Simpleline Documentation

Release 1.7

Jiri Konecny

Dec 07, 2020

Contents:

CHAPTER 1

Introduction

Simpleline is a text UI framework. Originally a part of the Anaconda installer project.

It is designed to be used with line-based machines and tools (e.g. serial console) so that every new line it appended to the bottom of the screen. Printed lines are never rewritten!

1.1 How to use

The best learning sources can be found in the examples directory in the GitHub repository and you can read the *Guide to Simpleline* section of this documentation. However, some basic usage of Simpleline will be shown here too, to get an idea of how Simpleline works:

```
from simpleline import App
from simpleline.render.screen import UIScreen
from simpleline.render.screen_handler import ScreenHandler
from simpleline.render.widgets import TextWidget


# UIScreen is the main building item for Simpleline. Every screen
# which will user see should be inherited from UIScreen.
class HelloWorld(UIScreen):

    def __init__(self):
        # Set title of the screen.
        super().__init__(title=u"Hello World")

    def refresh(self, args=None):
        # Fill the self.window attribute by the WindowContainer and set screen title␣
↪as header.
        super().refresh()
        widget = TextWidget("Body text")
        self.window.add_with_separator(widget)
```

(continues on next page)

(continued from previous page)

```
if __name__ == "__main__":
    # Initialize application (create scheduler and event loop).
    App.initialize()

    # Create our screen.
    screen = HelloWorld()

    # Schedule screen to the screen scheduler.
    # This can be called only after App.initialize().
    ScreenHandler.schedule_screen(screen)

    # Run the application. You must have some screen scheduled
    # otherwise it will end in an infinite loop.
    App.run()
```

The output from the simple *Hello World* example above:

```
$ ./run_example.sh 00_basic
================================================================================
================================================================================
Hello World

Body text

Please make a selection from the above ['c' to continue, 'q' to quit, 'r' to
refresh]:
```

If a user presses **r** and then **enter** to refresh, the same screen is printed again. This will be printed to a monitor:

```
$ ./run_example.sh 00_basic
================================================================================
================================================================================
Hello World

Body text

Please make a selection from the above ['c' to continue, 'q' to quit, 'r' to
refresh]: r
================================================================================
================================================================================
Hello World

Body text

Please make a selection from the above ['c' to continue, 'q' to quit, 'r' to
refresh]:
```

As you can see the whole screen is not rewritten – only printed again on the bottom. This is the expected behavior so the actual screen is always at the bottom but you can see the whole history. This behavior makes working with line based machines and tools much easier.

1.2 Dependencies

This is a Python3-only project. This code should not be difficult to migrate to Python2. However, there is no need from the community, so it is only compatible with Python3 at the moment. No special libraries are required to use this library. If you want to use glib event loop instead of the original one you need to install glib and Python3 gobject introspection.

If you want to run tests (make ci), you need to install Pocketlint and glib with gobject introspection for Python3.

CHAPTER 2

Guide to Simpleline

Simpleline is a text user interface framework written completely in Python 3 with a possibility to have *non-python event loops*. With the exception of optional event loops, Simpleline has almost no dependency on external libraries.

This UI is simple and easy to use. It is designed to be used with line-based machines and tools (e.g. serial console) so that every new line is appended to the bottom of the screen. Printed lines are never rewritten!

2.1 Basic components

For every application, the following parts are always required.

- The `App` static class to initialize and run application.
- The `ScreenHandler` static class for scheduling screens.
- The `UIScreen` based classes to create screens which will form the application.
- *Widgets* to show anything on the screens.
- *Containers* to position widgets on the screen.

Look at the next section to see how everything fits together.

2.2 How to create a simple application

Interaction with a user is necessary to have a useful UI framework. To show anything to a user the `UIScreen` class must be used. So we will subclass this class to create our screen and set a title for it:

```
class DividerScreen(UIScreen):

    def __init__(self):
        # Set title of the screen.
        super().__init__(title=u"Divider")
        self._message = 0
```

The `self._message` variable will be used later to show results to the user.

The screen's main purpose is to present content to a user. For this we need widgets and containers.

The *UIScreen.window* attribute is the most important part of the screen for rendering. It contains the *WindowContainer* container which is created and filled up by the *UIScreen.refresh()* method. Everything added to this container is printed to the monitor. We should override the *UIScreen.refresh()* method and call the parent's version to prepare the container. Then we can add *widgets* to the container. The screen will continue like this:

```
    def refresh(self, args=None):
        # Fill the self.window attribute by the WindowContainer and set screen title␣
↪as header.
        super().refresh()

        widget = TextWidget("Result: " + str(self._message))
        self.window.add_with_separator(widget)
```

The *WindowContainer.add_with_separator()* method will print a blank line after the *TextWidget's* text.

Now the user has the header and result printed on the screen but it would be nice to give them a hint about how to use the Divider screen. The best part of the screen for this is the *Prompt*. The *Prompt* class is responsible for guiding the user by giving them a set of possible options to choose from.

We will set the message of the prompt inside of the *UIScreen.prompt()* method. We also remove the default option to continue, because it functions the same as quitting when there is only one screen in the application.

```
    def prompt(self, args=None):
        # Change user prompt
        prompt = super().prompt()

        # Set message to the user prompt. Give a user hint how he/she may control our␣
↪application.
        prompt.set_message("Pass numbers to divider in a format: 'num / num'")

        # Remove continue option from the control. There is no need for that
        # when we have only one screen.
        prompt.remove_option('c')

        return prompt
```

When we are able to present our content to a user, we want to have the possibility to process user input. For this purpose there is the *UIScreen.input()* method. Input from a user is passed to this method, and the screen may process it, discard it or return it for further processing.

If input processing is not required and the screen should only be used for displaying information to a user, then the *UIScreen.input_required* property should be set to *False*. However, in this case you need to *close*, *redraw* or *push* a new screen manually. This can be done, for example, in the *UIScreen.show_all()* method.

```
    def input(self, args, key):
        """Process input from user and catch numbers with '/' symbol."""

        # Test if user passed valid input for divider.
        # This will basically take number + number and nothing else and only positive␣
↪numbers.
        groups = re.match(r'(\d+) *\/ *(\d+)$', key)
        if groups:
            num1 = int(groups[1])
```

(continues on next page)

(continued from previous page)

```
            num2 = int(groups[2])

            # Dividing by zero is not valid so we won't accept this input from the␣
↪user. New
            # input is then required from the user.
            if num2 == 0:
                return InputState.DISCARDED

            self._message = int(num1 / num2)

            # Because this input is processed we need to show this screen (show the␣
↪result)
            # again by returning PROCESSED_AND_REDRAW.
            # This will call the refresh method so our new result will be processed␣
↪inside
            # of the refresh() method.
            return InputState.PROCESSED_AND_REDRAW
        else:
            # Not input for our screen, try other default inputs. This will result in␣
↪the
            # same state as DISCARDED when no default option is used.
            return key
```

Our screen is finished. Next, we need to use it in our application. To run an application the *App* static class must be used.

This class will initialize an event loop and the scheduler by the *App.initialize()* method. When the application is initialized, we need to pass our screen to the screen stack (you can pass multiple screens but we have only one here). To pass a screen to the screen stack we will use *ScreenHandler*. For further explanation on screen scheduling, refer to the *Screen Handling* section.

```
if __name__ == "__main__":
    # Initialize application (create scheduler and event loop).
    App.initialize()

    # Create our screen.
    screen = DividerScreen()

    # Schedule screen to the screen scheduler.
    # This can be called only after App.initialize().
    ScreenHandler.schedule_screen(screen)

    # Run the application. You must have some screen scheduled
    # otherwise it will end in an infinite loop.
    App.run()
```

Well done! You have your first application in Simpleline.

2.3 Further reading

I would recommend everyone who wants to use Simpleline to look at the examples which is one of the best sources of information. Another place to look is the *Public API* documentation section.

CHAPTER 3

Public API

API listed here is the public API. Developers shouldn't use anything which is not mentioned here!

3.1 App

The App class is the heart of Simpleline. It holds the *event loop* instance and scheduler which are used by Simpleline. Both can be replaced if needed in the initialization phase by calling the `App.initialize()` method. However, by replacing the scheduler you are replacing most of the logic in Simpleline. Replacing the scheduler is not supported, since it is currently not a part of the public API.

All Simpleline applications must be run by the `App.run()` method. This method will start the event loop instance created in the initialization process.

If a reaction on application quit is required, please set `set_quit_callback` in the used event loop. This can be done by:

```
loop = App.get_event_loop()
loop.set_quit_callback(callback_function)
```

Do not instantiate the `App` **class!** It is designed to be used in a purely static way.

3.1.1 Application configuration

Configuration of the application is saved in the `GlobalConfiguration` class. This class can be created before `App.initialize()` is called and passed in as a parameter. This way the same configuration can be used between re-initialization of the application. The configuration can easily be changed, even while an application is running, by setting desired properties.

Look at the `GlobalConfiguration` to find out all the configuration possibilities.

3.1.2 App class

class simpleline.**App**

This is the main class for Simpleline library.

Do not create instance of this class. Use this class as static! The *initialize()* method must be called before use.

It is giving you access to the scheduler and event loop. You can have only one instance of this class in your application.

To create this instance call *App.initialize()* method. This method can also be used to reset settings in the App class to start with new event loop or scheduler.

classmethod get_configuration()

Get application defaults configuration object.

classmethod get_event_loop()

Get instance of class responsible for processing asynchronous events.

classmethod get_scheduler()

Get instance of class responsible for rendering of the screen.

classmethod initialize(*scheduler=None*, *event_loop=None*, *global_configuration=None*)

Create app instance inside of this class.

This method can be called multiple times to reset App settings.

Parameters

- **scheduler** (instance of *simpleline.render.screen_scheduler.ScreenScheduler*.) – scheduler used for rendering screens; if not specified use *simpleline.render.screen_scheduler.ScreenScheduler*.
- **event_loop** (object based on class *simpleline.event_loop.AbstractEventLoop*.) – event loop used for asynchronous tasks; if not specified use *simpleline.event_loop.main_loop.MainLoop*.
- **global_configuration** (object based on class *simpleline.global_configuration.GlobalConfiguration*) – instance of the global configuration object; if not specified use *simpleline.global_configuration.GlobalConfiguration*

classmethod is_initialized()

Is the App already initialized?

Returns True if the *App.initialized()* method was called, False otherwise.

classmethod run()

Run event loop.

Raise an exception if no screen is scheduled. This behavior can be changed by *should_run_with_empty_stack* global configuration option.

This is shortcut to *App.event_loop().run()*. :raises NothingScheduledError: when there is no screen scheduled

3.1.3 GlobalConfiguration class

class simpleline.global_configuration.**GlobalConfiguration**

Class for global configuration of application defaults.

All stored data are persistent between App.initialize() calls and can be used before this call.

clear_password_function()
Clear user defined password function and set the default.

Default: getpass.getpass function

clear_should_run_with_empty_stack()
Clear user defined test to run with an empty screen stack.

Default: False

clear_width()
Clear user defined width and set the default.

Default: 80 characters

password_function
Get function to get user passwords from a console.

Returns Function with one argument which is text representation of prompt.

should_run_with_empty_stack
Should test on empty screen stack when starting event loop.

Returns If False the App.run() call will end with an exception (default), True otherwise.

width
Get width of the application.

Returns int

3.2 UIScreen

The base class for creating a new screen. *UIScreen* is used for any user interaction. *UIScreen* uses *containers* and *widgets* to present information to a user, and the *UIScreen.input()* method to get input from a user to an application. Screens are pushed to the screen stack, which are then used to communicate with a user. See the *screen handling* section to find out more.

The methods *UIScreen.redraw()*, *UIScreen.close()*, *UIScreen.emit()* and *UIScreen.create_and_emit()* are asynchronous. These methods will create a signal which is passed to the event loop for later processing.

Beware, methods *UIScreen.redraw()* and *UIScreen.close()* can lead to unexpected behavior when multiple instances of these signals are emitted (methods are called multiple times). This can even crash your application.

3.2.1 Lifecycle

Every *UIScreen* has a distinct lifecycle: *uninitialized*, *initialized*, *draw*, *process input* and *closed*. If the screen has already been initialized (it was *drawn* to a monitor) and will be shown again, then the screen skips the *uninitialized* stage. The screen can cycle between *draw* and *process input* stages by calling the *UIScreen.redraw()* method after processing user input.

In case *UIScreen.redraw()* won't be called and no new screen is pushed, or this screen wasn't closed, then an application will stay in an infinite loop waiting for something to happen. This is correct behavior because there could be something in an event loop which will call *UIScreen.redraw()* later.

3.2.2 Rendering widgets

The *UIScreen.refresh()* method is the most important part of the *UIScreen*. It contains preparations for rendering (creating widgets and adding them to containers). The *UIScreen.refresh()* method will be called before anything is drawn on a monitor.

The *UIScreen.window* attribute, which is the *WindowContainer* instance, contains all items (widgets, containers) which are to be rendered by the screen. A new *WindowContainer* is created in the *UIScreen.refresh()* method for every screen redraw. The *UIScreen.title* attribute is passed to the *WindowContainer.title* property, and a developer can add *widgets* and other *containers* to present items to a user by calling *WindowContainer.add()* or *WindowContainer.add_with_separator()*. Multiple items can be added by calling these methods repeatedly.

When everything is prepared properly in the *UIScreen.refresh()* method, it needs to be drawn on the monitor for a user. This is handled by the *UIScreen.show_all()* method. This method works automatically, but it could also be useful for a developer, especially when the screen will not process input. In this case, the *UIScreen.input()* method is not called at all. Developers can override this method and call the parent class's *UIScreen.show_all()*, which will handle drawing the screen and any additional processing.

Redrawing the screen can be invoked by the *UIScreen.redraw()* method. However, this is not processed immediately. Instead it will be added to the event loop and processed later, when the loop is idle. Beware, after every *redraw* call the input is processed if not disabled, so if multiple redraw signals are emitted, than the application will crash.

The *UIScreen.redraw()* method is also invoked when a screen is *pushed* to the stack.

3.2.3 User input processing

If the screen shouldn't process user input, the *UIScreen.input_required* property needs to be set to *False*. *True* is the default value for this property.

After everything is printed to a monitor, the *UIScreen* will wait for user input. For this purpose there is *UIScreen.input()*, which is called when a user passes string input to a screen. The screen needs to react upon user input and return one of the options from the *InputState* enum or the user input string.

To accept the user input, *InputState.PROCESSED*, *InputState.PROCESSED_AND_REDRAW* or *InputState.PROCESSED_AND_CLOSE* should be returned. Addition to accepting user input the *InputState.PROCESSED_AND_REDRAW* value will also redraw active screen and *InputState.PROCESSED_AND_CLOSE* will close active screen. However, if *InputState.PROCESSED* is used then the developer is responsible for not ending in frozen application. The *UIScreen.refresh()* or the *UIScreen.close()* methods must be called manually.

In case the user input is invalid, the *InputState.DISCARDED* value should be returned. This will reject the user input and wait for another attempt. The *UIScreen.refresh* method will be called after 5 rejections and show the screen output again.

If the user input string is returned it will be checked for options of the *Prompt* instance which can either close the screen, refresh the screen (this will call *UIScreen.refresh()*) or quit the application.

3.2.4 Closing screens

There are several ways to close a screen. One is by calling *UIScreen.close()* or by pressing *c* to continue (with the default *Prompt* class). When the screen is closed the next screen on the stack will be shown. A screen can also be *replaced*. Then the original screen is removed from the stack without closing a screen. If a reaction on closing a screen is required then the *UIScreen.closed()* callback should be overridden.

Beware, when calling *UIScreen.close()* multiple times it will close multiple screens. This is because the close signal will always close the top screen on the screen stack.

3.2.5 UIScreen class

class simpleline.render.screen.**UIScreen**(*title=None*, *screen_height=30*)
Base class representing one TUI Screen.

Shares some API with anaconda's GUI to make it easy for devs to create similar UI with the familiar API.

close()
Emit signal to close this screen.

Add CloseScreenSignal to the event loop.

closed()
Callback when this screen is closed.

connect(*signal*, *callback*, *data=None*)
Connect this class method with given signal.

Parameters

- **signal** (class based on *simpleline.event_loop.AbstractSignal*) – signal class which you want to connect
- **callback** (*func(event_message, data)*) – the callback function
- **data** (*Anything*) – Data you want to pass to the callback

create_and_emit(*signal*)
Create the signal and emit it.

This is basically shortcut for calling *self.create_signal* and *self.emit*.

create_signal(*signal_class*, *priority=0*)
Create signal instance usable in the emit method.

Parameters

- **signal_class** (class based on *simpleline.event_loop.AbstractSignal*) – signal you want to use
- **priority** (*int*) – priority of the signal; please look on the *simpleline.event_loop.AbstractSignal.priority* for further info

emit(*signal*)
Emit the signal.

This will add *signal* to the event loop.

Parameters **signal** (instance of class based on *simpleline.event_loop.AbstractSignal*) – signal to emit

get_input_with_error_check(*args*)
Get user input and redraw if user add too many invalid inputs.

This method should be used only by ScreenScheduler.

Parameters **args** (*Anything.*) – Arguments passed in when scheduling this screen.

get_user_input(*message*, *hidden=False*)
Get immediately input from the user.

Use this with cautious. Never call this in middle of rendering or when other input is already waiting. It is recommended to use *self.input_required* instead.

Parameters

- **message** (*str*) – Message prompt for the user.
- **hidden** (*bool*) – Do not echo user input (password typing).

hide_user_input
Hide typed user input.

This is main solution how to ask for password.

Returns True if user input should be hidden. False otherwise (default).

input(*args*, *key*)
Method called to process input. If the input is not handled here, return it.

Parameters

- **key** (*str*) – input string to process
- **args** (*anything*) – optional argument passed from switch_screen calls

Returns return *simpleline.render.InputState.PROCESSED* if key was handled, *simpleline.render.InputState.DISCARDED* if the screen should not process input on the scheduler and key if you want it to.

Return type *simpleline.render.InputState* enum | str

input_required
Return if the screen requires input.

no_separator
Should we print separator for this screen?

Returns True to print separator before this screen (default). False do not print separator.

password_func
Get password function.

This is function with one argument to get password from command line.

prompt(*args=None*)
Return the text to be shown as prompt or handle the prompt and return None.

Parameters **args** (*anything*) – optional argument passed from switch_screen calls

Returns returns an instance of Prompt with text to be shown next to the prompt for input or None to skip further input processing

Return type Prompt instance|None

redraw()
Emit signal to initiate draw.

Add RenderScreenSignal to the event loop.

refresh(*args=None*)
Method which prepares the content desired on the screen to *self.window*.

Parameters **args** (*anything*) – optional argument passed from switch_screen calls

screen_ready
This screen is ready for use.

setup(*args*)
Do additional setup right before this screen is used.

It is mandatory to call this ancestor method in the child class to set ready status.

Parameters **args** (*array of values*) – arguments for the setup

Returns whether this screen should be scheduled or not

Return type bool

show_all()
Print WindowContainer in *self.window* with all its content.

title
Screen title.

window
Return WindowContainer instance.

3.2.6 InputState enum

class simpleline.render.screen.**InputState**
An enumeration.

DISCARDED = 0

PROCESSED = 1

PROCESSED_AND_CLOSE = 3

PROCESSED_AND_REDRAW = 2

3.3 Screen Handling

The *ScreenHandler* class is used to schedule a *UIScreen* to the screen stack.

Screen handling is an important part of using the Simpleline library, and it is recommended for a developer to get familiar with this principle. Screen handling is the gateway by which a developer manages the stack, by either adding or removing screens from it. There are many ways to add a screen to the stack. To remove a screen from the stack, the screen must be closed, or it can be replaced by another screen. To close a *UIScreen* a developer should call the *UIScreen.close()* method. A screen can also be closed when a user presses *c* (which can be disabled). This will close the screen automatically. When a screen is closed, the next screen on the top of the stack will be rendered. If the stack is empty then the application will close.

Do not instantiate the *ScreenHandler* **class!** All methods in the *ScreenHandler* class are class methods so the *ScreenHandler* shouldn't be instantiated at all.

The following operations can be used to schedule a screen.

3.3.1 Schedule screen

To schedule a screen use the *ScreenHandler.schedule_screen()* method. Scheduling a screen should be used on the first screen in your application before starting the event loop. The screen is added to the bottom of the screen stack, and it will be visible as the last screen in a stack.

This is the only way to add a screen to the screen stack without emitting a redraw call.

3.3.2 Push screen

To push a screen to the stack use the *ScreenHandler.push_screen()* method. Pushing a screen to the stack will place a screen on top of the stack, so it will be drawn on the next redraw call. The original screen will remain on the stack, so after the new screen is closed, the original screen will be rendered again. If you need to avoid this behavior please close the active screen before pushing a new screen to the stack.

The redraw signal will be emitted automatically when a screen is pushed.

3.3.3 Push screen as modal

To push a modal screen to the stack use the *ScreenHandler.push_screen_modal()* method. This behaves the same as *Push screen* with one important difference. The pushed screen will act as a modal screen. A modal screen has its own event loop, so event processing of the old loop is blocked until the modal screen is closed. The code processing is also blocked by the *ScreenHandler.push_screen_modal()* method call.

The *UIScreen.redraw()* method may be required if the original screen was already drawn before invoking the `ScreenHandler.push_screen_method()` method.

The redraw signal will be emitted automatically when a screen is pushed.

3.3.4 Replace screen

Replace an existing screen with a new screen. This behaves like *Push screen* but it replaces the original screen, so there is no need to close the original screen.

The redraw signal will be emitted automatically when a screen is replaced.

3.3.5 ScreenHandler class

class simpleline.render.screen_handler.**ScreenHandler**

classmethod push_screen(*ui_screen*, *args=None*)
Schedule screen to the active scheduler.

See: *simpleline.render.screen_scheduler.push_screen()*.

classmethod push_screen_modal(*ui_screen*, *args=None*)
Schedule screen to the active scheduler.

See: *simpleline.render.screen_scheduler.push_screen_modal()*.

classmethod replace_screen(*ui_screen*, *args=None*)
Schedule screen to the active scheduler.

See: *simpleline.render.screen_scheduler.replace_screen()*.

classmethod schedule_screen(*ui_screen*, *args=None*)
Schedule screen to the active scheduler.

See: *simpleline.render.screen_scheduler.schedule_screen()*.

3.4 Widgets

Widgets are the basic units to render items on a *screen*. Widgets can wrap a common text (*TextWidget*) or create empty lines (*SeparatorWidget*). They can also be more complex structures (*CheckboxWidget*). A new widget can also be created. See *Creating a custom widget*.

These widgets should be used in *Containers*.

3.4.1 Widgets classes

class simpleline.render.widgets.**TextWidget**(*text*)
Bases: *simpleline.render.widgets.Widget*

Class to handle wrapped text output.

clear()
Clears this widgets buffer and resets cursor.

content
Return a list (rows) of lists (columns) with one character elements.

draw(*w*, *row=None*, *col=None*, *block=False*)
Copy w widget's content to this widget's buffer at row, col position.

Parameters

- **w** (*class Widget*) – widget to take content from
- **row** (*int*) – row number to start at (default is at the cursor position)
- **col** (*int*) – column number to start at (default is at the cursor position)
- **block** (*boolean*) – when printing newline, start at column col (True) or at column 0 (False)

get_lines()
Return lines to write out in order to show this widget.

Returns lines representing this widget

Return type list(str)

height
The current height of the internal buffer.

render(*width*)
Renders the text widget limited to width number of columns.

Wraps to the next line when the text is longer.

Parameters **width** (*int*) – maximum width allocated to the string

set_cursor_position(*row*, *col*)
Set cursor position.

Parameters

- **row** (*int*) – row id, starts with 0 at the top of the screen
- **col** (*int*) – column id, starts with 0 on the left side of the screen

set_end()
Set the cursor to first column in new line at the end.

text
Contains text of this widget.

width
The current width of the internal buffer (id of the first empty column).

write(*text*, *row=None*, *col=None*, *width=None*, *block=False*, *wordwrap=False*)
Emulate the typing machine writing to this widget's buffer.

Parameters

- **text** (*str*) – text to type
- **row** (*int*) – row number to start at (default is at the cursor position)
- **col** (*int*) – column number to start at (default is at the cursor position)
- **width** (*int*) – wrap at "col" + "width" column (default is at self._max_width)
- **block** (*boolean*) – when printing newline, start at column col (True) or at column 0 (False)
- **wordwrap** (*boolean*) – wrap by words

class simpleline.render.widgets.**SeparatorWidget**(*lines=1*)
Bases: *simpleline.render.widgets.Widget*

Print empty line.

clear()
Clears this widgets buffer and resets cursor.

content
Return a list (rows) of lists (columns) with one character elements.

draw(*w*, *row=None*, *col=None*, *block=False*)
Copy w widget's content to this widget's buffer at row, col position.

Parameters

- **w** (*class Widget*) – widget to take content from
- **row** (*int*) – row number to start at (default is at the cursor position)
- **col** (*int*) – column number to start at (default is at the cursor position)
- **block** (*boolean*) – when printing newline, start at column col (True) or at column 0 (False)

get_lines()
Return lines to write out in order to show this widget.

Returns lines representing this widget

Return type list(str)

height
The current height of the internal buffer.

render(*width*)
Render empty line to the buffer.

Parameters **width** (*int*) – maximum width allocated to the string

set_cursor_position(*row*, *col*)
Set cursor position.

Parameters

- **row** (*int*) – row id, starts with 0 at the top of the screen
- **col** (*int*) – column id, starts with 0 on the left side of the screen

set_end()
Set the cursor to first column in new line at the end.

width
The current width of the internal buffer (id of the first empty column).

write(*text*, *row=None*, *col=None*, *width=None*, *block=False*, *wordwrap=False*)
Optimize write function.

To print just a blank line we don't need too much logic.

class simpleline.render.widgets.**EntryWidget**(*title*, *value=None*)
Bases: *simpleline.render.widgets.TextWidget*

This is the easy way how to generate entry items for containers.

If the numbering in a container is turned on the output looks like:

N) title value

Without numbering turned on:

title value

clear()
Clears this widgets buffer and resets cursor.

content
Return a list (rows) of lists (columns) with one character elements.

draw(*w*, *row=None*, *col=None*, *block=False*)
Copy w widget's content to this widget's buffer at row, col position.

Parameters

- **w** (*class Widget*) – widget to take content from
- **row** (*int*) – row number to start at (default is at the cursor position)
- **col** (*int*) – column number to start at (default is at the cursor position)
- **block** (*boolean*) – when printing newline, start at column col (True) or at column 0 (False)

get_lines()
Return lines to write out in order to show this widget.

Returns lines representing this widget

Return type list(str)

height
The current height of the internal buffer.

render(*width*)
Renders the text widget limited to width number of columns.

Wraps to the next line when the text is longer.

Parameters **width** (*int*) – maximum width allocated to the string

set_cursor_position(*row*, *col*)
Set cursor position.

Parameters

- **row** (*int*) – row id, starts with 0 at the top of the screen
- **col** (*int*) – column id, starts with 0 on the left side of the screen

set_end()
Set the cursor to first column in new line at the end.

text
Contains text of this widget.

width
The current width of the internal buffer (id of the first empty column).

write(*text*, *row=None*, *col=None*, *width=None*, *block=False*, *wordwrap=False*)
Emulate the typing machine writing to this widget's buffer.

Parameters

- **text** (*str*) – text to type
- **row** (*int*) – row number to start at (default is at the cursor position)
- **col** (*int*) – column number to start at (default is at the cursor position)
- **width** (*int*) – wrap at "col" + "width" column (default is at self._max_width)
- **block** (*boolean*) – when printing newline, start at column col (True) or at column 0 (False)
- **wordwrap** (*boolean*) – wrap by words

class simpleline.render.widgets.**CheckboxWidget**(*key='x'*, *title=None*, *text=None*, *completed=None*)
Bases: *simpleline.render.widgets.Widget*

Widget to show checkbox with (un)checked box, name and description.

clear()
Clears this widgets buffer and resets cursor.

completed
Returns the state of the checkbox, checked is True.

content
Return a list (rows) of lists (columns) with one character elements.

draw(*w*, *row=None*, *col=None*, *block=False*)
Copy w widget's content to this widget's buffer at row, col position.

Parameters

- **w** (*class Widget*) – widget to take content from
- **row** (*int*) – row number to start at (default is at the cursor position)
- **col** (*int*) – column number to start at (default is at the cursor position)
- **block** (*boolean*) – when printing newline, start at column col (True) or at column 0 (False)

get_lines()
Return lines to write out in order to show this widget.

Returns lines representing this widget

Return type list(str)

height
The current height of the internal buffer.

render(*width*)
Render the widget to internal buffer.

It should be max width characters wide.

set_cursor_position(*row*, *col*)
Set cursor position.

Parameters

- **row** (*int*) – row id, starts with 0 at the top of the screen
- **col** (*int*) – column id, starts with 0 on the left side of the screen

set_end()
Set the cursor to first column in new line at the end.

text
Contains the description text from the second line.

title
Returns the first line (main title) of the checkbox.

width
The current width of the internal buffer (id of the first empty column).

write(*text*, *row=None*, *col=None*, *width=None*, *block=False*, *wordwrap=False*)
Emulate the typing machine writing to this widget's buffer.

Parameters

- **text** (*str*) – text to type
- **row** (*int*) – row number to start at (default is at the cursor position)
- **col** (*int*) – column number to start at (default is at the cursor position)
- **width** (*int*) – wrap at "col" + "width" column (default is at self._max_width)
- **block** (*boolean*) – when printing newline, start at column col (True) or at column 0 (False)
- **wordwrap** (*boolean*) – wrap by words

class simpleline.render.widgets.**CenterWidget**(*w*)
Bases: *simpleline.render.widgets.Widget*

Class to handle horizontal centering of content.

clear()
Clears this widgets buffer and resets cursor.

content
Return a list (rows) of lists (columns) with one character elements.

draw(*w*, *row=None*, *col=None*, *block=False*)
Copy w widget's content to this widget's buffer at row, col position.

Parameters

- **w** (*class Widget*) – widget to take content from
- **row** (*int*) – row number to start at (default is at the cursor position)
- **col** (*int*) – column number to start at (default is at the cursor position)
- **block** (*boolean*) – when printing newline, start at column col (True) or at column 0 (False)

get_lines()
Return lines to write out in order to show this widget.

Returns lines representing this widget

Return type list(str)

height
The current height of the internal buffer.

render(*width*)
Render the centered widget to internal buffer.

Parameters **width** (*int*) – maximum width the widget should use

set_cursor_position(*row*, *col*)
Set cursor position.

Parameters

- **row** (*int*) – row id, starts with 0 at the top of the screen
- **col** (*int*) – column id, starts with 0 on the left side of the screen

set_end()
Set the cursor to first column in new line at the end.

width
The current width of the internal buffer (id of the first empty column).

write(*text*, *row=None*, *col=None*, *width=None*, *block=False*, *wordwrap=False*)
Emulate the typing machine writing to this widget's buffer.

Parameters

- **text** (*str*) – text to type
- **row** (*int*) – row number to start at (default is at the cursor position)
- **col** (*int*) – column number to start at (default is at the cursor position)
- **width** (*int*) – wrap at "col" + "width" column (default is at self._max_width)
- **block** (*boolean*) – when printing newline, start at column col (True) or at column 0 (False)
- **wordwrap** (*boolean*) – wrap by words

3.4.2 Creating a custom widget

To create a custom widget you should subclass the *Widget* class. The most important method for creating a customized widget is *Widget.render()*. This method is responsible for presenting a user textual information.

If the new widget is composed of other widgets the *Widget.draw()* method should be called on every widget used. The *Widget.write()* method should be called if the input is a string. These calls can be repeated multiple times. For an example please look at the existing implementation.

3.4.3 Base Widget class

class simpleline.render.widgets.**Widget**(*max_width=None*, *default=None*)
Bases: object

clear()
Clears this widgets buffer and resets cursor.

content
Return a list (rows) of lists (columns) with one character elements.

draw(*w*, *row=None*, *col=None*, *block=False*)
Copy w widget's content to this widget's buffer at row, col position.

Parameters

- **w** (*class Widget*) – widget to take content from
- **row** (*int*) – row number to start at (default is at the cursor position)
- **col** (*int*) – column number to start at (default is at the cursor position)
- **block** (*boolean*) – when printing newline, start at column col (True) or at column 0 (False)

get_lines()
Return lines to write out in order to show this widget.

Returns lines representing this widget

Return type list(str)

height
The current height of the internal buffer.

render(*width*)
Redraw the widget's self._buffer.

Parameters **width** (*int*) – the width of buffer requested by the caller

Commonly, call render of child widgets and then draw and write methods to copy their contents to self._buffer.

set_cursor_position(*row*, *col*)
Set cursor position.

Parameters

- **row** (*int*) – row id, starts with 0 at the top of the screen
- **col** (*int*) – column id, starts with 0 on the left side of the screen

set_end()
Set the cursor to first column in new line at the end.

width
The current width of the internal buffer (id of the first empty column).

write(*text*, *row=None*, *col=None*, *width=None*, *block=False*, *wordwrap=False*)
Emulate the typing machine writing to this widget's buffer.

Parameters

- **text** (*str*) – text to type
- **row** (*int*) – row number to start at (default is at the cursor position)

- **col** (*int*) – column number to start at (default is at the cursor position)
- **width** (*int*) – wrap at "col" + "width" column (default is at self._max_width)
- **block** (*boolean*) – when printing newline, start at column col (True) or at column 0 (False)
- **wordwrap** (*boolean*) – wrap by words

3.5 Containers

Containers are structures to hold widgets and handle automatic positioning. Containers are essentially widgets containing other widgets. Below is a collection of default containers which can position *widgets*, but they can do more (e.g. handle user input). Recursive composition of containers is supported.

Customized containers can also be created. See the *Creating a custom container* section.

3.5.1 Container classes

class simpleline.render.containers.**ListRowContainer**(*columns*, *items=None*, *columns_width=None*, *spacing=3*, *numbering=True*)

Bases: *simpleline.render.containers.Container*

Place widgets in rows automatically.

Compared to the ColumnWidget this is able to handle word wrapping correctly.

There is numbering N) automatically for all items. To disable this feature call *self.key_pattern = None*. If you want other numbering then look on *KeyPattern* class.

Widgets will be placed based on the number of columns in the following way:

1) w1 2) w2 3) w3 4) w4 5) w5 6) w6 ....

add(*item*, *callback=None*, *data=None*)

Add item to the Container.

Parameters

- **item** (Could be item (based on *simpleline.render.widgets.Widget*) or other container (based on *simpleline.render.containers.Container*).) – Add item to this container.
- **callback** (function `func(data).`) – Add callback for this item. This callback will be called when user activate this *item*.
- **data** – Data which will be passed to the callback.
- **data** – Anything.

Returns ID of the item in this Container.

Return type int

clear()

Clears this widgets buffer and resets cursor.

content

Return a list (rows) of lists (columns) with one character elements.

create_number_label(*item_id*)

Create TextWidget from KeyPattern.

Parameters **item_id** (*int*) – Create label for item with this id.

Returns Widget with label for the item with item_id.

Return type *simpleline.render.widgets.TextWidget* instance.

draw(*w*, *row=None*, *col=None*, *block=False*)
Copy w widget's content to this widget's buffer at row, col position.

Parameters

- **w** (*class Widget*) – widget to take content from
- **row** (*int*) – row number to start at (default is at the cursor position)
- **col** (*int*) – column number to start at (default is at the cursor position)
- **block** (*boolean*) – when printing newline, start at column col (True) or at column 0 (False)

get_lines()
Return lines to write out in order to show this widget.

Returns lines representing this widget

Return type list(str)

height
The current height of the internal buffer.

key_pattern
Return key pattern which will be used for items numbering.

Will return *None* if not set.

process_user_input(*key*)
Process input from the user if any of the items in the list was called.

This method must be called in *UIScreen.input()* method if list widget should call the callbacks.

Parameters **key** (*str*) – Key pressed from user.

Returns True if key was processed. False otherwise.

render(*width*)
Render widgets to it's internal buffer.

Parameters **width** (*int*) – the maximum width the item can use

Returns nothing

set_cursor_position(*row*, *col*)
Set cursor position.

Parameters

- **row** (*int*) – row id, starts with 0 at the top of the screen
- **col** (*int*) – column id, starts with 0 on the left side of the screen

set_end()
Set the cursor to first column in new line at the end.

size
Return items count.

width
The current width of the internal buffer (id of the first empty column).

write(*text*, *row=None*, *col=None*, *width=None*, *block=False*, *wordwrap=False*)
Emulate the typing machine writing to this widget's buffer.

Parameters

- **text** (*str*) – text to type
- **row** (*int*) – row number to start at (default is at the cursor position)
- **col** (*int*) – column number to start at (default is at the cursor position)
- **width** (*int*) – wrap at "col" + "width" column (default is at self._max_width)
- **block** (*boolean*) – when printing newline, start at column col (True) or at column 0 (False)
- **wordwrap** (*boolean*) – wrap by words

class simpleline.render.containers.**ListColumnContainer**(*columns*, *items=None*, *columns_width=None*, *spacing=3*, *numbering=True*)
Bases: *simpleline.render.containers.ListRowContainer*

Place widgets in columns automatically.

Compared to the ColumnWidget this is able to handle word wrapping correctly.

There is numbering N) automatically for all items. To disable this feature call *self.key_pattern = None*. If you want other numbering then look on *KeyPattern* class.

Widgets will be placed based on the number of columns in the following way:

1) w1 4) w4 7) w7
2) w2 5) w5 8) w8
3) w3 6) w6 9) w9

add(*item*, *callback=None*, *data=None*)
Add item to the Container.

Parameters

- **item** (Could be item (based on *simpleline.render.widgets.Widget*) or other container (based on *simpleline.render.containers.Container*).) – Add item to this container.
- **callback** (function `func(data).`) – Add callback for this item. This callback will be called when user activate this *item*.
- **data** – Data which will be passed to the callback.
- **data** – Anything.

Returns ID of the item in this Container.

Return type int

clear()
Clears this widgets buffer and resets cursor.

content
Return a list (rows) of lists (columns) with one character elements.

create_number_label(*item_id*)
Create TextWidget from KeyPattern.

Parameters **item_id** (*int*) – Create label for item with this id.

Returns Widget with label for the item with item_id.

Return type *simpleline.render.widgets.TextWidget* instance.

draw(*w*, *row=None*, *col=None*, *block=False*)
Copy w widget's content to this widget's buffer at row, col position.

Parameters

- **w** (*class Widget*) – widget to take content from
- **row** (*int*) – row number to start at (default is at the cursor position)
- **col** (*int*) – column number to start at (default is at the cursor position)
- **block** (*boolean*) – when printing newline, start at column col (True) or at column 0 (False)

get_lines()
Return lines to write out in order to show this widget.

Returns lines representing this widget

Return type list(str)

height
The current height of the internal buffer.

key_pattern
Return key pattern which will be used for items numbering.

Will return *None* if not set.

process_user_input(*key*)
Process input from the user if any of the items in the list was called.

This method must be called in *UIScreen.input()* method if list widget should call the callbacks.

Parameters **key** (*str*) – Key pressed from user.

Returns True if key was processed. False otherwise.

render(*width*)
Render widgets to it's internal buffer.

Parameters **width** (*int*) – the maximum width the item can use

Returns nothing

set_cursor_position(*row*, *col*)
Set cursor position.

Parameters

- **row** (*int*) – row id, starts with 0 at the top of the screen
- **col** (*int*) – column id, starts with 0 on the left side of the screen

set_end()
Set the cursor to first column in new line at the end.

size
Return items count.

width
The current width of the internal buffer (id of the first empty column).

write(*text*, *row=None*, *col=None*, *width=None*, *block=False*, *wordwrap=False*)
Emulate the typing machine writing to this widget's buffer.

Parameters

- **text** (*str*) – text to type
- **row** (*int*) – row number to start at (default is at the cursor position)
- **col** (*int*) – column number to start at (default is at the cursor position)
- **width** (*int*) – wrap at "col" + "width" column (default is at self._max_width)
- **block** (*boolean*) – when printing newline, start at column col (True) or at column 0 (False)
- **wordwrap** (*boolean*) – wrap by words

class simpleline.render.containers.**WindowContainer**(*title=None*)
Bases: *simpleline.render.containers.Container*

Base container for screens.

This can hold other containers or Widgets for rendering.

add(*item*, *callback=None*, *data=None*)
Add item to the Container.

Parameters

- **item** (Could be item (based on *simpleline.render.widgets.Widget*) or other container (based on *simpleline.render.containers.Container*).) – Add item to this container.
- **callback** (function `func(data)`.) – Add callback for this item. This callback will be called when user activate this *item*.
- **data** – Data which will be passed to the callback.
- **data** – Anything.

Returns ID of the item in this Container.

Return type int

add_separator(*lines=1*)
Add blank lines between widgets.

Parameters **lines** (*int greater than 0.*) – How many blank lines should be printed.

add_with_separator(*item*, *callback=None*, *data=None*, *blank_lines=1*)
Add widget and after widget add blank line.

This method will call *self.add(item, callback, data) self.add_separator(lines)*

Parameters

- **item** (Could be item (based on *simpleline.render.widgets.Widget*) or other container (based on *simpleline.render.containers.Container*).) – Add item to this container.
- **callback** (function `func(data)`.) – Add callback for this item. This callback will be called when user activate this *item*.
- **data** – Data which will be passed to the callback.
- **data** – Anything.
- **blank_lines** (*int greater than 0.*) – How many blank lines should be printed.

Returns ID of the item in this Container.

Return type int

clear()
Clears this widgets buffer and resets cursor.

content
Return a list (rows) of lists (columns) with one character elements.

create_number_label(*item_id*)
Create TextWidget from KeyPattern.

Parameters **item_id** (*int*) – Create label for item with this id.

Returns Widget with label for the item with item_id.

Return type *simpleline.render.widgets.TextWidget* instance.

draw(*w*, *row=None*, *col=None*, *block=False*)
Copy w widget's content to this widget's buffer at row, col position.

Parameters

- **w** (*class Widget*) – widget to take content from
- **row** (*int*) – row number to start at (default is at the cursor position)
- **col** (*int*) – column number to start at (default is at the cursor position)
- **block** (*boolean*) – when printing newline, start at column col (True) or at column 0 (False)

get_lines()
Return lines to write out in order to show this widget.

Returns lines representing this widget

Return type list(str)

height
The current height of the internal buffer.

key_pattern
Return key pattern which will be used for items numbering.

Will return *None* if not set.

process_user_input(*key*)
Process input from the user if any of the items in the list was called.

This method must be called in *UIScreen.input()* method if list widget should call the callbacks.

Parameters **key** (*str*) – Key pressed from user.

Returns True if key was processed. False otherwise.

render(*width*)
Render widgets to it's internal buffer.

Parameters **width** (*int*) – the maximum width the item can use

Returns nothing

set_cursor_position(*row*, *col*)
Set cursor position.

Parameters

- **row** (*int*) – row id, starts with 0 at the top of the screen
- **col** (*int*) – column id, starts with 0 on the left side of the screen

set_end()
Set the cursor to first column in new line at the end.

size
Return items count.

title
Title of WindowContainer.

width
The current width of the internal buffer (id of the first empty column).

write(*text*, *row=None*, *col=None*, *width=None*, *block=False*, *wordwrap=False*)
Emulate the typing machine writing to this widget's buffer.

Parameters

- **text** (*str*) – text to type
- **row** (*int*) – row number to start at (default is at the cursor position)
- **col** (*int*) – column number to start at (default is at the cursor position)
- **width** (*int*) – wrap at "col" + "width" column (default is at self._max_width)
- **block** (*boolean*) – when printing newline, start at column col (True) or at column 0 (False)
- **wordwrap** (*boolean*) – wrap by words

3.5.2 Creating a custom container

If an existing container is missing a required feature, a new, customized container can be created based on the *Container* class. Container creation is essentially the same as *custom widget creation* because containers are based on widgets. The main difference is that containers are working with widgets added by a developer when using the container. As an example, every *UIScreen* has a *WindowContainer*, and this is used as the main rendering point.

To create a customized container, the *Container.render()* method should be overridden and the positioning of widgets and containers should be done here. It can even enhance these widgets, for example, by adding numbering (this is done in *ListRowContainer* or *ListColumnContainer*). The *Container.render()* method should call the *Container.draw()* method. The *Container.draw()* method should be called for every widget placed. For a better understanding please refer to the existing implementation.

3.5.3 Base container class

class simpleline.render.containers.**Container**(*items=None*, *numbering=True*)
Bases: *simpleline.render.widgets.Widget*

Base class for containers which will do positioning of the widgets.

add(*item*, *callback=None*, *data=None*)
Add item to the Container.

Parameters

- **item** (Could be item (based on *simpleline.render.widgets.Widget*) or other container (based on *simpleline.render.containers.Container*).) – Add item to this container.
- **callback** (function `func(data)`.) – Add callback for this item. This callback will be called when user activate this *item*.
- **data** – Data which will be passed to the callback.
- **data** – Anything.

Returns ID of the item in this Container.

Return type int

clear()
Clears this widgets buffer and resets cursor.

content
Return a list (rows) of lists (columns) with one character elements.

create_number_label(*item_id*)
Create TextWidget from KeyPattern.

Parameters **item_id** (*int*) – Create label for item with this id.

Returns Widget with label for the item with item_id.

Return type *simpleline.render.widgets.TextWidget* instance.

draw(*w*, *row=None*, *col=None*, *block=False*)
Copy w widget's content to this widget's buffer at row, col position.

Parameters

- **w** (*class Widget*) – widget to take content from
- **row** (*int*) – row number to start at (default is at the cursor position)
- **col** (*int*) – column number to start at (default is at the cursor position)
- **block** (*boolean*) – when printing newline, start at column col (True) or at column 0 (False)

get_lines()
Return lines to write out in order to show this widget.

Returns lines representing this widget

Return type list(str)

height
The current height of the internal buffer.

key_pattern
Return key pattern which will be used for items numbering.

Will return *None* if not set.

process_user_input(*key*)
Process input from the user if any of the items in the list was called.

This method must be called in *UIScreen.input()* method if list widget should call the callbacks.

Parameters **key** (*str*) – Key pressed from user.

Returns True if key was processed. False otherwise.

render(*width*)
Redraw the widget's self._buffer.

Parameters **width** (*int*) – the width of buffer requested by the caller

Commonly, call render of child widgets and then draw and write methods to copy their contents to self._buffer.

set_cursor_position(*row*, *col*)
Set cursor position.

Parameters

- **row** (*int*) – row id, starts with 0 at the top of the screen
- **col** (*int*) – column id, starts with 0 on the left side of the screen

set_end()
Set the cursor to first column in new line at the end.

size
Return items count.

width
The current width of the internal buffer (id of the first empty column).

write(*text*, *row=None*, *col=None*, *width=None*, *block=False*, *wordwrap=False*)
Emulate the typing machine writing to this widget's buffer.

Parameters

- **text** (*str*) – text to type
- **row** (*int*) – row number to start at (default is at the cursor position)
- **col** (*int*) – column number to start at (default is at the cursor position)
- **width** (*int*) – wrap at "col" + "width" column (default is at self._max_width)
- **block** (*boolean*) – when printing newline, start at column col (True) or at column 0 (False)
- **wordwrap** (*boolean*) – wrap by words

3.6 Prompt

Class for prompting a user for input. New user options can be added by *Prompt.add_option()*, removed by *Prompt.remove_option()* or updated by *Prompt.update_option()*. A message for the user can also be set by the *Prompt.set_message()* method.

This class is used in the *UIScreen* class. The default instance always handles *r* (refresh), *c* (continue) and *q* (quit) and is created in the *UIScreen.prompt()* method. To create your own custom prompt please override the *UIScreen.prompt()* method.

3.6.1 Prompt class

class simpleline.render.prompt.**Prompt**(*message='Please make a selection from the above'*)
Class to create a prompt message with options.

add_continue_option(*description='to continue'*)
Add the option to continue.

add_help_option(*description='to help'*)
Add the option to help.

add_option(*key*, *description*)
Add an option to the prompt. Causes a warning if the option already exists.

Parameters

- **key** (*str*) – the key for choosing the option
- **description** (*str*) – the description of the option

add_quit_option(*description='to quit'*)
Add the option to quit.

add_refresh_option(*description='to refresh'*)
Add the option to refresh.

remove_option(*key*)
Remove an option with the given key.

Parameters **key** (*str*) – the key of the option

Returns the removed option

Return type str|None

set_message(*message*)
Set the prompt message.

Parameters **message** (*str|None*) – the message of the prompt

update_option(*key*, *description*)
Update an option in the prompt. Causes a warning if the option does not exist.

Parameters

- **key** (*str*) – the key for choosing the option
- **description** (*str*) – the description of the option

3.7 Advanced Widgets

Advanced widgets are *screens* which can be used for a specific purpose. For example, reading input from the user and testing acceptance conditions on this input, or asking the user a yes/no question.

3.7.1 Advanced widget classes

class simpleline.render.adv_widgets.**ErrorDialog**(*message*)
Bases: *simpleline.render.screen.UIScreen*

Dialog screen for reporting errors to user.

input(*args*, *key*)
This dialog is closed by any input.

And causes the program to quit.

prompt(*args=None*)
Return the text to be shown as prompt or handle the prompt and return None.

Parameters **args** (*anything*) – optional argument passed from switch_screen calls

Returns returns an instance of Prompt with text to be shown next to the prompt for input or None to skip further input processing

Return type Prompt instance|None

refresh(*args=None*)
Method which prepares the content desired on the screen to *self.window*.

Parameters **args** (*anything*) – optional argument passed from switch_screen calls

class simpleline.render.adv_widgets.**GetInputScreen**(*message*)
Bases: *simpleline.render.screen.UIScreen*

Screen for getting user input.

add_acceptance_condition(*acceptance_function*, *args=None*)
Add acceptance condition to the conditions list.

Parameters

- **acceptance_function** (*function(input, args) -> bool* - function which takes user input (string) and arguments (*args*) and return True when input is accepted or False if rejected so we will ask for a new input.) – Functions that accepts or rejects a user input.
- **args** (*Anything.*) – Second argument for *acceptance_function* the first one will be user input.

clear_acceptance_conditions()
Clear list of the acceptance conditions.

input(*args*, *key*)
Method called to process input. If the input is not handled here, return it.

Parameters

- **key** (*str*) – input string to process
- **args** (*anything*) – optional argument passed from switch_screen calls

Returns return *simpleline.render.InputState.PROCESSED* if key was handled, *simpleline.render.InputState.DISCARDED* if the screen should not process input on the scheduler and key if you want it to.

Return type *simpleline.render.InputState* enum | str

prompt(*args=None*)
Return the text to be shown as prompt or handle the prompt and return None.

Parameters **args** (*anything*) – optional argument passed from switch_screen calls

Returns returns an instance of Prompt with text to be shown next to the prompt for input or None to skip further input processing

Return type Prompt instance|None

refresh(*args=None*)
Method which prepares the content desired on the screen to *self.window*.

Parameters **args** (*anything*) – optional argument passed from switch_screen calls

value
User input.

class simpleline.render.adv_widgets.**GetPasswordInputScreen**(*message*)
Bases: *simpleline.render.adv_widgets.GetInputScreen*

Screen for getting user password input.

class simpleline.render.adv_widgets.**HelpScreen**(*help_path*)
Bases: *simpleline.render.screen.UIScreen*

Screen to display a help message.

input(*args*, *key*)
Handle user input.

prompt(*args=None*)
Return the text to be shown as prompt or handle the prompt and return None.

Parameters **args** (*anything*) – optional argument passed from switch_screen calls

Returns returns an instance of Prompt with text to be shown next to the prompt for input or None to skip further input processing

Return type Prompt instance|None

refresh(*args=None*)
Show the help.

class simpleline.render.adv_widgets.**PasswordDialog**(*message=None*)
Bases: *simpleline.render.screen.UIScreen*

Dialog screen for password input.

answer
The response can be None (no response) or the password entered.

input(*args*, *key*)
Method called to process input. If the input is not handled here, return it.

Parameters

- **key** (*str*) – input string to process
- **args** (*anything*) – optional argument passed from switch_screen calls

Returns return *simpleline.render.InputState.PROCESSED* if key was handled, *simpleline.render.InputState.DISCARDED* if the screen should not process input on the scheduler and key if you want it to.

Return type *simpleline.render.InputState* enum | str

prompt(*args=None*)
Return the text to be shown as prompt or handle the prompt and return None.

Parameters **args** (*anything*) – optional argument passed from switch_screen calls

Returns returns an instance of Prompt with text to be shown next to the prompt for input or None to skip further input processing

Return type Prompt instance|None

refresh(*args=None*)
Method which prepares the content desired on the screen to *self.window*.

Parameters **args** (*anything*) – optional argument passed from switch_screen calls

class simpleline.render.adv_widgets.**YesNoDialog**(*message*)
Bases: *simpleline.render.screen.UIScreen*

Dialog screen for Yes - No questions.

answer
The response can be True (yes), False (no) or None (no response).

input(*args*, *key*)
Method called to process input. If the input is not handled here, return it.

Parameters

- **key** (*str*) – input string to process
- **args** (*anything*) – optional argument passed from switch_screen calls

Returns return *simpleline.render.InputState.PROCESSED* if key was handled, *simpleline.render.InputState.DISCARDED* if the screen should not process input on the scheduler and key if you want it to.

Return type *simpleline.render.InputState* enum | str

prompt(*args=None*)
Return the text to be shown as prompt or handle the prompt and return None.

Parameters **args** (*anything*) – optional argument passed from switch_screen calls

Returns returns an instance of Prompt with text to be shown next to the prompt for input or None to skip further input processing

Return type Prompt instance|None

refresh(*args=None*)
Method which prepares the content desired on the screen to *self.window*.

Parameters **args** (*anything*) – optional argument passed from switch_screen calls

3.8 Event Loops

Event loops are the heart of Simpleline. Every event loop is based on the *AbstractEventLoop*, and they all work with *signals*. A signal is a message passed to the loop containing some information. Signals are passed to an event loop by calling *AbstractEventLoop.enqueue_signal()*. These signals are then processed by calling *AbstractEventLoop.process_signals()*. This method can be called by an application developer manually or by the *AbstractEventLoop.run()* method, which is called by *App.run()* to start the Simpleline-based application. When a signal is processed, all handlers attached to this signal are called. Signal handler assignment is done by the *AbstractEventLoop.register_signal_handler()* method.

Event loops can also be started recursively by *AbstractEventLoop.execute_new_loop()*. The old event loop is waiting for this event loop to stop. New loop execution is mandatory for modal screens to work, since they can't be interrupted by other screens. This new event loop is terminated by closing the last screen in the event loop or by calling the *AbstractEventLoop.close_loop()* method.

The last event loop should be terminated by closing the last screen in the screen stack or by calling *AbstractEventLoop.close_loop()*. In case a fatal error occurs the *AbstractEventLoop.force_quit()* method can be used to immediately kill the loop.

If a reaction on quitting the application (closing the last event loop) is required, the quit callback can be used. The quit callback can be set by the *AbstractEventLoop.set_quit_callback()* method.

The following event loops are supported by Simpleline, but you can also *Create your own loop* :

- *Main Loop*
- *GLib Event loop*

3.8.1 Main Loop

The main loop is the default event loop for Simpleline projects. The benefit of using *MainLoop* is that it isn't necessary to have any dependencies on other libraries. It is a lightweight event loop implemented completely in Python.

class simpleline.event_loop.main_loop.**MainLoop**
Bases: *simpleline.event_loop.AbstractEventLoop*

Default main event loop for the Simpleline.

This event loop can be replaced by your event loop by implementing *simpleline.event_loop.AbstractEventLoop* class.

close_loop()
Close active event loop.

Close an event loop created by the *execute_new_loop()* method.

enqueue_signal(*signal*)
Enqueue new event for processing.

Enqueue signal to the most inner queue (nearest to the active queue) where the *signal.source* belongs. If it belongs nowhere enqueue it to the active one.

This method is thread safe.

Parameters **signal** (*Instance based on AbstractEvent class.*) – Event which you want to add to the event queue for processing.

execute_new_loop(*signal*)
Starts the new event loop and pass *signal* in it.

This is required for processing a modal screens.

Parameters **signal** (The *AbstractSignal* based class.) – Signal passed to the new event loop.

force_quit()
Force quit all running event loops.

Kill all loop including inner loops (modal window). None of the Simpleline events will be processed anymore.

kill_app_with_traceback(*exception_signal*, *data=None*)
Print exception and screen stack and kill the application.

Parameters

- **exception_signal** (Instance of *simpleline.event_loop.signals.ExceptionSignal* class.) – ExceptionSignal encapsulating the original exception which will be passed to the sys.excepthook method.
- **data** (*Anything will be ignored.*) – To be usable as signal handler.

process_signals(*return_after=None*)
This method processes incoming async messages.

Process signals en-queued by the *self.enqueue_signal()* method. Call handlers registered to the signals by the *self.register_signal_handler()* method.

When *return_after* is specified then wait to the point when this signal is processed. NO warranty that this method will return immediately after the signal was processed!

Without *return_after* parameter this method will return after all queued signals with the highest priority will be processed.

The method is NOT thread safe!

Parameters **return_after** (*Class of the signal.*) – Wait on this signal to be processed.

register_signal_handler(*signal*, *callback*, *data=None*)
Register a callback which will be called when message "event" is encountered during process_events.

The callback has to accept two arguments: - the received message in the form of (type, [arguments]) - the data registered with the handler

Parameters

- **signal** (*Class based on the simpleline.event_loop.AbstractSignal class.*) – Signal class we want to react on.
- **callback** (*func(event_message, data)*) – The callback function.
- **data** (*Anything.*) – Optional data to pass to callback.

register_signal_source(*signal_source*)
Register source of signal for actual event queue.

Parameters **signal_source** (*simpleline.render.ui_screen.UIScreen.*) – Source for future signals.

run()
This methods starts the application.

Do not use self.mainloop() directly as run() handles all the required exceptions needed to keep nested mainloop working.

set_quit_callback(*callback*, *args=None*)
Call this callback when event loop quits.

Parameters

- **callback** (Function with one parameter data *func(data).*) – Call this callback when event loops ends (application quit).
- **args** (*Anything.*) – Arguments passed to the quit callback.

3.8.2 GLib Event loop

The GLib event loop was added in order to utilize existing event loops used by other libraries, for example, DBus connections. Simpleline with this loop should have the same behavior as with the *Main Loop*.

To use this loop you need to set it via the *App* class:

```
# Create Glib event loop.
glib_loop = GLibEventLoop()

# Use glib event loop instead of the original one.
# Everything else should behave the same as with the original Simpleline loop.
App.initialize(event_loop=glib_loop)
```

The GLib loop can be accessed by the *GLibEventLoop.active_main_loop* property, or by getting the default loop from GLib directly GLib.MainLoop().

class simpleline.event_loop.glib_event_loop.**GLibEventLoop**
Bases: *simpleline.event_loop.AbstractEventLoop*

active_main_loop
Return GLib mainloop object.

close_loop()
Close active event loop.

Close an event loop created by the *execute_new_loop()* method.

enqueue_signal(*signal*)
Enqueue new event for processing.

Parameters **signal** (*instance based on AbstractEvent class*) – signal which you want to add to the event queue for processing

execute_new_loop(*signal*)
Starts the new event loop and pass *signal* in it.

This is required for processing a modal screens.

Parameters **signal** (*AbstractSignal* based class) – signal passed to the new event loop

force_quit()
Force quit all running event loops.

Kill all loop including inner loops (modal window). None of the Simpleline events will be processed anymore.

kill_app_with_traceback(*exception_signal*, *data=None*)
Print exception and screen stack and kill the application.

Parameters

- **exception_signal** (Instance of *simpleline.event_loop.signals.ExceptionSignal* class.) – ExceptionSignal encapsulating the original exception which will be passed to the sys.excepthook method.
- **data** (*Anything will be ignored.*) – To be usable as signal handler.

process_signals(*return_after=None*)
This method processes incoming async messages.

Process signals en-queued by the *self.enqueue_signal()* method. Call handlers registered to the signals by the *self.register_signal_handler()* method.

When *return_after* is specified then wait to the point when this signal is processed. NO warranty that this method will return immediately after the signal was processed!

Without *return_after* parameter this method will return after all queued signals with the highest priority will be processed.

The method is NOT thread safe!

Parameters **return_after** (*Class of the signal.*) – Wait on this signal to be processed.

register_signal_handler(*signal*, *callback*, *data=None*)
Register a callback which will be called when message "event" is encountered during process_events.

The callback has to accept two arguments: - the received message in the form of (type, [arguments]) - the data registered with the handler

Parameters

- **signal** (*Class based on the simpleline.event_loop.AbstractSignal class.*) – Signal class we want to react on.

- **callback** (*func(event_message, data)*) – The callback function.
- **data** (*Anything.*) – Optional data to pass to callback.

register_signal_source(*signal_source*)
Register source of signal for actual event queue.

Parameters **signal_source** (*simpleline.render.ui_screen.UIScreen*) – Source for future signals.

run()
Starts the event loop.

set_quit_callback(*callback*, *args=None*)
Call this callback when event loop quits.

Parameters

- **callback** (Function with one parameter data *func(data)*.) – Call this callback when event loops ends (application quit).
- **args** (*Anything.*) – Arguments passed to the quit callback.

3.8.3 Create your own loop

If new loop support is required, it should inherit from *AbstractEventLoop* and implement the same behavior as the *Main Loop*. You can use existing tests from the event loops to start. If the new loop is stable enough, pull requests are always welcome at Simpleline repository.

class simpleline.event_loop.**AbstractEventLoop**

close_loop()
Close active event loop.

Close an event loop created by the *execute_new_loop()* method.

enqueue_signal(*signal*)
Enqueue new event for processing.

Parameters **signal** (*Instance based on AbstractEvent class.*) – Signal which you want to add to the event queue for processing.

execute_new_loop(*signal*)
Starts the new event loop and pass *signal* in it.

This is required for processing a modal screens.

Parameters **signal** (The *AbstractSignal* based class.) – Signal passed to the new event loop.

force_quit()
Force quit all running event loops.

Kill all loop including inner loops (modal window). None of the Simpleline events will be processed anymore.

kill_app_with_traceback(*exception_signal*, *data=None*)
Print exception and screen stack and kill the application.

Parameters

- **exception_signal** (Instance of *simpleline.event_loop.signals.ExceptionSignal* class.) – ExceptionSignal encapsulating the original exception which will be passed to the sys.excepthook method.

- **data** (*Anything will be ignored.*) – To be usable as signal handler.

process_signals(*return_after=None*)
This method processes incoming async messages.

Process signals enqueued by the *self.enqueue_signal()* method. Call handlers registered to the signals by the *self.register_signal_handler()* method.

When *return_after* is specified then wait to the point when this signal is processed. NO warranty that this method will return immediately after the signal was processed!

Without *return_after* parameter this method will return after all queued signals with the highest priority will be processed.

The method is NOT thread safe!

Parameters **return_after** (*Class of the signal.*) – Wait on this signal to be processed.

register_signal_handler(*signal*, *callback*, *data=None*)
Register a callback which will be called when message "event" is encountered during process_events.

The callback has to accept two arguments: - the received message in the form of (type, [arguments]) - the data registered with the handler

Parameters

- **signal** (*Class based on the simpleline.event_loop.AbstractSignal class.*) – Signal class we want to react on.
- **callback** (*func(event_message, data)*) – The callback function.
- **data** (*Anything.*) – Optional data to pass to callback.

register_signal_source(*signal_source*)
Register source of signal for actual event queue.

Parameters **signal_source** (*simpleline.render.ui_screen.UIScreen*) – Source for future signals.

run()
Starts the event loop.

set_quit_callback(*callback*, *args=None*)
Call this callback when event loop quits.

Parameters

- **callback** (Function with one parameter data *func(data).*) – Call this callback when event loops ends (application quit).
- **args** (*Anything.*) – Arguments passed to the quit callback.

3.9 Signals

This is a collection of signals that can be used in the *event loop*.

class simpleline.event_loop.signals.**ExceptionSignal**(*source*, *exception_info=None*)
Bases: *simpleline.event_loop.AbstractSignal*

Emit this signal when exception is raised.

This class must be created inside of exception handler or *exception_info* must be specified in creation process.

If you register handler for this exception then the Simpleline's exception handling is disabled!

priority
Priority of this event.

Values less than 0 denote higher priorities. Values greater than 0 denote lower priorities. Events from high priority sources are always processed before events from lower priority sources.

source
Source which emitted this event.

class simpleline.event_loop.signals.**InputReadySignal**(*source*, *input_handler_source*, *data*, *priority=0*, *success=True*)
Bases: *simpleline.event_loop.AbstractSignal*

Input from user is ready for processing.

priority
Priority of this event.

Values less than 0 denote higher priorities. Values greater than 0 denote lower priorities. Events from high priority sources are always processed before events from lower priority sources.

source
Source which emitted this event.

class simpleline.event_loop.signals.**RenderScreenSignal**(*source*, *priority=0*)
Bases: *simpleline.event_loop.AbstractSignal*

Render UIScreen to terminal.

priority
Priority of this event.

Values less than 0 denote higher priorities. Values greater than 0 denote lower priorities. Events from high priority sources are always processed before events from lower priority sources.

source
Source which emitted this event.

class simpleline.event_loop.signals.**CloseScreenSignal**(*source*, *priority=0*)
Bases: *simpleline.event_loop.AbstractSignal*

Close current screen.

priority
Priority of this event.

Values less than 0 denote higher priorities. Values greater than 0 denote lower priorities. Events from high priority sources are always processed before events from lower priority sources.

source
Source which emitted this event.

class simpleline.event_loop.signals.**InputReceivedSignal**(*source*, *data*, *priority=0*)
Bases: *simpleline.event_loop.AbstractSignal*

Raw input received.

This signal will be further processed and InputReadySignal should be enqueued soon. Most probably you are looking for InputReadySignal instead.

priority
Priority of this event.

Values less than 0 denote higher priorities. Values greater than 0 denote lower priorities. Events from high priority sources are always processed before events from lower priority sources.

source
Source which emitted this event.

3.9.1 Creating custom signals

New signals can be created by subclassing an existing signal class or *AbstractSignal*

class simpleline.event_loop.**AbstractSignal**(*source*, *priority=0*)
This class is base class for signals.

priority
Priority of this event.

Values less than 0 denote higher priorities. Values greater than 0 denote lower priorities. Events from high priority sources are always processed before events from lower priority sources.

source
Source which emitted this event.

3.10 Errors module

Collection of generic exception classes used everywhere in the project. The most important one is *SimplelineError*, which is the base exception for all the other exceptions used in the Simpleline project.

exception simpleline.errors.**SimplelineError**
Bases: Exception

Base exception for all other exceptions.

3.10.1 Render exceptions

Exceptions used for rendering errors.

exception simpleline.render.**RenderError**
Bases: *simpleline.errors.SimplelineError*

Exception raised when error in rendering happens.

exception simpleline.render.**RenderUnexpectedError**
Bases: *simpleline.render.RenderError*

Exception raised when something goes really wrong.

3.10.2 Event loop exceptions

Exceptions used for errors in event loops.

exception simpleline.event_loop.**ExitMainLoop**
Bases: *simpleline.errors.SimplelineError*

This exception ends the whole event loop.

3.11 Advanced Input

Warning: This section contains advanced input techniques which may lead to buggy code in your program if they are not used correctly.

The default technique to get user input is already described in *UIScreen*, and that should be the preferred way to obtain user input. However, if the default technique is not enough for your situation then read the text below to find out how to implement custom input.

3.11.1 Input handler classes

Input handler classes are classes created to obtain user input. If required `InputHandler` can block an application until user input is received.

This class can be instantiated everywhere in the code and used to ask for user input:

```
handler = InputHandler()
handler.get_input("Shut up and give me your input:")
handler.wait_for_input()
if handler.input_successful:
    user_input = handler.value
```

The `InputHandler.wait_on_input()` method will block code processing until user input is received. Input should always be checked before processing.

In case some other work needs to be done before user input is received, then pass a callback to the constructor of the `InputHandler` class and do not use the `InputHandler.wait_on_input()` method. However, the callback is using the *event loop* so it won't be called until event loop processing is active. If *concurrent input* is used then this callback might never get called!

If what a user types must not be displayed, then the `PasswordInputHandler` class should be used. It shares most of its implementation with the `InputHandler` but overrides how to obtain the code. For more info look at the `PasswordInputHandler` class documentation.

3.11.2 Concurrent input

Concurrent input is something which should be avoided. It drags unexpected behavior into an application and is hard to debug. However, there could be an instance when user input is required immediately, even when an application is already waiting for other input.

By default, every attempt for concurrent input will raise an exception and kill the application to prevent unexpected behavior. In order to allow for concurrent input, the `InputHandler.skip_concurrency_check` property must be set. After this property is disabled for the `InputHandler` instance, then the handler instance then it can support concurrent input.

The last registered concurrent input will result in dropping all other waiting inputs – even other waiting inputs with `InputHandler.skip_concurrency_check` will be dropped. The dropped waiting inputs will get a failed input signal to unblock `InputHandler.wait_on_input()` methods.

3.11.3 Creating a custom InputHandler

If the *InputHandler* class or the *PasswordInputHandler* class is not enough, developers can create their own handler. The structure of an input handler is based on two classes. First is a handler itself, and second is the request object it creates.

The handler class is used as an interface for the rest of the application. It also creates the requester instance. The requester object is a low-level implementation of obtaining user input. The requester object has to have the `get_input` method and should contain the `text_prompt` method. The `get_input` method is called in a separate thread and should prompt the user for input. The `text_prompt` method is required mainly for concurrent input, and it returns a string representation of the prompt.

For more details, please look at the implementation of the *InputHandler* class.

3.11.4 InputHandler class

class simpleline.input.input_handler.**InputHandler**(*callback=None*, *source=None*)

create_thread_object(*prompt*)
Create thread object containing all the information how to get user input.

Returns Instance of class inherited from *simpleline.input.InputThread*.

get_input(*prompt*)
Use prompt to ask for user input and wait (non-blocking) on user input.

This is an asynchronous call. If you want to wait for user input then use the *wait_on_input* method. If you want to get results asynchronously then register callback in constructor or by the *set_callback* method.

Check if user input was already received can be done by the *input_received* method call.

Parameters **prompt** (*String or Prompt instance.*) – Ask user what you want to get.

Returns User input.

Return type str

input_received()
Was user input already received?

Returns True if yes, False otherwise.

input_successful()
Was input successful?

Returns bool

set_callback(*callback*)
Set a callback to get user input asynchronously.

Parameters **callback** (*Method with 1 argument which is user input: def cb(user_input)*) – Callback called when user write their input.

skip_concurrency_check
Is this InputHandler skipping concurrency check?

:returns bool

source
Get source of this input.

Returns Anything probably UIScreen.

value
Return user input.

Returns String or None if no is input received.

wait_on_input()
Blocks execution till the user input is received.

Events will works as expected during this blocking.

Please check the *input_successful* method to test the input.

3.11.5 PasswordInputHandler class

class simpleline.input.input_handler.**PasswordInputHandler**(*callback=None*, *source=None*)

create_thread_object(*prompt*)
Return PasswordInputThread for getting user password.

get_input(*prompt*)
Use prompt to ask for user input and wait (non-blocking) on user input.

This is an asynchronous call. If you want to wait for user input then use the *wait_on_input* method. If you want to get results asynchronously then register callback in constructor or by the *set_callback* method.

Check if user input was already received can be done by the *input_received* method call.

Parameters **prompt** (*String or Prompt instance.*) – Ask user what you want to get.

Returns User input.

Return type str

input_received()
Was user input already received?

Returns True if yes, False otherwise.

input_successful()
Was input successful?

Returns bool

set_callback(*callback*)
Set a callback to get user input asynchronously.

Parameters **callback** (*Method with 1 argument which is user input: def cb(user_input)*) – Callback called when user write their input.

set_pass_func(*getpass_func*)
Set a function for getting passwords.

skip_concurrency_check
Is this InputHandler skipping concurrency check?

:returns bool

source
Get source of this input.

Returns Anything probably UIScreen.

value
Return user input.

Returns String or None if no is input received.

wait_on_input()
Blocks execution till the user input is received.

Events will works as expected during this blocking.

Please check the *input_successful* method to test the input.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

S

Index

A

C

D

E

F

G

H

I

K

L

M

N

P

R

S