layout.flip()),
```

<table>
<thead>
<tr>
<th>key</th>
<th>default</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>align</td>
<td>0</td>
<td>‘Which side master plane will be placed (one of MonadTall._left or MonadTall._right)’</td>
</tr>
<tr>
<td>border_focus</td>
<td>'#ff0000'</td>
<td>‘Border colour for the focused window.’</td>
</tr>
<tr>
<td>border_normal</td>
<td>'#000000'</td>
<td>‘Border colour for un-focused windows.’</td>
</tr>
<tr>
<td>border_width</td>
<td>2</td>
<td>‘Border width.’</td>
</tr>
<tr>
<td>change_ratio</td>
<td>0.05</td>
<td>‘Resize ratio’</td>
</tr>
<tr>
<td>change_size</td>
<td>20</td>
<td>‘Resize change in pixels’</td>
</tr>
<tr>
<td>margin</td>
<td>0</td>
<td>‘Margin of the layout’</td>
</tr>
<tr>
<td>max_ratio</td>
<td>0.75</td>
<td>‘The percent of the screen-space the master pane should occupy at maximum.’</td>
</tr>
<tr>
<td>min_ratio</td>
<td>0.25</td>
<td>‘The percent of the screen-space the master pane should occupy at minimum.’</td>
</tr>
<tr>
<td>min_secondary_size</td>
<td>85</td>
<td>‘minimum size in pixel for a secondary pane window’</td>
</tr>
<tr>
<td>name</td>
<td>'xmonadtall'</td>
<td>‘Name of this layout.’</td>
</tr>
<tr>
<td>new_at_current</td>
<td>False</td>
<td>‘Place new windows at the position of the active window.’</td>
</tr>
<tr>
<td>ratio</td>
<td>0.5</td>
<td>‘The percent of the screen-space the master pane should occupy by default.’</td>
</tr>
<tr>
<td>single_border_width</td>
<td>0.75</td>
<td>‘Border width for single window’</td>
</tr>
<tr>
<td>single_margin</td>
<td>None</td>
<td>‘Margin size for single window’</td>
</tr>
</tbody>
</table>
**RatioTile**

```python
class libqtile.layout.ratioTile.RatioTile(**config)
```

Tries to tile all windows in the width/height ratio passed in.

<table>
<thead>
<tr>
<th>Key</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>border_focus</td>
<td>'#0000ff'</td>
<td>'Border colour for the focused window.'</td>
</tr>
<tr>
<td>border_normal</td>
<td>'#000000'</td>
<td>'Border colour for un-focused windows.'</td>
</tr>
<tr>
<td>border_width</td>
<td>1</td>
<td>'Border width.'</td>
</tr>
<tr>
<td>fancy</td>
<td>False</td>
<td>'Use a different method to calculate window sizes.'</td>
</tr>
<tr>
<td>margin</td>
<td>0</td>
<td>'Margin of the layout'</td>
</tr>
<tr>
<td>name</td>
<td>'ratiotile'</td>
<td>'Name of this layout.'</td>
</tr>
<tr>
<td>ratio</td>
<td>1.618</td>
<td>'Ratio of the tiles'</td>
</tr>
<tr>
<td>ratio_increment</td>
<td>0.1</td>
<td>'Amount to increment per ratio increment'</td>
</tr>
</tbody>
</table>

**Slice**

```python
class libqtile.layout.slice.Slice(**config)
```

Slice layout

This layout cuts piece of screen_rect and places a single window on that piece, and delegates other window placement to other layout.

<table>
<thead>
<tr>
<th>Key</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fallback</td>
<td>&lt;libqtile.layout.Max object at 0x7feb673002d0&gt;</td>
<td>'Layout to be used for the non-slice area.'</td>
</tr>
<tr>
<td>match</td>
<td>None</td>
<td>'Match-object describing which window(s) to move to the slice.'</td>
</tr>
<tr>
<td>name</td>
<td>'slice'</td>
<td>'Name of this layout.'</td>
</tr>
<tr>
<td>side</td>
<td>'left'</td>
<td>'Position of the slice (left, right, top, bottom).'</td>
</tr>
<tr>
<td>width</td>
<td>256</td>
<td>'Slice width.'</td>
</tr>
</tbody>
</table>

**Stack**

```python
class libqtile.layout.stack.Stack(**config)
```

A layout composed of stacks of windows

The stack layout divides the screen_rect horizontally into a set of stacks. Commands allow you to switch between stacks, to next and previous windows within a stack, and to split a stack to show all windows in the stack, or unsplit it to show only the current window.

Unlike the columns layout the number of stacks is fixed.
### Tile

**class** `libqtile.layout.tile.Tile(**config)`

A layout with two stacks of windows dividing the screen

The Tile layout divides the screen_rect horizontally into two stacks. The maximum amount of “master” windows can be configured; surplus windows will be displayed in the slave stack on the right. Within their stacks, the windows will be tiled vertically. The windows can be rotated in their entirety by calling `up()` or `down()` or, if `shift_windows` is set to `True`, individually.

<table>
<thead>
<tr>
<th>key</th>
<th>default</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>autosplit</td>
<td>False</td>
<td>‘Auto split all new stacks.’</td>
</tr>
<tr>
<td>border_focus</td>
<td>'#0000ff'</td>
<td>‘Border colour for the focused window.’</td>
</tr>
<tr>
<td>border_normal</td>
<td>'#000000'</td>
<td>‘Border colour for un-focused windows.’</td>
</tr>
<tr>
<td>border_width</td>
<td>1</td>
<td>‘Border width.’</td>
</tr>
<tr>
<td>fair</td>
<td>False</td>
<td>‘Add new windows to the stacks in a round robin way.’</td>
</tr>
<tr>
<td>margin</td>
<td>0</td>
<td>‘Margin of the layout’</td>
</tr>
<tr>
<td>name</td>
<td>'stack'</td>
<td>‘Name of this layout.’</td>
</tr>
<tr>
<td>num_stacks</td>
<td>2</td>
<td>‘Number of stacks.’</td>
</tr>
</tbody>
</table>

### Key

<table>
<thead>
<tr>
<th>key</th>
<th>default</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>add_after_last</td>
<td>False</td>
<td>‘Add new clients after all the others. If this is True, it overrides add_on_top.’</td>
</tr>
<tr>
<td>add_on_top</td>
<td>True</td>
<td>‘Add new clients before all the others, potentially pushing other windows into slave stack.’</td>
</tr>
<tr>
<td>border_focus</td>
<td>'#0000ff'</td>
<td>‘Border colour for the focused window.’</td>
</tr>
<tr>
<td>border_normal</td>
<td>'#000000'</td>
<td>‘Border colour for un-focused windows.’</td>
</tr>
<tr>
<td>border_width</td>
<td>1</td>
<td>‘Border width.’</td>
</tr>
<tr>
<td>expand</td>
<td>True</td>
<td>‘Expand the master windows to the full screen width if no slaves are present.’</td>
</tr>
<tr>
<td>margin</td>
<td>0</td>
<td>‘Margin of the layout’</td>
</tr>
<tr>
<td>master_length</td>
<td>1</td>
<td>‘Amount of windows displayed in the master stack. Surplus windows will be moved to the slave stack.’</td>
</tr>
<tr>
<td>master_match</td>
<td>None</td>
<td>‘A Match object defining which window(s) should be kept masters.’</td>
</tr>
<tr>
<td>name</td>
<td>'tile'</td>
<td>‘Name of this layout.’</td>
</tr>
<tr>
<td>ratio</td>
<td>0.618</td>
<td>‘Width-percentage of screen size reserved for master windows.’</td>
</tr>
<tr>
<td>ratio_increment</td>
<td>0.05</td>
<td>‘By which amount to change ratio when cmd_decrease_ratio or cmd_increase_ratio are called.’</td>
</tr>
<tr>
<td>shift_windows</td>
<td>False</td>
<td>‘Allow to shift windows within the layout. If False, the layout will be rotated instead.’</td>
</tr>
</tbody>
</table>
TreeTab

```python
class libqtile.layout.tree.TreeTab(**config)
```

Tree Tab Layout

This layout works just like Max but displays tree of the windows at the left border of the screen_rect, which allows you to overview all opened windows. It's designed to work with `uzbl-browser` but works with other windows too.

The panel at the left border contains sections, each of which contains windows. Initially the panel looks like flat lists inside its section, and looks like trees if some of the windows are “moved” left or right.

For example, it looks like below with two sections initially:

```
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Section Foo</td>
</tr>
<tr>
<td>-------------------</td>
</tr>
<tr>
<td>Window A</td>
</tr>
<tr>
<td>-------------------</td>
</tr>
<tr>
<td>Window B</td>
</tr>
<tr>
<td>-------------------</td>
</tr>
<tr>
<td>Window C</td>
</tr>
<tr>
<td>-------------------</td>
</tr>
<tr>
<td>Section Bar</td>
</tr>
<tr>
<td>-------------------</td>
</tr>
</tbody>
</table>
```

And then it will look like below if “Window B” is moved right and “Window C” is moved right too:

```
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Section Foo</td>
</tr>
<tr>
<td>-------------------</td>
</tr>
<tr>
<td>Window A</td>
</tr>
<tr>
<td>-------------------</td>
</tr>
<tr>
<td>Window B</td>
</tr>
<tr>
<td>-------------------</td>
</tr>
<tr>
<td>Window C</td>
</tr>
<tr>
<td>-------------------</td>
</tr>
<tr>
<td>Section Bar</td>
</tr>
<tr>
<td>-------------------</td>
</tr>
</tbody>
</table>
```
<table>
<thead>
<tr>
<th>key</th>
<th>default</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>active_bg</td>
<td>'000080'</td>
<td>'Background color of active tab'</td>
</tr>
<tr>
<td>active_fg</td>
<td>'ffffff'</td>
<td>'Foreground color of active tab'</td>
</tr>
<tr>
<td>bg_color</td>
<td>'000000'</td>
<td>'Background color of tabs'</td>
</tr>
<tr>
<td>border_width</td>
<td>2</td>
<td>'Width of the border'</td>
</tr>
<tr>
<td>font</td>
<td>'sans'</td>
<td>'Font'</td>
</tr>
<tr>
<td>fontshadow</td>
<td>None</td>
<td>'Font shadow color, default is None (no shadow)'</td>
</tr>
<tr>
<td>fontsize</td>
<td>14</td>
<td>'Font pixel size.'</td>
</tr>
<tr>
<td>inactive_bg</td>
<td>'606060'</td>
<td>'Background color of inactive tab'</td>
</tr>
<tr>
<td>inactive_fg</td>
<td>'ffffff'</td>
<td>'Foreground color of inactive tab'</td>
</tr>
<tr>
<td>level_shift</td>
<td>8</td>
<td>'Shift for children tabs'</td>
</tr>
<tr>
<td>margin_left</td>
<td>6</td>
<td>'Left margin of tab panel'</td>
</tr>
<tr>
<td>margin_y</td>
<td>6</td>
<td>'Vertical margin of tab panel'</td>
</tr>
<tr>
<td>name</td>
<td>'treetab'</td>
<td>'Name of this layout.'</td>
</tr>
<tr>
<td>padding_left</td>
<td>6</td>
<td>'Left padding for tabs'</td>
</tr>
<tr>
<td>padding_x</td>
<td>6</td>
<td>'Left padding for tab label'</td>
</tr>
<tr>
<td>padding_y</td>
<td>2</td>
<td>'Top padding for tab label'</td>
</tr>
<tr>
<td>panel_width</td>
<td>150</td>
<td>'Width of the left panel'</td>
</tr>
<tr>
<td>previous_on_rm</td>
<td>False</td>
<td>'Focus previous window on close instead of first.'</td>
</tr>
<tr>
<td>section_bottom</td>
<td>6</td>
<td>'Bottom margin of section'</td>
</tr>
<tr>
<td>section_fg</td>
<td>'ffffff'</td>
<td>'Color of section label'</td>
</tr>
<tr>
<td>section_fontsize</td>
<td>11</td>
<td>'Font pixel size of section label'</td>
</tr>
<tr>
<td>section_left</td>
<td>4</td>
<td>'Left margin of section label'</td>
</tr>
<tr>
<td>section_padding</td>
<td>4</td>
<td>'Bottom of margin section label'</td>
</tr>
<tr>
<td>section_top</td>
<td>4</td>
<td>'Top margin of section label'</td>
</tr>
<tr>
<td>sections</td>
<td>['Default']</td>
<td>'Foreground color of inactive tab'</td>
</tr>
<tr>
<td>urgent_bg</td>
<td>'ff0000'</td>
<td>'Background color of urgent tab'</td>
</tr>
<tr>
<td>urgent_fg</td>
<td>'ffffff'</td>
<td>'Foreground color of urgent tab'</td>
</tr>
<tr>
<td>vspace</td>
<td>2</td>
<td>'Space between tabs'</td>
</tr>
</tbody>
</table>

**VerticalTile**

```python
class libqtile.layout.verticaltile.VerticalTile(**config)
```

Tiling layout that works nice on vertically mounted monitors

The available height gets divided by the number of panes, if no pane is maximized. If one pane has been maximized, the available height gets split in master- and secondary area. The maximized pane (master pane) gets the full height of the master area and the other panes (secondary panes) share the remaining space. The master area (at default 75%) can grow and shrink via keybindings.

```
-------------    --------------    ---
 |         |          | ---
<table>
<thead>
<tr>
<th>1</th>
<th>&lt;-- Panes</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>&lt;--------</td>
<td>1</td>
</tr>
<tr>
<td>-----------</td>
<td>----------</td>
<td>---</td>
</tr>
<tr>
<td>3</td>
<td>&lt;--------</td>
<td></td>
</tr>
</tbody>
</table>
```
Normal behavior. No One maximized pane in the master area maximized pane. No and two secondary panes in the specific areas. secondary area.

In some cases VerticalTile can be useful on horizontal mounted monitors two. For example if you want to have a webbrowser and a shell below it.

Suggested keybindings:

```python
Key([modkey], 'j', lazy.layout.down()),
Key([modkey], 'k', lazy.layout.up()),
Key([modkey], 'Tab', lazy.layout.next()),
Key([modkey, 'shift'], 'Tab', lazy.layout.next()),
Key([modkey, 'shift'], 'j', lazy.layout.shuffle_down()),
Key([modkey, 'shift'], 'k', lazy.layout.shuffle_up()),
Key([modkey], 'm', lazy.layout.maximize()),
Key([modkey], 'n', lazy.layout.normalize()),
```

<table>
<thead>
<tr>
<th>key</th>
<th>default</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>border_focus</td>
<td>'#FF0000'</td>
<td>'Border color for the focused window.'</td>
</tr>
<tr>
<td>border_normal</td>
<td>'#FPFFFF'</td>
<td>'Border color for un-focused windows.'</td>
</tr>
<tr>
<td>border_width</td>
<td>1</td>
<td>'Border width.'</td>
</tr>
<tr>
<td>margin</td>
<td>0</td>
<td>'Border margin.'</td>
</tr>
<tr>
<td>name</td>
<td>'verticaltile'</td>
<td>'Name of this layout.'</td>
</tr>
</tbody>
</table>

**Zoomy**

class libqtile.layout.zoomy.Zoomy(**config**)

A layout with single active windows, and few other previews at the right

<table>
<thead>
<tr>
<th>key</th>
<th>default</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>columnwidth</td>
<td>150</td>
<td>'Width of the right column'</td>
</tr>
<tr>
<td>margin</td>
<td>0</td>
<td>'Margin of the layout'</td>
</tr>
<tr>
<td>name</td>
<td>'zoomy'</td>
<td>'Name of this layout.'</td>
</tr>
<tr>
<td>property_big</td>
<td>'1.0'</td>
<td>'Property value to set on normal window'</td>
</tr>
<tr>
<td>property_name</td>
<td>'ZOOM'</td>
<td>'Property to set on zoomed window'</td>
</tr>
<tr>
<td>property_small</td>
<td>'0.1'</td>
<td>'Property value to set on zoomed window'</td>
</tr>
</tbody>
</table>
1.5.3 Built-in Widgets

1.5.4 Built-in Extensions
2.1 Scripting

2.1.1 Client-Server Scripting Model

Qtile has a client-server control model - the main Qtile instance listens on a named pipe, over which marshalled command calls and response data is passed. This allows Qtile to be controlled fully from external scripts. Remote interaction occurs through an instance of the `libqtile.command_interface.IPCCommandInterface` class. This class establishes a connection to the currently running instance of Qtile. A `libqtile.command_client.CommandClient` can use this connection to dispatch commands to the running instance. Commands then appear as methods with the appropriate signature on the `CommandClient` object. The object hierarchy is described in the `Commands API` section of this manual. Full command documentation is available through the `Qtile Shell`.

2.1.2 Example

Below is a very minimal example script that inspects the current Qtile instance, and returns the integer offset of the current screen.

```python
from libqtile.command_client import CommandClient
c = CommandClient()
print(c.screen.info()['_index'])
```

2.2 Commands API

Qtile’s command API is based on a graph of objects, where each object has a set of associated commands. The graph and object commands are used in a number of different places:

- Commands can be `bound to keys` in the Qtile configuration file.
- Commands can be `called through qshell`, the Qtile shell.
- The qsh can also be hooked into a Jupyter kernel called `iqshell`.
- Commands can be `called from a script` to interact with Qtile from Python.

If the explanation below seems a bit complex, please take a moment to explore the API using the `qshell` command shell. Command lists and detailed documentation can be accessed from its built-in help command.
2.2.1 Introduction: Object Graph

The objects in Qtile’s object graph come in seven flavours, matching the seven basic components of the window manager: layouts, windows, groups, bars, widgets, screens, and a special root node. Objects are addressed by a path specification that starts at the root, and follows the edges of the graph. This is what the graph looks like:

Each arrow can be read as “holds a reference to”. So, we can see that a widget object holds a reference to objects of type bar, screen and group. Let’s start with some simple examples of how the addressing works. Which particular objects we hold reference to depends on the context - for instance, widgets hold a reference to the screen that they appear on, and the bar they are attached to.

Let’s look at an example, starting at the root node. The following script runs the status command on the root node, which, in this case, is represented by the InteractiveCommandClient object:

```python
from libqtile.command_client import InteractiveCommandClient
c = InteractiveCommandClient()
```
The `InteractiveCommandClient` is a class that allows us to traverse the command graph using attributes to select child nodes or commands. In this example, we have resolved the `status()` command on the root object. The interactive command client will automatically find and connect to a running Qtile instance, and which it will use to dispatch the call and print out the return.

An alternative is to use the `CommandClient`, which allows for a more precise resolution of command graph objects, but is not as easy to interact with from a REPL:

```python
from libqtile.command_client import CommandClient
c = CommandClient()
print(c.call("status")())
```

Like the interactive client, the command client will automatically connect to a running Qtile instance. Here, we first resolve the `status()` command with the `.call("status")`, which simply located the function, then we can invoke the call with no arguments.

For the rest of this example, we will use the interactive command client. From the graph, we can see that the root node holds a reference to group nodes. We can access the “info” command on the current group like so:

```python
c.group.info()
```

To access a specific group, regardless of whether or not it is current, we use the Python mapping lookup syntax. This command sends group “b” to screen 1 (by the `libqtile.config.Group.to_screen()` method):

```python
c.group["b"]).to_screen(1)
```

In different contexts, it is possible to access a default object, where in other contexts a key is required. From the root of the graph, the current `group`, `layout`, `screen` and `window` can be accessed by simply leaving the key specifier out. The key specifier is mandatory for `widget` and `bar` nodes.

With this context, we can now drill down deeper in the graph, following the edges in the graphic above. To access the screen currently displaying group “b”, we can do this:

```python
c.group["b"]).screen.info()
```

Be aware, however, that group “b” might not currently be displayed. In that case, it has no associated screen, the path resolves to a non-existent node, and we get an exception:

```
libqtile.command.CommandError: No object screen in path 'group['b'].screen'
```

The graph is not a tree, since it can contain cycles. This path (redundantly) specifies the group belonging to the screen that belongs to group “b”:

```python
c.group["b"]).screen.group
```

This amount of connectivity makes it easy to reach out from a given object when callbacks and events fire on that object to related objects.
2.2.2 Keys

The key specifier for the various object types are as follows:

<table>
<thead>
<tr>
<th>Object</th>
<th>Key</th>
<th>Optional?</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>bar</td>
<td>“top”, “bottom”</td>
<td>No</td>
<td>c.screen.bar[“bottom”]</td>
</tr>
<tr>
<td>group</td>
<td>Name string</td>
<td>Yes</td>
<td>c.group[“one”]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>c.group</td>
</tr>
<tr>
<td>layout</td>
<td>Integer index</td>
<td>Yes</td>
<td>c.layout[2]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>c.layout</td>
</tr>
<tr>
<td>screen</td>
<td>Integer index</td>
<td>Yes</td>
<td>c.screen[1]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>c.screen</td>
</tr>
<tr>
<td>widget</td>
<td>Widget name</td>
<td>No</td>
<td>c.widget[“textbox”]</td>
</tr>
<tr>
<td>window</td>
<td>Integer window ID</td>
<td>Yes</td>
<td>c.window[123456]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>c.window</td>
</tr>
</tbody>
</table>

2.2.3 Digging Deeper: Command Objects

If you just want to script your Qtile window manager the above information, in addition to the documentation on the various scripting commands should be enough to get started. To develop the Qtile manager itself, we can dig into how Qtile represents these objects, which will lead to the way the commands are dispatched.

All of the configured objects setup by Qtile are CommandObject subclasses. These objects are so named because we can issue commands against them using the command scripting API. Looking through the code, the commands that are exposed are commands named cmd_* When writing custom layouts, widgets, or any other object, you can add your own custom cmd_* functions and they will be callable using the standard command infrastructure. An available command can be extracted by calling .command() with the name of the command.

In addition to having a set of associated commands, each command object also has a collection of items associated with it. This is what forms the graph that is shown above. For a given object type, the items() method returns all of the names of the associated objects of that type and whether or not there is a defaultable value. For example, from the root, .items("group") returns the name of all of the groups and that there is a default value, the currently focused group.

To navigate from one command object to the next, the .select() method is used. This method resolves a requested object from the command graph by iteratively selecting objects. A selector like [("group", "b"), ("screen", None)] would be to first resolve group “b”, then the screen associated to the group.
## 2.2.4 The Command Graph

In order to help in specifying command objects, there is the abstract command graph structure. The command graph structure allows us to address any valid command object and issue any command against it without needing to have any Qtile instance running or have anything to resolve the objects to. This is particularly useful when constructing lazy calls, where the Qtile instance does not exist to specify the path that will be resolved when the command is executed. The only limitation of traversing the command graph is that it must follow the allowed edges specified in the first section above.

Every object in the command graph is represented by a `CommandGraphNode`. Any call can be resolved from a given node. In addition, each node knows about all of the children objects that can be reached from it and have the ability to `.navigate()` to the other nodes in the command graph. Each of the object types are represented as `CommandGraphObject` types and the root node of the graph, the `CommandGraphRoot` represents the Qtile instance. When a call is performed on an object, it returns a `CommandGraphCall`. Each call will know its own name as well as be able to resolve the path through the command graph to be able to find itself.

Note that the command graph itself can standalone, there is no other functionality within Qtile that it relies on. While we could have started here and built up, it is helpful to understand the objects that the graph is meant to represent, as the graph is just a representation of a traversal of the real objects in a running Qtile window manager. In order to tie the running Qtile instance to the abstract command graph, we move on to the command interface.

## 2.2.5 Executing graph commands: Command Interface

The `CommandInterface` is what lets us take an abstract call on the command graph and resolve it against a running command object. Put another way, this is what takes the graph traversal `.group["b"].screen.info()` and executes the `info()` command against the addressed `screen` object. Additional functionality can be used to check that a given traversal resolves to actual objects and that the requested command actually exists. Note that by construction of the command graph, the traversals here must be feasible, even if they cannot be resolved for a given configuration state. For example, it is possible to check the screen associated to a group, even though the group may not be on a screen, but it is not possible to check the widget associated to a group.

The simplest form of the command interface is the `QtileCommandInterface`, which can take an in-process Qtile instance as the root `CommandObject` and execute requested commands. This is typically how we run the unit tests for Qtile.

The other primary example of this is the `IPCCommandInterface` which is able to then route all calls through an IPC client connected to a running Qtile instance. In this case, the command graph call can be constructed on the client side without having to dispatch to Qtile and once the call is constructed and deemed valid, the call can be executed.

In both of these cases, executing a command on a command interface will return the result of executing the command on a running Qtile instance. To support lazy execution, the `LazyCommandInterface` instead returns a `LazyCall` which is able to be resolved later by the running Qtile instance when it is configured to fire.

## 2.2.6 Tying it together: Command Client

So far, we have our running Command Objects and the Command Interface to dispatch commands against these objects as well as the Command Graph structure itself which encodes how to traverse the connections between the objects. The final component which ties everything together is the Command Client, which allows us to navigate through the graph to resolve objects, find their associated commands, and execute the commands against the held command interface.

The idea of the command client is that it is created with a reference into the command graph and a command interface. All navigation can be done against the command graph, and traversal is done by creating a new command client starting from the new node. When a command is executed against a node, that command is dispatched to the held command interface. The key decision here is how to perform the traversal. The command client exists in two different flavors: the standard `CommandClient` which is useful for handling more programmatic traversal of the graph, calling...
methods to traverse the graph, and the `InteractiveCommandClient` which behaves more like a standard Python object, traversing by accessing properties and performing key lookups.

Returning to our examples above, we now have the full context to see what is going on when we call:

```python
from libqtile.command_client import CommandClient

c = CommandClient()
print(c.call("status"))

from libqtile.command_client import InteractiveCommandClient

c = InteractiveCommandClient()
print(c.status())
```

In both cases, the command clients are constructed with the default command interface, which sets up an IPC connection to the running Qtile instance, and starts the client at the graph root. When we call `c.call("status")` or `c.status`, we navigate the command client to the `status` command on the root graph object. When these are invoked, the commands graph calls are dispatched via the IPC command interface and the results then sent back and printed on the local command line.

The power that can be realized by separating out the traversal and resolution of objects in the command graph from actually invoking or looking up any objects within the graph can be seen in the `lazy` module. By creating a lazy evaluated command client, we can expose the graph traversal and object resolution functionality via the same `InteractiveCommandClient` that is used to perform live command execution in the Qtile prompt.

### 2.3 Scripting Commands

Here is documented some of the commands available on objects in the command tree when running qshell or scripting commands to qtile. Note that this is an incomplete list, some objects, such as `layouts` and `widgets`, may implement their own set of commands beyond those given here.

#### 2.3.1 Qtile

```python
class libqtile.core.manager.Qtile(kore, config, eventloop, no_spawn=False, state=None)
```

This object is the root of the command graph

```python
cmd_add_rule(match_args, rule_args, min_priority=False)
```

Add a dgroup rule, returns rule_id needed to remove it

**Parameters**

- `match_args`: config.Match arguments
- `rule_args`: config.Rule arguments
- `min_priority`: If the rule is added with minimum priority (last) (default: False)

```python
cmd_addgroup(group, label=None, layout=None, layouts=None)
```

Add a group with the given name

```python
cmd_commands() → List[str]
```

Returns a list of possible commands for this object

Used by `__qsh__` for command completion and online help

```python
cmd_critical()
```

Set log level to CRITICAL

```python
cmd_debug()
```

Set log level to DEBUG
**cmd_delgroup** *(group)*
Delete a group with the given name

**cmd_display_kb** *(args)*
Display table of key bindings

**cmd_doc** *(name)* → str
Returns the documentation for a specified command name
Used by `__qsh__` to provide online help.

**cmd_error()**
Set log level to ERROR

**cmd_eval** *(code: str) → Tuple[bool, Optional[str]]*
Evaluates code in the same context as this function
Return value is tuple *(success, result)*, success being a boolean and result being a string representing the return value of `eval`, or `None` if `exec` was used instead.

**cmd_findwindow** *(prompt='window', widget='prompt')*
Launch prompt widget to find a window of the given name

**Parameters**
- **prompt**: Text with which to prompt user (default: “window”)
- **widget**: Name of the prompt widget (default: “prompt”)

**cmd_function** *(function, *args, **kwargs) → None*
Call a function with current object as argument

**cmd_get_info** ()
Prints info for all groups

**cmd_get_state** ()
Get pickled state for restarting qtile

**cmd_get_test_data** ()
Returns any content arbitrarily set in the `self.test_data` attribute. Useful in tests.

**cmd_groups** ()
Return a dictionary containing information for all groups

### Examples

```python
groups()
```

**cmd_hide_show_bar** *(position='all')*
Toggle visibility of a given bar

**Parameters**
- **position**: one of: “top”, “bottom”, “left”, “right”, or “all” (default: “all”)

**cmd_info** ()
Set log level to INFO

**cmd_internal_windows** ()
Return info for each internal window (bars, for example)

**cmd_items** *(name)* → Tuple[bool, List[str]]
Returns a list of contained items for the specified name
Used by __qsh__ to allow navigation of the object graph.

**cmd_list_widgets()**
List of all addressible widget names

**cmd_loglevel()**

**cmd_loglevelname()**

**cmd_next_layout (group=None)**
Switch to the next layout.

  **Parameters**

  **group**: Group name. If not specified, the current group is assumed

**cmd_next_screen()**
Move to next screen

**cmd_next_urgent()**
Focus next window with urgent hint

**cmd_pause()**
Drops into pdb

**cmd_prev_layout (group=None)**
Switch to the previous layout.

  **Parameters**

  **group**: Group name. If not specified, the current group is assumed

**cmd_prev_screen()**
Move to the previous screen

**cmd_qtile_info()**
Returns a dictionary of info on the Qtile instance

**cmd_qtilecmd (prompt='command', widget='prompt', messenger='xmessage') → None**
Execute a Qtile command using the client syntax

  Tab completion aids navigation of the command tree

  **Parameters**

  **prompt**: Text to display at the prompt (default: “command: “)

  **widget**: Name of the prompt widget (default: “prompt”)  

  **messenger**: Command to display output, set this to None to disable (default: “xmessage”)

**cmd_remove_rule (rule_id)**
Remove a dgroup rule by rule_id

**cmd_restart ()**
Restart qtile

**cmd_run_extension (extension)**
Run extensions

**cmd_run_external (full_path)**
Run external Python script

**cmd_screens ()**
Return a list of dictionaries providing information on all screens
**cmd_shutdown()**
Quit Qtile

**cmd_simulate_keypress**(modifiers, key)
Simulates a keypress on the focused window.

**Parameters**

- **modifiers**: A list of modifier specification strings. Modifiers can be one of “shift”, “lock”, “control” and “mod1” - “mod5”.
- **key**: Key specification.

**Examples**

simulate_keypress([“control”, “mod2”], “k”)

**cmd_spawn**(cmd, shell=False)
Run cmd, in a shell or not (default).

cmd may be a string or a list (similar to subprocess.Popen).

**Examples**

spawn(“firefox”)
spawn([“xterm”, “-T”, “Temporary terminal”])

**cmd_spawncmd**(prompt=’spawn’, widget=’prompt’, command=’%s’, complete=’cmd’, shell=True)
Spawn a command using a prompt widget, with tab-completion.

**Parameters**

- **prompt**: Text with which to prompt user (default: “spawn: ”).
- **widget**: Name of the prompt widget (default: “prompt”).
- **command**: command template (default: “%s”).
- **complete**: Tab completion function (default: “cmd”)

**cmd_status()**
Return “OK” if Qtile is running

**cmd_switch_groups**(groupa, groupb)
Switch position of groupa to groupb

**cmd_switchgroup**(prompt=’group’, widget=’prompt’)
Launch prompt widget to switch to a given group to the current screen

**Parameters**

- **prompt**: Text with which to prompt user (default: “group”)
- **widget**: Name of the prompt widget (default: “prompt”)

**cmd_sync()**
Sync the X display. Should only be used for development

**cmd_to_layout_index**(index, group=None)
Switch to the layout with the given index in self.layouts.

**Parameters**

2.3. Scripting Commands
**index**: Index of the layout in the list of layouts.

**group**: Group name. If not specified, the current group is assumed.

```python
cmd_to_screen(n)
```
Warp focus to screen n, where n is a 0-based screen number

**Examples**

to_screen(0)

```python
cmd_togroup(prompt='group', widget='prompt')
```
Launch prompt widget to move current window to a given group

**Parameters**

- **prompt**: Text with which to prompt user (default: “group”)
- **widget**: Name of the prompt widget (default: “prompt”)

```python
cmd_tracemalloc_dump()
```
Dump tracemalloc snapshot

```python
cmd_tracemalloc_toggle()
```
Toggle tracemalloc status

Running tracemalloc is required for qtile-top

```python
cmd_validate_config()
```

```python
cmd_warning()
```
Set log level to WARNING

```python
cmd_windows()
```
Return info for each client window

### 2.3.2 Bar

```python
class libqtile.bar.Bar(widgets, size, **config)
```
A bar, which can contain widgets

**Parameters**

- **widgets**: A list of widget objects.
- **size**: The “thickness” of the bar, i.e. the height of a horizontal bar, or the width of a vertical bar.

<table>
<thead>
<tr>
<th>key</th>
<th>default</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>background</td>
<td>'#000000'</td>
<td>'Background colour.'</td>
</tr>
<tr>
<td>margin</td>
<td>0</td>
<td>'Space around bar as int or list of ints [N E S W].'</td>
</tr>
<tr>
<td>opacity</td>
<td>1</td>
<td>'Bar window opacity.'</td>
</tr>
</tbody>
</table>

```python
cmd_commands() → List[str]
```
Returns a list of possible commands for this object

Used by __qsh__ for command completion and online help
cmd_doc(name) → str
Returns the documentation for a specified command name
Used by __qsh__ to provide online help.

cmd_eval(code: str) → Tuple[bool, Optional[str]]
Evaluates code in the same context as this function
Return value is tuple (success, result), success being a boolean and result being a string representing the return value of eval, or None if exec was used instead.

cmd_fake_button_press(screen, position, x, y, button=1)
Fake a mouse-button-press on the bar. Co-ordinates are relative to the top-left corner of the bar.
:screen The integer screen offset :position One of “top”, “bottom”, “left”, or “right”

cmd_function(function, *args, **kwargs) → None
Call a function with current object as argument

cmd_info()
Info for this object.

cmd_items(name) → Tuple[bool, List[str]]
Returns a list of contained items for the specified name
Used by __qsh__ to allow navigation of the object graph.

2.3.3 Group

class libqtile.config.Group(name, matches=None, exclusive=False, spawn=None, layout=None, layouts=None, persist=True, init=True, layout_opts=None, screen_affinity=None, position=9223372036854775807, label=None)
Represents a “dynamic” group
These groups can spawn apps, only allow certain Matched windows to be on them, hide when they’re not in use, etc. Groups are identified by their name.

Parameters

name [string] the name of this group
matches [default None] list of Match objects whose windows will be assigned to this group
exclusive [boolean] when other apps are started in this group, should we allow them here or not?
spawn [boolean] when other apps are started in this group, this will be exec() d when the group is created, you can pass either a program name or a list of programs to exec()
layout [string] the name of default layout for this group (e.g. ‘max’ or ‘stack’). This is the name specified for a particular layout in config.py or if not defined it defaults in general the class name in all lower case.
layouts [list] the group layouts list overriding global layouts. Use this to define a separate list of layouts for this particular group.
persist [boolean] should this group stay alive with no member windows?
init [boolean] is this group alive when qtile starts?
position [int] group position
label [string] the display name of the group. Use this to define a display name other than name of the group. If set to None, the display name is set to the name.

2.3.4 Screen

class libqtile.config.Screen(top: Optional[Union[libqtile.bar.Bar, libqtile.bar.Gap]] = None, 
bottom: Optional[Union[libqtile.bar.Bar, libqtile.bar.Gap]] = None, 
left: Optional[Union[libqtile.bar.Bar, libqtile.bar.Gap]] = None, 
right: Optional[Union[libqtile.bar.Bar, libqtile.bar.Gap]] = None, 
wallpaper: Optional[str] = None, 
wallpaper_mode: Optional[str] = None, 
x: Optional[int] = None, 
y: Optional[int] = None, 
width: Optional[int] = None, 
height: Optional[int] = None)

A physical screen, and its associated paraphernalia.

Define a screen with a given set of Bars of a specific geometry. Note that bar.Bar objects can only be placed at the top or the bottom of the screen (bar.Gap objects can be placed anywhere). Also, x, y, width, and height aren’t specified usually unless you are using ‘fake screens’.

The wallpaper parameter, if given, should be a path to an image file. How this image is painted to the screen is specified by the wallpaper_mode parameter. By default, the image will be placed at the screens origin and retain its own dimensions. If the mode is ‘fill’, the image will be centred on the screen and resized to fill it. If the mode is ’stretch’, the image is stretched to fit all of it into the screen.

```
cmd_commands() \rightarrow List[str]
    Returns a list of possible commands for this object
    Used by __qsh__ for command completion and online help

cmd_doc(name) \rightarrow str
    Returns the documentation for a specified command name
    Used by __qsh__ to provide online help.

cmd_eval(code: str) \rightarrow Tuple[bool, Optional[str]]
    Evaluates code in the same context as this function
    Return value is tuple (success, result), success being a boolean and result being a string representing the return value of eval, or None if exec was used instead.

cmd_function(function, *args, **kwargs) \rightarrow None
    Call a function with current object as argument

cmd_info() \rightarrow bool, List[str]]
    Returns a dictionary of info for this screen.

cmd_items(name) \rightarrow List[str]]
    Returns a list of contained items for the specified name
    Used by __qsh__ to allow navigation of the object graph.

cmd_next_group(skip_empty=False, skip_managed=False)
    Switch to the next group

cmd_prev_group(skip_empty=False, skip_managed=False)
    Switch to the previous group

cmd_resize(x=None, y=None, w=None, h=None)
    Resize the screen

cmd_toggle_group(group_name=None)
    Switch to the selected group or to the previously active one
```
cmd_togglegroup (groupName=None)
    Switch to the selected group or to the previously active one
    Deprecated: use toggle_group()

2.3.5 Window

class libqtile.window.Window (window, qtile)

    cmd_bring_to_front()
    cmd_commands () → List[str]
        Returns a list of possible commands for this object
        Used by __qsh__ for command completion and online help
    cmd_disable_floating()
    cmd_disable_fullscreen()
    cmd_doc (name) → str
        Returns the documentation for a specified command name
        Used by __qsh__ to provide online help.
    cmd_down_opacity()
    cmd_enable_floating()
    cmd_enable_fullscreen()
    cmd_eval (code: str) → Tuple[bool, Optional[str]]
        Evaluates code in the same context as this function
        Return value is tuple (success, result), success being a boolean and result
        being a string representing the return value of eval, or None if exec was used
        instead.
    cmd_focus (warp=None)
        Focuses the window.
    cmd_function (function, *args, **kwargs) → None
        Call a function with current object as argument
    cmd_get_position()
    cmd_get_size()
    cmd_hints()
        Returns the X11 hints (WM_HINTS and WM_SIZE_HINTS) for this window.
    cmd_info()
        Returns a dictionary of info for this object
    cmd_inspect()
        Tells you more than you ever wanted to know about a window
    cmd_items (name) → Tuple[bool, List[str]]
        Returns a list of contained items for the specified name
        Used by __qsh__ to allow navigation of the object graph.
cmd_kill()
    Kill this window
    Try to do this politely if the client support this, otherwise be brutal.

cmd_match(*args, **kwargs)

cmd_move_floating(dx, dy)
    Move window by dx and dy

cmd_opacity(opacity)

cmd_resize_floating(dw, dh)
    Add dw and dh to size of window

cmd_set_position(dx, dy)

cmd_set_position_floating(x, y)
    Move window to x and y

cmd_set_size_floating(w, h)
    Set window dimensions to w and h

cmd_static(screen, x=None, y=None, width=None, height=None)
    Makes this window a static window, attached to a Screen
    If any of the arguments are left unspecified, the values given by the window itself are used instead. So, for
    a window that’s aware of its appropriate size and location (like dzen), you don’t have to specify anything.

cmd_toggle_floating()

cmd_toggle_fullscreen()

cmd_toggle_maximize()

cmd_toggle_minimize()

cmd_togroup(groupName=None, *, switch_group=False)
    Move window to a specified group.
    If groupName is not specified, we assume the current group. If switch_group is True, also switch to that
group.

Examples

Move window to current group:

togroup()

Move window to group “a”:

togroup("a")

Move window to group “a”, and switch to group “a”:

togroup("a", switch_group=True)

cmd_toscreen(index=None)
    Move window to a specified screen.
    If index is not specified, we assume the current screen
Examples

Move window to current screen:

toscreen()

Move window to screen 0:

toscreen(0)

cmd_up_opacity()

2.4 Keybindings in images

2.4.1 Default configuration
Keybindings for Qtile
Modifiers: mod4, shift

Keybindings for Qtile
Modifiers: mod4, control
2.4.2 Generate your own images

Qtile provides a tiny helper script to generate keybindings images from a config file. In the repository, the script is located under `scripts/gen-keybinding-img`.

This script accepts a configuration file and an output directory. If no argument is given, the default configuration will be used and files will be placed in the same directory where the command has been run.

```bash
usage: gen-keybinding-img [-h] [-c CONFIGFILE] [-o OUTPUT_DIR]
Qtile keybindings image generator

optional arguments:
  -h, --help            show this help message and exit
  -c CONFIGFILE, --config CONFIGFILE
                        use specified configuration file. If no presented
default will be used
  -o OUTPUT_DIR, --output-dir OUTPUT_DIR
                        set directory to export all images to
```
3.1 Contributing

3.1.1 Reporting bugs

Perhaps the easiest way to contribute to Qtile is to report any bugs you run into on the GitHub issue tracker. Useful bug reports are ones that get bugs fixed. A useful bug report normally has two qualities:

1. **Reproducible.** If your bug is not reproducible it will never get fixed. You should clearly mention the steps to reproduce the bug. Do not assume or skip any reproducing step. Described the issue, step-by-step, so that it is easy to reproduce and fix.

2. **Specific.** Do not write a essay about the problem. Be Specific and to the point. Try to summarize the problem in minimum words yet in effective way. Do not combine multiple problems even they seem to be similar. Write different reports for each problem.

Ensure to include any appropriate log entries from ~/.local/share/qtile/qtile.log and/or ~/.xsession-errors!

3.1.2 Writing code

To get started writing code for Qtile, check out our guide to *Hacking on Qtile*.

Submit a pull request

You’ve done your hacking and are ready to submit your patch to Qtile. Great! Now it’s time to submit a pull request to our issue tracker on GitHub.

**Important:** Pull requests are not considered complete until they include all of the following:

- **Code** that conforms to PEP8.
- **Unit tests** that pass locally and in our CI environment (More below).
- **Documentation** updates on an as needed basis.

Feel free to add your contribution (no matter how small) to the appropriate place in the CHANGELOG as well!
Unit testing

We must test each *unit* of code to ensure that new changes to the code do not break existing functionality. The framework we use to test Qtile is pytest. How pytest works is outside of the scope of this documentation, but there are tutorials online that explain how it is used.

Our tests are written inside the `test` folder at the top level of the repository. Reading through these, you can get a feel for the approach we take to test a given unit. Most of the tests involve an object called `manager`. This is the test manager (defined in `test/conftest.py`), which exposes a command client at `manager.c` that we use to test a Qtile instance running in a separate thread as if we were using a command client from within a running Qtile session.

For any Qtile-specific question on testing, feel free to ask on our issue tracker or on IRC (#qtile on irc.oftc.net).

3.2 Hacking on Qtile

3.2.1 Requirements

Any reasonably recent version of these should work, so you can probably just install them from your package manager.

- `pytest`
- `Xephyr`
- `xrandr`, `xcalc`, `xeyes` and `xclock` (*x11-apps* on Ubuntu)

On Ubuntu, if testing on Python 3, this can be done with:

```
sudo apt-get install python3-pytest xserver-xephyr x11-apps
```

On ArchLinux, the X11 requirements are installed with:

```
sudo pacman -S xorg-xrandr xorg-xcalc xorg-xeyes xorg-xclock
```

To build the documentation, you will also need to install `graphviz`. On ArchLinux, you can install it with `sudo pacman -S graphviz`.

3.2.2 Building cffi module

Qtile ships with a small in-tree pangocairo binding built using cffi, `pangocffi.py`, and also binds to xcursor with cffi. The bindings are not built at run time and will have to be generated manually when the code is downloaded or when any changes are made to the cffi library. This can be done by calling:

```
./scripts/ffibuild
```

3.2.3 Setting up the environment

In the root of the project, run `. /dev.sh`. It will create a virtualenv called `venv`.

Activate this virtualenv with `. venv/bin/activate`. Deactivate it with the `deactivate` command.
3.2.4 Building the documentation

Activate your virtualenv, and install the graphviz Python package:

```
pip install graphviz
```

Go into the docs/ directory and run
```
pip install -r requirements.txt
```

Build the documentation with
```
make html
```

Check the result by opening _build/html/index.html in your browser.

3.2.5 Development and testing

In practice, the development cycle looks something like this:

1. make minor code change
2. run appropriate test: `pytest tests/test_module.py` or `pytest -k PATTERN`
3. GOTO 1, until hackage is complete
4. run entire test suite: `pytest`
5. commit

Of course, your patches should also pass the unit tests as well (i.e. `make check`). These will be run by ci on every pull request so you can see whether or not your contribution passes.

3.2.6 Coding style

While not all of our code follows PEP8, we do try to adhere to it where possible. All new code should be PEP8 compliant.

The `make lint` command will run a linter with our configuration over libqtile to ensure your patch complies with reasonable formatting constraints. We also request that git commit messages follow the standard format.

3.2.7 Deprecation policy

When a widget API is changed, you should deprecate the change using `libqtile.widget.base.deprecated` to warn users, in addition to adding it to the appropriate place in the changelog. We will typically remove deprecated APIs one tag after they are deprecated.

3.2.8 Using Xephyr

Qtile has a very extensive test suite, using the Xephyr nested X server. When tests are run, a nested X server with a nested instance of Qtile is fired up, and then tests interact with the Qtile instance through the client API. The fact that we can do this is a great demonstration of just how completely scriptable Qtile is. In fact, Qtile is designed expressly to be scriptable enough to allow unit testing in a nested environment.

The Qtile repo includes a tiny helper script to let you quickly pull up a nested instance of Qtile in Xephyr, using your current configuration. Run it from the top-level of the repository, like this:

```
./scripts/xephyr
```

Change the screen size by setting the SCREEN_SIZE environment variable. Default: 800x600. Example:
SCREEN_SIZE=1920x1080 ./scripts/xephyr

Change the log level by setting the LOG_LEVEL environment variable. Default: INFO. Example:

LOG_LEVEL=DEBUG ./scripts/xephyr

The script will also pass any additional options to Qtile. For example, you can use a specific configuration file like this:

./scripts/xephyr -c ~/.config/qtile/other_config.py

Once the Xephyr window is running and focused, you can enable capturing the keyboard shortcuts by hitting Control+Shift. Hitting them again will disable the capture and let you use your personal keyboard shortcuts again.

You can close the Xephyr window by enabling the capture of keyboard shortcuts and hit Mod4+Control+Q. Mod4 (or Mod) is usually the Super key (or Windows key). You can also close the Xephyr window by running qtile-cmd -o cmd -f shutdown in a terminal (from inside the Xephyr window of course).

You don’t need to run the Xephyr script in order to run the tests as the test runner will launch its own Xephyr instances.

### 3.2.9 Second X Session

Some users prefer to test Qtile in a second, completely separate X session: Just switch to a new tty and run startx normally to use the ~/.xinitrc X startup script.

It’s likely though that you want to use a different, customized startup script for testing purposes, for example ~/.config/qtile/xinitrc. You can do so by launching X with:

```
startx ~/.config/qtile/xinitrc
```

`startx` deals with multiple X sessions automatically. If you want to use xinit instead, you need to first copy /etc/X11/xinit/xserverrc to ~/.xserverrc: when launching it, you have to specify a new session number:

```
xinit ~/.config/qtile/xinitrc -- :1
```

Examples of custom X startup scripts are available in qtile-examples.

### 3.2.10 Debugging in PyCharm

Make sure to have all the requirements installed and your development environment setup.

PyCharm should automatically detect the venv virtualenv when opening the project. If you are using another virtualenv, just instruct PyCharm to use it in Settings -> Project: qtile -> Project interpreter.

In the project tree, on the left, right-click on the libqtile folder, and click on Mark Directory as -> Sources Root.

Next, add a Configuration using a Python template with these fields:

- **Script path**: bin/qtile, or the absolute path to it
- **Parameters**: `-c libqtile/resources/default_config.py`, or nothing if you want to use your own config file in `~/.config/qtile/config.py`
- **Environment variables**: `PYTHONUNBUFFERED=1;DISPLAY=:1`
- **Working directory**: the root of the project
• Add contents root to PYTHONPATH: yes
• Add source root to PYTHONPATH: yes

Then, in a terminal, run:

```
Xephyr +extension RANDR -screen 1920x1040 :1 -ac &
```

Note that we used the same display, :1, in both the terminal command and the PyCharm configuration environment variables. Feel free to change the screen size to fit your own screen.

Finally, place your breakpoints in the code and click on Debug!

Once you finished debugging, you can close the Xephyr window with `kill PID` (use the `jobs` builtin to get its PID).

### 3.2.11 Debugging in VSCode

Make sure to have all the requirements installed and your development environment setup.

Open the root of the repo in VSCode. If you have created it, VSCode should detect the `venv` virtualenv, if not, select it.

Create a `launch.json` file with the following lines.

```json
{
   "version": "0.2.0",
   "configurations": [
   {
      "name": "Python: Qtile",
      "type": "python",
      "request": "launch",
      "program": "${workspaceFolder}/bin/qtile",
      "cwd": "${workspaceFolder}",
      "args": ["-c", "libqtile/resources/default_config.py"],
      "console": "integratedTerminal",
      "env": {"PYTHONUNBUFFERED":"1", "DISPLAY":":1"}
   }
   ]
}
```

Then, in a terminal, run:

```
Xephyr +extension RANDR -screen 1920x1040 :1 -ac &
```

Note that we used the same display, :1, in both the terminal command and the VSCode configuration environment variables. Then debug usually in VSCode. Feel free to change the screen size to fit your own screen.

### 3.2.12 Resources

Here are a number of resources that may come in handy:

• Inter-Client Conventions Manual
• Extended Window Manager Hints
• A reasonable basic Xlib Manual
3.2.13 Troubleshoot

Cairo errors

When running the Xephyr script (`./scripts/xephyr`), you might see tracebacks with attribute errors like the following or similar:

```
AttributeError: cffi library 'libcairo.so.2' has no function, constant or global
→variable named 'cairo_xcb_surface_create'
```

If it happens, it might be because the `cairocffi` and `xcffib` dependencies were installed in the wrong order.

To fix this:

1. uninstall them from your environment: with `pip uninstall cairocffi xcffib` if using a virtualenv, or with your system package-manager if you installed the development version of Qtile system-wide.
2. re-install them sequentially (again, with pip or with your package-manager):

   ```
   pip install xcffib
   pip install --no-cache-dir cairocffi
   ```

See this issue comment for more information.

If you are using your system package-manager and the issue still happens, the packaging of `cairocffi` might be broken for your distribution. Try to contact the persons responsible for `cairocffi`'s packaging on your distribution, or to install it from the sources with `xcffib` available.

DBus/GObject errors

When running the Xephyr script (`./scripts/xephyr`), you might see a line in the output like the following or similar:

```
libqtile manager.py:setup_python_dbus():L310 importing dbus/gobject failed, dbus:
→will not work.
```

If it happens, it might be because you are missing some dependencies on your system and/or in your Qtile virtualenv.

To fix this:

1. Follow the installation instructions of PyGObject. There are methods for several Linux distributions: pick yours.
2. There are instructions for system-wide installation and virtualenv installation: pick the relevant one, depending on how you installed the development version of Qtile (usually in a virtualenv).
3. Optionally re-install Qtile’s dependencies:

   ```
   pip install -r requirements.txt
   pip install -r requirements-dev.txt
   ```
Fonts errors

When running the test suite or the Xephyr script (`./scripts/xephyr`), you might see errors in the output like the following or similar:

- **Xephyr script:**

```bash
xterm: cannot load font "-Misc-Fixed-medium-R---13-120-75-75-C-120-ISO10646-1"
xterm: cannot load font "-misc-fixed-medium-r-semicondensed--13-120-75-75-c-60-iso10646-1"
```

- **pytest:**

```bash
-------- Captured stderr setup --------
Warning: Cannot convert string "8x13" to type FontStruct
Warning: Unable to load any usable ISO8859 font
Warning: Unable to load any usable ISO8859 font
Error: Aborting: no font found
-------- Captured stderr teardown --------
Qtile exited with exitcode: -9
```

If it happens, it might be because you’re missing fonts on your system.

On ArchLinux, you can fix this by installing `xorg-fonts-misc`:

```bash
sudo pacman -S xorg-fonts-misc
```

Try to search for “xorg fonts misc” with your distribution name on the internet to find how to install them.
4.1 Frequently Asked Questions

4.1.1 Why the name Qtile?

Users often wonder, why the Q? Does it have something to do with Qt? No. Below is an IRC excerpt where cortesi explains the great trial that ultimately brought Qtile into existence, thanks to the benevolence of the Open Source Gods. Praise be to the OSG!

ramnes: what does Qtile mean?
ramnes: what's the Q?
@tych0: ramnes: it doesn't :)
@tych0: cortesi was just looking for the first letter that wasn't registered in a domain name with "tile" as a suffix
@tych0: qtile it was :)
cortesi: tych0, dx: we really should have something more compelling to explain the name. one day i was swimming at manly beach in sydney, where i lived at the time. suddenly, i saw an enormous great white right beside me. it went for my leg with massive, gaping jaws, but quick as a flash, i thumb-punched it in both eyes. when it reared back in agony, i saw that it had a jagged, gnarly scar on its stomach... a scar shaped like the letter "Q".
cortesi: while it was distracted, i surfed a wave to shore. i knew that i had to dedicate my next open source project to the ocean gods, in thanks for my lucky escape. and thus, qtile got its name...

4.1.2 When I first start xterm/urxvt/rxvt containing an instance of Vim, I see text and layout corruption. What gives?

Vim is not handling terminal resizes correctly. You can fix the problem by starting your xterm with the “-wf” option, like so:

```
xterm -wf -e vim
```

Alternatively, you can just cycle through your layouts a few times, which usually seems to fix it.
4.1.3 How do I know which modifier specification maps to which key?

To see a list of modifier names and their matching keys, use the `xmodmap` command. On my system, the output looks like this:

```
$ xmodmap
xmodmap: up to 3 keys per modifier, (keycodes in parentheses):

  shift   Shift_L (0x32), Shift_R (0x3e)
  lock    Caps_Lock (0x9)
  control Control_L (0x25), Control_R (0x69)
  mod1    Alt_L (0x40), Alt_R (0x6c), Meta_L (0xcd)
  mod2    Num_Lock (0x4d)
  mod3
  mod4    Super_L (0xce), Hyper_L (0xcf)
  mod5    ISO_Level3_Shift (0x5c), Mode_switch (0xcb)
```

4.1.4 My “pointer mouse cursor” isn’t the one I expect it to be!

Qtile should set the default cursor to left_ptr, you must install xcb-util-cursor if you want support for themed cursors.

4.1.5 LibreOffice menus don’t appear or don’t stay visible

A workaround for problem with the mouse in libreoffice is setting the environment variable `SAL_USE_VCLPLUGIN=gen`. It is dependet on your system configuration where to do this. e.g. Arch-Linux with libreoffice-fresh in `/etc/profile.d/libreoffice-fresh.sh`.

4.2 License

This project is distributed under the MIT license.

Copyright (c) 2008, Aldo Cortesi All rights reserved.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the “Software”), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

- genindex
INDEX

A
addgroup () (libqtile.hook.subscribe method), 31

B
Bar (class in libqtile.bar), 17
Bsp (class in libqtile.layout.bsp), 35

C
change_group () (libqtile.hook.subscribe method), 31
Click (class in libqtile.config), 14
client_focus () (libqtile.hook.subscribe method), 31
client_killed () (libqtile.hook.subscribe method), 31
client_managed () (libqtile.hook.subscribe method), 31
client_mouse_enter () (libqtile.hook.subscribe method), 32
client_name_updated () (libqtile.hook.subscribe method), 32
client_new () (libqtile.hook.subscribe method), 32
client_urgent_hint_changed () (libqtile.hook.subscribe method), 32
cmd_add_rule () (libqtile.core.manager.Qtile method), 54

cmd_addgroup () (libqtile.core.manager.Qtile method), 54

cmd_bring_to_front () (libqtile.window.Window method), 61
cmd_commands () (libqtile.bar.Bar method), 58
cmd_commands () (libqtile.config.Screen method), 60
cmd_commands () (libqtile.core.manager.Qtile method), 54
cmd_commands () (libqtile.window.Window method), 61
cmd_critical () (libqtile.core.manager.Qtile method), 54

cmd_debug () (libqtile.core.manager.Qtile method), 54

cmd_delgroup () (libqtile.core.manager.Qtile method), 54

cmd_disable_fullscreen () (libqtile.window.Window method), 61
cmd_display_kb () (libqtile.core.manager.Qtile method), 55
cmd_doc () (libqtile.bar.Bar method), 58
cmd_doc () (libqtile.config.Screen method), 60
cmd_doc () (libqtile.core.manager.Qtile method), 55
cmd_doc () (libqtile.window.Window method), 61
cmd_down_opacity () (libqtile.window.Window method), 61
cmd_enable_fullscreen () (libqtile.window.Window method), 61
cmd_enable_fullscreen () (libqtile.core.manager.Qtile method), 55
cmd_eval () (libqtile.core.manager.Qtile method), 55
cmd_eval () (libqtile.config.Screen method), 60
cmd_eval () (libqtile.core.manager.Qtile method), 55
cmd_eval () (libqtile.window.Window method), 61
cmd_fake_button_press () (libqtile.bar.Bar method), 59
cmd_findwindow () (libqtile.core.manager.Qtile method), 55
cmd_focus () (libqtile.window.Window method), 61
cmd_function () (libqtile.bar.Bar method), 59
cmd_function () (libqtile.config.Screen method), 60
cmd_function () (libqtile.core.manager.Qtile method), 55
cmd_function () (libqtile.window.Window method), 61
cmd_get_info () (libqtile.core.manager.Qtile method), 55
cmd_get_position () (libqtile.window.Window method), 61
cmd_get_size () (libqtile.window.Window method), 61
cmd_get_state () (libqtile.core.manager.Qtile method), 55
cmd_get_test_data () (libqtile.core.manager.Qtile method), 55
cmd_groups () (libqtile.core.manager.Qtile method), 55
Index
Qtile (class in libqtile.core.manager), 54

R
RatioTile (class in libqtile.layout.ratiotile), 42
restart() (libqtile.hook.subscribe method), 33
Rule (class in libqtile.config), 9

S
ScratchPad (class in libqtile.config), 10
Screen (class in libqtile.config), 17
screen_change() (libqtile.hook.subscribe method), 33
selection_change() (libqtile.hook.subscribe method), 34
selection_notify() (libqtile.hook.subscribe method), 34
setgroup() (libqtile.hook.subscribe method), 34
shutdown() (libqtile.hook.subscribe method), 34
simple_key_binder() (in module libqtile.dgroups), 9
Slice (class in libqtile.layout.slice), 42
Stack (class in libqtile.layout.stack), 42
startup() (libqtile.hook.subscribe method), 34
startup_complete() (libqtile.hook.subscribe method), 34
startup_once() (libqtile.hook.subscribe method), 34

T
Tile (class in libqtile.layout.tile), 43
TreeTab (class in libqtile.layout.tree), 44

V
VerticalTile (class in libqtile.layout.verticaltile), 45

W
Window (class in libqtile.window), 61
window_name_change() (libqtile.hook.subscribe method), 34

Z
Zoomy (class in libqtile.layout.zoomy), 46