prompt_toolkit Documentation

Release 1.0.14

Jonathan Slenders

Dec 24, 2017

Contents

prompt_toolkit is a library for building powerful interactive command lines and terminal applications in Python.

It can be a pure Python replacement for GNU readline, but it can be much more than that.

Some features:

- Syntax highlighting of the input while typing. (For instance, with a Pygments lexer.)
- Multi-line input editing.
- Advanced code completion.
- Selecting text for copy/paste. (Both Emacs and Vi style.)
- Mouse support for cursor positioning and scrolling.
- Auto suggestions. (Like fish shell.)
- No global state.

Like readline:

- Both Emacs and Vi key bindings.
- Reverse and forward incremental search.
- Works well with Unicode double width characters. (Chinese input.)

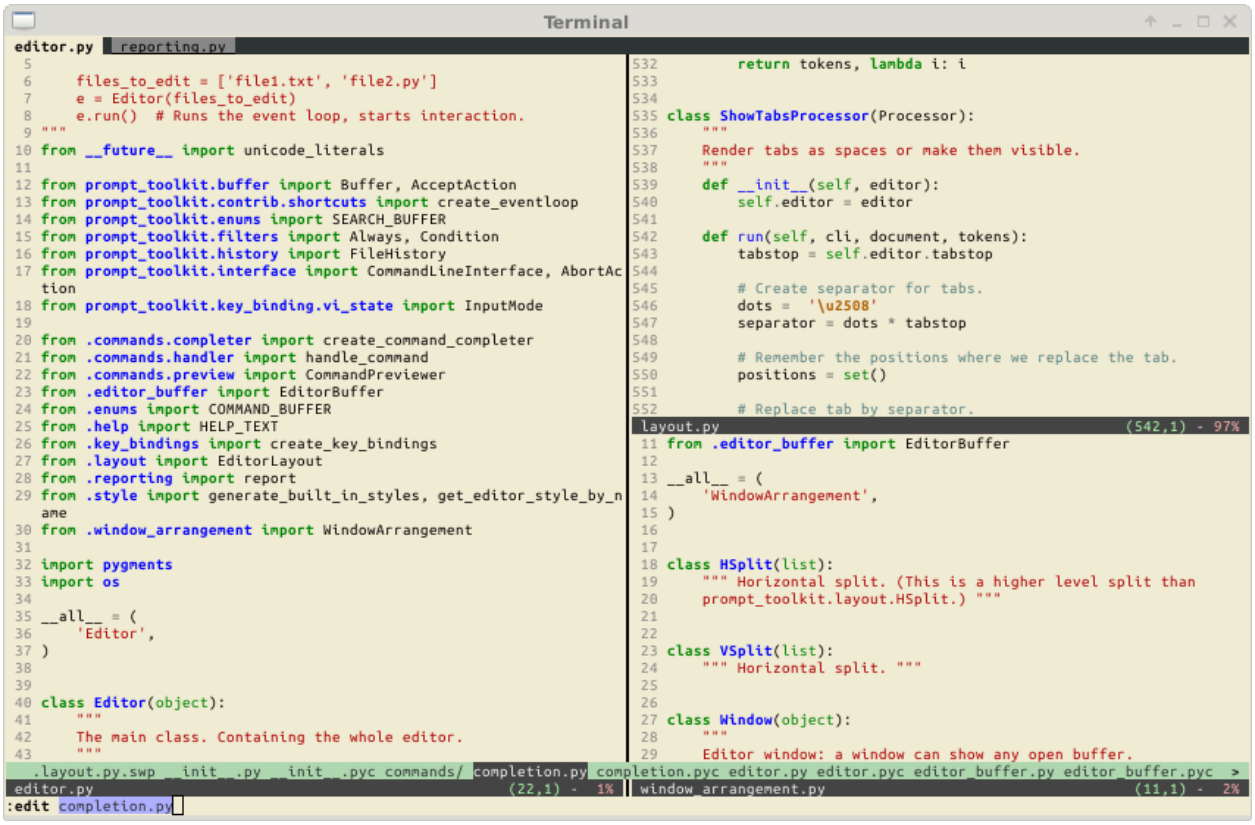
Works everywhere:

- Pure Python. Runs on all Python versions from 2.6 up to 3.4.
- Runs on Linux, OS X, OpenBSD and Windows systems.
- Lightweight, the only dependencies are Pygments, six and wcwidth.
- No assumptions about I/O are made. Every prompt_toolkit application should also run in a telnet/ssh server or an asyncio process.

CHAPTER 1

Two use cases: prompts versus full screen terminal applications

`prompt_toolkit` was in the first place meant to be a replacement for readline. However, when it became more mature, we realised that all the components for full screen applications are there and `prompt_toolkit` is very capable of handling many use cases. Pyvim and pymux are examples of full screen applications.



Basically, at the core, `prompt_toolkit` has a layout engine, that supports horizontal and vertical splits as well as floats, where each "window" can display a user control. The API for user controls is simple yet powerful.

When `prompt_toolkit` is used to simply read some input from the user, it uses a rather simple built-in layout. One that displays the default input buffer and the prompt, a float for the autocompletions and a toolbar for input validation which is hidden by default.

For full screen applications, usually we build the layout ourself, because it's very custom.

Further, there is a very flexible key binding system that can be programmed for all the needs of full screen applications.

CHAPTER 2

Installation

```
pip install prompt_toolkit
```

For Conda, do:

```
conda install -c https://conda.anaconda.org/conda-forge prompt_toolkit
```

CHAPTER 3

Getting started

The following snippet is the most simple example, it uses the `prompt()` function to asks the user for input and returns the text. Just like `(raw_)input`.

```
from __future__ import unicode_literals
from prompt_toolkit import prompt

text = prompt('Give me some input: ')
print('You said: %s' % text)
```

For more information, start reading the *building prompts* section.

CHAPTER 4

Thanks to:

Thanks to all the contributors for making prompt_toolkit possible.

Also, a special thanks to the Pygments and wcwidth libraries.

CHAPTER 5

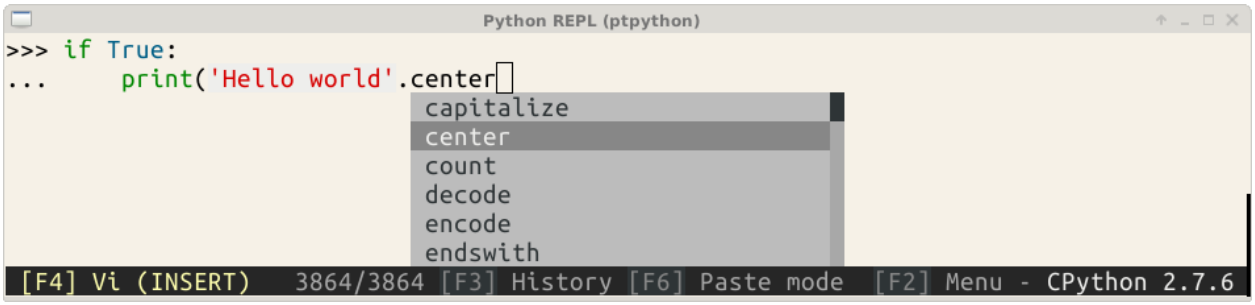
Table of contents

5.1 Gallery

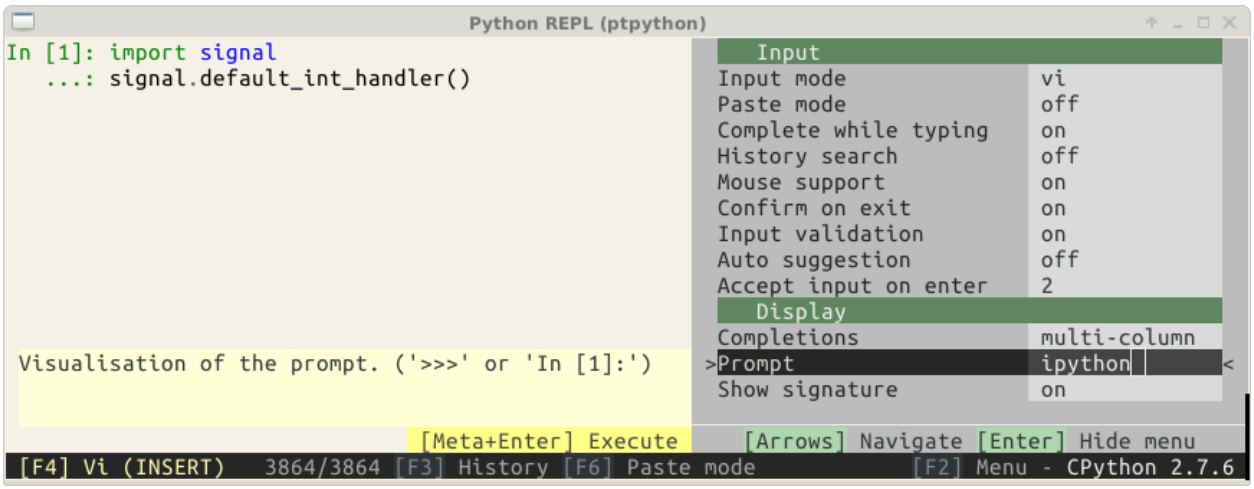
Showcase, demonstrating the possibilities of prompt_toolkit.

5.1.1 Ptpython, a Python REPL

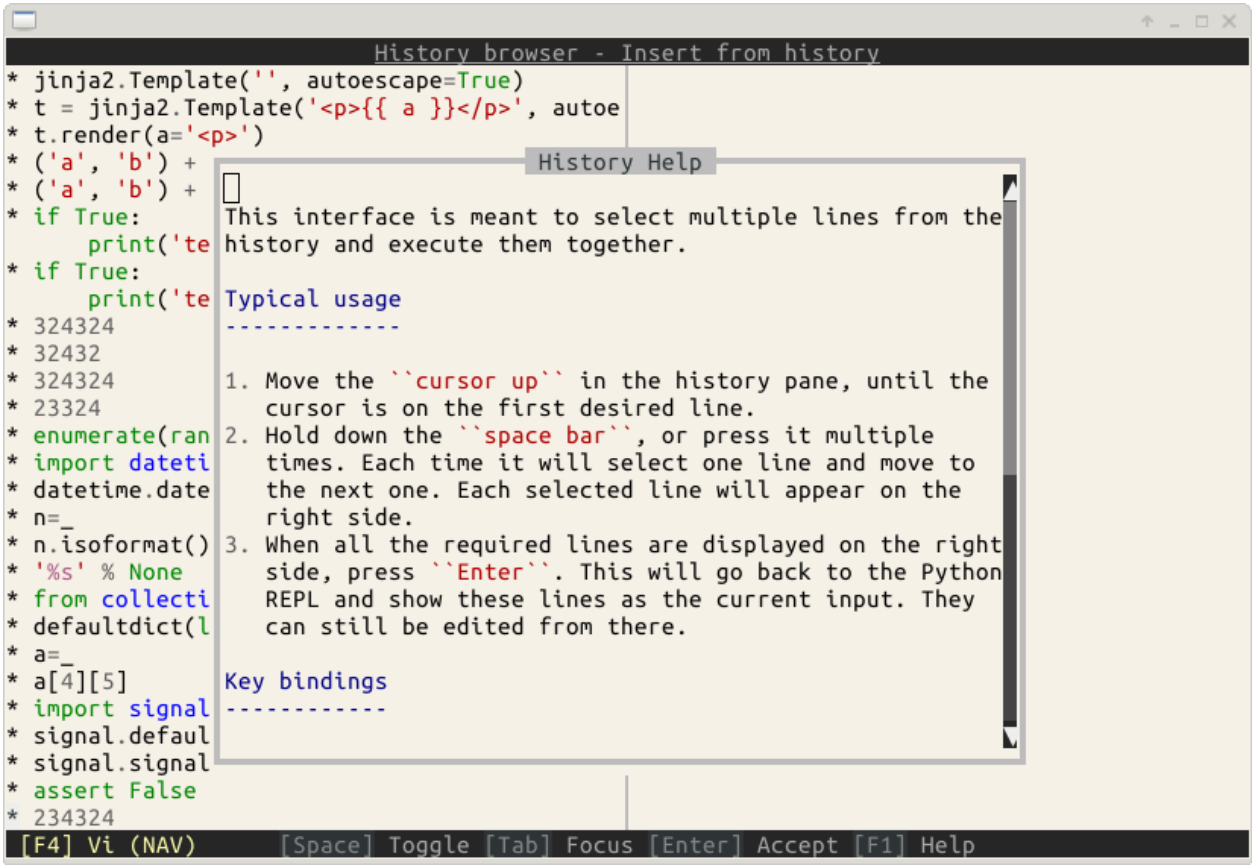
The prompt:



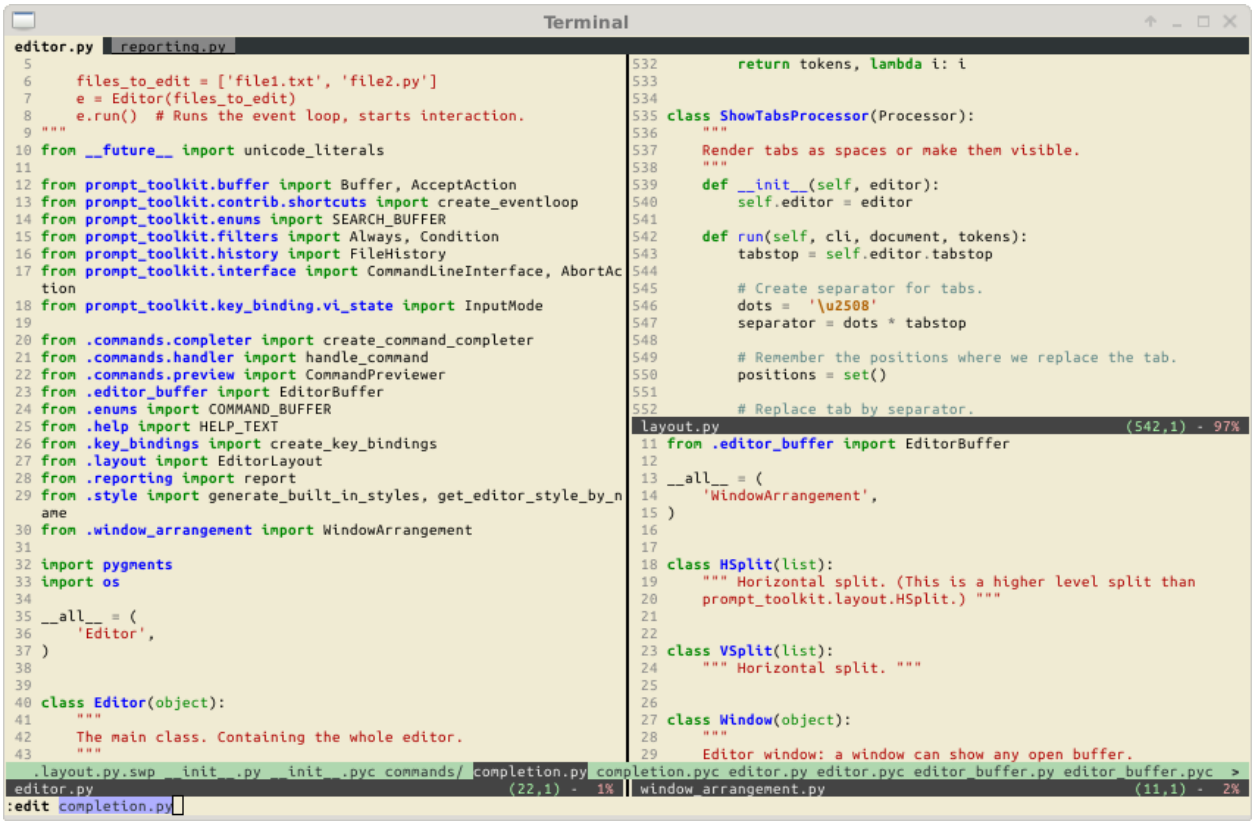
The configuration menu of ptpython.



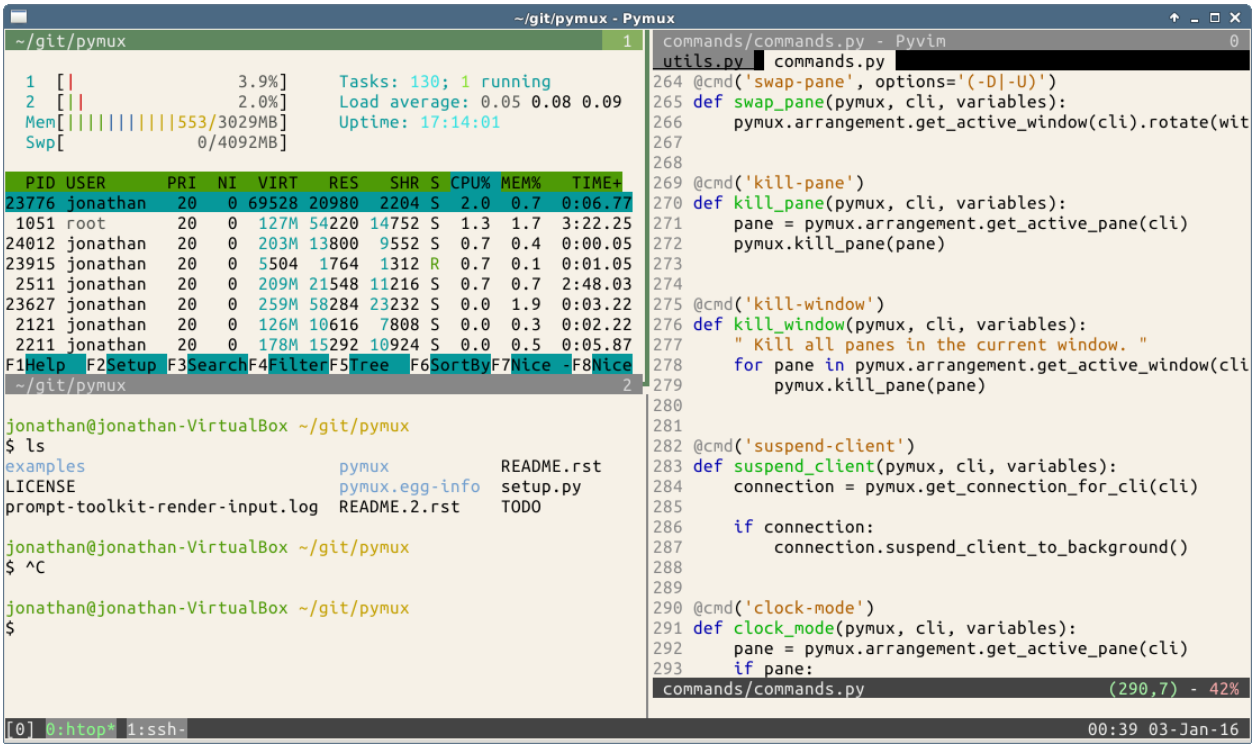
The history page with its help. (This is a full-screen layout.)



5.1.2 Pyvim, a Vim clone



5.1.3 Pymux, a terminal multiplexer (like tmux) in Python



5.2 Building prompts

This page is about building prompts. Pieces of code that we can embed in a program for asking the user for input. If you want to use *prompt_toolkit* for building full screen terminal applications, it is probably still a good idea to read this first, before heading to the *building full screen applications* page.

5.2.1 Hello world

The following snippet is the most simple example, it uses the `prompt()` function to asks the user for input and returns the text. Just like `(raw_)input`.

```
from __future__ import unicode_literals
from prompt_toolkit import prompt

text = prompt('Give me some input: ')
print('You said: %s' % text)
```

What we get here is a simple prompt that supports the Emacs key bindings like readline, but further nothing special. However, `prompt()` has a lot of configuration options. In the following sections, we will discover all these parameters.

Note: `prompt_toolkit` expects unicode strings everywhere. If you are using Python 2, make sure that all strings which are passed to `prompt_toolkit` are unicode strings (and not bytes). Either use `from __future__ import unicode_literals` or explicitely put a small `'u'` in front of every string.

5.2.2 Syntax highlighting

Adding syntax highlighting is as simple as adding a lexer. All of the Pygments lexers can be used after wrapping them in a *PygmentsLexer*. It is also possible to create a custom lexer by implementing the *Lexer* abstract base class.

```
from pygments.lexers import HtmlLexer
from prompt_toolkit.shortcuts import prompt
from prompt_toolkit.layout.lexers import PygmentsLexer

text = prompt('Enter HTML', lexer=PygmentsLexer(HtmlLexer))
print('You said: %s' % text)
```

5.2.3 Colors

The colors for syntax highlighting are defined by a `Style` instance. By default, a neutral built-in style is used, but any style instance can be passed to the *prompt()* function. A simple way to create a style, is by using the `style_from_dict` function:

```
from pygments.lexers import HtmlLexer
from prompt_toolkit.shortcuts import prompt
from prompt_toolkit.token import Token
from prompt_toolkit.styles import style_from_dict
from prompt_toolkit.layout.lexers import PygmentsLexer

our_style = style_from_dict({
    Token.Comment:   '#888888 bold',
    Token.Keyword:   '#ff88ff bold',
})

text = prompt('Enter HTML: ', lexer=PygmentsLexer(HtmlLexer),
              style=our_style)
```

The style dictionary is very similar to the Pygments `styles` dictionary, with a few differences:

- The *roman*, *sans*, *mono* and *border* options are not ignored.
- The style has a few additions: *blink*, *noblink*, *reverse* and *noreverse*.
- Colors can be in the *#ff0000* format, but they can be one of the built-in ANSI color names as well. In that case, they map directly to the 16 color palette of the terminal.

Using a Pygments style

All Pygments style classes can be used as well, when they are wrapped through `style_from_pygments()`.

Suppose we'd like to use a Pygments style, for instance `pygments.styles.tango.TangoStyle`, that is possible like this:

Creating a custom style could be done like this:

```
from prompt_toolkit.shortcuts import prompt
from prompt_toolkit.styles import style_from_pygments
from prompt_toolkit.layout.lexers import PygmentsLexer

from pygments.styles.tango import TangoStyle
from pygments.token import Token
from pygments.lexers import HtmlLexer

our_style = style_from_pygments(TangoStyle, {
    Token.Comment:   '#888888 bold',
    Token.Keyword:   '#ff88ff bold',
})

text = prompt('Enter HTML: ', lexer=PygmentsLexer(HtmlLexer),
              style=our_style)
```

Coloring the prompt itself

It is possible to add some colors to the prompt itself. For this, we need a `get_prompt_tokens` function. This function takes a *`CommandLineInterface`* instance as input (ignore that for now) and it should return a list of `(Token, text)` tuples. Each token is a Pygments token and can be styled individually.

```
from prompt_toolkit.shortcuts import prompt
from prompt_toolkit.styles import style_from_dict

example_style = style_from_dict({
    # User input.
    Token:          '#ff0066',

    # Prompt.
    Token.Username: '#884444',
    Token.At:       '#00aa00',
    Token.Colon:    '#00aa00',
    Token.Pound:    '#00aa00',
    Token.Host:     '#000088 bg:#aaaaff',
    Token.Path:     '#884444 underline',
})

def get_prompt_tokens(cli):
    return [
        (Token.Username, 'john'),
        (Token.At,       '@'),
        (Token.Host,     'localhost'),
        (Token.Colon,    ':'),
        (Token.Path,     '/user/john'),
        (Token.Pound,    '# '),
    ]

text = prompt(get_prompt_tokens=get_prompt_tokens, style=example_style)
```

By default, colors are taking from the 256 color palette. If you want to have 24bit true color, this is possible by adding the `true_color=True` option to the `prompt` function.

```
text = prompt(get_prompt_tokens=get_prompt_tokens, style=example_style,
              true_color=True)
```

Printing text (output) in color

Besides prompting for input, we often have to print some output in color. This is simple with the *print_tokens()* function.

```
from prompt_toolkit.shortcuts import print_tokens
from prompt_toolkit.styles import style_from_dict
from pygments.token import Token

# Create a stylesheet.
style = style_from_dict({
    Token.Hello: '#ff0066',
    Token.World: '#44ff44 italic',
})

# Make a list of (Token, text) tuples.
tokens = [
    (Token.Hello, 'Hello '),
    (Token.World, 'World'),
    (Token, '\n'),
]

# Print the result.
print_tokens(tokens, style=style)
```

5.2.4 Autocompletion

Autocompletion can be added by passing a `completer` parameter. This should be an instance of the *Completer* abstract base class. `WordCompleter` is an example of a completer that implements that interface.

```
from prompt_toolkit import prompt
from prompt_toolkit.contrib.completers import WordCompleter

html_completer = WordCompleter(['<html>', '<body>', '<head>', '<title>'])
text = prompt('Enter HTML: ', completer=html_completer)
print('You said: %s' % text)
```

`WordCompleter` is a simple completer that completes the last word before the cursor with any of the given words.

A custom completer

For more complex examples, it makes sense to create a custom completer. For instance:

```
from prompt_toolkit import prompt
from prompt_toolkit.completion import Completer, Completion

class MyCustomCompleter(Completer):
    def get_completions(self, document, complete_event):
        yield Completion('completion', start_position=0)

text = prompt('> ', completer=MyCustomCompleter())
```

A *Completer* class has to implement a generator named *get_completions()* that takes a *Document* and yields the current *Completion* instances. Each completion contains a portion of text, and a position.

The position is used in for fixing text before the cursor. Pressing the tab key could for instance turn parts of the input from lowercase to uppercase. This makes sense for a case insensitive completer. Or in case of a fuzzy completion, it could fix typos. When `start_position` is something negative, this amount of characters will be deleted and replaced.

5.2.5 Input validation

A prompt can have a validator attached. This is some code that will check whether the given input is acceptable and it will only return it if that's the case. Otherwise it will show an error message and move the cursor to a given possition.

A validator should implements the *Validator* abstract base class. This requires only one method, named `validate` that takes a *Document* as input and raises *ValidationError* when the validation fails.

```
from prompt_toolkit.validation import Validator, ValidationError
from prompt_toolkit import prompt

class NumberValidator(Validator):
    def validate(self, document):
        text = document.text

        if text and not text.isdigit():
            i = 0

            # Get index of fist non numeric character.
            # We want to move the cursor here.
```

```
            for i, c in enumerate(text):
                if not c.isdigit():
                    break

            raise ValidationError(message='This input contains non-numeric characters
↪',
                                  cursor_position=i)


number = int(prompt('Give a number: ', validator=NumberValidator()))
print('You said: %i' % number)
```

5.2.6 History

A *History* object keeps track of all the previously entered strings. When nothing is passed into the *prompt()* function, it will start with an empty history each time again. Usually, however, for a REPL, you want to keep the same history between several calls to *prompt()*. This is possible by instantiating a *History* object and passing that to each `prompt` call.

```
from prompt_toolkit.history import InMemoryHistory
from prompt_toolkit import prompt

history = InMemoryHistory()

while True:
    prompt(history=history)
```

To persist a history to disk, use *FileHistory* instead instead of *InMemoryHistory*.

5.2.7 Auto suggestion

Auto suggestion is a way to propose some input completions to the user like the fish shell.

Usually, the input is compared to the history and when there is another entry starting with the given text, the completion will be shown as gray text behind the current input. Pressing the right arrow → will insert this suggestion.

Note: When suggestions are based on the history, don't forget to share one *History* object between consecutive *prompt()* calls.

Example:

```
from prompt_toolkit import prompt
from prompt_toolkit.history import InMemoryHistory
from prompt_toolkit.auto_suggest import AutoSuggestFromHistory

history = InMemoryHistory()

while True:
    text = prompt('> ', history=history, auto_suggest=AutoSuggestFromHistory())
    print('You said: %s' % text)
```

A suggestion does not have to come from the history. Any implementation of the *AutoSuggest* abstract base class can be passed as an argument.

5.2.8 Adding a bottom toolbar

Adding a bottom toolbar is as easy as passing a `get_bottom_toolbar_tokens` function to *prompt()*. The function is called every time the prompt is rendered (at least on every key stroke), so the bottom toolbar can be used to display dynamic information. It receives a *CommandLineInterface* and should return a list of tokens. The toolbar is always erased when the prompt returns.

```
from prompt_toolkit import prompt
from prompt_toolkit.styles import style_from_dict
from prompt_toolkit.token import Token

def get_bottom_toolbar_tokens(cli):
    return [(Token.Toolbar, ' This is a toolbar. ')]

style = style_from_dict({
    Token.Toolbar: '#ffffff bg:#333333',
})

text = prompt('> ', get_bottom_toolbar_tokens=get_bottom_toolbar_tokens,
              style=style)
print('You said: %s' % text)
```

The default token is `Token.Toolbar` and that will also be used to fill the background of the toolbar. *Styling* can be done by pointing to that token.

5.2.9 Adding a right prompt

The *prompt()* function has out of the box support for right prompts as well. People familiar to ZSH could recognise this as the *RPROMPT* option.

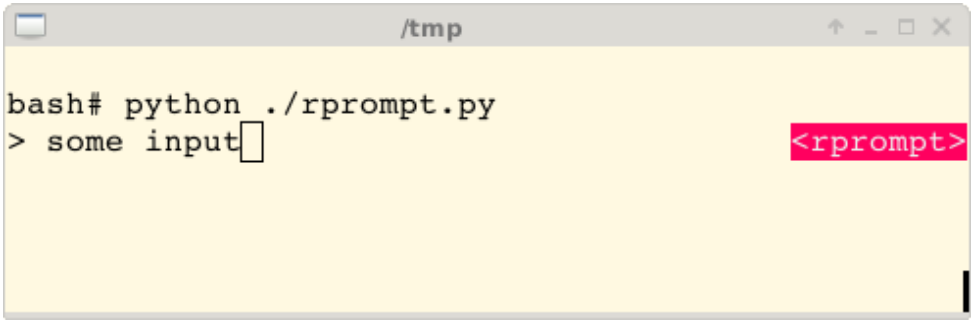
So, similar to adding a bottom toolbar, we can pass a `get_rprompt_tokens` callable.

```
from prompt_toolkit import prompt
from prompt_toolkit.styles import style_from_dict
from prompt_toolkit.token import Token

example_style = style_from_dict({
    Token.RPrompt: 'bg:#ff0066 #ffffff',
})

def get_rprompt_tokens(cli):
    return [
        (Token, ' '),
        (Token.RPrompt, '<rprompt>'),
    ]
```

```
answer = prompt('> ', get_rprompt_tokens=get_rprompt_tokens,
                style=example_style)
```



5.2.10 Vi input mode

Prompt-toolkit supports both Emacs and Vi key bindings, similar to Readline. The *prompt()* function will use Emacs bindings by default. This is done because on most operating systems, also the Bash shell uses Emacs bindings by default, and that is more intuitive. If however, Vi binding are required, just pass `vi_mode=True`.

```
from prompt_toolkit import prompt

prompt('> ', vi_mode=True)
```

5.2.11 Adding custom key bindings

The *prompt()* function accepts an optional `key_bindings_registry` argument. This should be a *Registry* instance which hold all of the key bindings.

It would be possible to create such a *Registry* class ourself, but usually, for a prompt, we would like to have at least the basic (Emacs/Vi) bindings and start from there. That's what the *KeyBindingManager* class does.

An example of a prompt that prints `'hello world'` when `Control-T` is pressed.

```
from prompt_toolkit import prompt
from prompt_toolkit.key_binding.manager import KeyBindingManager
from prompt_toolkit.keys import Keys

manager = KeyBindingManager.for_prompt()

@manager.registry.add_binding(Keys.ControlT)
def _(event):
    def print_hello():
        print('hello world')
    event.cli.run_in_terminal(print_hello)

text = prompt('> ', key_bindings_registry=manager.registry)
print('You said: %s' % text)
```

Note that we use *run_in_terminal()*. This ensures that the output of the print-statement and the prompt don't mix up.

Enable key bindings according to a condition

Often, some key bindings can be enabled or disabled according to a certain condition. For instance, the Emacs and Vi bindings will never be active at the same time, but it is possible to switch between Emacs and Vi bindings at run time.

In order to enable a key binding according to a certain condition, we have to pass it a *CLIFilter*, usually a *Condition* instance. (*Read more about filters.*)

```
from prompt_toolkit import prompt
from prompt_toolkit.filters import Condition
from prompt_toolkit.key_binding.manager import KeyBindingManager
from prompt_toolkit.keys import Keys

manager = KeyBindingManager.for_prompt()

def is_active(cli):
    " Only activate key binding on the second half of each minute. "
    return datetime.datetime.now().second > 30

@manager.registry.add_binding(Keys.ControlT, filter=Condition(is_active))
def _(event):
    # ...
    pass

prompt('> ', key_bindings_registry=manager.registry)
```

Dynamically switch between Emacs and Vi mode

The *CommandLineInterface* has an `editing_mode` attribute. We can change the key bindings by changing this attribute from `EditingMode.VI` to `EditingMode.EMACS`.

```
from prompt_toolkit import prompt
from prompt_toolkit.filters import Condition
from prompt_toolkit.key_binding.manager import KeyBindingManager
from prompt_toolkit.keys import Keys

def run():
    # Create a set of key bindings.
    manager = KeyBindingManager.for_prompt()

    # Add an additional key binding for toggling this flag.
    @manager.registry.add_binding(Keys.F4)
    def _(event):
        " Toggle between Emacs and Vi mode. "
        cli = event.cli

        if cli.editing_mode == EditingMode.VI:
            cli.editing_mode = EditingMode.EMACS
        else:
            cli.editing_mode = EditingMode.VI

    # Add a toolbar at the bottom to display the current input mode.
    def get_bottom_toolbar_tokens(cli):
        " Display the current input mode. "
        text = 'Vi' if cli.editing_mode == EditingMode.VI else 'Emacs'
        return [
            (Token.Toolbar, ' [F4] %s ' % text)
```

```
        ]

    prompt('> ', key_bindings_registry=manager.registry,
           get_bottom_toolbar_tokens=get_bottom_toolbar_tokens)

run()
```

5.2.12 Other prompt options

Multiline input

Reading multiline input is as easy as passing the `multiline=True` parameter.

```
from prompt_toolkit import prompt

prompt('> ', multiline=True)
```

A side effect of this is that the enter key will now insert a newline instead of accepting and returning the input. The user will now have to press `Meta+Enter` in order to accept the input. (Or `Escape` followed by `Enter`.)

It is possible to specify a continuation prompt. This works by passing a `get_continuation_tokens` callable to `prompt`. This function can return a list of `(Token, text)` tuples. The width of the returned text should not exceed the given width. (The width of the prompt margin is defined by the prompt.)

```
def get_continuation_tokens(cli, width):
    return [(Token, '.' * width)]

prompt('> ', multiline=True,
       get_continuation_tokens=get_continuation_tokens)
```

Passing a default

A default value can be given:

```
from prompt_toolkit import prompt
import getpass

prompt('What is your name: ', default='%s' % getpass.getuser())
```

Mouse support

There is limited mouse support for positioning the cursor, for scrolling (in case of large multiline inputs) and for clicking in the autocompletion menu.

Enabling can be done by passing the `mouse_support=True` option.

```
from prompt_toolkit import prompt
import getpass

prompt('What is your name: ', mouse_support=True)
```

Line wrapping

Line wrapping is enabled by default. This is what most people are used to and this is what GNU readline does. When it is disabled, the input string will scroll horizontally.

```
from prompt_toolkit import prompt
import getpass

prompt('What is your name: ', wrap_lines=False)
```

Password input

When the `is_password=True` flag has been given, the input is replaced by asterisks (`*` characters).

```
from prompt_toolkit import prompt
import getpass

prompt('Enter password: ', is_password=True)
```

5.2.13 Prompt in an `asyncio` application

For asyncio applications, it's very important to never block the eventloop. However, *`prompt()`* is blocking, and calling this would freeze the whole application. A quick fix is to call this function via the asyncio `eventloop.run_in_executor`, but that would cause the user interface to run in another thread. (If we have custom key bindings for instance, it would be better to run them in the same thread as the other code.)

The answer is to run the prompt_toolkit interface on top of the asyncio event loop. Prompting the user for input is as simple as calling *`prompt_async()`*.

```
from prompt_toolkit import prompt_async

async def my_coroutine():
    while True:
        result = await prompt_async('Say something: ', patch_stdout=True)
        print('You said: %s' % result)
```

The `patch_stdout=True` parameter is optional, but it's recommended, because other coroutines could print to stdout. This option ensures that other output won't destroy the prompt.

5.3 Building full screen applications

prompt_toolkit can be used to create complex full screen terminal applications. Typically, an application consists of a layout (to describe the graphical part) and a set of key bindings.

The sections below describe the components required for full screen applications (or custom, non full screen applications), and how to assemble them together.

> **Warning:** This is going to change.
>
> The information below is still up to date, but we are planning to refactor some of the internal architecture of prompt_toolkit, to make it easier to build full screen applications. This will however be backwards-incompatible. The refactoring should probably be complete somewhere around half 2017.

5.3.1 Running the application

To run our final Full screen Application, we need three I/O objects, and an *Application* instance. These are passed as arguments to *CommandLineInterface*.

The three I/O objects are:

- An *EventLoop* instance. This is basically a while-true loop that waits for user input, and when it receives something (like a key press), it will send that to the application.
- An *Input* instance. This is an abstraction on the input stream (stdin).
- An *Output* instance. This is an abstraction on the output stream, and is called by the renderer.

The input and output objects are optional. However, the eventloop is always required.

We'll come back to what the *Application* instance is later.

So, the only thing we actually need in order to run an application is the following:

```
from prompt_toolkit.interface import CommandLineInterface
from prompt_toolkit.application import Application
from prompt_toolkit.shortcuts import create_eventloop

loop = create_eventloop()
application = Application()
cli = CommandLineInterface(application=application, eventloop=loop)
# cli.run()
print('Exiting')
```

Note: In the example above, we don't run the application yet, as otherwise it will hang indefinitely waiting for a signal to exit the event loop. This is why the *cli.run()* part is commented.

(Actually, it would accept the *Enter* key by default. But that's only because by default, a buffer called *DEFAULT_BUFFER* has the focus; its *AcceptAction* is configured to return the result when accepting, and there is a default *Enter* key binding that calls the *AcceptAction* of the currently focussed buffer. However, the content of the *DEFAULT_BUFFER* buffer is not yet visible, so it's hard to see what's going on.)

Let's now bind a keyboard shortcut to exit:

5.3.2 Key bindings

In order to react to user actions, we need to create a registry of keyboard shortcuts to pass to our *Application*. The easiest way to do so, is to create a *KeyBindingManager*, and then attach handlers to our desired keys. `Keys` contains a few predefined keyboards shortcut that can be useful.

To create a *registry*, we can simply instantiate a *KeyBindingManager* and take its *registry* attribute:

```
from prompt_toolkit.key_binding.manager import KeyBindingManager
manager = KeyBindingManager()
registry = manager.registry
```

Update the *Application* constructor, and pass the registry as one of the argument.

```
application = Application(key_bindings_registry=registry)
```

To register a new keyboard shortcut, we can use the *add_binding()* method as a decorator of the key handler:

```
from prompt_toolkit.keys import Keys

@registry.add_binding(Keys.ControlQ, eager=True)
def exit_(event):
    """
    Pressing Ctrl-Q will exit the user interface.

    Setting a return value means: quit the event loop that drives the user
    interface and return this value from the `CommandLineInterface.run()` call.
    """
    event.cli.set_return_value(None)
```

In this particular example we use `eager=True` to trigger the callback as soon as the shortcut *Ctrl-Q* is pressed. The callback is named `exit_` for clarity, but it could have been named _ (underscore) as well, because the we won't refer to this name.

5.3.3 Creating a layout

A *layout* is a composition of *Container* and *UIControl* that will describe the disposition of various element on the user screen.

Various Layouts can refer to *Buffers* that have to be created and pass to the application separately. This allow an application to have its layout changed, without having to reconstruct buffers. You can imagine for example switching from an horizontal to a vertical split panel layout and vice versa,

There are two types of classes that have to be combined to construct a layout:

- **containers** (*Container* instances), which arrange the layout
- **user controls** (*UIControl* instances), which generate the actual content

---

Note: An important difference:

- containers use *absolute coordinates*, and paint on a *Screen* instance.
- user controls create a *UIContent* instance. This is a collection of lines that represent the actual content. A *UIControl* is not aware of the screen.

Abstract base class	Examples
Container	*HSplit VSplit FloatContainer Window*
UIControl	*BufferControl TokenListControl FillControl*

The *Window* class itself is particular: it is a *Container* that can contain a *UIControl*. Thus, it's the adaptor between the two.

The *Window* class also takes care of scrolling the content if the user control created a *Screen* that is larger than what was available to the *Window*.

Here is an example of a layout that displays the content of the default buffer on the left, and displays `"Hello world"` on the right. In between it shows a vertical line:

```
from prompt_toolkit.enums import DEFAULT_BUFFER
from prompt_toolkit.layout.containers import VSplit, Window
from prompt_toolkit.layout.controls import BufferControl, FillControl,␣
↪TokenListControl
from prompt_toolkit.layout.dimension import LayoutDimension as D
```

```
from pygments.token import Token

layout = VSplit([
    # One window that holds the BufferControl with the default buffer on the
    # left.
    Window(content=BufferControl(buffer_name=DEFAULT_BUFFER)),

    # A vertical line in the middle. We explicitely specify the width, to make
    # sure that the layout engine will not try to divide the whole width by
    # three for all these windows. The `FillControl` will simply fill the whole
    # window by repeating this character.
    Window(width=D.exact(1),
           content=FillControl('|', token=Token.Line)),

    # Display the text 'Hello world' on the right.
    Window(content=TokenListControl(
        get_tokens=lambda cli: [(Token, 'Hello world')])),
])
```

The previous section explains how to create an application, you can just pass the currently created *layout* when you create the `Application` instance using the `layout=` keyword argument.

```
app = Application(..., layout=layout, ...)
```

The rendering flow

Understanding the rendering flow is important for understanding how *Container* and *UIControl* objects interact. We will demonstrate it by explaining the flow around a *BufferControl*.

Note: A *BufferControl* is a *UIControl* for displaying the content of a *Buffer*. A buffer is the object that holds any editable region of text. Like all controls, it has to be wrapped into a *Window*.

Let's take the following code:

```
from prompt_toolkit.enums import DEFAULT_BUFFER
from prompt_toolkit.layout.containers import Window
from prompt_toolkit.layout.controls import BufferControl

Window(content=BufferControl(buffer_name=DEFAULT_BUFFER))
```

What happens when a *Renderer* objects wants a *Container* to be rendered on a certain *Screen*?

The visualisation happens in several steps:

1. The *Renderer* calls the *write_to_screen()* method of a *Container*. This is a request to paint the layout in a rectangle of a certain size.

   The *Window* object then requests the *UIControl* to create a *UIContent* instance (by calling *create_content()*). The user control receives the dimensions of the window, but can still decide to create more or less content.

   Inside the *create_content()* method of *UIControl*, there are several steps:

   (a) First, the buffer's text is passed to the *lex_document()* method of a *Lexer*. This returns a function which for a given line number, returns a token list for that line (that's a list of `(Token, text)` tuples).

(b) The token list is passed through a list of *Processor* objects. Each processor can do a transformation for each line. (For instance, they can insert or replace some text.)

(c) The *UIControl* returns a *UIContent* instance which generates such a token lists for each lines.

The *Window* receives the *UIContent* and then:

5. It calculates the horizontal and vertical scrolling, if applicable (if the content would take more space than what is available).
6. The content is copied to the correct absolute position *Screen*, as requested by the *Renderer*. While doing this, the *Window* can possible wrap the lines, if line wrapping was configured.

Note that this process is lazy: if a certain line is not displayed in the *Window*, then it is not requested from the *UIContent*. And from there, the line is not passed through the processors or even asked from the *Lexer*.

Input processors

An *Processor* is an object that processes the tokens of a line in a *BufferControl* before it's passed to a *UIContent* instance.

Some build-in processors:

Processor	Usage:
HighlightSearchProcessor	Highlight the current search results.
HighlightSelectionProcessor	Highlight the selection.
PasswordProcessor	Display input as asterisks. ($\star$ characters).
BracketsMismatchProcessor	Highlight open/close mismatches for brackets.
BeforeInput	Insert some text before.
AfterInput	Insert some text after.
AppendAutoSuggestion	Append auto suggestion text.
ShowLeadingWhiteSpaceProcessor	Visualise leading whitespace.
ShowTrailingWhiteSpaceProcessor	Visualise trailing whitespace.
TabsProcessor	Visualise tabs as *n* spaces, or some symbols.

The `TokenListControl`

Custom user controls

The Window class

The *Window* class exposes many interesting functionality that influences the behaviour of user controls.

5.3.4 Buffers

5.3.5 The focus stack

5.3.6 The `Application` instance

The *Application* instance is where all the components for a prompt_toolkit application come together.

Note: Actually, not *all* the components; just everything that is not dependent on I/O (i.e. all components except for the eventloop and the input/output objects).

This way, it's possible to create an *Application* instance and later decide to run it on an asyncio eventloop or in a telnet server.

```
from prompt_toolkit.application import Application

application = Application(
    layout=layout,
    key_bindings_registry=registry,

    # Let's add mouse support as well.
    mouse_support=True,

    # For fullscreen:
    use_alternate_screen=True)
```

We are talking about full screen applications, so it's important to pass `use_alternate_screen=True`. This switches to the alternate terminal buffer.

5.3.7 Filters (reactivity)

Many places in *prompt_toolkit* expect a boolean. For instance, for determining the visibility of some part of the layout (it can be either hidden or visible), or a key binding filter (the binding can be active on not) or the `wrap_lines` option of *BufferControl*, etc.

These booleans however are often dynamic and can change at runtime. For instance, the search toolbar should only be visible when the user is actually searching (when the search buffer has the focus). The `wrap_lines` option could be changed with a certain key binding. And that key binding could only work when the default buffer got the focus.

In *prompt_toolkit*, we decided to reduce the amount of state in the whole framework, and apply a simple kind of reactive programming to describe the flow of these booleans as expressions. (It's one-way only: if a key binding needs to know whether it's active or not, it can follow this flow by evaluating an expression.)

There are two kind of expressions:

- *SimpleFilter*, which wraps an expression that takes no input, and evaluates to a boolean.
- *CLIFilter*, which takes a *CommandLineInterface* as input.

Most code in prompt_toolkit that expects a boolean will also accept a *CLIFilter*.

One way to create a *CLIFilter* instance is by creating a *Condition*. For instance, the following condition will evaluate to `True` when the user is searching:

```
from prompt_toolkit.filters import Condition
from prompt_toolkit.enums import DEFAULT_BUFFER

is_searching = Condition(lambda cli: cli.is_searching)
```

This filter can then be used in a key binding, like in the following snippet:

```
from prompt_toolkit.key_binding.manager import KeyBindingManager

manager = KeyBindingManager.for_prompt()

@manager.registry.add_binding(Keys.ControlT, filter=is_searching)
def _(event):
    # Do, something, but only when searching.
    pass
```

There are many built-in filters, ready to use:

- `HasArg`
- `HasCompletions`
- `HasFocus`
- `InFocusStack`
- `HasSearch`
- `HasSelection`
- `HasValidationError`
- `IsAborting`
- `IsDone`
- `IsMultiline`
- `IsReadOnly`
- `IsReturning`
- `RendererHeightIsKnown`

Further, these filters can be chained by the `&` and `|` operators or negated by the `~` operator.

Some examples:

```
from prompt_toolkit.key_binding.manager import KeyBindingManager
from prompt_toolkit.filters import HasSearch, HasSelection

manager = KeyBindingManager()

@manager.registry.add_binding(Keys.ControlT, filter=~is_searching)
def _(event):
    # Do, something, but not when when searching.
    pass

@manager.registry.add_binding(Keys.ControlT, filter=HasSearch() | HasSelection())
def _(event):
    # Do, something, but not when when searching.
    pass
```

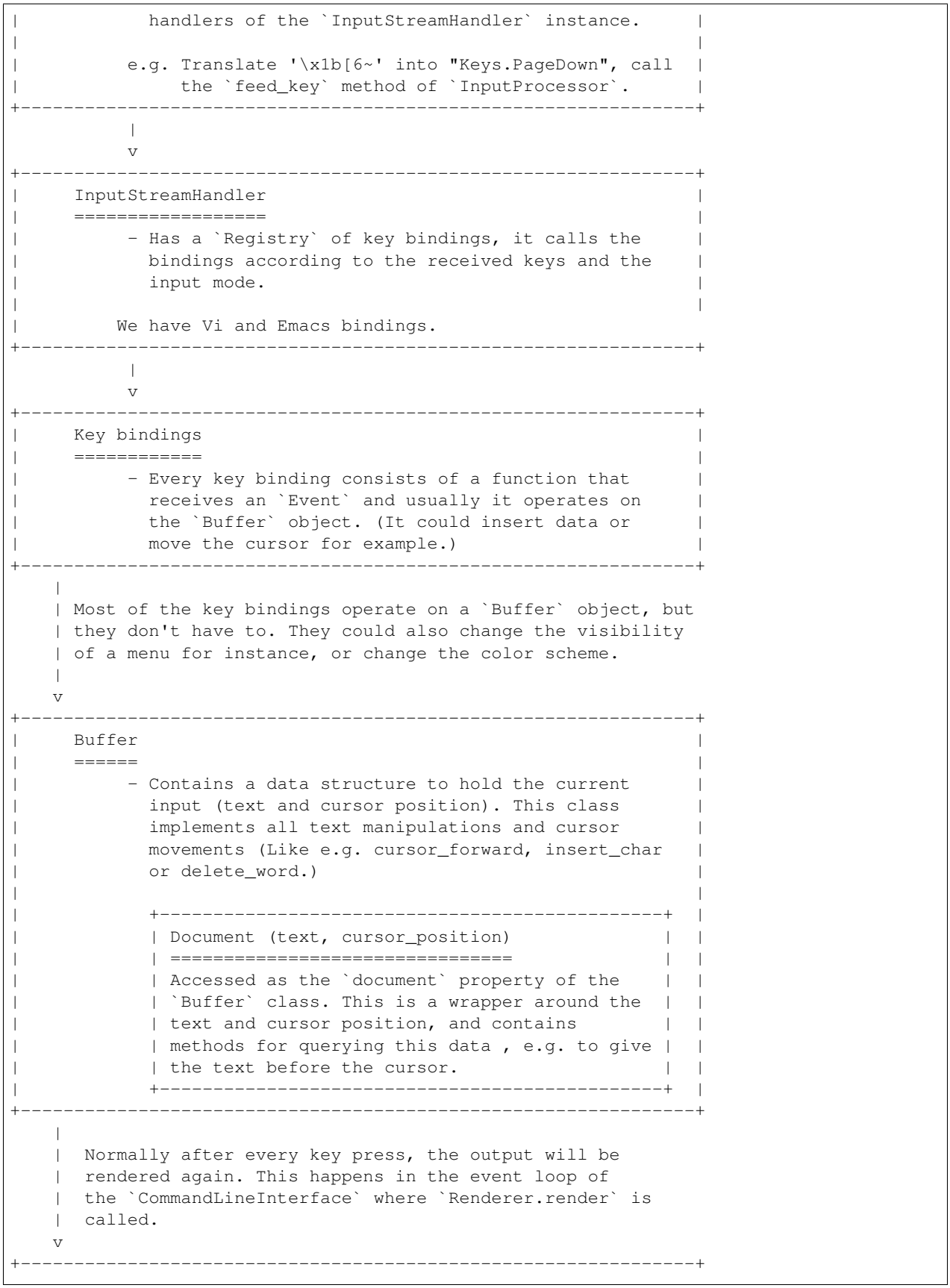
5.3.8 Input hooks

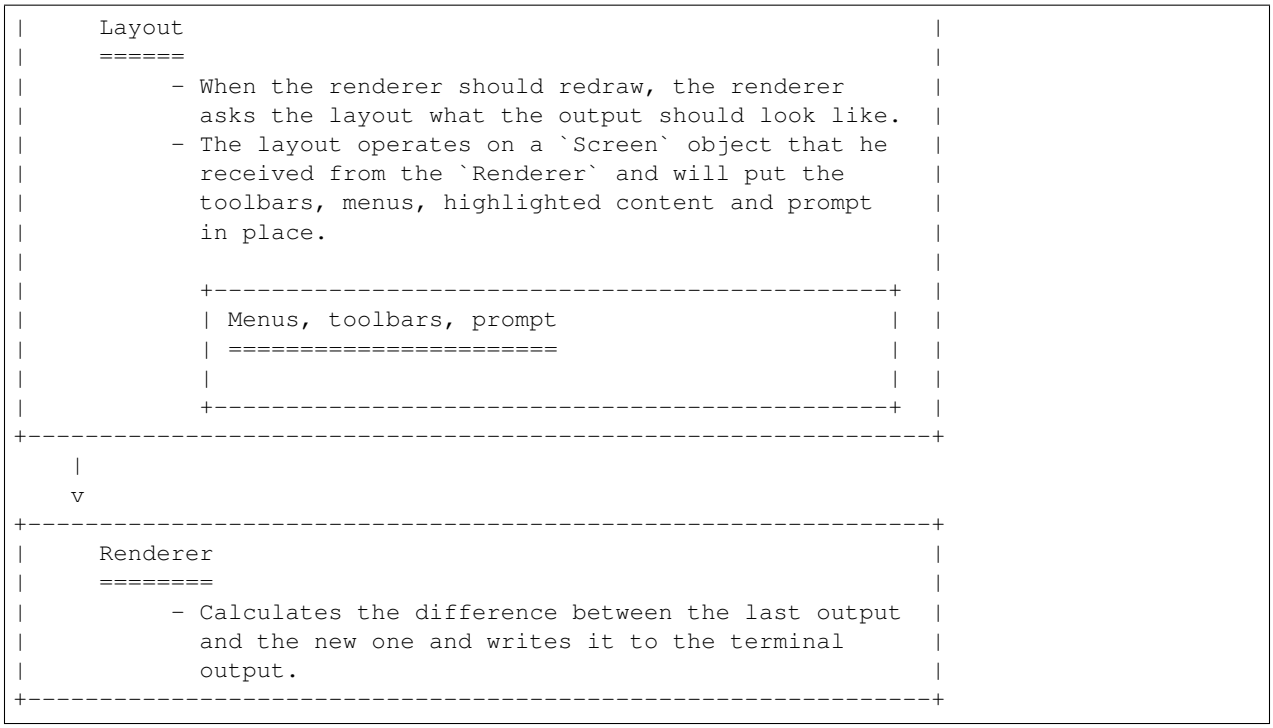
5.3.9 Running on the `asyncio` event loop

5.4 Architecture

TODO: this is a little outdated.

```
+--------------------------------------------------------------+
|     InputStream                                              |
|     ===========                                              |
|          - Parses the input stream coming from a VT100       |
|            compatible terminal. Translates it into data input |
|            and control characters. Calls the corresponding   |
```


handlers of the `InputStreamHandler` instance.
e.g. Translate '\x1b[6~' into "Keys.PageDown", call the `feed_key` method of `InputProcessor`.
InputStreamHandler
- Has a `Registry` of key bindings, it calls the bindings according to the received keys and the input mode.
We have Vi and Emacs bindings.
Key bindings
- Every key binding consists of a function that receives an `Event` and usually it operates on the `Buffer` object. (It could insert data or move the cursor for example.)
Most of the key bindings operate on a `Buffer` object, but they don't have to. They could also change the visibility of a menu for instance, or change the color scheme.
Buffer
- Contains a data structure to hold the current input (text and cursor position). This class implements all text manipulations and cursor movements (Like e.g. cursor_forward, insert_char or delete_word.)
Document (text, cursor_position)
Accessed as the `document` property of the `Buffer` class. This is a wrapper around the text and cursor position, and contains methods for querying this data , e.g. to give the text before the cursor.
Normally after every key press, the output will be rendered again. This happens in the event loop of the `CommandLineInterface` where `Renderer.render` is called.



5.5 Reference

5.5.1 Application

class prompt_toolkit.application.**AbortAction**
Actions to take on an Exit or Abort exception.

class prompt_toolkit.application.**Application**(*layout=None*, *buffer=None*, *buffers=None*, *initial_focussed_buffer=u'DEFAULT_BUFFER'*, *style=None*, *key_bindings_registry=None*, *clipboard=None*, *on_abort=u'raise-exception'*, *on_exit=u'raise-exception'*, *use_alternate_screen=False*, *mouse_support=False*, *get_title=None*, *paste_mode=False*, *ignore_case=False*, *editing_mode=u'EMACS'*, *erase_when_done=False*, *reverse_vi_search_direction=False*, *on_input_timeout=None*, *on_start=None*, *on_stop=None*, *on_reset=None*, *on_initialize=None*, *on_buffer_changed=None*, *on_render=None*, *on_invalidate=None*)
Application class to be passed to a *CommandLineInterface*.

This contains all customizable logic that is not I/O dependent. (So, what is independent of event loops, input and output.)

This way, such an *Application* can run easily on several *CommandLineInterface* instances, each with a different I/O backends. that runs for instance over telnet, SSH or any other I/O backend.

Parameters

- **layout** – A *Container* instance.
- **buffer** – A *Buffer* instance for the default buffer.
- **initial_focussed_buffer** – Name of the buffer that is focussed during start-up.
- **key_bindings_registry** – *BaseRegistry* instance for the key bindings.
- **clipboard** – *Clipboard* to use.
- **on_abort** – What to do when Control-C is pressed.
- **on_exit** – What to do when Control-D is pressed.
- **use_alternate_screen** – When True, run the application on the alternate screen buffer.
- **get_title** – Callable that returns the current title to be displayed in the terminal.
- **erase_when_done** – (bool) Clear the application output when it finishes.
- **reverse_vi_search_direction** – Normally, in Vi mode, a '/' searches forward and a '?' searches backward. In readline mode, this is usually reversed.

Filters:

Parameters

- **mouse_support** – (*CLIFilter* or boolean). When True, enable mouse support.
- **paste_mode** – *CLIFilter* or boolean.
- **ignore_case** – *CLIFilter* or boolean.
- **editing_mode** – `EditingMode`.

Callbacks (all of these should accept a *CommandLineInterface* object as input.)

Parameters

- **on_input_timeout** – Called when there is no input for x seconds. (Fired when any eventloop.onInputTimeout is fired.)
- **on_start** – Called when reading input starts.
- **on_stop** – Called when reading input ends.
- **on_reset** – Called during reset.
- **on_buffer_changed** – Called when the content of a buffer has been changed.
- **on_initialize** – Called after the *CommandLineInterface* initializes.
- **on_render** – Called right after rendering.
- **on_invalidate** – Called when the UI has been invalidated.

5.5.2 Buffer

Data structures for the Buffer. It holds the text, cursor position, history, etc...

exception `prompt_toolkit.buffer.`**EditReadOnlyBuffer**
Attempt editing of read-only *Buffer*.

class prompt_toolkit.buffer.**AcceptAction**(*handler=None*)

What to do when the input is accepted by the user. (When Enter was pressed in the command line.)

Parameters **handler** – (optional) A callable which takes a *CommandLineInterface* and *Document*. It is called when the user accepts input.

is_returnable

True when there is something handling accept.

classmethod run_in_terminal(*handler*, *render_cli_done=False*)

Create an *AcceptAction* that runs the given handler in the terminal.

Parameters **render_cli_done** – When True, render the interface in the 'Done' state first, then execute the function. If False, erase the interface instead.

validate_and_handle(*cli*, *buffer*)

Validate buffer and handle the accept action.

class prompt_toolkit.buffer.**Buffer**(*completer=None*, *auto_suggest=None*, *history=None*, *validator=None*, *tempfile_suffix=u''*, *is_multiline=False*, *complete_while_typing=False*, *enable_history_search=False*, *initial_document=None*, *accept_action=<prompt_toolkit.buffer.AcceptAction object>*, *read_only=False*, *on_text_changed=None*, *on_text_insert=None*, *on_cursor_position_changed=None*)

The core data structure that holds the text and cursor position of the current input line and implements all text manupulations on top of it. It also implements the history, undo stack and the completion state.

Parameters

- **completer** – *Completer* instance.
- **history** – *History* instance.
- **tempfile_suffix** – Suffix to be appended to the tempfile for the 'open in editor' function.

Events:

Parameters

- **on_text_changed** – When the buffer text changes. (Callable on None.)
- **on_text_insert** – When new text is inserted. (Callable on None.)
- **on_cursor_position_changed** – When the cursor moves. (Callable on None.)

Filters:

Parameters

- **is_multiline** – *SimpleFilter* to indicate whether we should consider this buffer a multiline input. If so, key bindings can decide to insert newlines when pressing [Enter]. (Instead of accepting the input.)
- **complete_while_typing** – *SimpleFilter* instance. Decide whether or not to do asynchronous autocompleting while typing.
- **enable_history_search** – *SimpleFilter* to indicate when up-arrow partial string matching is enabled. It is adviced to not enable this at the same time as *complete_while_typing*, because when there is an autocompletion found, the up arrows usually browse through the completions, rather than through the history.
- **read_only** – *SimpleFilter*. When True, changes will not be allowed.

append_to_history()
Append the current input to the history. (Only if valid input.)

apply_completion(*completion*)
Insert a given completion.

apply_search(*search_state*, *include_current_position=True*, *count=1*)
Apply search. If something is found, set *working_index* and *cursor_position*.

auto_down(*count=1*, *go_to_start_of_line_if_history_changes=False*)
If we're not on the last line (of a multiline input) go a line down, otherwise go forward in history. (If nothing is selected.)

auto_up(*count=1*, *go_to_start_of_line_if_history_changes=False*)
If we're not on the first line (of a multiline input) go a line up, otherwise go back in history. (If nothing is selected.)

cancel_completion()
Cancel completion, go back to the original text.

complete_next(*count=1*, *disable_wrap_around=False*)
Browse to the next completions. (Does nothing if there are no completion.)

complete_previous(*count=1*, *disable_wrap_around=False*)
Browse to the previous completions. (Does nothing if there are no completion.)

copy_selection(*_cut=False*)
Copy selected text and return *ClipboardData* instance.

cursor_down(*count=1*)
(for multiline edit). Move cursor to the next line.

cursor_up(*count=1*)
(for multiline edit). Move cursor to the previous line.

cut_selection()
Delete selected text and return *ClipboardData* instance.

delete(*count=1*)
Delete specified number of characters and Return the deleted text.

delete_before_cursor(*count=1*)
Delete specified number of characters before cursor and return the deleted text.

document
Return *Document* instance from the current text, cursor position and selection state.

document_for_search(*search_state*)
Return a *Document* instance that has the text/cursor position for this search, if we would apply it. This will be used in the *BufferControl* to display feedback while searching.

get_search_position(*search_state*, *include_current_position=True*, *count=1*)
Get the cursor position for this search. (This operation won't change the *working_index*. It's won't go through the history. Vi text objects can't span multiple items.)

go_to_completion(*index*)
Select a completion from the list of current completions.

go_to_history(*index*)
Go to this item in the history.

history_backward(*count=1*)
Move backwards through history.

history_forward(*count=1*)
Move forwards through the history.

Parameters **count** – Amount of items to move forward.

insert_line_above(*copy_margin=True*)
Insert a new line above the current one.

insert_line_below(*copy_margin=True*)
Insert a new line below the current one.

insert_text(*data*, *overwrite=False*, *move_cursor=True*, *fire_event=True*)
Insert characters at cursor position.

Parameters **fire_event** – Fire *on_text_insert* event. This is mainly used to trigger autocompletion while typing.

join_next_line(*separator=u' '*)
Join the next line to the current one by deleting the line ending after the current line.

join_selected_lines(*separator=u' '*)
Join the selected lines.

newline(*copy_margin=True*)
Insert a line ending at the current position.

open_in_editor(*cli*)
Open code in editor.

Parameters **cli** – *CommandLineInterface* instance.

paste_clipboard_data(*data*, *paste_mode=u'EMACS'*, *count=1*)
Insert the data from the clipboard.

reset(*initial_document=None*, *append_to_history=False*)

Parameters **append_to_history** – Append current input to history first.

save_to_undo_stack(*clear_redo_stack=True*)
Safe current state (input text and cursor position), so that we can restore it by calling undo.

set_completions(*completions*, *go_to_first=True*, *go_to_last=False*)
Start completions. (Generate list of completions and initialize.)

set_document(*value*, *bypass_readonly=False*)
Set *Document* instance. Like the `document` property, but accept an `bypass_readonly` argument.

Parameters **bypass_readonly** – When True, don't raise an *EditReadOnlyBuffer* exception, even when the buffer is read-only.

start_history_lines_completion()
Start a completion based on all the other lines in the document and the history.

start_selection(*selection_type=u'CHARACTERS'*)
Take the current cursor position as the start of this selection.

swap_characters_before_cursor()
Swap the last two characters before the cursor.

transform_current_line(*transform_callback*)
Apply the given transformation function to the current line.

Parameters **transform_callback** – callable that takes a string and return a new string.

transform_lines(*line_index_iterator*, *transform_callback*)
Transforms the text on a range of lines. When the iterator yield an index not in the range of lines that the document contains, it skips them silently.

To uppercase some lines:

```
new_text = transform_lines(range(5,10), lambda text: text.upper())
```

Parameters

- **line_index_iterator** – Iterator of line numbers (int)
- **transform_callback** – callable that takes the original text of a line, and return the new text for this line.

Returns The new text.

transform_region(*from_*, *to*, *transform_callback*)
Transform a part of the input string.

Parameters

- **from** – (int) start position.
- **to** – (int) end position.
- **transform_callback** – Callable which accepts a string and returns the transformed string.

validate()
Returns *True* if valid.

yank_last_arg(*n=None*)
Like *yank_nth_arg*, but if no argument has been given, yank the last word by default.

yank_nth_arg(*n=None*, *_yank_last_arg=False*)
Pick nth word from previous history entry (depending on current *yank_nth_arg_state*) and insert it at current position. Rotate through history if called repeatedly. If no *n* has been given, take the first argument. (The second word.)

Parameters **n** – (None or int), The index of the word from the previous line to take.

prompt_toolkit.buffer.**indent**(*buffer*, *from_row*, *to_row*, *count=1*)
Indent text of a *Buffer* object.

prompt_toolkit.buffer.**unindent**(*buffer*, *from_row*, *to_row*, *count=1*)
Unindent text of a *Buffer* object.

prompt_toolkit.buffer.**reshape_text**(*buffer*, *from_row*, *to_row*)
Reformat text, taking the width into account. *to_row* is included. (Vi 'gq' operator.)

5.5.3 Selection

Data structures for the selection.

class prompt_toolkit.selection.**SelectionType**
Type of selection.

class prompt_toolkit.selection.**SelectionState**(*original_cursor_position=0*, *type=u'CHARACTERS'*)
State of the current selection.

Parameters

- **original_cursor_position** – int
- **type** – *SelectionType*

5.5.4 Clipboard

Clipboard for command line interface.

class prompt_toolkit.clipboard.base.**Clipboard**

Abstract baseclass for clipboards. (An implementation can be in memory, it can share the X11 or Windows keyboard, or can be persistent.)

get_data()

Return clipboard data.

rotate()

For Emacs mode, rotate the kill ring.

set_data(*data*)

Set data to the clipboard.

Parameters **data** – *ClipboardData* instance.

set_text(*text*)

Shortcut for setting plain text on clipboard.

class prompt_toolkit.clipboard.base.**ClipboardData**(*text=u''*, *type=u'CHARACTERS'*)

Text on the clipboard.

Parameters

- **text** – string
- **type** – *SelectionType*

class prompt_toolkit.clipboard.in_memory.**InMemoryClipboard**(*data=None*, *max_size=60*)

Default clipboard implementation. Just keep the data in memory.

This implements a kill-ring, for Emacs mode.

5.5.5 Auto completion

class prompt_toolkit.completion.**Completion**(*text*, *start_position=0*, *display=None*, *display_meta=None*, *get_display_meta=None*)

Parameters

- **text** – The new string that will be inserted into the document.
- **start_position** – Position relative to the cursor_position where the new text will start. The text will be inserted between the start_position and the original cursor position.
- **display** – (optional string) If the completion has to be displayed differently in the completion menu.
- **display_meta** – (Optional string) Meta information about the completion, e.g. the path or source where it's coming from.
- **get_display_meta** – Lazy *display_meta*. Retrieve meta information only when meta is displayed.

new_completion_from_position(*position*)
(Only for internal use!) Get a new completion by splitting this one. Used by *CommandLineInterface* when it needs to have a list of new completions after inserting the common prefix.

class prompt_toolkit.completion.**Completer**
Base class for completer implementations.

get_completions(*document*, *complete_event*)
Yield *Completion* instances.

Parameters

- **document** – *Document* instance.
- **complete_event** – *CompleteEvent* instance.

class prompt_toolkit.completion.**CompleteEvent**(*text_inserted=False*, *completion_requested=False*)
Event that called the completer.

Parameters

- **text_inserted** – When True, it means that completions are requested because of a text insert. (*Buffer.complete_while_typing*.)
- **completion_requested** – When True, it means that the user explicitely pressed the *Tab* key in order to view the completions.

These two flags can be used for instance to implemented a completer that shows some completions when `Tab` has been pressed, but not automatically when the user presses a space. (Because of *complete_while_typing*.)

prompt_toolkit.completion.**get_common_complete_suffix**(*document*, *completions*)
Return the common prefix for all completions.

5.5.6 Document

The *Document* that implements all the text operations/querying.

class prompt_toolkit.document.**Document**(*text=u''*, *cursor_position=None*, *selection=None*)
This is a immutable class around the text and cursor position, and contains methods for querying this data, e.g. to give the text before the cursor.

This class is usually instantiated by a *Buffer* object, and accessed as the *document* property of that class.

Parameters

- **text** – string
- **cursor_position** – int
- **selection** – *SelectionState*

char_before_cursor
Return character before the cursor or an empty string.

current_char
Return character under cursor or an empty string.

current_line
Return the text on the line where the cursor is. (when the input consists of just one line, it equals *text*.

current_line_after_cursor
Text from the cursor until the end of the line.

current_line_before_cursor
Text from the start of the line until the cursor.

cursor_position
The document cursor position.

cursor_position_col
Current column. (0-based.)

cursor_position_row
Current row. (0-based.)

cut_selection()
Return a (*Document*, *ClipboardData*) tuple, where the document represents the new document when the selection is cut, and the clipboard data, represents whatever has to be put on the clipboard.

empty_line_count_at_the_end()
Return number of empty lines at the end of the document.

end_of_paragraph(*count=1*, *after=False*)
Return the end of the current paragraph. (Relative cursor position.)

find(*sub*, *in_current_line=False*, *include_current_position=False*, *ignore_case=False*, *count=1*)
Find *text* after the cursor, return position relative to the cursor position. Return *None* if nothing was found.

Parameters **count** – Find the n-th occurance.

find_all(*sub*, *ignore_case=False*)
Find all occurances of the substring. Return a list of absolute positions in the document.

find_backwards(*sub*, *in_current_line=False*, *ignore_case=False*, *count=1*)
Find *text* before the cursor, return position relative to the cursor position. Return *None* if nothing was found.

Parameters **count** – Find the n-th occurance.

find_boundaries_of_current_word(*WORD=False*, *include_leading_whitespace=False*, *include_trailing_whitespace=False*)
Return the relative boundaries (startpos, endpos) of the current word under the cursor. (This is at the current line, because line boundaries obviously don't belong to any word.) If not on a word, this returns (0,0)

find_enclosing_bracket_left(*left_ch*, *right_ch*, *start_pos=None*)
Find the left bracket enclosing current position. Return the relative position to the cursor position.

When *start_pos* is given, don't look past the position.

find_enclosing_bracket_right(*left_ch*, *right_ch*, *end_pos=None*)
Find the right bracket enclosing current position. Return the relative position to the cursor position.

When *end_pos* is given, don't look past the position.

find_matching_bracket_position(*start_pos=None*, *end_pos=None*)
Return relative cursor position of matching [, (, { or < bracket.

When *start_pos* or *end_pos* are given. Don't look past the positions.

find_next_matching_line(*match_func*, *count=1*)
Look downwards for empty lines. Return the line index, relative to the current line.

find_next_word_beginning(*count=1*, *WORD=False*)
Return an index relative to the cursor position pointing to the start of the next word. Return *None* if nothing was found.

find_next_word_ending(*include_current_position=False*, *count=1*, *WORD=False*)
Return an index relative to the cursor position pointing to the end of the next word. Return *None* if nothing was found.

find_previous_matching_line(*match_func*, *count=1*)
Look upwards for empty lines. Return the line index, relative to the current line.

find_previous_word_beginning(*count=1*, *WORD=False*)
Return an index relative to the cursor position pointing to the start of the previous word. Return *None* if nothing was found.

find_previous_word_ending(*count=1*, *WORD=False*)
Return an index relative to the cursor position pointing to the end of the previous word. Return *None* if nothing was found.

find_start_of_previous_word(*count=1*, *WORD=False*)
Return an index relative to the cursor position pointing to the start of the previous word. Return *None* if nothing was found.

get_column_cursor_position(*column*)
Return the relative cursor position for this column at the current line. (It will stay between the boundaries of the line in case of a larger number.)

get_cursor_down_position(*count=1*, *preferred_column=None*)
Return the relative cursor position (character index) where we would be if the user pressed the arrow-down button.

Parameters **preferred_column** – When given, go to this column instead of staying at the current column.

get_cursor_left_position(*count=1*)
Relative position for cursor left.

get_cursor_right_position(*count=1*)
Relative position for cursor_right.

get_cursor_up_position(*count=1*, *preferred_column=None*)
Return the relative cursor position (character index) where we would be if the user pressed the arrow-up button.

Parameters **preferred_column** – When given, go to this column instead of staying at the current column.

get_end_of_document_position()
Relative position for the end of the document.

get_end_of_line_position()
Relative position for the end of this line.

get_start_of_document_position()
Relative position for the start of the document.

get_start_of_line_position(*after_whitespace=False*)
Relative position for the start of this line.

get_word_before_cursor(*WORD=False*)
Give the word before the cursor. If we have whitespace before the cursor this returns an empty string.

get_word_under_cursor(*WORD=False*)
Return the word, currently below the cursor. This returns an empty string when the cursor is on a whitespace region.

has_match_at_current_position(*sub*)
True when this substring is found at the cursor position.

insert_after(*text*)
Create a new document, with this text inserted after the buffer. It keeps selection ranges and cursor position in sync.

insert_before(*text*)
Create a new document, with this text inserted before the buffer. It keeps selection ranges and cursor position in sync.

is_cursor_at_the_end
True when the cursor is at the end of the text.

is_cursor_at_the_end_of_line
True when the cursor is at the end of this line.

last_non_blank_of_current_line_position()
Relative position for the last non blank character of this line.

leading_whitespace_in_current_line
The leading whitespace in the left margin of the current line.

line_count
Return the number of lines in this document. If the document ends with a trailing n, that counts as the beginning of a new line.

lines
Array of all the lines.

lines_from_current
Array of the lines starting from the current line, until the last line.

on_first_line
True when we are at the first line.

on_last_line
True when we are at the last line.

paste_clipboard_data(*data*, *paste_mode=u'EMACS'*, *count=1*)
Return a new *Document* instance which contains the result if we would paste this data at the current cursor position.

Parameters

- **paste_mode** – Where to paste. (Before/after/emacs.)
- **count** – When >1, Paste multiple times.

selection
SelectionState object.

selection_range()
Return (from, to) tuple of the selection. start and end position are included.

This doesn't take the selection type into account. Use *selection_ranges* instead.

selection_range_at_line(*row*)
If the selection spans a portion of the given line, return a (from, to) tuple. Otherwise, return None.

selection_ranges()
Return a list of (from, to) tuples for the selection or none if nothing was selected. start and end position are always included in the selection.

This will yield several (from, to) tuples in case of a BLOCK selection.

start_of_paragraph(*count=1*, *before=False*)
Return the start of the current paragraph. (Relative cursor position.)

text
The document text.

translate_index_to_position(*index*)
Given an index for the text, return the corresponding (row, col) tuple. (0-based. Returns (0, 0) for index=0.)

translate_row_col_to_index(*row*, *col*)
Given a (row, col) tuple, return the corresponding index. (Row and col params are 0-based.)

Negative row/col values are turned into zero.

5.5.7 Enums

5.5.8 History

class prompt_toolkit.history.**FileHistory**(*filename*)
History class that stores all strings in a file.

class prompt_toolkit.history.**History**
Base History interface.

append(*string*)
Append string to history.

class prompt_toolkit.history.**InMemoryHistory**
History class that keeps a list of all strings in memory.

5.5.9 Interface

The main *CommandLineInterface* class and logic.

class prompt_toolkit.interface.**AbortAction**
Actions to take on an Exit or Abort exception.

class prompt_toolkit.interface.**CommandLineInterface**(*application*, *eventloop=None*, *input=None*, *output=None*)
Wrapper around all the other classes, tying everything together.

Typical usage:

```
application = Application(...)
cli = CommandLineInterface(application, eventloop)
result = cli.run()
print(result)
```

Parameters

- **application** – *Application* instance.
- **eventloop** – The *EventLoop* to be used when *run* is called. The easiest way to create an eventloop is by calling *create_eventloop()*.
- **input** – *Input* instance.
- **output** – *Output* instance. (Probably Vt100_Output or Win32Output.)

abort()
Set abort. When Control-C has been pressed.

add_buffer(*name*, *buffer*, *focus=False*)
Insert a new buffer.

current_buffer
The currently focussed *Buffer*.

(This returns a dummy *Buffer* when none of the actual buffers has the focus. In this case, it's really not practical to check for *None* values or catch exceptions every time.)

current_buffer_name
The name of the current *Buffer*. (Or *None*.)

exit()
Set exit. When Control-D has been pressed.

focus(*buffer_name*)
Focus the buffer with the given name on the focus stack.

in_paste_mode
True when we are in paste mode.

invalidate()
Thread safe way of sending a repaint trigger to the input event loop.

is_aborting
`True` when the abort flag as been set.

is_exiting
`True` when the exit flag as been set.

is_ignoring_case
True when we currently ignore casing.

is_returning
`True` when a return value has been set.

is_searching
True when we are searching.

patch_stdout_context(*raw=False*, *patch_stdout=True*, *patch_stderr=True*)
Return a context manager that will replace `sys.stdout` with a proxy that makes sure that all printed text will appear above the prompt, and that it doesn't destroy the output from the renderer.

Parameters

- **patch_stdout** – Replace *sys.stdout*.
- **patch_stderr** – Replace *sys.stderr*.

pop_focus()
Pop from the focus stack.

print_tokens(*tokens*, *style=None*)
Print a list of (Token, text) tuples to the output. (When the UI is running, this method has to be called through *run_in_terminal*, otherwise it will destroy the UI.)

Parameters **style** – Style class to use. Defaults to the active style in the CLI.

push_focus(*buffer_name*)
Push to the focus stack.

request_redraw()
Thread safe way of sending a repaint trigger to the input event loop.

reset(*reset_current_buffer=False*)
Reset everything, for reading the next input.

Parameters **reset_current_buffer** – XXX: not used anymore. The reason for having this option in the past was when this CommandLineInterface is run multiple times, that we could reset the buffer content from the previous run. This is now handled in the AcceptAction.

return_value()
Get the return value. Not that this method can throw an exception.

run(*reset_current_buffer=False*, *pre_run=None*)
Read input from the command line. This runs the eventloop until a return value has been set.

Parameters

- **reset_current_buffer** – XXX: Not used anymore.
- **pre_run** – Callable that is called right after the reset has taken place. This allows custom initialisation.

run_application_generator(*coroutine*, *render_cli_done=False*)
EXPERIMENTAL Like *run_in_terminal*, but takes a generator that can yield Application instances.

Example:

def f(): yield Application1(...) print('...') yield Application2(...)

cli.run_in_terminal_async(f)

The values which are yielded by the given coroutine are supposed to be *Application* instances that run in the current CLI, all other code is supposed to be CPU bound, so except for yielding the applications, there should not be any user interaction or I/O in the given function.

run_async(*reset_current_buffer=True*, *pre_run=None*)
Same as *run*, but this returns a coroutine.

This is only available on Python >3.3, with asyncio.

run_in_terminal(*func*, *render_cli_done=False*)
Run function on the terminal above the prompt.

What this does is first hiding the prompt, then running this callable (which can safely output to the terminal), and then again rendering the prompt which causes the output of this function to scroll above the prompt.

Parameters

- **func** – The callable to execute.
- **render_cli_done** – When True, render the interface in the 'Done' state first, then execute the function. If False, erase the interface first.

Returns the result of *func*.

run_sub_application(*application*, *done_callback=None*, *erase_when_done=False*, *_from_application_generator=False*)
Run a sub *Application*.

This will suspend the main application and display the sub application until that one returns a value. The value is returned by calling *done_callback* with the result.

The sub application will share the same I/O of the main application. That means, it uses the same input and output channels and it shares the same event loop.

Note: Technically, it gets another Eventloop instance, but that is only a proxy to our main event loop. The reason is that calling 'stop' –which returns the result of an application when it's done– is handled differently.

---

run_system_command(*command*)
Run system command (While hiding the prompt. When finished, all the output will scroll above the prompt.)

Parameters **command** – Shell command to be executed.

set_abort()
Set abort. When Control-C has been pressed.

set_exit()
Set exit. When Control-D has been pressed.

set_return_value(*document*)
Set a return value. The eventloop can retrieve the result it by calling *return_value*.

start_completion(*buffer_name=None*, *select_first=False*, *select_last=False*, *insert_common_part=False*, *complete_event=None*)
Start asynchronous autocompletion of this buffer. (This will do nothing if a previous completion was still in progress.)

stdout_proxy(*raw=False*)
Create an `_StdoutProxy` class which can be used as a patch for *sys.stdout*. Writing to this proxy will make sure that the text appears above the prompt, and that it doesn't destroy the output from the renderer.

Parameters **raw** – (*bool*) When True, vt100 terminal escape sequences are not removed/escaped.

suspend_to_background(*suspend_group=True*)
(Not thread safe – to be called from inside the key bindings.) Suspend process.

Parameters **suspend_group** – When true, suspend the whole process group. (This is the default, and probably what you want.)

terminal_title
Return the current title to be displayed in the terminal. When this in *None*, the terminal title remains the original.

5.5.10 Keys

5.5.11 Style

Styling for prompt_toolkit applications.

5.5.12 Reactive

Prompt_toolkit is designed a way that the amount of changing state is reduced to a minimum. Where possible, code is written in a pure functional way. In general, this results in code where the flow is very easy to follow: the value of a variable can be deducted from its first assignment.

However, often, practicality and performance beat purity and some classes still have a changing state. In order to not having to care too much about transferring states between several components we use some reactive programming. Actually some kind of data binding.

We introduce two types:

- Filter: for binding a boolean state. They can be chained using & and | operators. Have a look in the `filters` module. Resolving the actual value of a filter happens by calling it.
- Integer: for binding integer values. Reactive operations (like addition and substraction) are not suppported. Resolving the actual value happens by casting it to int, like `int(integer)`. This way, it is possible to use normal integers as well for static values.

class prompt_toolkit.reactive.**Integer**
Reactive integer – anything that can be resolved to an `int`.

classmethod from_callable(*func*)
Create an Integer-like object that calls the given function when it is resolved to an int.

5.5.13 Shortcuts

Shortcuts for retrieving input from the user.

If you are using this library for retrieving some input from the user (as a pure Python replacement for GNU readline), probably for 90% of the use cases, the *prompt()* function is all you need. It's the easiest shortcut which does a lot of the underlying work like creating a *CommandLineInterface* instance for you.

When is this not sufficient:

- When you want to have more complicated layouts (maybe with sidebars or multiple toolbars. Or visibility of certain user interface controls according to some conditions.)
- When you wish to have multiple input buffers. (If you would create an editor like a Vi clone.)
- Something else that requires more customization than what is possible with the parameters of *prompt*.

In that case, study the code in this file and build your own *CommandLineInterface* instance. It's not too complicated.

prompt_toolkit.shortcuts.**create_eventloop**(*inputhook=None*, *recognize_win32_paste=True*)
Create and return an *EventLoop* instance for a *CommandLineInterface*.

prompt_toolkit.shortcuts.**create_output**(*stdout=None*, *true_color=False*, *ansi_colors_only=None*)
Return an *Output* instance for the command line.

Parameters

- **true_color** – When True, use 24bit colors instead of 256 colors. (*bool* or *SimpleFilter*.)
- **ansi_colors_only** – When True, restrict to 16 ANSI colors only. (*bool* or *SimpleFilter*.)

prompt_toolkit.shortcuts.**create_prompt_layout**(*message=u”*, *lexer=None*, *is_password=False*, *reserve_space_for_menu=8*, *get_prompt_tokens=None*, *get_continuation_tokens=None*, *get_rprompt_tokens=None*, *get_bottom_toolbar_tokens=None*, *display_completions_in_columns=False*, *extra_input_processors=None*, *multiline=False*, *wrap_lines=True*)

Create a *Container* instance for a prompt.

Parameters

- **message** – Text to be used as prompt.
- **lexer** – *Lexer* to be used for the highlighting.
- **is_password** – *bool* or *CLIFilter*. When True, display input as ‘*’.
- **reserve_space_for_menu** – Space to be reserved for the menu. When >0, make sure that a minimal height is allocated in the terminal, in order to display the completion menu.
- **get_prompt_tokens** – An optional callable that returns the tokens to be shown in the menu. (To be used instead of a *message*.)
- **get_continuation_tokens** – An optional callable that takes a CommandLineInterface and width as input and returns a list of (Token, text) tuples to be used for the continuation.
- **get_bottom_toolbar_tokens** – An optional callable that returns the tokens for a toolbar at the bottom.
- **display_completions_in_columns** – *bool* or *CLIFilter*. Display the completions in multiple columns.
- **multiline** – *bool* or *CLIFilter*. When True, prefer a layout that is more adapted for multiline input. Text after newlines is automatically indented, and search/arg input is shown below the input, instead of replacing the prompt.
- **wrap_lines** – *bool* or *CLIFilter*. When True (the default), automatically wrap long lines instead of scrolling horizontally.

prompt_toolkit.shortcuts.**create_prompt_application**(*message=u''*, *multiline=False*, *wrap_lines=True*, *is_password=False*, *vi_mode=False*, *editing_mode=u'EMACS'*, *complete_while_typing=True*, *enable_history_search=False*, *lexer=None*, *enable_system_bindings=False*, *enable_open_in_editor=False*, *validator=None*, *completer=None*, *reserve_space_for_menu=8*, *auto_suggest=None*, *style=None*, *history=None*, *clipboard=None*, *get_prompt_tokens=None*, *get_continuation_tokens=None*, *get_rprompt_tokens=None*, *get_bottom_toolbar_tokens=None*, *display_completions_in_columns=False*, *get_title=None*, *mouse_support=False*, *extra_input_processors=None*, *key_bindings_registry=None*, *on_abort=u'raise-exception'*, *on_exit=u'raise-exception'*, *accept_action=<prompt_toolkit.buffer.AcceptAction object>*, *erase_when_done=False*, *default=u''*)

Create an `Application` instance for a prompt.

(It is meant to cover 90% of the prompt use cases, where no extreme customization is required. For more complex input, it is required to create a custom `Application` instance.)

Parameters

- **message** – Text to be shown before the prompt.
- **mulitiline** – Allow multiline input. Pressing enter will insert a newline. (This requires Meta+Enter to accept the input.)
- **wrap_lines** – *bool* or *CLIFilter*. When True (the default), automatically wrap long lines instead of scrolling horizontally.
- **is_password** – Show asterisks instead of the actual typed characters.
- **editing_mode** – `EditingMode.VI` or `EditingMode.EMACS`.
- **vi_mode** – *bool*, if True, Identical to `editing_mode=EditingMode.VI`.
- **complete_while_typing** – *bool* or *SimpleFilter*. Enable autocompletion while typing.
- **enable_history_search** – *bool* or *SimpleFilter*. Enable up-arrow parting string matching.
- **lexer** – *Lexer* to be used for the syntax highlighting.
- **validator** – *Validator* instance for input validation.
- **completer** – *Completer* instance for input completion.

- **reserve_space_for_menu** – Space to be reserved for displaying the menu. (0 means that no space needs to be reserved.)
- **auto_suggest** – *AutoSuggest* instance for input suggestions.
- **style** – `Style` instance for the color scheme.
- **enable_system_bindings** – *bool* or *CLIFilter*. Pressing Meta+'!' will show a system prompt.
- **enable_open_in_editor** – *bool* or *CLIFilter*. Pressing 'v' in Vi mode or C-X C-E in emacs mode will open an external editor.
- **history** – *History* instance.
- **clipboard** – *Clipboard* instance. (e.g. *InMemoryClipboard*)
- **get_bottom_toolbar_tokens** – Optional callable which takes a *CommandLineInterface* and returns a list of tokens for the bottom toolbar.
- **display_completions_in_columns** – *bool* or *CLIFilter*. Display the completions in multiple columns.
- **get_title** – Callable that returns the title to be displayed in the terminal.
- **mouse_support** – *bool* or *CLIFilter* to enable mouse support.
- **default** – The default text to be shown in the input buffer. (This can be edited by the user.)

prompt_toolkit.shortcuts.**prompt**(*message=u''*, ***kwargs*)

Get input from the user and return it.

This is a wrapper around a lot of `prompt_toolkit` functionality and can be a replacement for *raw_input*. (or GNU readline.)

If you want to keep your history across several calls, create one *History* instance and pass it every time.

This function accepts many keyword arguments. Except for the following, they are a proxy to the arguments of *create_prompt_application()*.

Parameters

- **patch_stdout** – Replace `sys.stdout` by a proxy that ensures that print statements from other threads won't destroy the prompt. (They will be printed above the prompt instead.)
- **return_asyncio_coroutine** – When True, return a asyncio coroutine. (Python >3.3)
- **true_color** – When True, use 24bit colors instead of 256 colors.
- **refresh_interval** – (number; in seconds) When given, refresh the UI every so many seconds.

prompt_toolkit.shortcuts.**prompt_async**(*message=u''*, ***kwargs*)

Similar to *prompt()*, but return an asyncio coroutine instead.

prompt_toolkit.shortcuts.**create_confirm_application**(*message*)

Create a confirmation *Application* that returns True/False.

prompt_toolkit.shortcuts.**run_application**(*application*, *patch_stdout=False*, *return_asyncio_coroutine=False*, *true_color=False*, *refresh_interval=0*, *eventloop=None*)

Run a prompt toolkit application.

Parameters

- **patch_stdout** – Replace `sys.stdout` by a proxy that ensures that print statements from other threads won't destroy the prompt. (They will be printed above the prompt instead.)
- **return_asyncio_coroutine** – When True, return a asyncio coroutine. (Python >3.3)
- **true_color** – When True, use 24bit colors instead of 256 colors.
- **refresh_interval** – (number; in seconds) When given, refresh the UI every so many seconds.

prompt_toolkit.shortcuts.**confirm**(*message=u'Confirm (y or n) '*)
Display a confirmation prompt.

prompt_toolkit.shortcuts.**print_tokens**(*tokens*, *style=None*, *true_color=False*, *file=None*)
Print a list of (Token, text) tuples in the given style to the output. E.g.:

```
style = style_from_dict({
    Token.Hello: '#ff0066',
    Token.World: '#884444 italic',
})
tokens = [
    (Token.Hello, 'Hello'),
    (Token.World, 'World'),
]
print_tokens(tokens, style=style)
```

Parameters

- **tokens** – List of `(Token, text)` tuples.
- **style** – `Style` instance for the color scheme.
- **true_color** – When True, use 24bit colors instead of 256 colors.
- **file** – The output file. This can be *sys.stdout* or *sys.stderr*.

prompt_toolkit.shortcuts.**clear**()
Clear the screen.

5.5.14 Validation

Input validation for a *Buffer*. (Validators will be called before accepting input.)

class prompt_toolkit.validation.**ConditionalValidator**(*validator*, *filter*)
Validator that can be switched on/off according to a filter. (This wraps around another validator.)

exception prompt_toolkit.validation.**ValidationError**(*cursor_position=0*, *message=u''*)
Error raised by *Validator.validate()*.

Parameters

- **cursor_position** – The cursor position where the error occured.
- **message** – Text.

class prompt_toolkit.validation.**Validator**
Abstract base class for an input validator.

validate(*document*)
Validate the input. If invalid, this should raise a *ValidationError*.

Parameters **document** – *Document* instance.

5.5.15 Auto suggestion

Fish-style like auto-suggestion.

While a user types input in a certain buffer, suggestions are generated (asynchronously.) Usually, they are displayed after the input. When the cursor presses the right arrow and the cursor is at the end of the input, the suggestion will be inserted.

class prompt_toolkit.auto_suggest.**Suggestion**(*text*)
Suggestion returned by an auto-suggest algorithm.

Parameters **text** – The suggestion text.

class prompt_toolkit.auto_suggest.**AutoSuggest**
Base class for auto suggestion implementations.

get_suggestion(*cli*, *buffer*, *document*)
Return *None* or a *Suggestion* instance.

We receive both `buffer` and `document`. The reason is that auto suggestions are retrieved asynchronously. (Like completions.) The buffer text could be changed in the meantime, but `document` contains the buffer document like it was at the start of the auto suggestion call. So, from here, don't access `buffer.text`, but use `document.text` instead.

Parameters

- **buffer** – The *Buffer* instance.
- **document** – The *Document* instance.

class prompt_toolkit.auto_suggest.**AutoSuggestFromHistory**
Give suggestions based on the lines in the history.

class prompt_toolkit.auto_suggest.**ConditionalAutoSuggest**(*auto_suggest*, *filter*)
Auto suggest that can be turned on and of according to a certain condition.

5.5.16 Renderer

Renders the command line on the console. (Redraws parts of the input line that were changed.)

class prompt_toolkit.renderer.**Renderer**(*style*, *output*, *use_alternate_screen=False*, *mouse_support=False*)
Typical usage:

```
output = Vt100_Output.from_pty(sys.stdout)
r = Renderer(style, output)
r.render(cli, layout=...)
```

clear()
Clear screen and go to 0,0

erase(*leave_alternate_screen=True*, *erase_title=True*)
Hide all output and put the cursor back at the first line. This is for instance used for running a system command (while hiding the CLI) and later resuming the same CLI.)

Parameters

- **leave_alternate_screen** – When True, and when inside an alternate screen buffer, quit the alternate screen.
- **erase_title** – When True, clear the title from the title bar.

height_is_known
True when the height from the cursor until the bottom of the terminal is known. (It's often nicer to draw bottom toolbars only if the height is known, in order to avoid flickering when the CPR response arrives.)

render(*cli*, *layout*, *is_done=False*)
Render the current interface to the output.

Parameters **is_done** – When True, put the cursor at the end of the interface. We won't print any changes to this part.

report_absolute_cursor_row(*row*)
To be called when we know the absolute cursor position. (As an answer of a "Cursor Position Request" response.)

request_absolute_cursor_position()
Get current cursor position. For vt100: Do CPR request. (answer will arrive later.) For win32: Do API call. (Answer comes immediately.)

rows_above_layout
Return the number of rows visible in the terminal above the layout.

prompt_toolkit.renderer.**print_tokens**(*output*, *tokens*, *style*)
Print a list of (Token, text) tuples in the given style to the output.

5.5.17 Layout

Container for the layout. (Containers can contain other containers or user interface controls.)

class prompt_toolkit.layout.containers.**Container**
Base class for user interface layout.

preferred_height(*cli*, *width*, *max_available_height*)
Return a *LayoutDimension* that represents the desired height for this container.

Parameters **cli** – *CommandLineInterface*.

preferred_width(*cli*, *max_available_width*)
Return a *LayoutDimension* that represents the desired width for this container.

Parameters **cli** – *CommandLineInterface*.

reset()
Reset the state of this container and all the children. (E.g. reset scroll offsets, etc...)

walk(*cli*)
Walk through all the layout nodes (and their children) and yield them.

write_to_screen(*cli*, *screen*, *mouse_handlers*, *write_position*)
Write the actual content to the screen.

Parameters

- **cli** – *CommandLineInterface*.
- **screen** – *Screen*
- **mouse_handlers** – MouseHandlers.

class prompt_toolkit.layout.containers.**HSplit**(*children*, *window_too_small=None*, *get_dimensions=None*, *report_dimensions_callback=None*)

Several layouts, one stacked above/under the other.

Parameters

- **children** – List of child *Container* objects.
- **window_too_small** – A *Container* object that is displayed if there is not enough space for all the children. By default, this is a "Window too small" message.
- **get_dimensions** – (*None* or a callable that takes a *CommandLineInterface* and returns a list of *LayoutDimension* instances.) By default the dimensions are taken from the children and divided by the available space. However, when *get_dimensions* is specified, this is taken instead.
- **report_dimensions_callback** – When rendering, this function is called with the *CommandLineInterface* and the list of used dimensions. (As a list of integers.)

walk(*cli*)

Walk through children.

write_to_screen(*cli*, *screen*, *mouse_handlers*, *write_position*)

Render the prompt to a *Screen* instance.

Parameters **screen** – The *Screen* class to which the output has to be written.

class prompt_toolkit.layout.containers.**VSplit**(*children*, *window_too_small=None*, *get_dimensions=None*, *report_dimensions_callback=None*)

Several layouts, one stacked left/right of the other.

Parameters

- **children** – List of child *Container* objects.
- **window_too_small** – A *Container* object that is displayed if there is not enough space for all the children. By default, this is a "Window too small" message.
- **get_dimensions** – (*None* or a callable that takes a *CommandLineInterface* and returns a list of *LayoutDimension* instances.) By default the dimensions are taken from the children and divided by the available space. However, when *get_dimensions* is specified, this is taken instead.
- **report_dimensions_callback** – When rendering, this function is called with the *CommandLineInterface* and the list of used dimensions. (As a list of integers.)

walk(*cli*)

Walk through children.

write_to_screen(*cli*, *screen*, *mouse_handlers*, *write_position*)

Render the prompt to a *Screen* instance.

Parameters **screen** – The *Screen* class to which the output has to be written.

class prompt_toolkit.layout.containers.**FloatContainer**(*content*, *floats*)

Container which can contain another container for the background, as well as a list of floating containers on top of it.

Example Usage:

```
FloatContainer(content=Window(...),
               floats=[
                   Float(xcursor=True,
                         ycursor=True,
                         layout=CompletionMenu(...))
               ])
```

preferred_height(*cli*, *width*, *max_available_height*)
Return the preferred height of the float container. (We don't care about the height of the floats, they should always fit into the dimensions provided by the container.)

walk(*cli*)
Walk through children.

class prompt_toolkit.layout.containers.**Float**(*top=None*, *right=None*, *bottom=None*, *left=None*, *width=None*, *height=None*, *get_width=None*, *get_height=None*, *xcursor=False*, *ycursor=False*, *content=None*, *hide_when_covering_content=False*)
Float for use in a *FloatContainer*.

Parameters

- **content** – *Container* instance.
- **hide_when_covering_content** – Hide the float when it covers content underneath.

class prompt_toolkit.layout.containers.**Window**(*content*, *width=None*, *height=None*, *get_width=None*, *get_height=None*, *dont_extend_width=False*, *dont_extend_height=False*, *left_margins=None*, *right_margins=None*, *scroll_offsets=None*, *allow_scroll_beyond_bottom=False*, *wrap_lines=False*, *get_vertical_scroll=None*, *get_horizontal_scroll=None*, *always_hide_cursor=False*, *cursorline=False*, *cursorcolumn=False*, *get_colorcolumns=None*, *cursorline_token=Token.CursorLine*, *cursorcolumn_token=Token.CursorColumn*)
Container that holds a control.

Parameters

- **content** – *UIControl* instance.
- **width** – *LayoutDimension* instance.
- **height** – *LayoutDimension* instance.
- **get_width** – callable which takes a *CommandLineInterface* and returns a *LayoutDimension*.
- **get_height** – callable which takes a *CommandLineInterface* and returns a *LayoutDimension*.
- **dont_extend_width** – When *True*, don't take up more width then the preferred width reported by the control.

- **dont_extend_height** – When *True*, don't take up more width then the preferred height reported by the control.
- **left_margins** – A list of *Margin* instance to be displayed on the left. For instance: *NumberredMargin* can be one of them in order to show line numbers.
- **right_margins** – Like *left_margins*, but on the other side.
- **scroll_offsets** – *ScrollOffsets* instance, representing the preferred amount of lines/columns to be always visible before/after the cursor. When both top and bottom are a very high number, the cursor will be centered vertically most of the time.
- **allow_scroll_beyond_bottom** – A *bool* or *CLIFilter* instance. When True, allow scrolling so far, that the top part of the content is not visible anymore, while there is still empty space available at the bottom of the window. In the Vi editor for instance, this is possible. You will see tildes while the top part of the body is hidden.
- **wrap_lines** – A *bool* or *CLIFilter* instance. When True, don't scroll horizontally, but wrap lines instead.
- **get_vertical_scroll** – Callable that takes this window instance as input and returns a preferred vertical scroll. (When this is *None*, the scroll is only determined by the last and current cursor position.)
- **get_horizontal_scroll** – Callable that takes this window instance as input and returns a preferred vertical scroll.
- **always_hide_cursor** – A *bool* or *CLIFilter* instance. When True, never display the cursor, even when the user control specifies a cursor position.
- **cursorline** – A *bool* or *CLIFilter* instance. When True, display a cursorline.
- **cursorcolumn** – A *bool* or *CLIFilter* instance. When True, display a cursorcolumn.
- **get_colorcolumns** – A callable that takes a *CommandLineInterface* and returns a a list of `ColorColumn` instances that describe the columns to be highlighted.
- **cursorline_token** – The token to be used for highlighting the current line, if *cursorline* is True.
- **cursorcolumn_token** – The token to be used for highlighting the current line, if *cursorcolumn* is True.

write_to_screen(*cli*, *screen*, *mouse_handlers*, *write_position*)

Write window to screen. This renders the user control, the margins and copies everything over to the absolute position at the given screen.

class prompt_toolkit.layout.containers.**WindowRenderInfo**(*ui_content*, *horizontal_scroll*, *vertical_scroll*, *window_width*, *window_height*, *configured_scroll_offsets*, *visible_line_to_row_col*, *rowcol_to_yx*, *x_offset*, *y_offset*, *wrap_lines*)

Render information, for the last render time of this control. It stores mapping information between the input buffers (in case of a *BufferControl*) and the actual render position on the output screen.

(Could be used for implementation of the Vi 'H' and 'L' key bindings as well as implementing mouse support.)

Parameters

- **ui_content** – The original *UIContent* instance that contains the whole input, without clipping. (ui_content)
- **horizontal_scroll** – The horizontal scroll of the *Window* instance.
- **vertical_scroll** – The vertical scroll of the *Window* instance.
- **window_width** – The width of the window that displays the content, without the margins.
- **window_height** – The height of the window that displays the content.
- **configured_scroll_offsets** – The scroll offsets as configured for the *Window* instance.
- **visible_line_to_row_col** – Mapping that maps the row numbers on the displayed screen (starting from zero for the first visible line) to (row, col) tuples pointing to the row and column of the *UIContent*.
- **rowcol_to_yx** – Mapping that maps (row, column) tuples representing coordinates of the `UIContent` to (y, x) absolute coordinates at the rendered screen.

applied_scroll_offsets
Return a *ScrollOffsets* instance that indicates the actual offset. This can be less than or equal to what's configured. E.g, when the cursor is completely at the top, the top offset will be zero rather than what's configured.

bottom_visible
True when the bottom of the buffer is visible.

center_visible_line(*before_scroll_offset=False*, *after_scroll_offset=False*)
Like *first_visible_line*, but for the center visible line.

content_height
The full height of the user control.

cursor_position
Return the cursor position coordinates, relative to the left/top corner of the rendered screen.

displayed_lines
List of all the visible rows. (Line numbers of the input buffer.) The last line may not be entirely visible.

first_visible_line(*after_scroll_offset=False*)
Return the line number (0 based) of the input document that corresponds with the first visible line.

full_height_visible
True when the full height is visible (There is no vertical scroll.)

get_height_for_line(*lineno*)
Return the height of the given line. (The height that it would take, if this line became visible.)

input_line_to_visible_line
Return the dictionary mapping the line numbers of the input buffer to the lines of the screen. When a line spans several rows at the screen, the first row appears in the dictionary.

last_visible_line(*before_scroll_offset=False*)
Like *first_visible_line*, but for the last visible line.

top_visible
True when the top of the buffer is visible.

vertical_scroll_percentage
Vertical scroll as a percentage. (0 means: the top is visible, 100 means: the bottom is visible.)

class prompt_toolkit.layout.containers.**ConditionalContainer**(*content*, *filter*)
Wrapper around any other container that can change the visibility. The received *filter* determines whether the given container should be displayed or not.

Parameters

- **content** – *Container* instance.
- **filter** – *CLIFilter* instance.

class prompt_toolkit.layout.containers.**ScrollOffsets**(*top=0*, *bottom=0*, *left=0*, *right=0*)
Scroll offsets for the *Window* class.

Note that left/right offsets only make sense if line wrapping is disabled.

User interface Controls for the layout.

class prompt_toolkit.layout.controls.**BufferControl**(*buffer_name=u'DEFAULT_BUFFER'*, *input_processors=None*, *lexer=None*, *preview_search=False*, *search_buffer_name=u'SEARCH_BUFFER'*, *get_search_state=None*, *menu_position=None*, *default_char=None*, *focus_on_click=False*)
Control for visualising the content of a *Buffer*.

Parameters

- **input_processors** – list of *Processor*.
- **lexer** – *Lexer* instance for syntax highlighting.
- **preview_search** – *bool* or *CLIFilter*: Show search while typing.
- **get_search_state** – Callable that takes a CommandLineInterface and returns the SearchState to be used. (If not CommandLineInterface.search_state.)
- **buffer_name** – String representing the name of the buffer to display.
- **default_char** – *Char* instance to use to fill the background. This is transparent by default.
- **focus_on_click** – Focus this buffer when it's click, but not yet focussed.

create_content(*cli*, *width*, *height*)
Create a UIContent.

mouse_handler(*cli*, *mouse_event*)
Mouse handler for this control.

preferred_width(*cli*, *max_available_width*)
This should return the preferred width.

Note: We don't specify a preferred width according to the content, because it would be too expensive. Calculating the preferred width can be done by calculating the longest line, but this would require applying all the processors to each line. This is unfeasible for a larger document, and doing it for small documents only would result in inconsistent behaviour.

class prompt_toolkit.layout.controls.**FillControl**(*character=None*, *token=Token*, *char=None*, *get_char=None*)
Fill whole control with characters with this token. (Also helpful for debugging.)

Parameters

- **char** – *Char* instance to use for filling.
- **get_char** – A callable that takes a CommandLineInterface and returns a *Char* object.

class prompt_toolkit.layout.controls.**TokenListControl**(*get_tokens*, *default_char=None*, *get_default_char=None*, *align_right=False*, *align_center=False*, *has_focus=False*)

Control that displays a list of (Token, text) tuples. (It's mostly optimized for rather small widgets, like toolbars, menus, etc...)

Mouse support:

The list of tokens can also contain tuples of three items, looking like: (Token, text, handler). When mouse support is enabled and the user clicks on this token, then the given handler is called. That handler should accept two inputs: (CommandLineInterface, MouseEvent) and it should either handle the event or return *NotImplemented* in case we want the containing Window to handle this event.

Parameters

- **get_tokens** – Callable that takes a *CommandLineInterface* instance and returns the list of (Token, text) tuples to be displayed right now.
- **default_char** – default *Char* (character and Token) to use for the background when there is more space available than *get_tokens* returns.
- **get_default_char** – Like *default_char*, but this is a callable that takes a *prompt_toolkit.interface.CommandLineInterface* and returns a *Char* instance.
- **has_focus** – *bool* or *CLIFilter*, when this evaluates to *True*, this UI control will take the focus. The cursor will be shown in the upper left corner of this control, unless *get_token* returns a `Token.SetCursorPosition` token somewhere in the token list, then the cursor will be shown there.

mouse_handler(*cli*, *mouse_event*)

Handle mouse events.

(When the token list contained mouse handlers and the user clicked on on any of these, the matching handler is called. This handler can still return *NotImplemented* in case we want the *Window* to handle this particular event.)

preferred_width(*cli*, *max_available_width*)

Return the preferred width for this control. That is the width of the longest line.

class prompt_toolkit.layout.controls.**UIControl**

Base class for all user interface controls.

create_content(*cli*, *width*, *height*)

Generate the content for this user control.

Returns a *UIContent* instance.

has_focus(*cli*)

Return `True` when this user control has the focus.

If so, the cursor will be displayed according to the cursor position reported by *UIControl.create_content()*. If the created content has the property `show_cursor=False`, the cursor will be hidden from the output.

mouse_handler(*cli*, *mouse_event*)
Handle mouse events.

When *NotImplemented* is returned, it means that the given event is not handled by the *UIControl* itself. The *Window* or key bindings can decide to handle this event as scrolling or changing focus.

Parameters

- **cli** – *CommandLineInterface* instance.
- **mouse_event** – *MouseEvent* instance.

move_cursor_down(*cli*)
Request to move the cursor down. This happens when scrolling down and the cursor is completely at the top.

move_cursor_up(*cli*)
Request to move the cursor up.

class prompt_toolkit.layout.controls.**UIContent**(*get_line=None*, *line_count=0*, *cursor_position=None*, *menu_position=None*, *show_cursor=True*, *default_char=None*)
Content generated by a user control. This content consists of a list of lines.

Parameters

- **get_line** – Callable that returns the current line. This is a list of (Token, text) tuples.
- **line_count** – The number of lines.
- **cursor_position** – a *Point* for the cursor position.
- **menu_position** – a *Point* for the menu position.
- **show_cursor** – Make the cursor visible.
- **default_char** – The default *Char* for filling the background.

get_height_for_line(*lineno*, *width*)
Return the height that a given line would need if it is rendered in a space with the given width.

Layout dimensions are used to give the minimum, maximum and preferred dimensions for containers and controls.

class prompt_toolkit.layout.dimension.**LayoutDimension**(*min=None*, *max=None*, *weight=1*, *preferred=None*)
Specified dimension (width/height) of a user control or window.

The layout engine tries to honor the preferred size. If that is not possible, because the terminal is larger or smaller, it tries to keep in between min and max.

Parameters

- **min** – Minimum size.
- **max** – Maximum size.
- **weight** – For a VSplit/HSplit, the actual size will be determined by taking the proportion of weights from all the children. E.g. When there are two children, one width a weight of 1, and the other with a weight of 2. The second will always be twice as big as the first, if the min/max values allow it.

- **preferred** – Preferred size.

classmethod exact(*amount*)
Return a *LayoutDimension* with an exact size. (min, max and preferred set to `amount`).

prompt_toolkit.layout.dimension.**sum_layout_dimensions**(*dimensions*)
Sum a list of *LayoutDimension* instances.

prompt_toolkit.layout.dimension.**max_layout_dimensions**(*dimensions*)
Take the maximum of a list of *LayoutDimension* instances.

Lexer interface and implementation. Used for syntax highlighting.

class prompt_toolkit.layout.lexers.**Lexer**
Base class for all lexers.

lex_document(*cli*, *document*)
Takes a *Document* and returns a callable that takes a line number and returns the tokens for that line.

class prompt_toolkit.layout.lexers.**SimpleLexer**(*token=Token*, *default_token=None*)
Lexer that doesn't do any tokenizing and returns the whole input as one token.

Parameters **token** – The *Token* for this lexer.

class prompt_toolkit.layout.lexers.**PygmentsLexer**(*pygments_lexer_cls*, *sync_from_start=True*, *syntax_sync=None*)
Lexer that calls a pygments lexer.

Example:

```
from pygments.lexers import HtmlLexer
lexer = PygmentsLexer(HtmlLexer)
```

Note: Don't forget to also load a Pygments compatible style. E.g.:

```
from prompt_toolkit.styles.from_pygments import style_from_pygments
from pygments.styles import get_style_by_name
style = style_from_pygments(get_style_by_name('monokai'))
```

Parameters

- **pygments_lexer_cls** – A *Lexer* from Pygments.
- **sync_from_start** – Start lexing at the start of the document. This will always give the best results, but it will be slow for bigger documents. (When the last part of the document is display, then the whole document will be lexed by Pygments on every key stroke.) It is recommended to disable this for inputs that are expected to be more than 1,000 lines.
- **syntax_sync** – *SyntaxSync* object.

classmethod from_filename(*filename*, *sync_from_start=True*)
Create a *Lexer* from a filename.

lex_document(*cli*, *document*)
Create a lexer function that takes a line number and returns the list of (Token, text) tuples as the Pygments lexer returns for that line.

class prompt_toolkit.layout.lexers.**SyntaxSync**
Syntax synchroniser. This is a tool that finds a start position for the lexer. This is especially important when editing big documents; we don't want to start the highlighting by running the lexer from the beginning of the file. That is very slow when editing.

get_sync_start_position(*document*, *lineno*)
Return the position from where we can start lexing as a (row, column) tuple.

Parameters

- **document** – *Document* instance that contains all the lines.
- **lineno** – The line that we want to highlight. (We need to return this line, or an earlier position.)

class prompt_toolkit.layout.lexers.**SyncFromStart**
Always start the syntax highlighting from the beginning.

class prompt_toolkit.layout.lexers.**RegexSync**(*pattern*)
Synchronize by starting at a line that matches the given regex pattern.

classmethod from_pygments_lexer_cls(*lexer_cls*)
Create a *RegexSync* instance for this Pygments lexer class.

get_sync_start_position(*document*, *lineno*)
Scan backwards, and find a possible position to start.

Margin implementations for a *Window*.

class prompt_toolkit.layout.margins.**Margin**
Base interface for a margin.

create_margin(*cli*, *window_render_info*, *width*, *height*)
Creates a margin. This should return a list of (Token, text) tuples.

Parameters

- **cli** – *CommandLineInterface* instance.
- **window_render_info** – *WindowRenderInfo* instance, generated after rendering and copying the visible part of the *UIControl* into the *Window*.
- **width** – The width that's available for this margin. (As reported by *get_width()*.)
- **height** – The height that's available for this margin. (The height of the *Window*.)

get_width(*cli*, *get_ui_content*)
Return the width that this margin is going to consume.

Parameters

- **cli** – *CommandLineInterface* instance.
- **get_ui_content** – Callable that asks the user control to create a *UIContent* instance. This can be used for instance to obtain the number of lines.

class prompt_toolkit.layout.margins.**NumberedMargin**(*relative=False*, *display_tildes=False*)
Margin that displays the line numbers.

Parameters

- **relative** – Number relative to the cursor position. Similar to the Vi 'relativenumber' option.
- **display_tildes** – Display tildes after the end of the document, just like Vi does.

class prompt_toolkit.layout.margins.**ScrollbarMargin**(*display_arrows=False*)
Margin displaying a scrollbar.

Parameters **display_arrows** – Display scroll up/down arrows.

class prompt_toolkit.layout.margins.**ConditionalMargin**(*margin*, *filter*)
Wrapper around other *Margin* classes to show/hide them.

class prompt_toolkit.layout.margins.**PromptMargin**(*get_prompt_tokens*, *get_continuation_tokens=None*, *show_numbers=False*)
Create margin that displays a prompt. This can display one prompt at the first line, and a continuation prompt (e.g, just dots) on all the following lines.

Parameters

- **get_prompt_tokens** – Callable that takes a CommandLineInterface as input and returns a list of (Token, type) tuples to be shown as the prompt at the first line.
- **get_continuation_tokens** – Callable that takes a CommandLineInterface and a width as input and returns a list of (Token, type) tuples for the next lines of the input.
- **show_numbers** – (bool or *CLIFilter*) Display line numbers instead of the continuation prompt.

get_width(*cli*, *ui_content*)
Width to report to the *Window*.

class prompt_toolkit.layout.menus.**MultiColumnCompletionsMenu**(*min_rows=3*, *suggested_max_column_width=30*, *show_meta=True*, *extra_filter=True*)
Container that displays the completions in several columns. When *show_meta* (a *CLIFilter*) evaluates to True, it shows the meta information at the bottom.

Processors are little transformation blocks that transform the token list from a buffer before the BufferControl will render it to the screen.

They can insert tokens before or after, or highlight fragments by replacing the token types.

class prompt_toolkit.layout.processors.**Processor**
Manipulate the tokens for a given line in a *BufferControl*.

apply_transformation(*cli*, *document*, *lineno*, *source_to_display*, *tokens*)
Apply transformation. Returns a *Transformation* instance.

Parameters

- **cli** – *CommandLineInterface* instance.
- **lineno** – The number of the line to which we apply the processor.
- **source_to_display** – A function that returns the position in the *tokens* for any position in the source string. (This takes previous processors into account.)
- **tokens** – List of tokens that we can transform. (Received from the previous processor.)

has_focus(*cli*)
Processors can override the focus. (Used for the reverse-i-search prefix in DefaultPrompt.)

class prompt_toolkit.layout.processors.**Transformation**(*tokens*, *source_to_display=None*, *display_to_source=None*)
Transformation result, as returned by *Processor.apply_transformation()*.

Important: Always make sure that the length of *document.text* is equal to the length of all the text in *tokens*!

Parameters

- **tokens** – The transformed tokens. To be displayed, or to pass to the next processor.
- **source_to_display** – Cursor position transformation from original string to transformed string.
- **display_to_source** – Cursor position transformed from source string to original string.

class prompt_toolkit.layout.processors.**HighlightSearchProcessor**(*preview_search=False*, *search_buffer_name=u'SEARCH_BUF*, *get_search_state=None*)

Processor that highlights search matches in the document. Note that this doesn't support multiline search matches yet.

Parameters **preview_search** – A Filter; when active it indicates that we take the search text in real time while the user is typing, instead of the last active search state.

class prompt_toolkit.layout.processors.**HighlightSelectionProcessor**

Processor that highlights the selection in the document.

class prompt_toolkit.layout.processors.**PasswordProcessor**(*char=u'*'*)

Processor that turns masks the input. (For passwords.)

Parameters **char** – (string) Character to be used. "*" by default.

class prompt_toolkit.layout.processors.**HighlightMatchingBracketProcessor**(*chars=u'[](){}<>'*, *max_cursor_distance=10*

When the cursor is on or right after a bracket, it highlights the matching bracket.

Parameters **max_cursor_distance** – Only highlight matching brackets when the cursor is within this distance. (From inside a *Processor*, we can't know which lines will be visible on the screen. But we also don't want to scan the whole document for matching brackets on each key press, so we limit to this value.)

class prompt_toolkit.layout.processors.**DisplayMultipleCursors**(*buffer_name*)

When we're in Vi block insert mode, display all the cursors.

class prompt_toolkit.layout.processors.**BeforeInput**(*get_tokens*)

Insert tokens before the input.

Parameters **get_tokens** – Callable that takes a *CommandLineInterface* and returns the list of tokens to be inserted.

classmethod static(*text*, *token=Token*)

Create a *BeforeInput* instance that always inserts the same text.

class prompt_toolkit.layout.processors.**AfterInput**(*get_tokens*)

Insert tokens after the input.

Parameters **get_tokens** – Callable that takes a *CommandLineInterface* and returns the list of tokens to be appended.

classmethod static(*text*, *token=Token*)

Create a *AfterInput* instance that always inserts the same text.

class prompt_toolkit.layout.processors.**AppendAutoSuggestion**(*buffer_name=None*, *token=Token.AutoSuggestion*)

Append the auto suggestion to the input. (The user can then press the right arrow the insert the suggestion.)

Parameters **buffer_name** – The name of the buffer from where we should take the auto suggestion. If not given, we take the current buffer.

class prompt_toolkit.layout.processors.**ConditionalProcessor**(*processor*, *filter*)
Processor that applies another processor, according to a certain condition. Example:

```
# Create a function that returns whether or not the processor should
# currently be applied.
def highlight_enabled(cli):
    return true_or_false

# Wrapt it in a `ConditionalProcessor` for usage in a `BufferControl`.
BufferControl(input_processors=[
    ConditionalProcessor(HighlightSearchProcessor(),
                         Condition(highlight_enabled))])
```

Parameters

- **processor** – *Processor* instance.
- **filter** – *CLIFilter* instance.

class prompt_toolkit.layout.processors.**ShowLeadingWhiteSpaceProcessor**(*get_char=None*, *token=Token.LeadingWhiteSpac*
Make leading whitespace visible.

Parameters

- **get_char** – Callable that takes a CommandLineInterface instance and returns one character.
- **token** – Token to be used.

class prompt_toolkit.layout.processors.**ShowTrailingWhiteSpaceProcessor**(*get_char=None*, *token=Token.TrailingWhiteSpa*
Make trailing whitespace visible.

Parameters

- **get_char** – Callable that takes a CommandLineInterface instance and returns one character.
- **token** – Token to be used.

class prompt_toolkit.layout.processors.**TabsProcessor**(*tabstop=4*, *get_char1=None*, *get_char2=None*, *token=Token.Tab*)
Render tabs as spaces (instead of ^I) or make them visible (for instance, by replacing them with dots.)

Parameters

- **tabstop** – (Integer) Horizontal space taken by a tab.
- **get_char1** – Callable that takes a *CommandLineInterface* and return a character (text of length one). This one is used for the first space taken by the tab.
- **get_char2** – Like *get_char1*, but for the rest of the space.

prompt_toolkit.layout.utils.**token_list_len**(*tokenlist*)
Return the amount of characters in this token list.

Parameters **tokenlist** – List of (token, text) or (token, text, mouse_handler) tuples.

prompt_toolkit.layout.utils.**token_list_width**(*tokenlist*)
Return the character width of this token list. (Take double width characters into account.)

Parameters **tokenlist** – List of (token, text) or (token, text, mouse_handler) tuples.

prompt_toolkit.layout.utils.**token_list_to_text**(*tokenlist*)
Concatenate all the text parts again.

prompt_toolkit.layout.utils.**explode_tokens**(*tokenlist*)
Turn a list of (token, text) tuples into another list where each string is exactly one character.

It should be fine to call this function several times. Calling this on a list that is already exploded, is a null operation.

Parameters **tokenlist** – List of (token, text) tuples.

prompt_toolkit.layout.utils.**split_lines**(*tokenlist*)
Take a single list of (Token, text) tuples and yield one such list for each line. Just like str.split, this will yield at least one item.

Parameters **tokenlist** – List of (token, text) or (token, text, mouse_handler) tuples.

prompt_toolkit.layout.utils.**find_window_for_buffer_name**(*cli*, *buffer_name*)
Look for a *Window* in the Layout that contains the *BufferControl* for the given buffer and return it. If no such Window is found, return None.

class prompt_toolkit.layout.screen.**Point**(*y*, *x*)

x
Alias for field number 1

y
Alias for field number 0

class prompt_toolkit.layout.screen.**Size**(*rows*, *columns*)

columns
Alias for field number 1

rows
Alias for field number 0

class prompt_toolkit.layout.screen.**Screen**(*default_char=None*, *initial_width=0*, *initial_height=0*)
Two dimentional buffer of *Char* instances.

replace_all_tokens(*token*)
For all the characters in the screen. Set the token to the given *token*.

class prompt_toolkit.layout.screen.**Char**(*char=u' '*, *token=Token*)
Represent a single character in a *Screen*.

This should be considered immutable.

5.5.18 Token

The Token class, interchangeable with `pygments.token`.

A *Token* has some semantics for a piece of text that is given a style through a `Style` class. A pygments lexer for instance, returns a list of (Token, text) tuples. Each fragment of text has a token assigned, which when combined with a style sheet, will determine the fine style.

5.5.19 Filters

Filters decide whether something is active or not (they decide about a boolean state). This is used to enable/disable features, like key bindings, parts of the layout and other stuff. For instance, we could have a *HasSearch* filter attached to some part of the layout, in order to show that part of the user interface only while the user is searching.

Filters are made to avoid having to attach callbacks to all event in order to propagate state. However, they are lazy, they don't automatically propagate the state of what they are observing. Only when a filter is called (it's actually a callable), it will calculate its value. So, its not really reactive programming, but it's made to fit for this framework.

One class of filters observe a *CommandLineInterface* instance. However, they are not attached to such an instance. (We have to pass this instance to the filter when calling it.) The reason for this is to allow declarative programming: for key bindings, we can attach a filter to a key binding without knowing yet which *CommandLineInterface* instance it will observe in the end. Examples are *HasSearch* or *IsExiting*.

Another class of filters doesn't take anything as input. And a third class of filters are universal, for instance *Always* and *Never*. It is impossible to mix the first and the second class, because that would mean mixing filters with a different signature.

Filters can be chained using `&` and `|` operations, and inverted using the `~` operator, for instance:

```
filter = HasFocus('default') & ~ HasSelection()
```

class prompt_toolkit.filters.**Filter**
Filter to activate/deactivate a feature, depending on a condition. The return value of `__call__` will tell if the feature should be active.

test_args(**args*)
Test whether this filter can be called with the following argument list.

class prompt_toolkit.filters.**CLIFilter**
Abstract base class for filters that accept a *CommandLineInterface* argument. It cannot be instantiated, it's only to be used for instance assertions, e.g.:

```
isinstance(my_filter, CliFilter)
```

class prompt_toolkit.filters.**SimpleFilter**
Abstract base class for filters that don't accept any arguments.

class prompt_toolkit.filters.**Condition**(*func*)
Turn any callable (which takes a cli and returns a boolean) into a Filter.

This can be used as a decorator:

```
@Condition
def feature_is_active(cli):  # `feature_is_active` becomes a Filter.
    return True
```

Parameters **func** – Callable which takes either a *CommandLineInterface* or nothing and returns a boolean. (Depending on what it takes, this will become a *Filter* or *CLIFilter*.)

5.5.20 Key binding

Key bindings registry.

A *Registry* object is a container that holds a list of key bindings. It has a very efficient internal data structure for checking which key bindings apply for a pressed key.

Typical usage:

```
r = Registry()

@r.add_binding(Keys.ControlX, Keys.ControlC, filter=INSERT)
def handler(event):
    # Handle ControlX-ControlC key sequence.
    pass
```

It is also possible to combine multiple registries. We do this in the default key bindings. There are some registries that contain Emacs bindings, while others contain the Vi bindings. They are merged together using a *MergedRegistry*.

We also have a *ConditionalRegistry* object that can enable/disable a group of key bindings at once.

class prompt_toolkit.key_binding.registry.**BaseRegistry**
Interface for a Registry.

class prompt_toolkit.key_binding.registry.**Registry**
Key binding registry.

add_binding(**keys*, ***kwargs*)
Decorator for annotating key bindings.

Parameters

- **filter** – *CLIFilter* to determine when this key binding is active.
- **eager** – *CLIFilter* or *bool*. When True, ignore potential longer matches when this key binding is hit. E.g. when there is an active eager key binding for Ctrl-X, execute the handler immediately and ignore the key binding for Ctrl-X Ctrl-E of which it is a prefix.
- **save_before** – Callable that takes an *Event* and returns True if we should save the current buffer, before handling the event. (That's the default.)

get_bindings_for_keys(*keys*)
Return a list of key bindings that can handle this key. (This return also inactive bindings, so the *filter* still has to be called, for checking it.)

Parameters **keys** – tuple of keys.

get_bindings_starting_with_keys(*keys*)
Return a list of key bindings that handle a key sequence starting with *keys*. (It does only return bindings for which the sequences are longer than *keys*. And like *get_bindings_for_keys*, it also includes inactive bindings.)

Parameters **keys** – tuple of keys.

remove_binding(*function*)
Remove a key binding.

This expects a function that was given to *add_binding* method as parameter. Raises *ValueError* when the given function was not registered before.

class prompt_toolkit.key_binding.registry.**ConditionalRegistry**(*registry=None*, *filter=True*)
Wraps around a *Registry*. Disable/enable all the key bindings according to the given (additional) filter.:

```
@Condition
def setting_is_true(cli):
    return True  # or False

registy = ConditionalRegistry(registry, setting_is_true)
```

When new key bindings are added to this object. They are also enable/disabled according to the given *filter*.

Parameters

- **registries** – List of *Registry* objects.
- **filter** – *CLIFilter* object.

class prompt_toolkit.key_binding.registry.**MergedRegistry**(*registries*)

Merge multiple registries of key bindings into one.

This class acts as a proxy to multiple *Registry* objects, but behaves as if this is just one bigger *Registry*.

Parameters **registries** – List of *Registry* objects.

DEPRECATED: Use *prompt_toolkit.key_binding.defaults.load_key_bindings* instead.

KeyBindingManager is a utility (or shortcut) for loading all the key bindings in a key binding registry, with a logic set of filters to quickly to quickly change from Vi to Emacs key bindings at runtime.

You don't have to use this, but it's practical.

Usage:

```
manager = KeyBindingManager()
app = Application(key_bindings_registry=manager.registry)
```

class prompt_toolkit.key_binding.manager.**KeyBindingManager**(*registry=None*, *enable_vi_mode=None*, *enable_all=True*, *get_search_state=None*, *enable_abort_and_exit_bindings=False*, *enable_system_bindings=False*, *enable_search=False*, *enable_open_in_editor=False*, *enable_extra_page_navigation=False*, *enable_auto_suggest_bindings=False*)

Utility for loading all key bindings into memory.

Parameters

- **registry** – Optional *Registry* instance.
- **enable_abort_and_exit_bindings** – Filter to enable Ctrl-C and Ctrl-D.
- **enable_system_bindings** – Filter to enable the system bindings (meta-! prompt and Control-Z suspension.)
- **enable_search** – Filter to enable the search bindings.
- **enable_open_in_editor** – Filter to enable open-in-editor.
- **enable_open_in_editor** – Filter to enable open-in-editor.
- **enable_extra_page_navigation** – Filter for enabling extra page navigation. (Bindings for up/down scrolling through long pages, like in Emacs or Vi.)
- **enable_auto_suggest_bindings** – Filter to enable fish-style suggestions.
- **enable_vi_mode** – Deprecated!

classmethod for_prompt(***kw*)
Create a `KeyBindingManager` with the defaults for an input prompt. This activates the key bindings for abort/exit (Ctrl-C/Ctrl-D), incremental search and auto suggestions.

(Not for full screen applications.)

class prompt_toolkit.key_binding.vi_state.**ViState**
Mutable class to hold the state of the Vi navigation.

reset(*mode=u'vi-insert'*)
Reset state, go back to the given mode. INSERT by default.

5.5.21 Eventloop

class prompt_toolkit.eventloop.base.**EventLoop**
Eventloop interface.

add_reader(*fd*, *callback*)
Start watching the file descriptor for read availability and then call the callback.

call_from_executor(*callback*, *_max_postpone_until=None*)
Call this function in the main event loop. Similar to Twisted's `callFromThread`.

Parameters **_max_postpone_until** – *None* or *time.time* value. For interal use. If the eventloop is saturated, consider this task to be low priority and postpone maximum until this timestamp. (For instance, repaint is done using low priority.)

Note: In the past, this used to be a datetime.datetime instance, but apparently, executing *time.time* is more efficient: it does fewer system calls. (It doesn't read /etc/localtime.)

close()
Clean up of resources. Eventloop cannot be reused a second time after this call.

remove_reader(*fd*)
Stop watching the file descriptor for read availability.

run(*stdin*, *callbacks*)
Run the eventloop until stop() is called. Report all input/timeout/terminal-resize events to the callbacks.

Parameters

- **stdin** – *Input* instance.
- **callbacks** – *EventLoopCallbacks* instance.

run_as_coroutine(*stdin*, *callbacks*)
Similar to *run*, but this is a coroutine. (For asyncio integration.)

run_in_executor(*callback*)
Run a long running function in a background thread. (This is recommended for code that could block the event loop.) Similar to Twisted's `deferToThread`.

stop()
Stop the *run* call. (Normally called by *CommandLineInterface*, when a result is available, or Abort/Quit has been called.)

class prompt_toolkit.eventloop.posix.**PosixEventLoop**(*inputhook=None*, *selector=<class 'prompt_toolkit.eventloop.select.AutoSelector'>*)
Event loop for posix systems (Linux, Mac os X).

add_reader(*fd*, *callback*)
Add read file descriptor to the event loop.

call_from_executor(*callback*, *_max_postpone_until=None*)
Call this function in the main event loop. Similar to Twisted's `callFromThread`.

Parameters **_max_postpone_until** – *None* or *time.time* value. For interal use. If the eventloop is saturated, consider this task to be low priority and postpone maximum until this timestamp. (For instance, repaint is done using low priority.)

received_winch()
Notify the event loop that SIGWINCH has been received

remove_reader(*fd*)
Remove read file descriptor from the event loop.

run(*stdin*, *callbacks*)
The input 'event loop'.

run_in_executor(*callback*)
Run a long running function in a background thread. (This is recommended for code that could block the event loop.) Similar to Twisted's `deferToThread`.

stop()
Stop the event loop.

Eventloop for integration with Python3 asyncio.

Note that we can't use "yield from", because the package should be installable under Python 2.6 as well, and it should contain syntactically valid Python 2.6 code.

class prompt_toolkit.eventloop.asyncio_base.**AsyncioTimeout**(*timeout*, *callback*, *loop*)
Call the *timeout* function when the timeout expires. Every call of the *reset* method, resets the timeout and starts a new timer.

reset()
Reset the timeout. Starts a new timer.

stop()
Ignore timeout. Don't call the callback anymore.

class prompt_toolkit.eventloop.callbacks.**EventLoopCallbacks**
This is the glue between the *EventLoop* and *CommandLineInterface*.

run() takes an *EventLoopCallbacks* instance and operates on that one, driving the interface.

5.5.22 Input and output

Abstraction of CLI Input.

class prompt_toolkit.input.**Input**
Abstraction for any input.

An instance of this class can be given to the constructor of a *CommandLineInterface* and will also be passed to the *EventLoop*.

cooked_mode()
Context manager that turns the input into cooked mode.

fileno()
Fileno for putting this in an event loop.

raw_mode()
Context manager that turns the input into raw mode.

read()
Return text from the input.

class prompt_toolkit.input.**StdinInput**(*stdin=None*)
Simple wrapper around stdin.

class prompt_toolkit.input.**PipeInput**
Input that is send through a pipe. This is useful if we want to send the input programatically into the interface, but still use the eventloop.

Usage:

```
input = PipeInput()
input.send('inputdata')
```

close()
Close pipe fds.

send(*data*)
Send text to the input.

send_text(*data*)
Send text to the input.

Interface for an output.

class prompt_toolkit.output.**Output**
Base class defining the output interface for a *Renderer*.

Actual implementations are Vt100_Output and Win32Output.

ask_for_cpr()
Asks for a cursor position report (CPR). (VT100 only.)

bell()
Sound bell.

clear_title()
Clear title again. (or restore previous title.)

cursor_backward(*amount*)
Move cursor *amount* place backward.

cursor_down(*amount*)
Move cursor *amount* place down.

cursor_forward(*amount*)
Move cursor *amount* place forward.

cursor_goto(*row=0*, *column=0*)
Move cursor position.

cursor_up(*amount*)
Move cursor *amount* place up.

disable_autowrap()
Disable auto line wrapping.

disable_bracketed_paste()
For vt100 only.

disable_mouse_support()
Disable mouse.

enable_autowrap()
Enable auto line wrapping.

enable_bracketed_paste()
For vt100 only.

enable_mouse_support()
Enable mouse.

encoding()
Return the encoding for this output, e.g. 'utf-8'. (This is used mainly to know which characters are supported by the output the data, so that the UI can provide alternatives, when required.)

enter_alternate_screen()
Go to the alternate screen buffer. (For full screen applications).

erase_down()
Erases the screen from the current line down to the bottom of the screen.

erase_end_of_line()
Erases from the current cursor position to the end of the current line.

erase_screen()
Erases the screen with the background colour and moves the cursor to home.

fileno()
Return the file descriptor to which we can write for the output.

flush()
Write to output stream and flush.

hide_cursor()
Hide cursor.

quit_alternate_screen()
Leave the alternate screen buffer.

reset_attributes()
Reset color and styling attributes.

set_attributes(*attrs*)
Set new color and styling attributes.

set_title(*title*)
Set terminal title.

show_cursor()
Show cursor.

write(*data*)
Write text (Terminal escape sequences will be removed/escaped.)

write_raw(*data*)
Write text.

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Prompt_toolkit was created by Jonathan Slenders.

Python Module Index

p

Index

D

E

F

G

H

I

Q

R

S

T

U

V

W

X

Y