prompt_toolkit Documentation

Release 2.0.4

Jonathan Slenders

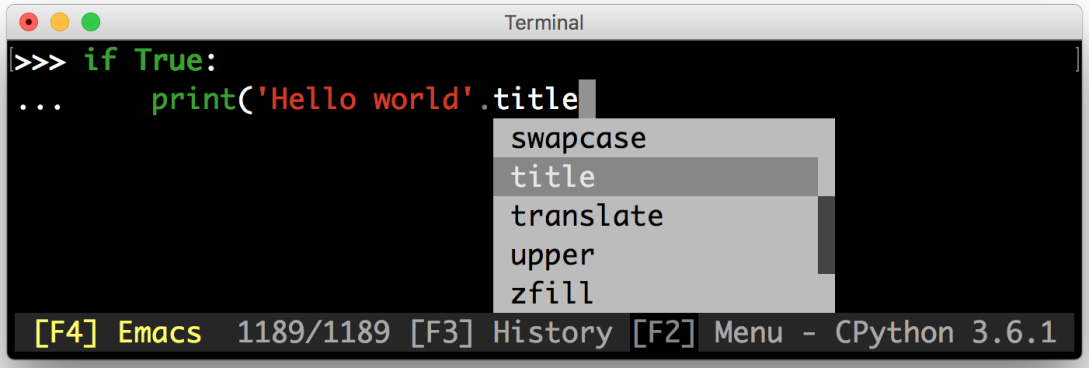
Jul 22, 2018

Contents

Warning: Notice that this is the prompt_toolkit 2.0 documentation. It is incompatible with the 1.0 branch, but much better in many regards. Please read *Upgrading to prompt_toolkit 2.0* for more information.

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

It can be a very advanced pure Python replacement for GNU readline, but it can also be used for building full screen applications.



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.

Have a look at *the gallery* to get an idea of what is possible.

CHAPTER 1

Getting started

Go to *getting started* and build your first prompt.

CHAPTER 2

Thanks to:

A special thanks to all the contributors for making prompt_toolkit possible.

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

CHAPTER 3

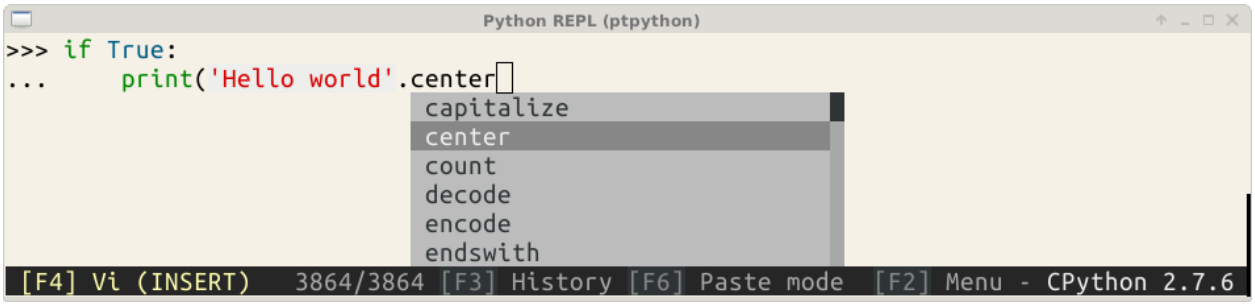
Table of contents

3.1 Gallery

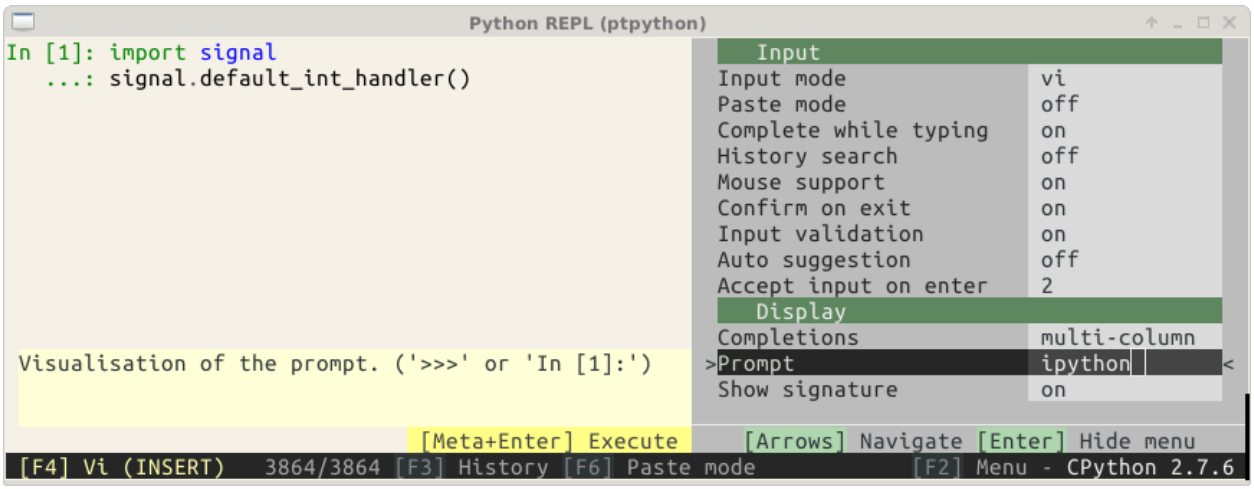
Showcase, demonstrating the possibilities of prompt_toolkit.

3.1.1 Ptpython, a Python REPL

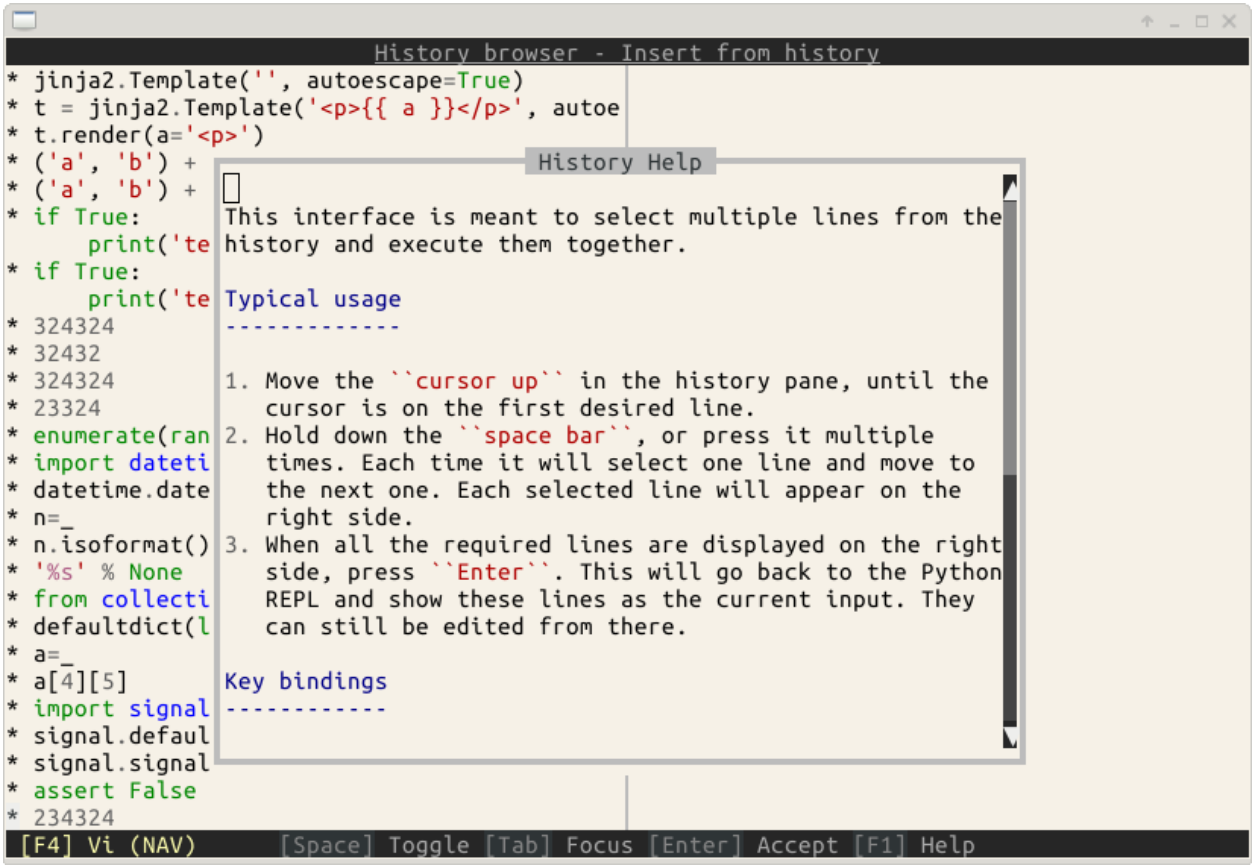
The prompt:



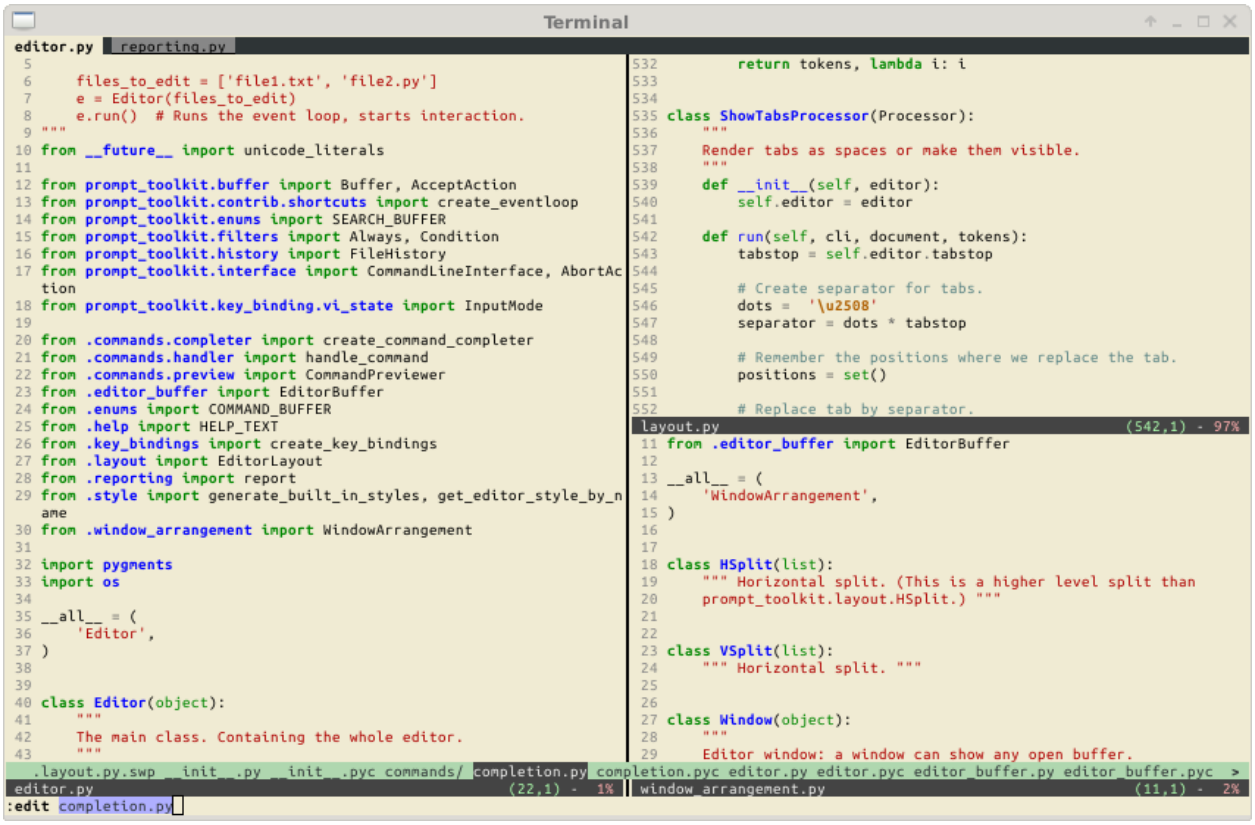
The configuration menu of ptpython.



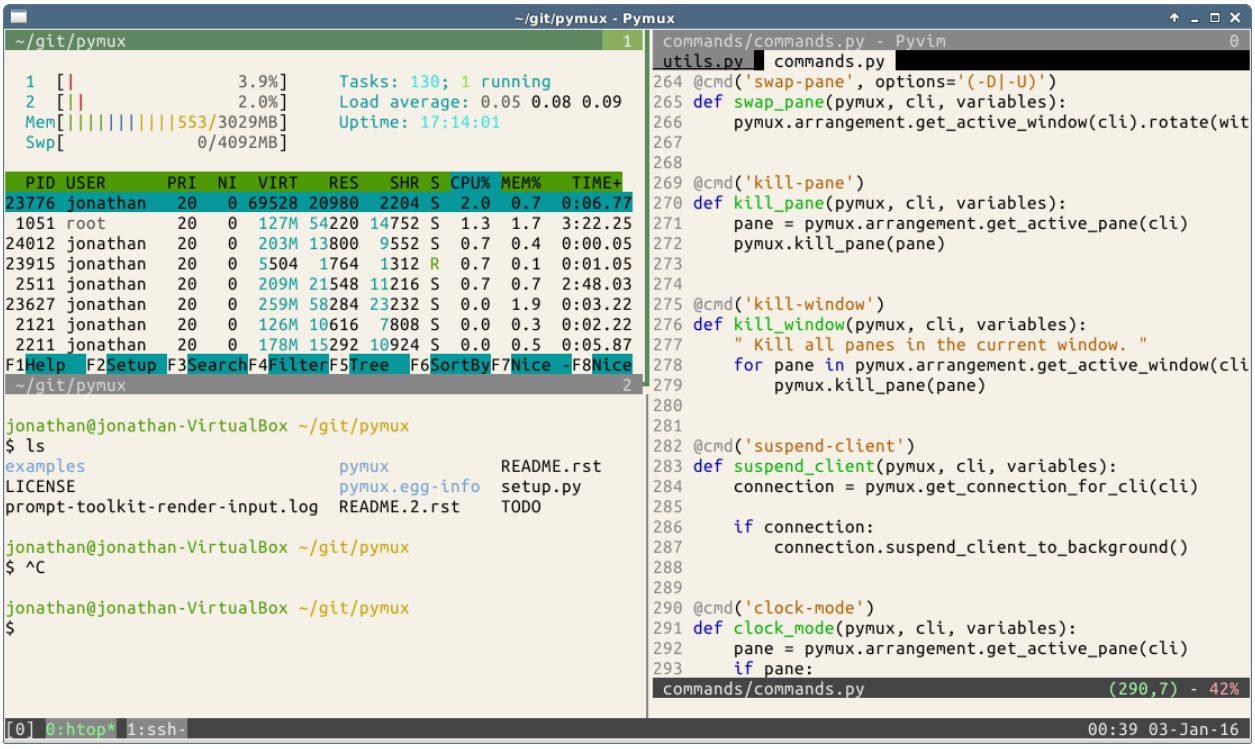
The history page with its help. (This is a full-screen layout.)



3.1.2 Pyvim, a Vim clone



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



3.2 Getting started

3.2.1 Installation

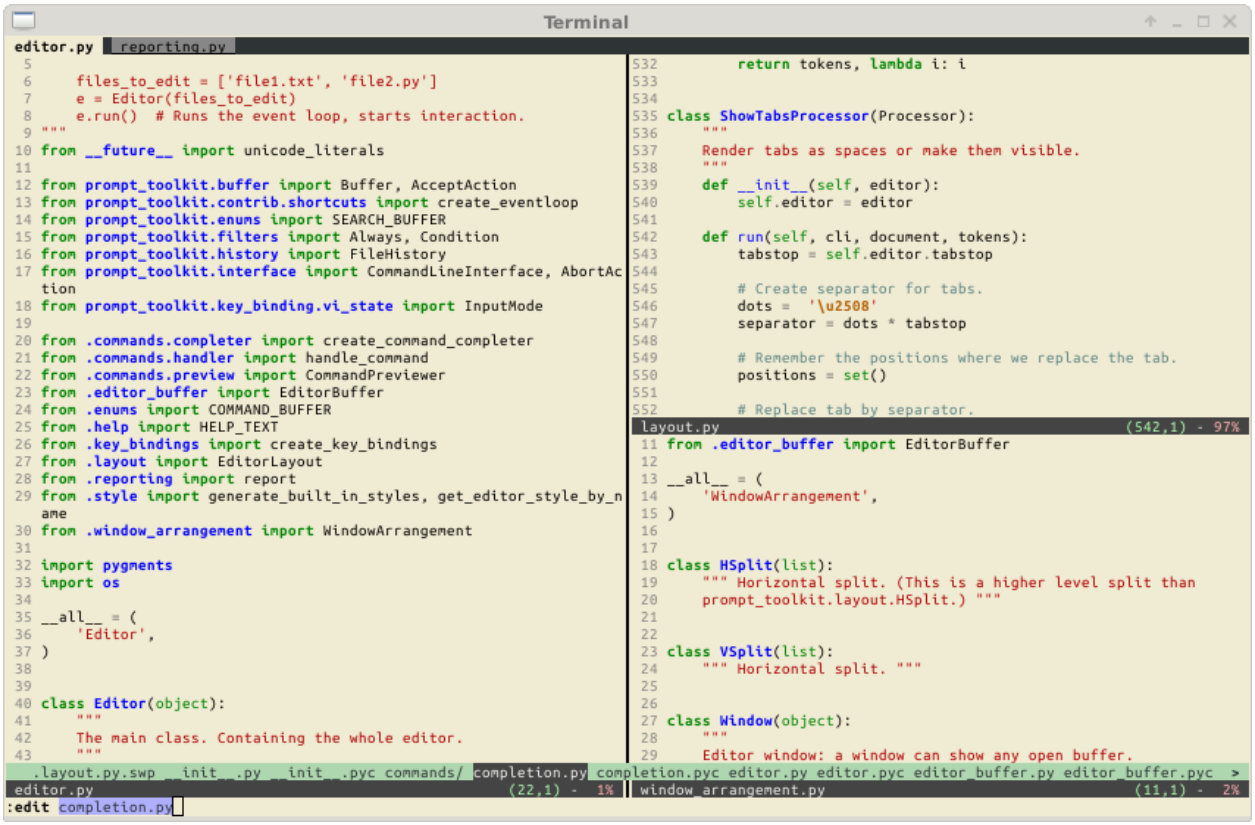
```
pip install prompt_toolkit
```

For Conda, do:

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

3.2.2 Several 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 situations. 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 as a readline replacement, (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 a custom layout ourselves.

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

3.2.3 A simple prompt

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)
```

3.2.4 Learning *prompt_toolkit*

In order to learn and understand *prompt_toolkit*, it is best to go through the all sections in the order below. Also don't forget to have a look at all the examples examples in the repository.

- First, *learn how to print text*. This is important, because it covers how to use "formatted text", which is something you'll use whenever you want to use colors anywhere.
- Secondly, go through the *asking for input* section. This is useful for almost any use case, even for full screen applications. It covers autocompletions, syntax highlighting, key bindings, and so on.
- Then, learn about *Dialogs*, which is easy and fun.
- Finally, learn about *full screen applications* and read through *the advanced topics*.

3.3 Upgrading to prompt_toolkit 2.0

Prompt_toolkit 2.0 is not compatible with 1.0, however you probably want to upgrade your applications. This page explains why we have these differences and how to upgrade.

If you experience some difficulties or you feel that some information is missing from this page, don't hesitate to open a GitHub issue for help.

3.3.1 Why all these breaking changes?

After more and more custom prompt_toolkit applications were developed, it became clear that prompt_toolkit 1.0 was not flexible enough for certain use cases. Mostly, the development of full screen applications was not really natural. All the important components, like the rendering, key bindings, input and output handling were present, but the API was in the first place designed for simple command line prompts. This was mostly notably in the following two places:

- First, there was the focus which was always pointing to a `Buffer` (or text input widget), but in full screen applications there are other widgets, like menus and buttons which can be focused.
- And secondly, it was impossible to make reusable UI components. All the key bindings for the entire applications were stored together in one `KeyBindings` object, and similar, all `Buffer` objects were stored together in one dictionary. This didn't work well. You want reusable components to define their own key bindings and everything. It's the idea of encapsulation.

For simple prompts, the changes wouldn't be that invasive, but given that there would be some, I took the opportunity to fix a couple of other things. For instance:

- In prompt_toolkit 1.0, we translated \r into \n during the input processing. This was not a good idea, because some people wanted to handle these keys individually. This makes sense if you keep in mind that they correspond to *Control-M* and *Control-J*. However, we couldn't fix this without breaking everyone's enter key, which happens to be the most important key in prompts.

Given that we were going to break compatibility anyway, we changed a couple of other important things that both effect both simple prompt applications and full screen applications. These are the most important:

- We no longer depend on Pygments for styling. While we like Pygments, it was not flexible enough to provide all the styling options that we need, and the Pygments tokens were not ideal for styling anything besides tokenized text.

  Instead we created something similar to CSS. All UI components can attach classnames to themselves, as well as define an inline style. The final style is then computed by combining the inline styles, the classnames and the style sheet.

  There are still adaptors available for using Pygments lexers as well as for Pygments styles.
- The way that key bindings were defined was too complex. `KeyBindingsManager` was too complex and no longer exists. Every set of key bindings is now a `KeyBindings` object and multiple of these can be merged together at any time. The runtime performance remains the same, but it's now easier for users.

- The separation between the `CommandLineInterface` and *`Application`* class was confusing and in the end, didn't really had an advantage. These two are now merged together in one *`Application`* class.
- We no longer pass around the active `CommandLineInterface`. This was one of the most annoying things. Key bindings need it in order to change anything and filters need it in order to evaluate their state. It was pretty annoying, especially because there was usually only one application active at a time. So, *`Application`* became a `TaskLocal`. That is like a global variable, but scoped in the current coroutine or context. The way this works is still not 100% correct, but good enough for the projects that need it (like Pymux), and hopefully Python will get support for this in the future thanks to PEP521, PEP550 or PEP555.

All of these changes have been tested for many months, and I can say with confidence that prompt_toolkit 2.0 is a better prompt_toolkit.

3.3.2 Some new features

Apart from the breaking changes above, there are also some exciting new features.

- We now support vt100 escape codes for Windows consoles on Windows 10. This means much faster rendering, and full color support.
- We have a concept of formatted text. This is an object that evaluates to styled text. Every input that expects some text, like the message in a prompt, or the text in a toolbar, can take any kind of formatted text as input. This means you can pass in a plain string, but also a list of *(style, text)* tuples (similar to a Pygments tokenized string), or an *`HTML`* object. This simplifies many APIs.
- New utilities were added. We now have function for printing formatted text and an experimental module for displaying progress bars.
- Autocompletion, input validation, and auto suggestion can now either be asynchronous or synchronous. By default they are synchronous, but by wrapping them in *`ThreadedCompleter`*, *`ThreadedValidator`* or *`ThreadedAutoSuggest`*, they will become asynchronous by running in a background thread.

  Furter, if the autocompletion code runs in a background thread, we will show the completions as soon as they arrive. This means that the autocompletion algorithm could for instance first yield the most trivial completions and then take time to produce the completions that take more time.

3.3.3 Upgrading

More guidelines on how to upgrade will follow.

AbortAction has been removed

Prompt_toolkit 1.0 had an argument `abort_action` for both the `Application` class as well as for the `prompt` function. This has been removed. The recommended way to handle this now is by capturing `KeyboardInterrupt` and `EOFError` manually.

Calling *create_eventloop* usually not required anymore

Prompt_toolkit 2.0 will automatically create the appropriate event loop when it's needed for the first time. There is no need to create one and pass it around. If you want to run an application on top of asyncio (without using an executor), it still needs to be activated by calling *`use_asyncio_event_loop()`* at the beginning.

Pygments styles and tokens

prompt_toolkit 2.0 no longer depends on Pygments, but that definitely doesn't mean that you can't use any Pygments functionality anymore. The only difference is that Pygments stuff needs to be wrapped in an adaptor to make it compatible with the native prompt_toolkit objects.

- For instance, if you have a list of `(pygments.Token, text)` tuples for formatting, then this needs to be wrapped in a *PygmentsTokens* object. This is an adaptor that turns it into prompt_toolkit "formatted text". Feel free to keep using this.
- Pygments lexers need to be wrapped in a *PygmentsLexer*. This will convert the list of Pygments tokens into prompt_toolkit formatted text.
- If you have a Pygments style, then this needs to be converted as well. A Pygments style class can be converted in a prompt_toolkit *Style* with the `style_from_pygments_cls()` function (which used to be called `style_from_pygments`). A Pygments style dictionary can be converted using `style_from_pygments_dict()`.

  Multiple styles can be merged together using *merge_styles()*.

Asynchronous autocompletion

By default, prompt_toolkit 2.0 completion is now synchronous. If you still want asynchronous auto completion (which is often good thing), then you have to wrap the completer in a *ThreadedCompleter*.

Filters

We don't distiguish anymore between *CLIFilter* and *SimpleFilter*, because the application object is no longer passed around. This means that all filters are a *Filter* from now on.

All filters have been turned into functions. For instance, *IsDone* became *is_done* and *HasCompletions* became *has_completions*.

This was done because almost all classes were called without any arguments in the *__init__* causing additional braces everywhere. This means that *HasCompletions()* has to be replaced by *has_completions* (without parenthesis).

The few filters that took arguments as input, became functions, but still have to be called with the given arguments.

For new filters, it is recommended to use the *@Condition* decorator, rather then inheriting from *Filter*. For instance:

```
from prompt_toolkit.filter import Condition

@Condition
def my_filter();
    return True  # Or False
```

3.4 Printing (and using) formatted text

Prompt_toolkit ships with a *print_formatted_text()* function that's meant to be (as much as possible) compatible with the built-in print function, but on top of that, also supports colors and formatting.

On Linux systems, this will output VT100 escape sequences, while on Windows it will use Win32 API calls or VT100 sequences, depending on what is available.

Note: This page is also useful if you'd like to learn how to use formatting in other places, like in a prompt or a toolbar. Just like *print_formatted_text()* takes any kind of "formatted text" as input, prompts and toolbars also accept "formatted text".

3.4.1 Printing plain text

The print function can be imported as follows:

```
from __future__ import unicode_literals
from prompt_toolkit import print_formatted_text

print_formatted_text('Hello world')
```

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 explicitly put a small `'u'` in front of every string.

You can replace the built in `print` function as follows, if you want to.

```
from __future__ import unicode_literals, print_function
from prompt_toolkit import print_formatted_text as print

print('Hello world')
```

Note: If you're using Python 2, make sure to add `from __future__ import print_function`. Otherwise, it will not be possible to import a function named `print`.

3.4.2 Formatted text

There are several ways to display colors:

- By creating an *HTML* object.
- By creating an *ANSI* object that contains ANSI escape sequences.
- By creating a list of `(style, text)` tuples.
- By creating a list of `(pygments.Token, text)` tuples, and wrapping it in *PygmentsTokens*.

An instance of any of these three kinds of objects is called "formatted text". There are various places in prompt toolkit, where we accept not just plain text (as a strings), but also formatted text.

HTML

HTML can be used to indicate that a string contains HTML-like formatting. It recognizes the basic tags for bold, italic and underline: `<b>`, `<i>` and `<u>`.

```
from __future__ import unicode_literals, print_function
from prompt_toolkit import print_formatted_text, HTML

print_formatted_text(HTML('<b>This is bold</b>'))
print_formatted_text(HTML('<i>This is italic</i>'))
print_formatted_text(HTML('<u>This is underlined</u>'))
```

Further, it's possible to use tags for foreground colors:

```
# Colors from the ANSI palette.
print_formatted_text(HTML('<ansired>This is red</ansired>'))
print_formatted_text(HTML('<ansigreen>This is green</ansigreen>'))

# Named colors (256 color palette, or true color, depending on the output).
print_formatted_text(HTML('<skyblue>This is sky blue</skyblue>'))
print_formatted_text(HTML('<seagreen>This is sea green</seagreen>'))
print_formatted_text(HTML('<violet>This is violet</violet>'))
```

Both foreground and background colors can also be specified setting the *fg* and *bg* attributes of any HTML tag:

```
# Colors from the ANSI palette.
print_formatted_text(HTML('<aaa fg="ansiwhite" bg="ansigreen">White on green</aaa>'))
```

Underneath, all HTML tags are mapped to classes from a stylesheet, so you can assign a style for a custom tag.

```
from __future__ import unicode_literals, print_function
from prompt_toolkit import print_formatted_text, HTML
from prompt_toolkit.styles import Style

style = Style.from_dict({
    'aaa': '#ff0066',
    'bbb': '#44ff00 italic',
})

print_formatted_text(HTML('<aaa>Hello</aaa> <bbb>world</bbb>!'), style=style)
```

ANSI

Some people like to use the VT100 ANSI escape sequences to generate output. Natively, this is however only supported on VT100 terminals, but prompt_toolkit can parse these, and map them to formatted text instances. This means that they will work on Windows as well. The *ANSI* class takes care of that.

```
from __future__ import unicode_literals, print_function
from prompt_toolkit import print_formatted_text, ANSI

print_formatted_text(ANSI('\x1b[31mhello \x1b[32mworld'))
```

Keep in mind that even on a Linux VT100 terminal, the final output produced by prompt_toolkit, is not necessarily exactly the same. Depending on the color depth, it is possible that colors are mapped to different colors, and unknown tags will be removed.

(style, text) tuples

Internally, both *HTML* and *ANSI* objects are mapped to a list of `(style, text)` tuples. It is however also possible to create such a list manually with *FormattedText* class. This is a little more verbose, but it's probably the most

powerful way of expressing formatted text.

```
from __future__ import unicode_literals, print_function
from prompt_toolkit import print_formatted_text
from prompt_toolkit.formatted_text import FormattedText

text = FormattedText([
    ('#ff0066', 'Hello'),
    ('', ' '),
    ('#44ff00 italic', 'World'),
])

print_formatted_text(text)
```

Similar to the *HTML* example, it is also possible to use class names, and separate the styling in a style sheet.

```
from __future__ import unicode_literals, print_function
from prompt_toolkit import print_formatted_text
from prompt_toolkit.formatted_text import FormattedText
from prompt_toolkit.styles import Style

# The text.
text = FormattedText([
    ('class:aaa', 'Hello'),
    ('', ' '),
    ('class:bbb', 'World'),
])

# The style sheet.
style = Style.from_dict({
    'aaa': '#ff0066',
    'bbb': '#44ff00 italic',
})

print_formatted_text(text, style=style)
```

Pygments `(Token, text)` tuples

When you have a list of Pygments `(Token, text)` tuples, then these can be printed by wrapping them in a *PygmentsTokens* object.

```
from pygments.token import Token
from prompt_toolkit import print_formatted_text
from prompt_toolkit.formatted_text import PygmentsTokens

text = [
    (Token.Keyword, 'print'),
    (Token.Punctuation, '('),
    (Token.Literal.String.Double, '"'),
    (Token.Literal.String.Double, 'hello'),
    (Token.Literal.String.Double, '"'),
    (Token.Punctuation, ')'),
    (Token.Text, '\n'),
]

print_formatted_text(PygmentsTokens(text))
```

Similarly, it is also possible to print the output of a Pygments lexer:

```
import pygments
from pygments.token import Token
from pygments.lexers.python import PythonLexer

from prompt_toolkit.formatted_text import PygmentsTokens
from prompt_toolkit import print_formatted_text

# Printing the output of a pygments lexer.
tokens = list(pygments.lex('print("Hello")', lexer=PythonLexer()))
print_formatted_text(PygmentsTokens(tokens))
```

Prompt_toolkit ships with a default colorscheme which styles it just like Pygments would do, but if you'd like to change the colors, keep in mind that Pygments tokens map to classnames like this:

pygments.Token	prompt_toolkit classname
• Token.Keyword • Token.Punctuation • Token.Literal.String.Double • Token.Text • Token	• "class:pygments.keyword" • "class:pygments.punctuation" • "class:pygments.literal.string.double" • "class:pygments.text" • "class:pygments"

A classname like `pygments.literal.string.double` is actually decomposed in the following four classnames: `pygments`, `pygments.literal`, `pygments.literal.string` and `pygments.literal.string.double`. The final style is computed by combining the style for these four classnames. So, changing the style from these Pygments tokens can be done as follows:

```
from prompt_toolkit.styles import Style

style = Style.from_dict({
    'pygments.keyword': 'underline',
    'pygments.literal.string': 'bg:#00ff00 #ffffff',
})
print_formatted_text(PygmentsTokens(tokens), style=style)
```

to_formatted_text

A useful function to know about is *to_formatted_text()*. This ensures that the given input is valid formatted text. While doing so, an additional style can be applied as well.

```
from prompt_toolkit.formatted_text import to_formatted_text, HTML
from prompt_toolkit import print_formatted_text

html = HTML('<aaa>Hello</aaa> <bbb>world</bbb>!')
text = to_formatted_text(html, style='class:my_html bg:#00ff00 italic')

print_formatted_text(text)
```

3.5 Asking for input (prompts)

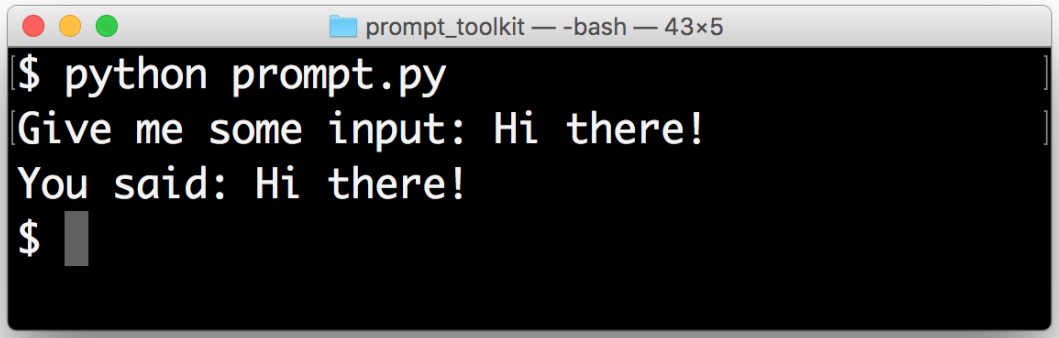
This page is about building prompts. Pieces of code that we can embed in a program for asking the user for input. Even if you want to use *prompt_toolkit* for building full screen terminal applications, it is probably still a good idea to read this first, before heading to the *building full screen applications*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

In this page, we will cover autocompletion, syntax highlighting, key bindings, and so on.

3.5.1 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 explicitly put a small `'u'` in front of every string.

3.5.2 The *PromptSession* object

Instead of calling the `prompt()` function, it's also possible to create a `PromptSession` instance followed by calling its `prompt()` method for every input call. This creates a kind of an input session.

```
from prompt_toolkit import PromptSession

# Create prompt object.
session = PromptSession()

# Do multiple input calls.
text1 = session.prompt()
text2 = session.prompt()
```

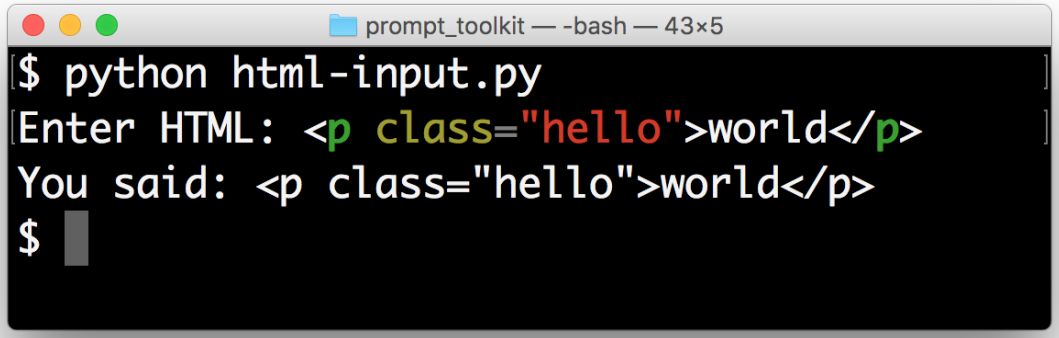
This has mainly two advantages:

- The input history will be kept between consecutive `prompt()` calls.
- The `PromptSession()` instance and its `prompt()` method take about the same arguments, like all the options described below (highlighting, completion, etc...). So if you want to ask for multiple inputs, but each input call needs about the same arguments, they can be passed to the `PromptSession()` instance as well, and they can be overridden by passing values to the `prompt()` method.

3.5.3 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.html import HtmlLexer
from prompt_toolkit.shortcuts import prompt
from prompt_toolkit.lexers import PygmentsLexer

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



The default Pygments colorscheme is included as part of the default style in prompt_toolkit. If you want to use another Pygments style along with the lexer, you can do the following:

```
from pygments.lexers.html import HtmlLexer
from pygments.styles import get_style_by_name
from prompt_toolkit.shortcuts import prompt
from prompt_toolkit.lexers import PygmentsLexer
```

(continues on next page)

(continued from previous page)

```
from prompt_toolkit.styles.pygments import style_from_pygments_cls

style = style_from_pygments_cls(get_style_by_name('monokai'))
text = prompt('Enter HTML: ', lexer=PygmentsLexer(HtmlLexer), style=style,
              include_default_pygments_style=False)
print('You said: %s' % text)
```

We pass `include_default_pygments_style=False`, because otherwise, both styles will be merged, possibly giving slightly different colors in the outcome for cases where where our custom Pygments style doesn't specify a color.

3.5.4 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 *`from_dict()`* function:

```
from pygments.lexers.html import HtmlLexer
from prompt_toolkit.shortcuts import prompt
from prompt_toolkit.styles import Style
from prompt_toolkit.lexers import PygmentsLexer

our_style = style.from_dict({
    'pygments.comment':   '#888888 bold',
    'pygments.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 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.

Read more about styling.

Using a Pygments style

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

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_cls, merge_styles
from prompt_toolkit.lexers import PygmentsLexer

from pygments.styles.tango import TangoStyle
from pygments.lexers.html import HtmlLexer
```

(continues on next page)

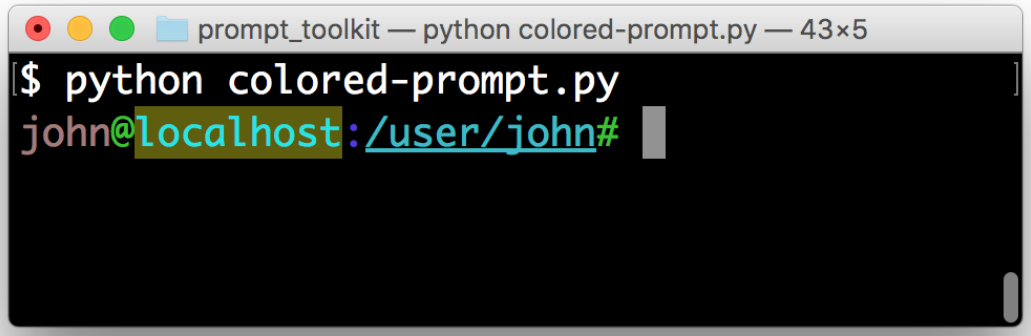
(continued from previous page)

```
our_style = merge_styles([
    style_from_pygments_cls(TangoStyle),
    Style.from_dict({
        'pygments.comment': '#888888 bold',
        'pygments.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 to build some *formatted text*. One way of doing is is by creating a list of style/text tuples. In the following example, we use class names to refer to the style.

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

style = Style.from_dict({
    # User input (default text).
    '':          '#ff0066',

    # Prompt.
    'username': '#884444',
    'at':       '#00aa00',
    'colon':    '#0000aa',
    'pound':    '#00aa00',
    'host':     '#00ffff bg:#444400',
    'path':     'ansicyan underline',
})

message = [
    ('class:username', 'john'),
    ('class:at',       '@'),
    ('class:host',     'localhost'),
    ('class:colon',    ':'),
    ('class:path',     '/user/john'),
    ('class:pound',    '# '),
]

text = prompt(message, style=style)
```



The *message* can be any kind of formatted text, as discussed *here*. It can also be a callable that returns some formatted text.

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.

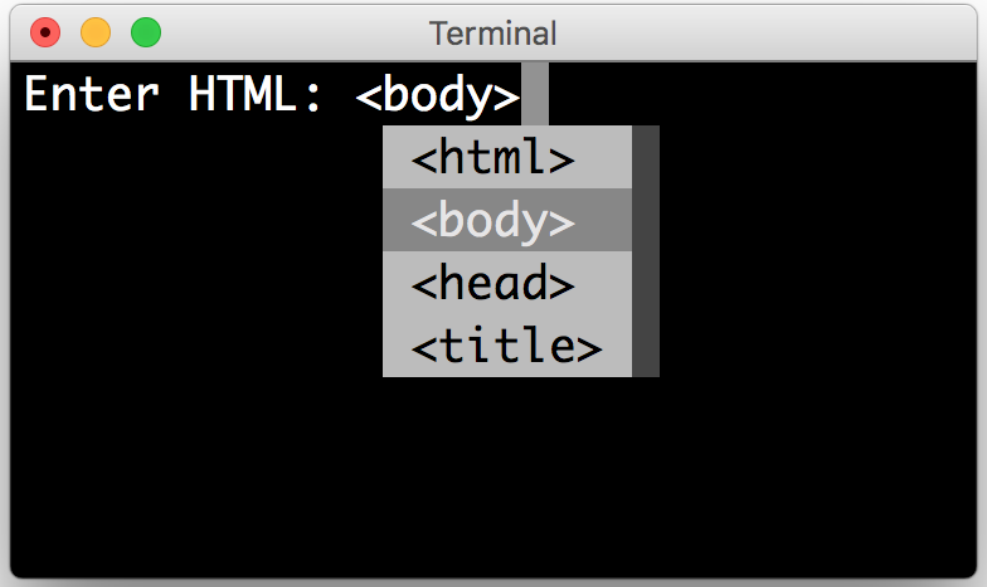
```
text = prompt(message, style=style, true_color=True)
```

3.5.5 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.completion 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.



Note: Note that in prompt_toolkit 2.0, the auto completion became synchronous. This means that if it takes a long time to compute the completions, that this will block the event loop and the input processing.

For heavy completion algorithms, it is recommended to wrap the completer in a *ThreadedCompleter* in order to run it in a background thread.

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.

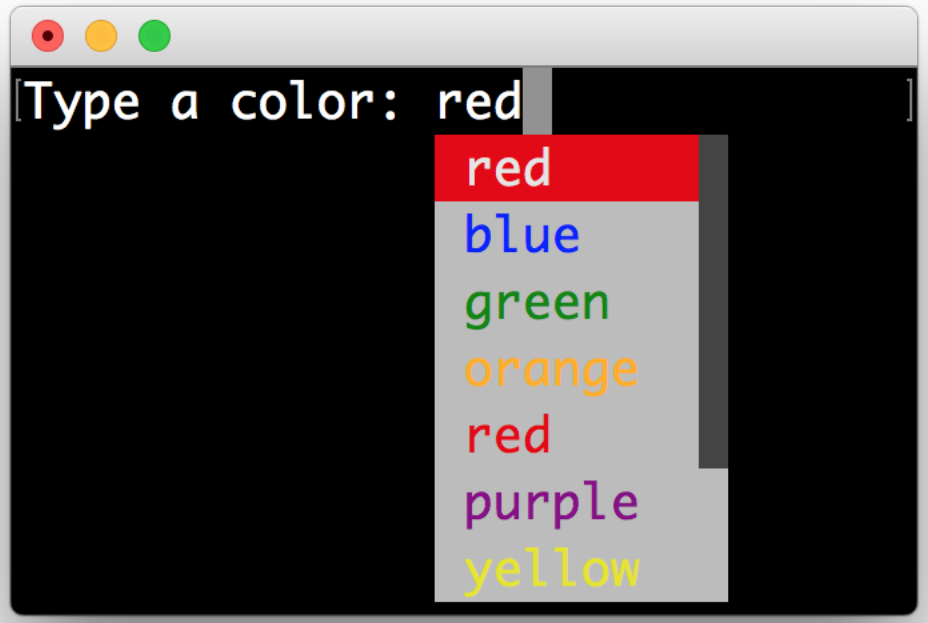
The position is used for fixing text before the cursor. Pressing the tab key could for instance turn parts of the input from lowercase to uppercase. This makes sense for a case insensitive completer. Or in case of a fuzzy completion, it could fix typos. When `start_position` is something negative, this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

Styling individual completions

Each completion can provide a custom style, which is used when it is rendered in the completion menu or toolbar. This is possible by passing a style to each *Completion* instance.

```
from prompt_toolkit.completion import Completer, Completion

class MyCustomCompleter(Completer):
    def get_completions(self, document, complete_event):
        # Display this completion, black on yellow.
        yield Completion('completion1', start_position=0,
                         style='bg:ansiyellow fg:ansiblack')

        # Underline completion.
        yield Completion('completion2', start_position=0,
                         style='underline')

        # Specify class name, which will be looked up in the style sheet.
        yield Completion('completion3', start_position=0,
                         style='class:special-completion')
```

The "colorful-prompts.py" example uses completion styling:



Complete while typing

Autcompletions can be generated automatically while typing or when the user presses the tab key. This can be configured with the `complete_while_typing` option:

```
text = prompt('Enter HTML: ', completer=my_completer,
              complete_while_typing=True)
```

Notice that this setting is incompatible with the `enable_history_search` option. The reason for this is that the up and down key bindings would conflict otherwise. So, make sure to disable history search for this.

Asynchronous completion

When generating the completions takes a lot of time, it's better to do this in a background thread. This is possible by wrapping the completer in a *ThreadedCompleter*, but also by passing the *complete_in_thread=True* argument.

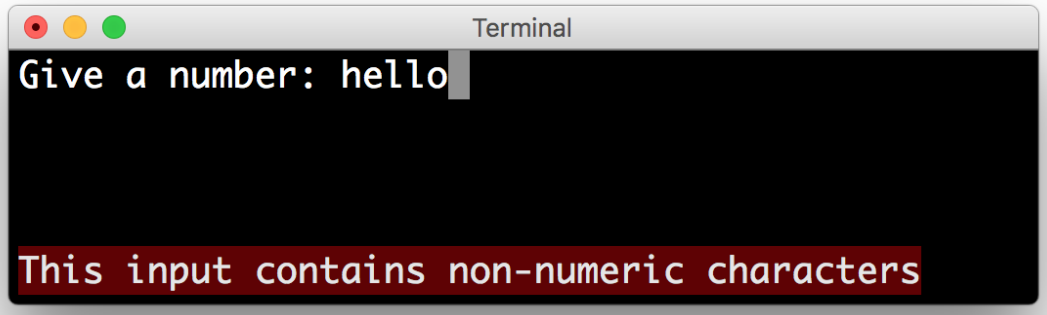
```
text = prompt('> ', completer=MyCustomCompleter(), complete_in_thread=True)
```

3.5.6 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 position.

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)
```



By default, the input is only validated when the user presses the enter key, but prompt_toolkit can also validate in real-time while typing:

```
prompt('Give a number: ', validator=NumberValidator(),
       validate_while_typing=True)
```

If the input validation contains some heavy CPU intensive code, but you don't want to block the event loop, then it's recommended to wrap the validator class in a *ThreadedValidator*.

Validator from a callable

Instead of implementing the *Validator* abstract base class, it is also possible to start from a simple function and use the *from_callable()* classmethod. This is easier and sufficient for probably 90% of the validators. It looks as follows:

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

def is_number(text):
    return text.isdigit()

validator = Validator.from_callable(
    is_number,
    error_message='This input contains non-numeric characters',
    move_cursor_to_end=True)

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

We define a function that takes a string, and tells whether it's valid input or not by returning a boolean. *from_callable()* turns that into a *Validator* instance. Notice that setting the cursor position is not possible this way.

3.5.7 History

A *History* object keeps track of all the previously entered strings, so that the up-arrow can reveal previously entered items.

The recommended way is to use a *PromptSession*, which uses an *InMemoryHistory* for the entire session by default. The following example has a history out of the box:

```
from prompt_toolkit import PromptSession

session = PromptSession()

while True:
    session.prompt()
```

To persist a history to disk, use a *FileHistory* instead instead of the default *InMemoryHistory*. This history object can be passed either to a *PromptSession* or to the *prompt()* function. For instance:

```
from prompt_toolkit import PromptSession
from prompt_toolkit.history import FileHistory

session = PromptSession(history=FileHistory('~/.myhistory'))

while True:
    session.prompt()
```

3.5.8 Auto suggestion

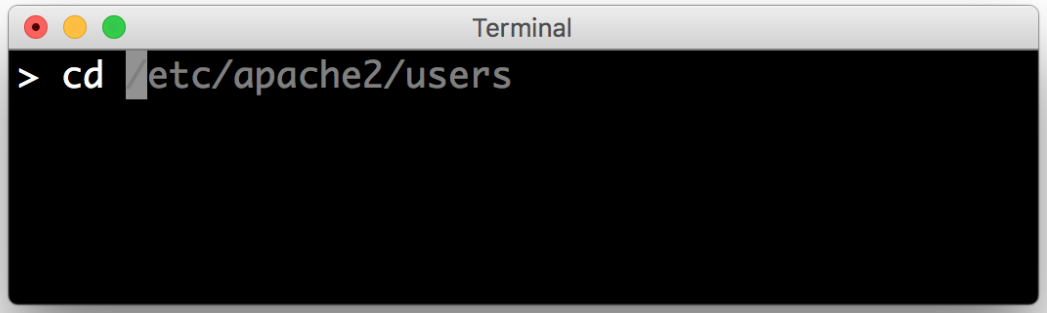
Auto suggestion is a way to propose some input completions to the user like the fish shell.

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 → or `c-e` will insert this suggestion, `alt-f` will insert the first word of the suggestion.

Note: When suggestions are based on the history, don't forget to share one *History* object between consecutive *prompt()* calls. Using a *PromptSession* does this for you.

Example:

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

session = PromptSession()

while True:
    text = session.prompt('> ', 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.

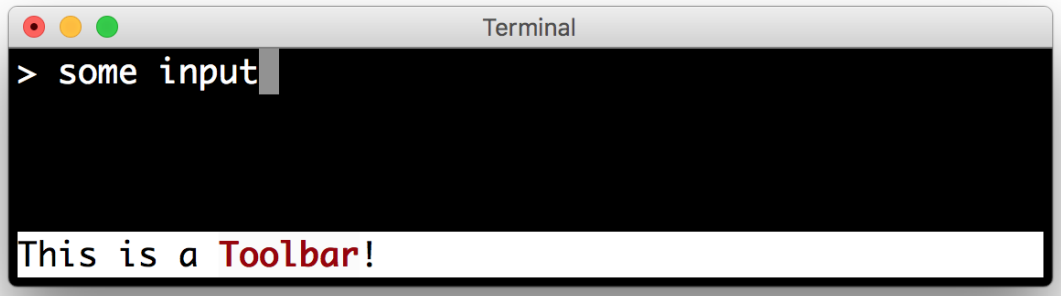
3.5.9 Adding a bottom toolbar

Adding a bottom toolbar is as easy as passing a `bottom_toolbar` argument to *prompt()*. This argument be either plain text, *formatted text* or a callable that returns plain or formatted text.

When a function is given, it will be called every time the prompt is rendered, so the bottom toolbar can be used to display dynamic information.

The toolbar is always erased when the prompt returns. Here we have an example of a callable that returns an *HTML* object. By default, the toolbar has the **reversed style**, which is why we are setting the background instead of the foreground.

```
from prompt_toolkit import prompt
from prompt_toolkit.formatted_text import HTML

def bottom_toolbar():
    return HTML('This is a <b><style bg="ansired">Toolbar</style></b>!')

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



Similar, we could use a list of style/text tuples.

```
from prompt_toolkit import prompt
from prompt_toolkit.styles import Style

def bottom_toolbar():
    return [('class:bottom-toolbar', ' This is a toolbar. ')]

style = Style.from_dict({
    'bottom-toolbar': '#ffffff bg:#333333',
})

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

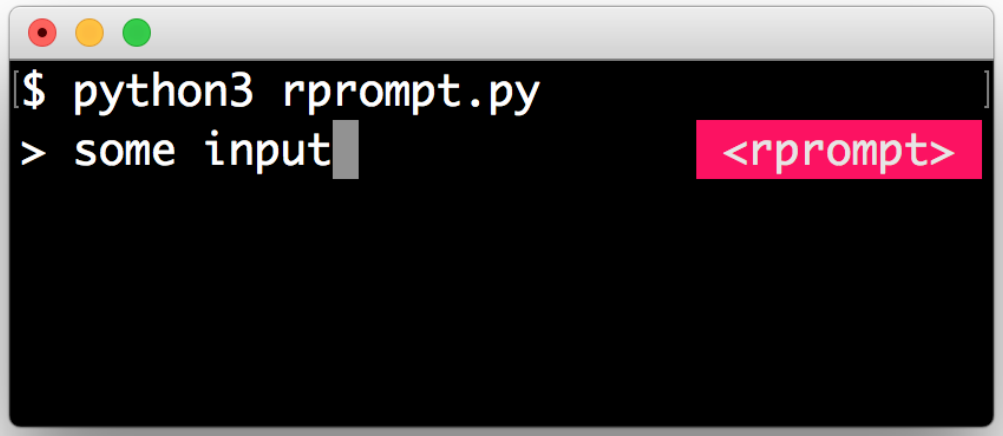
The default class name is `bottom-toolbar` and that will also be used to fill the background of the toolbar.

3.5.10 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 an `rprompt` argument. This can be either plain text, *formatted text* or a callable which returns either.

```
from prompt_toolkit import prompt
from prompt_toolkit.styles import Style

example_style = Style.from_dict({
    'rprompt': 'bg:#ff0066 #ffffff',
})

def get_rprompt():
    return '<rprompt>'

answer = prompt('> ', rprompt=get_rprompt, style=example_style)
```



The `get_rprompt` function can return any kind of formatted text such as *HTML*. it is also possible to pass text directly to the `rprompt` argument of the *prompt()* function. It does not have to be a callable.

3.5.11 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)
```

3.5.12 Adding custom key bindings

By default, every prompt already has a set of key bindings which implements the usual Vi or Emacs behaviour. We can extend this by passing another *KeyBindings* instance to the `key_bindings` argument of the *prompt()* function or the *PromptSession* class.

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

```
from prompt_toolkit import prompt
from prompt_toolkit.application import run_in_terminal
from prompt_toolkit.key_binding import KeyBindings

bindings = KeyBindings()

@bindings.add('c-t')
def _(event):
    " Say 'hello' when `c-t` is pressed. "
    def print_hello():
```

(continues on next page)

(continued from previous page)

```
        print('hello world')
    run_in_terminal(print_hello)

@bindings.add('c-x')
def _(event):
    " Exit when `c-x` is pressed. "
    event.app.exit()

text = prompt('> ', key_bindings=bindings)
print('You said: %s' % text)
```

Note that we use *run_in_terminal()* for the first key binding. This ensures that the output of the print-statement and the prompt don't mix up. If the key bindings doesn't print anything, then it can be handled directly without nesting functions.

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 *Filter*, usually a *Condition* instance. (*Read more about filters*.)

```
from prompt_toolkit import prompt
from prompt_toolkit.filters import Condition
from prompt_toolkit.key_binding import KeyBindings

bindings = KeyBindings()

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

@bindings.add('c-t', filter=is_active)
def _(event):
    # ...
    pass

prompt('> ', key_bindings=bindings)
```

Dynamically switch between Emacs and Vi mode

The *Application* 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.application.current import get_app
from prompt_toolkit.filters import Condition
from prompt_toolkit.key_binding import KeyBindings

def run():
    # Create a set of key bindings.
    bindings = KeyBindings()
```

(continues on next page)

(continued from previous page)

```
    # Add an additional key binding for toggling this flag.
    @bindings.add('f4')
    def _(event):
        " Toggle between Emacs and Vi mode. "
        app = event.app

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

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

    prompt('> ', key_bindings=bindings, bottom_toolbar=bottom_toolbar)

run()
```

Read more about key bindings . . .

Using control-space for completion

An popular short cut that people sometimes use it to use control-space for opening the autocompletion menu instead of the tab key. This can be done with the following key binding.

```
kb = KeyBindings()

@kb.add('c-space')
def _(event):
    " Initialize autocompletion, or select the next completion. "
    buff = event.app.current_buffer
    if buff.complete_state:
        buff.complete_next()
    else:
        buff.start_completion(select_first=False)
```

3.5.13 Other prompt options

Multiline input

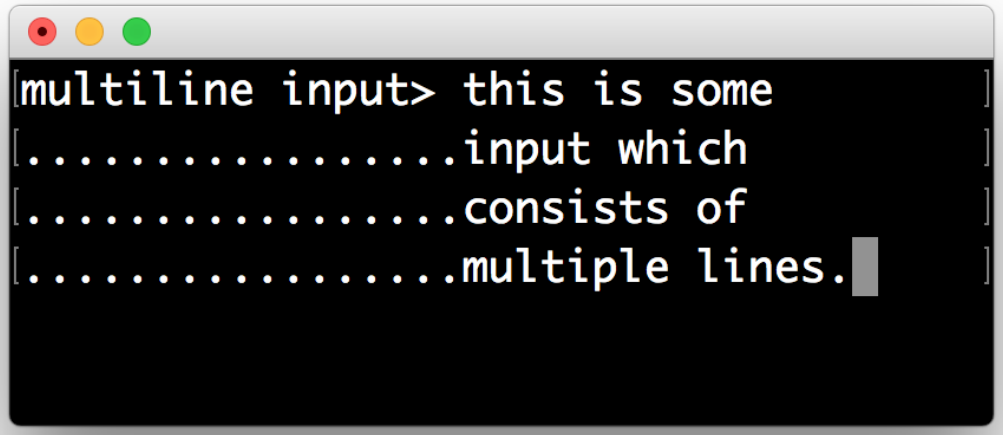
Reading multiline input is as easy as passing the `multiline=True` parameter.

```
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 `prompt_continuation` callable to `prompt()`. This function is supposed to return *formatted text*, or a list of `(style, 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.)

```
from prompt_toolkit import prompt

def prompt_continuation(width, line_number, is_soft_wrap):
    return '.' * width
    # Or: return [('', '.' * width)]

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



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
```

(continues on next page)

(continued from previous page)

```
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)
```

3.5.14 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. First we have to tell prompt_toolkit to use the asyncio event loop. Then prompting the user for input is as simple as calling *prompt()* with the *async_=True* argument.

```
from prompt_toolkit import prompt
from prompt_toolkit.eventloop.defaults import use_asyncio_event_loop
from prompt_toolkit.patch_stdout import patch_stdout

# Tell prompt_toolkit to use the asyncio event loop.
use_asyncio_event_loop()

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

The *patch_stdout()* context manager is optional, but it's recommended, because other coroutines could print to stdout. This ensures that other output won't destroy the prompt.

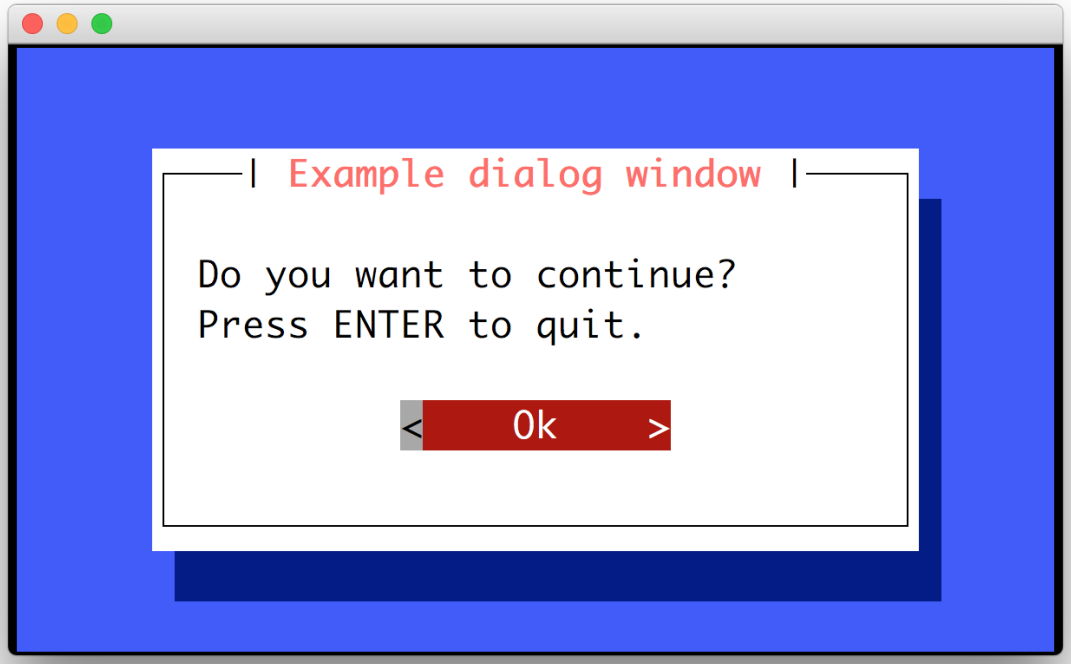
3.6 Dialogs

Prompt_toolkit ships with a high level API for displaying dialogs, similar to the Whiptail program, but in pure Python.

3.6.1 Message box

Use the *message_dialog()* function to display a simple message box. For instance:

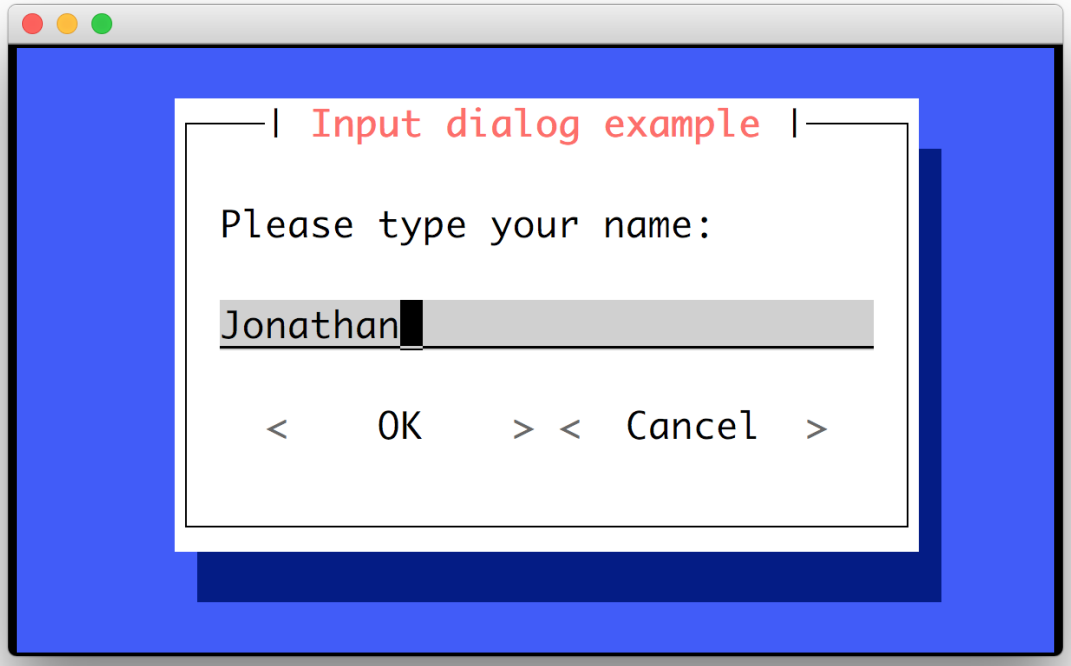
```
from prompt_toolkit.shortcuts import message_dialog

message_dialog(
    title='Example dialog window',
    text='Do you want to continue?\nPress ENTER to quit.')
```



3.6.2 Input box

The *input_dialog()* function can display an input box. It will return the user input as a string.

```
from prompt_toolkit.shortcuts import input_dialog

text = input_dialog(
    title='Input dialog example',
    text='Please type your name:')
```

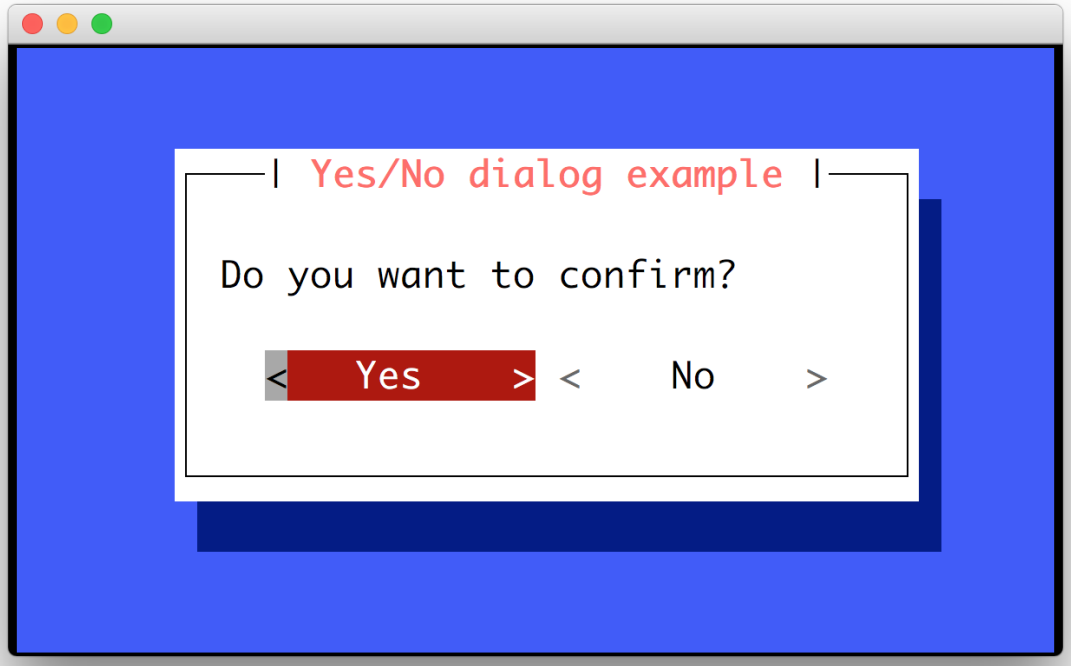


The `password=True` option can be passed to the *`input_dialog()`* function to turn this into a password input box.

3.6.3 Yes/No confirmation dialog

The *`yes_no_dialog()`* function displays a yes/no confirmation dialog. It will return a boolean according to the selection.

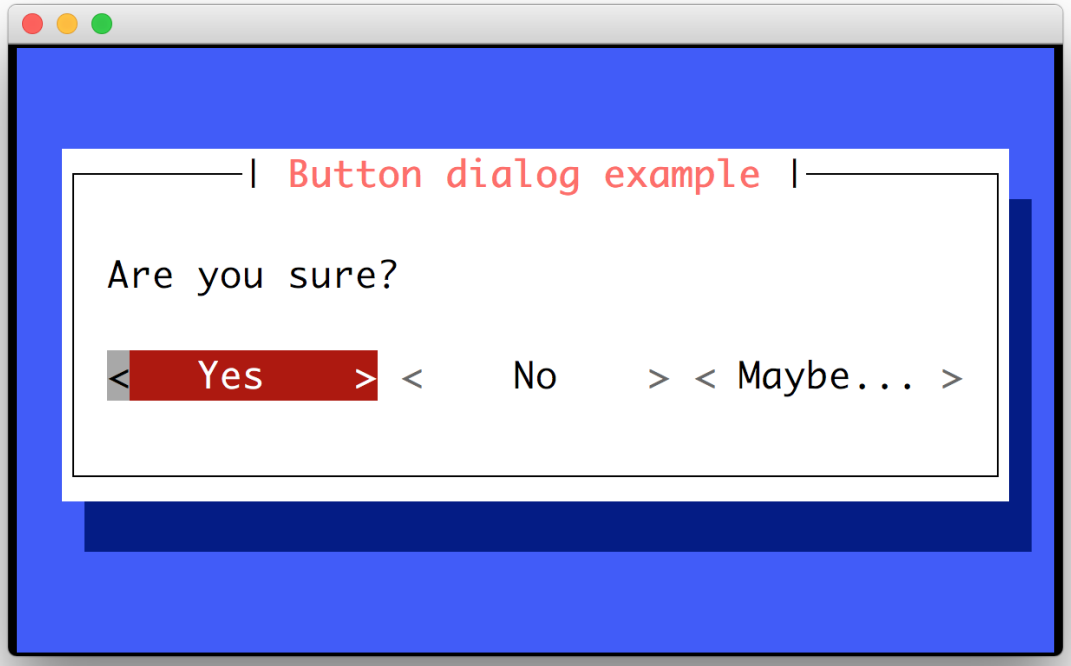
```
from prompt_toolkit.shortcuts import yes_no_dialog

result = yes_no_dialog(
    title='Yes/No dialog example',
    text='Do you want to confirm?')
```



3.6.4 Button dialog

The *button_dialog()* function displays a dialog with choices offered as buttons. Buttons are indicated as a list of tuples, each providing the label (first) and return value if clicked (second).

```
from prompt_toolkit.shortcuts import button_dialog

result = button_dialog(
    title='Button dialog example',
    text='Do you want to confirm?',
    buttons=[
        ('Yes', True),
        ('No', False),
        ('Maybe...', None)
    ],
)
```



3.6.5 Styling of dialogs

A custom *Style* instance can be passed to all dialogs to override the default style. Also, text can be styled by passing an *HTML* object.

```
from prompt_toolkit.formatted_text import HTML
from prompt_toolkit.shortcuts import message_dialog
from prompt_toolkit.styles import Style

example_style = Style.from_dict({
    'dialog':             'bg:#88ff88',
    'dialog frame-label': 'bg:#ffffff #000000',
    'dialog.body':        'bg:#000000 #00ff00',
    'dialog shadow':      'bg:#00aa00',
})

message_dialog(
    title=HTML('<style bg="blue" fg="white">Styled</style> '
               '<style fg="ansired">dialog</style> window'),
    text='Do you want to continue?\nPress ENTER to quit.',
    style=example_style)
```

3.7 Progress bars

Prompt_toolkit ships with a high level API for displaying progress bars, inspired by tqdm

Warning: The API for the prompt_toolkit progress bars is still very new and can possibly change in the future. It is usable and tested, but keep this in mind when upgrading.

Remember that the examples directory of the prompt_toolkit repository ships with many progress bar examples as well.

3.7.1 Simple progress bar

Creating a new progress bar can be done by calling the *ProgressBar* context manager.

The progress can be displayed for any iterable. This works by wrapping the iterable (like `range`) with the *ProgressBar* context manager itself. This way, the progress bar knows when the next item is consumed by the forloop and when progress happens.

```
from prompt_toolkit.shortcuts import ProgressBar
import time


with ProgressBar() as pb:
```

(continues on next page)

(continued from previous page)

```
    for i in pb(range(800)):
        time.sleep(.01)
```

```
$ python simple-progress-bar.py
  39.1% [============>                ] 313/800 eta [00:05]
```

Keep in mind that not all iterables can report their total length. This happens with a typical generator. In that case, you can still pass the total as follows in order to make displaying the progress possible:

```
def some_iterable():
    yield ...

with ProgressBar() as pb:
    for i in pb(some_iterable, total=1000):
        time.sleep(.01)
```

3.7.2 Multiple parallel tasks

A prompt_toolkit *ProgressBar* can display the progress of multiple tasks running in parallel. Each task can run in a separate thread and the *ProgressBar* user interface runs in its own thread.

Notice that we set the "daemon" flag for both threads that run the tasks. This is because control-c will stop the progress and quit our application. We don't want the application to wait for the background threads to finish. Whether you want this depends on the application.

```
from prompt_toolkit.shortcuts import ProgressBar
import time
import threading


with ProgressBar() as pb:
    # Two parallel tasks.
    def task_1():
        for i in pb(range(100)):
            time.sleep(.05)

    def task_2():
        for i in pb(range(150)):
            time.sleep(.08)

    # Start threads.
    t1 = threading.Thread(target=task_1)
    t2 = threading.Thread(target=task_2)
    t1.daemon = True
    t2.daemon = True
    t1.start()
    t2.start()

    # Wait for the threads to finish. We use a timeout for the join() call,
    # because on Windows, join cannot be interrupted by Control-C or any other
    # signal.
    for t in [t1, t2]:
        while t.is_alive():
            t.join(timeout=.5)
```

```
$ python two-tasks.py
 100.0% [================================================================================>] 100/100 eta [00:00]
  51.3% [=========================================>                                       ]  77/150 eta [00:05]
```

3.7.3 Adding a title and label

Each progress bar can have one title, and for each task an individual label. Both the title and the labels can be *formatted text*.

```
from prompt_toolkit.shortcuts import ProgressBar
from prompt_toolkit.formatted_text import HTML
import time

title = HTML('Downloading <style bg="yellow" fg="black">4 files...</style>')
label = HTML('<ansired>some file</ansired>: ')

with ProgressBar(title=title) as pb:
    for i in pb(range(800), label=label):
        time.sleep(.01)
```

```
$ python colored-title-and-label.py
Downloading 4 files...
some file:   31.5% [===>          ] 252/800 eta [00:05]
```

3.7.4 Formatting the progress bar

The visualisation of a `ProgressBar` can be customized by using a different sequence of formatters. The default formatting looks something like this:

```
from prompt_toolkit.shortcuts.progress_bar.formatters import *

default_formatting = [
    Label(),
    Text(' '),
    Percentage(),
    Text(' '),
    Bar(),
    Text(' '),
    Progress(),
    Text(' '),
    Text('eta [', style='class:time-left'),
    TimeLeft(),
    Text(']', style='class:time-left'),
    Text(' '),
]
```

That sequence of `Formatter` can be passed to the *formatter* argument of `ProgressBar`. So, we could change this and modify the progress bar to look like an apt-get style progress bar:

```
from prompt_toolkit.shortcuts import ProgressBar
from prompt_toolkit.styles import Style
from prompt_toolkit.shortcuts.progress_bar import formatters
import time
```

(continues on next page)

(continued from previous page)

```
style = Style.from_dict({
    'label': 'bg:#ffff00 #000000',
    'percentage': 'bg:#ffff00 #000000',
    'current': '#448844',
    'bar': '',
})


custom_formatters = [
    formatters.Label(),
    formatters.Text(': [', style='class:percentage'),
    formatters.Percentage(),
    formatters.Text(']', style='class:percentage'),
    formatters.Text(' '),
    formatters.Bar(sym_a='#', sym_b='#', sym_c='.'),
    formatters.Text('  '),
]

with ProgressBar(style=style, formatters=custom_formatters) as pb:
    for i in pb(range(1600), label='Installing'):
        time.sleep(.01)
```

```
$ python styled-apt-get-install.py
Installing: [ 64.4%] [####################.........]
```

3.7.5 Adding key bindings and toolbar

Like other prompt_toolkit applications, we can add custom key bindings, by passing a *KeyBindings* object:

```
from prompt_toolkit import HTML
from prompt_toolkit.key_binding import KeyBindings
from prompt_toolkit.patch_stdout import patch_stdout
from prompt_toolkit.shortcuts import ProgressBar

import time

bottom_toolbar = HTML(' <b>[f]</b> Print "f" <b>[x]</b> Abort.')

# Create custom key bindings first.
kb = KeyBindings()
cancel = [False]

@kb.add('f')
def _(event):
    print('You pressed `f`.')

@kb.add('x')
def _(event):
    " Send Abort (control-c) signal. "
    cancel[0] = True
    os.kill(os.getpid(), signal.SIGINT)

# Use `patch_stdout`, to make sure that prints go above the
# application.
```

(continues on next page)

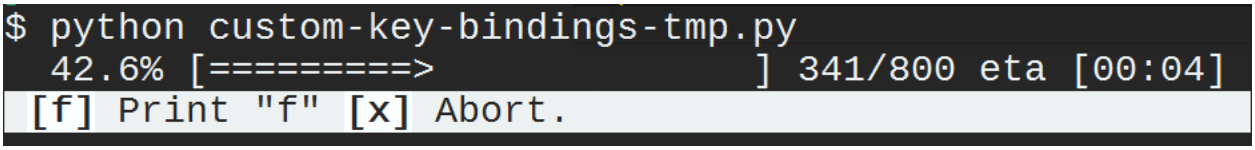
(continued from previous page)

```
with patch_stdout():
    with ProgressBar(key_bindings=kb, bottom_toolbar=bottom_toolbar) as pb:
        for i in pb(range(800)):
            time.sleep(.01)

            # Stop when the cancel flag has been set.
            if cancel[0]:
                break
```

Notice that we use *patch_stdout()* to make printing text possible while the progress bar is displayed. This ensures that printing happens above the progress bar.

Further, when "x" is pressed, we set a cancel flag, which stops the progress. It would also be possible to send *SIGINT* to the mean thread, but that's not always considered a clean way of cancelling something.

In the example above, we also display a toolbar at the bottom which shows the key bindings.



Read more about key bindings . . .

3.8 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.

Before going through this page, it could be helpful to go through *asking for input* (prompts) first. Many things that apply to an input prompt, like styling, key bindings and so on, also apply to full screen applications.

Note: Also remember that the `examples` directory of the prompt_toolkit repository contains plenty of examples. Each example is supposed to explain one idea. So, this as well should help you get started.

Don't hesitate to open a GitHub issue if you feel that a certain example is missing.

3.8.1 A simple application

Every prompt_toolkit application is an instance of an *Application* object. The simplest full screen example would look like this:

```
from prompt_toolkit import Application

app = Application(full_screen=True)
app.run()
```

This will display a dummy application that says "No layout specified. Press ENTER to quit.".

Note: If we wouldn't set the `full_screen` option, the application would not run in the alternate screen buffer, and only consume the least amount of space required for the layout.

An application consists of several components. The most important are:

- I/O objects: the input and output device.
- The layout: this defines the graphical structure of the application. For instance, a text box on the left side, and a button on the right side. You can also think of the layout as a collection of 'widgets'.
- A style: this defines what colors and underline/bold/italic styles are used everywhere.
- A set of key bindings.

We will discuss all of these in more detail them below.

3.8.2 I/O objects

Every *Application* instance requires an I/O objects for input and output:

- An *Input* instance, which is an abstraction of the input stream (stdin).
- An *Output* instance, which is an abstraction of the output stream, and is called by the renderer.

Both are optional and normally not needed to pass explicitly. Usually, the default works fine.

There is a third I/O object which is also required by the application, but not passed inside. This is the event loop, 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 the appropriate handler, like for instance, a key binding.

When *run()* is called, the event loop will run until the application is done. An application will quit when *exit()* is called.

3.8.3 The layout

A layered layout architecture

There are several ways to create a prompt_toolkit layout, depending on how customizable you want things to be. In fact, there are several layers of abstraction.

- The most low-level way of creating a layout is by combining *Container* and *UIControl* objects.

  Examples of *Container* objects are *VSplit* (vertical split), *HSplit* (horizontal split) and *FloatContainer*. These containers arrange the layout and can split it in multiple regions. Each container can recursively contain multiple other containers. They can be combined in any way to define the "shape" of the layout.

  The *Window* object is a special kind of container that can contain a *UIControl* object. The *UIControl* object is responsible for the generation of the actual content. The *Window* object acts as an adaptor between the *UIControl* and other containers, but it's also responsible for the scrolling and line wrapping of the content.

  Examples of *UIControl* objects are *BufferControl* for showing the content of an editable/scrollable buffer, and *FormattedTextControl* for displaying (*formatted*) text.

  Normally, it is never needed to create new *UIControl* or *Container* classes, but instead you would create the layout by composing instances of the existing built-ins.

- A higher level abstraction of building a layout is by using "widgets". A widget is a reusable layout component that can contain multiple containers and controls. It should have a `__pt__container__` function, which is supposed to return the root container for this widget. Prompt_toolkit contains a couple of widgets like *TextArea*, *Button*, *Frame*, *VerticalLine* and so on.
- The highest level abstractions can be found in the `shortcuts` module. There we don't have to think about the layout, controls and containers at all. This is the simplest way to use prompt_toolkit, but is only meant for specific use cases, like a prompt or a simple dialog window.

Containers and controls

The biggest difference between containers and controls is that containers arrange the layout by splitting the screen in many regions, while controls are responsible for generating the actual content.

Note: Under the hood, the difference is:

- 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 FormattedTextControl*

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 and wrapping the lines if needed.

Finally, there is the *Layout* class which wraps the whole layout. This is responsible for keeping track of which window has the focus.

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 import Application
from prompt_toolkit.buffer import Buffer
from prompt_toolkit.layout.containers import VSplit, Window
from prompt_toolkit.layout.controls import BufferControl, FormattedTextControl
from prompt_toolkit.layout.layout import Layout

buffer1 = Buffer()  # Editable buffer.

root_container = VSplit([
    # One window that holds the BufferControl with the default buffer on
    # the left.
    Window(content=BufferControl(buffer=buffer1)),

    # A vertical line in the middle. We explicitly 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 window will simply fill its
    # content by repeating this character.
    Window(width=1, char='|'),

    # Display the text 'Hello world' on the right.
    Window(content=FormattedTextControl(text='Hello world')),
```

(continues on next page)

(continued from previous page)

```
])

layout = Layout(root_container)

app = Application(layout=layout, full_screen=True)
app.run() # You won't be able to Exit this app
```

Notice that if you execute this right now, there is no way to quit this application yet. This is something we explain in the next section below.

More complex layouts can be achieved by nesting multiple *VSplit*, *HSplit* and *FloatContainer* objects.

If you want to make some part of the layout only visible when a certain condition is satisfied, use a *ConditionalContainer*.

Focusing windows

Focussing something can be done by calling the *focus()* method. This method is very flexible and accepts a *Window*, a *Buffer*, a `UIControl` and more.

In the following example, we use *get_app()* for getting the active application.

```
from prompt_toolkit.application import get_app

# This window was created earlier.
w = Window()

# ...

# Now focus it.
get_app().layout.focus(w)
```

Changing the focus is something which is typically done in a key binding, so read on to see how to define key bindings.

3.8.4 Key bindings

In order to react to user actions, we need to create a *KeyBindings* object and pass that to our *Application*.

There are two kinds of key bindings:

- Global key bindings, which are always active.
- Key bindings that belong to a certain `UIControl` and are only active when this control is focused. Both *BufferControl* *FormattedTextControl* take a `key_bindings` argument.

Global key bindings

Key bindings can be passed to the application as follows:

```
from prompt_toolkit import Application
from prompt_toolkit.key_binding import KeyBindings

kb = KeyBindings()
app = Application(key_bindings=kb)
app.run()
```

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

```
from prompt_toolkit import Application
from prompt_toolkit.key_binding import KeyBindings

kb = KeyBindings()

@kb.add('c-q')
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.app.exit()

app = Application(key_bindings=kb, full_screen=True)
app.run()
```

The callback function is named `exit_` for clarity, but it could have been named _ (underscore) as well, because the we won't refer to this name.

Read more about key bindings ...

Modal containers

All container objects, like *VSplit* and *HSplit* take a `modal` argument.

If this flag has been set, then key bindings from the parent account are not taken into account if one of the children windows has the focus.

This is useful in a complex layout, where many controls have their own key bindings, but you only want to enable the key bindings for a certain region of the layout.

The global key bindings are always active.

3.8.5 More about the Window class

As said earlier, a *Window* is a *Container* that wraps a *UIControl*, like a *BufferControl* or *FormattedTextControl*.

Note: Basically, windows are the leafs in the tree structure that represent the UI.

A *Window* provides a "view" on the *UIControl*, which provides lines of content. The window is in the first place responsible for the line wrapping and scrolling of the content, but there are much more options.

- Adding left or right margins. These are used for displaying scroll bars or line numbers.
- There are the *cursorline* and *cursorcolumn* options. These allow highlighting the line or column of the cursor position.
- Alignment of the content. The content can be left aligned, right aligned or centered.
- Finally, the background can be filled with a default character.

3.8.6 More about buffers and *BufferControl*

Input processors

A *Processor* is used to postprocess the content of a *BufferControl* before it's displayed. It can for instance highlight matching brackets or change the visualisation of tabs and so on.

A *Processor* operates on individual lines. Basically, it takes a (formatted) line and produces a new (formatted) line.

Some build-in processors:

Processor	Usage:
HighlightSearchProcessor	Highlight the current search results.
HighlightSelectionProcessor	Highlight the selection.
PasswordProcessor	Display input as asterisks. (* 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.

A *BufferControl* takes only one processor as input, but it is possible to "merge" multiple processors into one with the *merge_processors()* function.

3.9 Tutorials

3.9.1 Tutorial: Build an SQLite REPL

The aim of this tutorial is to build an interactive command line interface for an SQLite database using prompt_toolkit.

First, install the library using pip, if you haven't done this already.

```
pip install prompt_toolkit
```

Read User Input

Let's start accepting input using the *prompt()* function. This will ask the user for input, and echo back whatever the user typed. We wrap it in a `main()` function as a good practise.

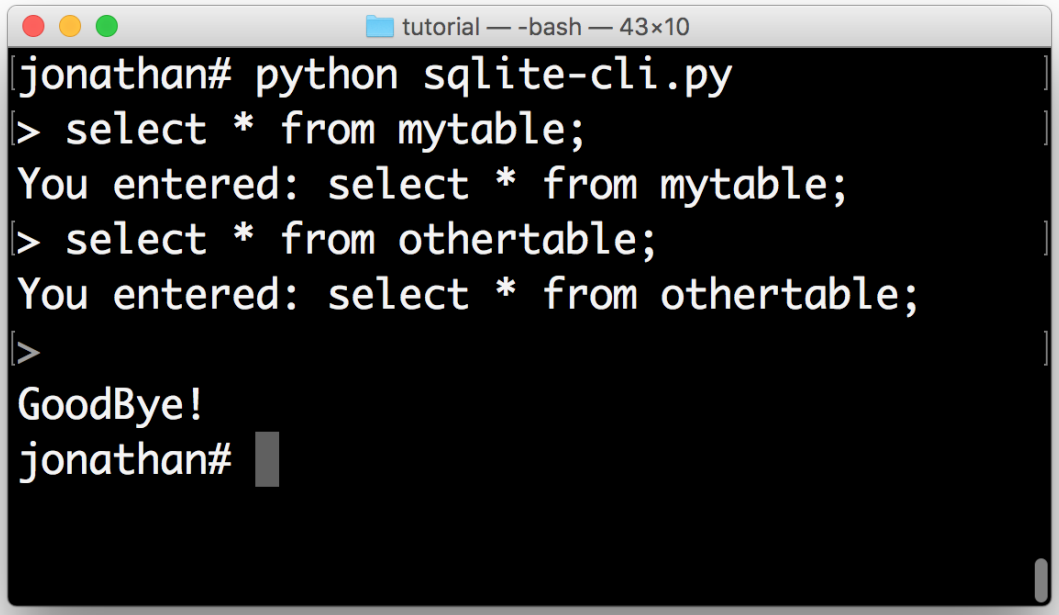
```
from __future__ import unicode_literals
from prompt_toolkit import prompt

def main():
    text = prompt('> ')
    print('You entered:', text)

if __name__ == '__main__':
    main()
```

Loop The REPL

Now we want to call the *prompt()* method in a loop. In order to keep the history, the easiest way to do it is to use a *PromptSession*. This uses an *InMemoryHistory* underneath that keeps track of the history, so that if the user presses the up-arrow, he'll see the previous entries.

The *prompt()* method raises `KeyboardInterrupt` when ControlC has been pressed and `EOFError` when ControlD has been pressed. This is what people use for cancelling commands and exiting in a REPL. The try/except below handles these error conditions and make sure that we go to the next iteration of the loop or quit the loop respectively.

```
from __future__ import unicode_literals
from prompt_toolkit import PromptSession

def main():
    session = PromptSession()

    while True:
        try:
            text = session.prompt('> ')
        except KeyboardInterrupt:
            continue
        except EOFError:
            break
        else:
            print('You entered:', text)
    print('GoodBye!')

if __name__ == '__main__':
    main()
```



Syntax Highlighting

This is where things get really interesting. Let's step it up a notch by adding syntax highlighting to the user input. We know that users will be entering SQL statements, so we can leverage the Pygments library for coloring the input. The `lexer` parameter allows us to set the syntax lexer. We're going to use the `SqlLexer` from the Pygments library for highlighting.

Notice that in order to pass a Pygments lexer to prompt_toolkit, it needs to be wrapped into a *`PygmentsLexer`*.

```
from __future__ import unicode_literals
from prompt_toolkit import PromptSession
from prompt_toolkit.lexers import PygmentsLexer
from pygments.lexers.sql import SqlLexer

def main():
    session = PromptSession(lexer=PygmentsLexer(SqlLexer)

    while True:
        try:
            text = session.prompt('> ')
        except KeyboardInterrupt:
            continue
        except EOFError:
            break
        else:
            print('You entered:', text)
    print('GoodBye!')
```

(continues on next page)

(continued from previous page)

```
if __name__ == '__main__':
    main()
```

Auto-completion

Now we are going to add auto completion. We'd like to display a drop down menu of possible keywords when the user starts typing.

We can do this by creating an *sql_completer* object from the *WordCompleter* class, defining a set of *keywords* for the auto-completion.

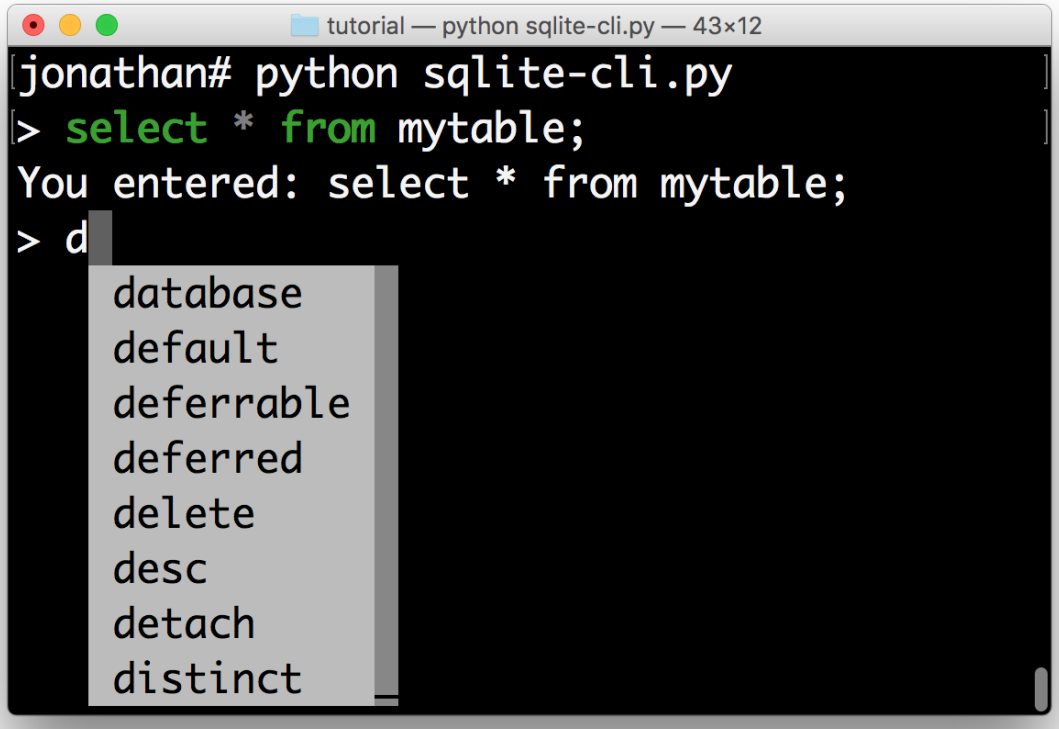
Like the lexer, this `sql_completer` instance can be passed to either the *PromptSession* class or the *prompt()* method.

```
from __future__ import unicode_literals
from prompt_toolkit import PromptSession
from prompt_toolkit.completion import WordCompleter
from prompt_toolkit.lexers import PygmentsLexer
from pygments.lexers.sql import SqlLexer

sql_completer = WordCompleter([
    'abort', 'action', 'add', 'after', 'all', 'alter', 'analyze', 'and',
    'as', 'asc', 'attach', 'autoincrement', 'before', 'begin', 'between',
    'by', 'cascade', 'case', 'cast', 'check', 'collate', 'column',
    'commit', 'conflict', 'constraint', 'create', 'cross', 'current_date',
    'current_time', 'current_timestamp', 'database', 'default',
    'deferrable', 'deferred', 'delete', 'desc', 'detach', 'distinct',
    'drop', 'each', 'else', 'end', 'escape', 'except', 'exclusive',
    'exists', 'explain', 'fail', 'for', 'foreign', 'from', 'full', 'glob',
    'group', 'having', 'if', 'ignore', 'immediate', 'in', 'index',
    'indexed', 'initially', 'inner', 'insert', 'instead', 'intersect',
    'into', 'is', 'isnull', 'join', 'key', 'left', 'like', 'limit',
    'match', 'natural', 'no', 'not', 'notnull', 'null', 'of', 'offset',
```

(continues on next page)

(continued from previous page)

```
    'on', 'or', 'order', 'outer', 'plan', 'pragma', 'primary', 'query',
    'raise', 'recursive', 'references', 'regexp', 'reindex', 'release',
    'rename', 'replace', 'restrict', 'right', 'rollback', 'row',
    'savepoint', 'select', 'set', 'table', 'temp', 'temporary', 'then',
    'to', 'transaction', 'trigger', 'union', 'unique', 'update', 'using',
    'vacuum', 'values', 'view', 'virtual', 'when', 'where', 'with',
    'without'], ignore_case=True)

def main():
    session = PromptSession(
        lexer=PygmentsLexer(SqlLexer), completer=sql_completer)

    while True:
        try:
            text = session.prompt('> ')
        except KeyboardInterrupt:
            continue
        except EOFError:
            break
        else:
            print('You entered:', text)
    print('GoodBye!')

if __name__ == '__main__':
    main()
```



In about 30 lines of code we got ourselves an auto completing, syntax highlighting REPL. Let's make it even better.

Styling the menus

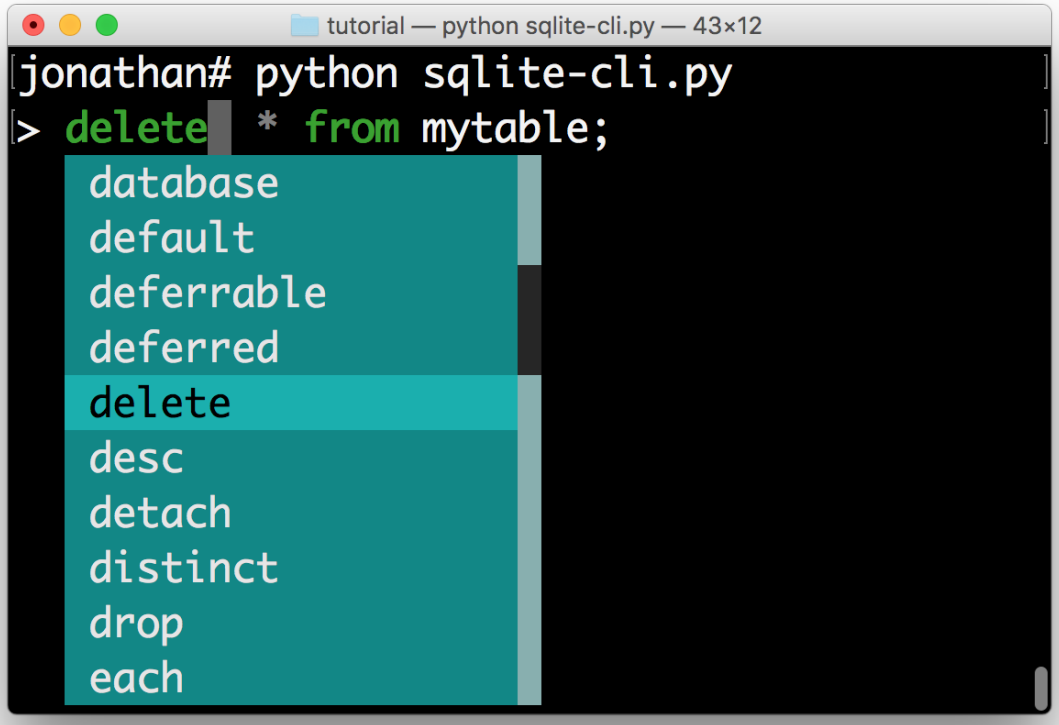
If we want, we can now change the colors of the completion menu. This is possible by creating a *Style* instance and passing it to the *prompt()* function.

```
from __future__ import unicode_literals
from prompt_toolkit import PromptSession
from prompt_toolkit.completion import WordCompleter
from prompt_toolkit.lexers import PygmentsLexer
from prompt_toolkit.styles import Style
from pygments.lexers.sql import SqlLexer

sql_completer = WordCompleter([
    'abort', 'action', 'add', 'after', 'all', 'alter', 'analyze', 'and',
    'as', 'asc', 'attach', 'autoincrement', 'before', 'begin', 'between',
    'by', 'cascade', 'case', 'cast', 'check', 'collate', 'column',
    'commit', 'conflict', 'constraint', 'create', 'cross', 'current_date',
    'current_time', 'current_timestamp', 'database', 'default',
    'deferrable', 'deferred', 'delete', 'desc', 'detach', 'distinct',
    'drop', 'each', 'else', 'end', 'escape', 'except', 'exclusive',
    'exists', 'explain', 'fail', 'for', 'foreign', 'from', 'full', 'glob',
    'group', 'having', 'if', 'ignore', 'immediate', 'in', 'index',
    'indexed', 'initially', 'inner', 'insert', 'instead', 'intersect',
    'into', 'is', 'isnull', 'join', 'key', 'left', 'like', 'limit',
    'match', 'natural', 'no', 'not', 'notnull', 'null', 'of', 'offset',
    'on', 'or', 'order', 'outer', 'plan', 'pragma', 'primary', 'query',
    'raise', 'recursive', 'references', 'regexp', 'reindex', 'release',
    'rename', 'replace', 'restrict', 'right', 'rollback', 'row',
    'savepoint', 'select', 'set', 'table', 'temp', 'temporary', 'then',
    'to', 'transaction', 'trigger', 'union', 'unique', 'update', 'using',
    'vacuum', 'values', 'view', 'virtual', 'when', 'where', 'with',
    'without'], ignore_case=True)

style = Style.from_dict({
    'completion-menu.completion': 'bg:#008888 #ffffff',
    'completion-menu.completion.current': 'bg:#00aaaa #000000',
    'scrollbar.background': 'bg:#88aaaa',
    'scrollbar.button': 'bg:#222222',
})

def main():
   session = PromptSession(
       lexer=PygmentsLexer(SqlLexer), completer=sql_completer, style=style)

   while True:
       try:
           text = session.prompt('> ')
       except KeyboardInterrupt:
           continue
       except EOFError:
           break
       else:
           print('You entered:', text)
   print('GoodBye!')
```

(continues on next page)

(continued from previous page)

```
if __name__ == '__main__':
    main()
```



All that's left is hooking up the sqlite backend, which is left as an exercise for the reader. Just kidding... Keep reading.

Hook up Sqlite

This step is the final step to make the SQLite REPL actually work. It's time to relay the input to SQLite.

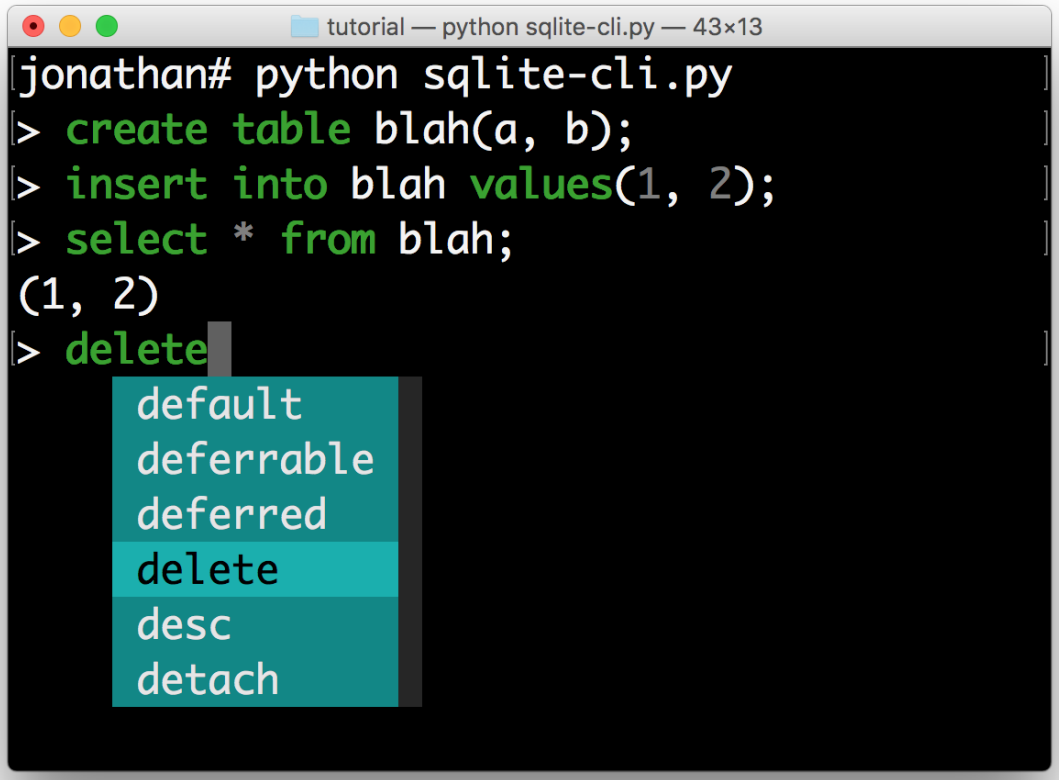
Obviously I haven't done the due diligence to deal with the errors. But it gives a good idea of how to get started.

```
#!/usr/bin/env python
from __future__ import unicode_literals
import sys
import sqlite3

from prompt_toolkit import PromptSession
from prompt_toolkit.completion import WordCompleter
from prompt_toolkit.lexers import PygmentsLexer
from prompt_toolkit.styles import Style
from pygments.lexers.sql import SqlLexer

sql_completer = WordCompleter([
    'abort', 'action', 'add', 'after', 'all', 'alter', 'analyze', 'and',
```

(continues on next page)

(continued from previous page)

```
    'as', 'asc', 'attach', 'autoincrement', 'before', 'begin', 'between',
    'by', 'cascade', 'case', 'cast', 'check', 'collate', 'column',
    'commit', 'conflict', 'constraint', 'create', 'cross', 'current_date',
    'current_time', 'current_timestamp', 'database', 'default',
    'deferrable', 'deferred', 'delete', 'desc', 'detach', 'distinct',
    'drop', 'each', 'else', 'end', 'escape', 'except', 'exclusive',
    'exists', 'explain', 'fail', 'for', 'foreign', 'from', 'full', 'glob',
    'group', 'having', 'if', 'ignore', 'immediate', 'in', 'index',
    'indexed', 'initially', 'inner', 'insert', 'instead', 'intersect',
    'into', 'is', 'isnull', 'join', 'key', 'left', 'like', 'limit',
    'match', 'natural', 'no', 'not', 'notnull', 'null', 'of', 'offset',
    'on', 'or', 'order', 'outer', 'plan', 'pragma', 'primary', 'query',
    'raise', 'recursive', 'references', 'regexp', 'reindex', 'release',
    'rename', 'replace', 'restrict', 'right', 'rollback', 'row',
    'savepoint', 'select', 'set', 'table', 'temp', 'temporary', 'then',
    'to', 'transaction', 'trigger', 'union', 'unique', 'update', 'using',
    'vacuum', 'values', 'view', 'virtual', 'when', 'where', 'with',
    'without'], ignore_case=True)

style = Style.from_dict({
    'completion-menu.completion': 'bg:#008888 #ffffff',
    'completion-menu.completion.current': 'bg:#00aaaa #000000',
    'scrollbar.background': 'bg:#88aaaa',
    'scrollbar.button': 'bg:#222222',
})

def main(database):
    connection = sqlite3.connect(database)
    session = PromptSession(
        lexer=PygmentsLexer(SqlLexer), completer=sql_completer, style=style)

    while True:
        try:
            text = session.prompt('> ')
        except KeyboardInterrupt:
            continue  # Control-C pressed. Try again.
        except EOFError:
            break  # Control-D pressed.

        with connection:
            try:
                messages = connection.execute(text)
            except Exception as e:
                print(repr(e))
            else:
                for message in messages:
                    print(message)

    print('GoodBye!')

if __name__ == '__main__':
    if len(sys.argv) < 2:
        db = ':memory:'
    else:
        db = sys.argv[1]

    main(db)
```



I hope that gives an idea of how to get started on building command line interfaces.

The End.

3.10 Advanced topics

3.10.1 Move about key bindings

This page contains a few additional notes about key bindings.

Key bindings can be defined as follows by creating a *KeyBindings* instance:

```
from prompt_toolkit.key_binding import KeyBindings

bindings = KeyBindings()

@bindings.add('a')
def _(event):
    " Do something if 'a' has been pressed. "
    ...
```

(continues on next page)

(continued from previous page)

```
@bindings.add('c-t')
def _(event):
    " Do something if Control-T has been pressed. "
    ...
```

Note: `c-q` (control-q) and `c-s` (control-s) are often captured by the terminal, because they were used traditionally for software flow control. When this is enabled, the application will automatically freeze when `c-s` is pressed, until `c-q` is pressed. It won't be possible to bind these keys.

In order to disable this, execute type the following in your shell, or even add it to your *.bashrc*.

```
stty -ixon
```

Key bindings can even consist of a sequence of multiple keys. The binding is only triggered when all the keys in this sequence are pressed.

```
@bindings.add('a', 'b')
def _(event):
    " Do something if 'a' is pressed and then 'b' is pressed. "
    ...
```

If the user presses only *a*, then nothing will happen until either a second key (like *b*) has been pressed or until the timeout expires (see later).

List of special keys

Besides literal characters, any of the following keys can be used in a key binding:

Name	Possible keys
Escape	`escape`
Arrows	`left`, `right`, `up`, `down`
Navigation	`home`, `end`, `delete`, `pageup`, `pagedown`, `insert`
Control+lowercase	`c-a`, `c-b`, `c-c`, `c-d`, `c-e`, `c-f`, `c-g`, `c-h`, `c-i`, `c-j`, `c-k`, `c-l`, `c-m`, `c-n`, `c-o`, `c-p`, `c-q`, `c-r`, `c-s`, `c-t`, `c-u`, `c-v`, `c-w`, `c-x`, `c-y`, `c-z`
Control+uppercase	`c-A`, `c-B`, `c-C`, `c-D`, `c-E`, `c-F`, `c-G`, `c-H`, `c-I`, `c-J`, `c-K`, `c-L`, `c-M`, `c-N`, `c-O`, `c-P`, `c-Q`, `c-R`, `c-S`, `c-T`, `c-U`, `c-V`, `c-W`, `c-X`, `c-Y`, `c-Z`
Control + arrow	`c-left`, `c-right`, `c-up`, `c-down`
Other control keys	`c-@`, `c-\`, `c-]`, `c-^`, `c-_`, `c-delete`
Shift + arrow	`s-left`, `s-right`, `s-up`, `s-down`
Other shift keys	`s-delete`, `s-tab`
F-keys	`f1`, `f2`, `f3`, `f4`, `f5`, `f6`, `f7`, `f8`, `f9`, `f10`, `f11`, `f12`, `f13`, `f14`, `f15`, `f16`, `f17`, `f18`, `f19`, `f20`, `f21`, `f22`, `f23`, `f24`

There are a couple of useful aliases as well:

c-h	backspace
c-@	c-space
c-m	enter
c-i	tab

Note: Note that the supported keys are limited to what typical VT100 terminals offer. Binding `c-7` (control + number 7) for instance is not supported.

Binding alt+something, option+something or meta+something

Vt100 terminals translate the alt key into a leading `escape` key. For instance, in order to handle `alt-f`, we have to handle `escape` + `f`. Notice that we receive this as two individual keys. This means that it's exactly the same as first typing `escape` and then typing `f`. Something this alt-key is also known as option or meta.

In code that looks as follows:

```
@bindings.add('escape', 'f')
def _(event):
    " Do something if alt-f or meta-f have been pressed. "
```

Wildcards

Sometimes you want to catch any key that follows after a certain key stroke. This is possible by binding the '<any>' key:

```
@bindings.add('a', '<any>')
def _(event):
    ...
```

This will handle *aa*, *ab*, *ac*, etcetera. The key binding can check the *event* object for which keys exactly have been pressed.

Attaching a filter (condition)

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

```
from prompt_toolkit.filters import Condition

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

@bindings.add('c-t', filter=is_active)
def _(event):
    # ...
    pass
```

The key binding will be ignored when this condition is not satisfied.

ConditionalKeyBindings: Disabling a set of key bindings

Sometimes you want to enable or disable a whole set of key bindings according to a certain condition. This is possible by wrapping it in a *ConditionalKeyBindings* object.

```
from prompt_toolkit.key_binding import ConditionalKeyBindings

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

 bindings = ConditionalKeyBindings(
     key_bindings=my_bindings,
     filter=is_active)
```

If the condition is not satisfied, all the key bindings in *my_bindings* above will be ignored.

Merging key bindings

Sometimes you have different parts of your application generate a collection of key bindings. It is possible to merge them together through the *merge_key_bindings()* function. This is preferred above passing a *KeyBindings* object around and having everyone populate it.

```
from prompt_toolkit.key_binding import merge_key_bindings

bindings = merge_key_bindings([
    bindings1,
    bindings2,
])
```

Eager

Usually not required, but if ever you have to override an existing key binding, the *eager* flag can be useful.

Suppose that there is already an active binding for *ab* and you'd like to add a second binding that only handles *a*. When the user presses only *a*, prompt_toolkit has to wait for the next key press in order to know which handler to call.

By passing the *eager* flag to this second binding, we are actually saying that prompt_toolkit shouldn't wait for longer matches when all the keys in this key binding are matched. So, if *a* has been pressed, this second binding will be called, even if there's an active *ab* binding.

```
@bindings.add('a', 'b')
def binding_1(event):
    ...

@bindings.add('a', eager=True)
def binding_2(event):
    ...
```

This is mainly useful in order to conditionally override another binding.

Timeouts

There are two timeout settings that effect the handling of keys.

- `Application.ttimeoutlen`: Like Vim's *ttimeoutlen* option. When to flush the input (For flushing escape keys.) This is important on terminals that use vt100 input. We can't distinguish the escape key from for instance the left-arrow key, if we don't know what follows after "x1b". This little timer will consider "x1b" to be escape if nothing did follow in this time span. This seems to work like the *ttimeoutlen* option in Vim.
- `KeyProcessor.timeoutlen`: like Vim's *timeoutlen* option. This can be *None* or a float. For instance, suppose that we have a key binding AB and a second key binding A. If the uses presses A and then waits, we don't handle this binding yet (unless it was marked 'eager'), because we don't know what will follow. This timeout is the maximum amount of time that we wait until we call the handlers anyway. Pass *None* to disable this timeout.

Recording macros

Both Emacs and Vi mode allow macro recording. By default, all key presses are recorded during a macro, but it is possible to exclude certain keys by setting the *record_in_macro* parameter to *False*:

```
@bindings.add('c-t', record_in_macro=False)
def _(event):
    # ...
    pass
```

Creating new Vi text objects and operators

We tried very hard to ship prompt_toolkit with as many as possible Vi text objects and operators, so that text editing feels as natural as possible to Vi users.

If you wish to create a new text object or key binding, that is actually possible. Check the *custom-vi-operator-and-text-object.py* example for more information.

Processing *.inputrc*

GNU readline can be configured using an *.inputrc* configuration file. This can could key bindings as well as certain settings. Right now, prompt_toolkit doesn't support *.inputrc* yet, but it should be possible in the future.

3.10.2 More about styling

This page will attempt to explain in more detail how to use styling in prompt_toolkit.

To some extent, it is very similar to how Pygments styling works.

Style strings

Many user interface controls, like *Window* accept a `style` argument which can be used to pass the formatting as a string. For instance, we can select a foreground color:

- `"fg:ansired"` (ANSI color palette)
- `"fg:ansiblue"` (ANSI color palette)
- `"fg:#ffaa33"` (hexadecimal notation)
- `"fg:darkred"` (named color)

Or a background color:

- `"bg:ansired"` (ANSI color palette)
- `"bg:#ffaa33"` (hexadecimal notation)

Or we can add one of the following flags:

- `"bold"`
- `"italic"`
- `"underline"`
- `"blink"`
- `"reverse"` (reverse foreground and background on the terminal.)
- `"hidden"`

Or their negative variants:

- `"nobold"`
- `"noitalic"`
- `"nounderline"`
- `"noblink"`
- `"noreverse"`
- `"nohidden"`

All of these formatting options can be combined as well:

- `"fg:ansiyellow bg:black bold underline"`

The style string can be given to any user control directly, or to a *`Container`* object from where it will propagate to all its children. A style defined by a parent user control can be overridden by any of its children. The parent can for instance say `style="bold underline"` where a child overrides this style partly by specifying `style="nobold bg:ansired"`.

Note: These styles are actually compatible with Pygments styles, with additional support for *reverse* and *blink*. Further, we ignore flags like *roman*, *sans*, *mono* and *border*.

The following ANSI colors are available (both for foreground and background):

```
# Low intensity, dark.  (One or two components 0x80, the other 0x00.)
ansiblack, ansired, ansigreen, ansiyellow, ansiblue
ansimagenta, 'ansicyan, ansigray

# High intensity, bright.
ansibrightblack, ansibrightred, ansibrightgreen, ansibrightyellow
ansibrightblue, ansibrightmagenta, ansibrightcyan, ansiwhite
```

In order to know which styles are actually used in an application, it is possible to call `get_used_style_strings()`, when the application is done.

Class names

Like we do for web design, it is not a good habit to specify all styling inline. Instead, we can attach class names to UI controls and have a style sheet that refers to these class names. The `Style` can be passed as an argument to the *`Application`*.

```
from prompt_toolkit.layout import VSplit, Window
from prompt_toolkit.style import Style

layout = VSplit([
    Window(BufferControl(...), style='class:left'),
    HSplit([
        Window(BufferControl(...), style='class:top'),
        Window(BufferControl(...), style='class:bottom'),
    ], style='class:right')
])

style = Style([
     ('left': 'bg:ansired'),
     ('top': 'fg:#00aaaa'),
     ('bottom': 'underline bold'),
 ])
```

It is possible to add multiple class names to an element. That way we'll combine the styling for these class names. Multiple classes can be passed by using a comma separated list, or by using the `class:` prefix twice.

```
Window(BufferControl(...), style='class:left,bottom'),
Window(BufferControl(...), style='class:left class:bottom'),
```

It is possible to combine class names and inline styling. The order in which the class names and inline styling is specified determines the order of priority. In the following example for instance, we'll take first the style of the "header" class, and then override that with a red background color.

```
Window(BufferControl(...), style='class:header bg:red'),
```

Dot notation in class names

The dot operator has a special meaning in a class name. If we write: `style="class:a.b.c"`, then this will actually expand to the following: `style="class:a class:a.b class:a.b.c"`.

This is mainly added for Pygments lexers, which specify "Tokens" like this, but it's useful in other situations as well.

Multiple classes in a style sheet

A style sheet can be more complex as well. We can for instance specify two class names. The following will underline the left part within the header, or whatever has both the class "left" and the class "header" (the order doesn't matter).

```
style = Style([
     ('header left': 'underline'),
 ])
```

If you have a dotted class, then it's required to specify the whole path in the style sheet (just typing `c` or `b.c` doesn't work if the class is `a.b.c`):

```
style = Style([
     ('a.b.c': 'underline'),
 ])
```

It is possible to combine this:

```
style = Style([
     ('header body left.text': 'underline'),
 ])
```

Evaluation order of rules in a style sheet

The style is determined as follows:

- First, we concatenate all the style strings from the root control through all the parents to the child in one big string. (Things at the right take precedence anyway.)

  E.g: `class:body bg:#aaaaaa #000000 class:header.focused class:left.text.highlighted underline`

- Then we go through this style from left to right, starting from the default style. Inline styling is applied directly.

  If we come across a class name, then we generate all combinations of the class names that we collected so far (this one and all class names to the left), and for each combination which includes the new class name, we look for matching rules in our style sheet. All these rules are then applied (later rules have higher priority).

  If we find a dotted class name, this will be expanded in the individual names (like `class:left class:left.text class:left.text.highlighted`), and all these are applied like any class names.

- Then this final style is applied to this user interface element.

Using a dictionary as a style sheet

The order of the rules in a style sheet is meaningful, so typically, we use a list of tuples to specify the style. But is also possible to use a dictionary as a style sheet. This makes sense for Python 3.6, where dictionaries remember their ordering. An `OrderedDict` works as well.

```
from prompt_toolkit.style import Style

style = Style.from_dict({
     'header body left.text': 'underline',
})
```

Loading a style from Pygments

Pygments has a slightly different notation for specifying styles, because it maps styling to Pygments "Tokens". A Pygments style can however be loaded and used as follows:

```
from prompt_toolkit.styles.from_pygments import style_from_pygments_cls
from pygments.styles import get_style_by_name

style = style_from_pygments_cls(get_style_by_name('monokai'))
```

Merging styles together

Multiple `Style` objects can be merged together as follows:

```
from prompt_toolkit.styles import merge_styles

style = merge_styles([
    style1,
    style2,
    style3
])
```

Color depths

There are four different levels of color depths available:

1 bit	Black and white	`ColorDepth.DEPTH_1_BIT`	`ColorDepth.MONOCHROME`
4 bit	ANSI colors	`ColorDepth.DEPTH_4_BIT`	`ColorDepth.ANSI_COLORS_ONLY`
8 bit	256 colors	`ColorDepth.DEPTH_8_BIT`	`ColorDepth.DEFAULT`
24 bit	True colors	`ColorDepth.DEPTH_24_BIT`	`ColorDepth.TRUE_COLOR`

By default, 256 colors are used, because this is what most terminals support these days. If the `TERM` enviroment variable is set to `linux` or `eterm-color`, then only ANSI colors are used, because of these terminals. 24 bit true color output needs to be enabled explicitely. When 4 bit color output is chosen, all colors will be mapped to the closest ANSI color.

Setting the default color depth for any prompt_toolkit application can be done by setting the `PROMPT_TOOLKIT_COLOR_DEPTH` environment variable. You could for instance copy the following into your *.bashrc* file.

```
# export PROMPT_TOOLKIT_COLOR_DEPTH=DEPTH_1_BIT
export PROMPT_TOOLKIT_COLOR_DEPTH=DEPTH_4_BIT
# export PROMPT_TOOLKIT_COLOR_DEPTH=DEPTH_8_BIT
# export PROMPT_TOOLKIT_COLOR_DEPTH=DEPTH_24_BIT
```

An application can also decide to set the color depth manually by passing a *`ColorDepth`* value to the *`Application`* object:

```
from prompt_toolkit.output.color_depth import ColorDepth

app = Application(
    color_depth=ColorDepth.ANSI_COLORS_ONLY,
    # ...
)
```

3.10.3 Filters

Many places in *prompt_toolkit* require a boolean value that can change over time. For instance:

- to specify whether a part of the layout needs to be visible or not;
- or to decide whether a certain key binding needs to be active or not;
- or the `wrap_lines` option of *`BufferControl`*;
- etcetera.

These booleans 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.)

The (abstract) base class is *Filter*, which wraps an expression that takes no input and evaluates to a boolean. Getting the state of a filter is done by simply calling it.

An example

The most obvious way to create such a *Filter* instance is by creating a *Condition* instance from a function. For instance, the following condition will evaluate to `True` when the user is searching:

```
from prompt_toolkit.application.current import get_app
from prompt_toolkit.filters import Condition

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

A different way of writing this, is by using the decorator syntax:

```
from prompt_toolkit.application.current import get_app
from prompt_toolkit.filters import Condition

@Condition
def is_searching():
    return get_app().is_searching
```

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

```
from prompt_toolkit.key_binding import KeyBindings

kb = KeyBindings()

@kb.add('c-t', filter=is_searching)
def _(event):
    # Do, something, but only when searching.
    pass
```

If we want to know the boolean value of this filter, we have to call it like a function:

```
print(is_searching())
```

Built-in filters

There are many built-in filters, ready to use. All of them have a lowercase name, because they represent the wrapped function underneath, and can be called as a function.

- `has_arg`
- `has_completions`
- *has_focus*
- `buffer_has_focus`

- has_selection
- has_validation_error
- is_aborting
- is_done
- is_read_only
- is_multiline
- renderer_height_is_known
- *in_editing_mode*
- in_paste_mode
- vi_mode
- vi_navigation_mode
- vi_insert_mode
- vi_insert_multiple_mode
- vi_replace_mode
- vi_selection_mode
- vi_waiting_for_text_object_mode
- vi_digraph_mode
- emacs_mode
- emacs_insert_mode
- emacs_selection_mode
- is_searching
- control_is_searchable
- vi_search_direction_reversed

Combining filters

Filters can be chained with the `&` (AND) and `|` (OR) operators and negated with the `~` (negation) operator.

Some examples:

```
from prompt_toolkit.key_binding import KeyBindings
from prompt_toolkit.filters import has_selection, has_selection

kb = KeyBindings()

@kb.add('c-t', filter=~is_searching)
def _(event):
    " Do something, but not while searching. "
    pass

@kb.add('c-t', filter=has_search | has_selection)
def _(event):
    " Do something, but only when searching or when there is a selection. "
    pass
```

to_filter

Finally, in many situations you want your code to expose an API that is able to deal with both booleans as well as filters. For instance, when for most users a boolean works fine because they don't need to change the value over time, while some advanced users want to be able this value to a certain setting or event that does changes over time.

In order to handle both use cases, there is a utility called *to_filter()*.

This is a function that takes either a boolean or an actual *Filter* instance, and always returns a *Filter*.

```
from prompt_toolkit.filters.utils import to_filter

# In each of the following three examples, 'f' will be a `Filter`
# instance.
f = to_filter(True)
f = to_filter(False)
f = to_filter(Condition(lambda: True))
f = to_filter(has_search | has_selection)
```

3.10.4 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
from prompt_toolkit.buffer import Buffer

b = Buffer(name=DEFAULT_BUFFER)
Window(content=BufferControl(buffer=b))
```

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:

   (b) 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 "formatted text list" for that line (that's a list of `(style_string, text)` tuples).

   (c) This 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, highlight the selection or search string, etc...)

   (d) 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*.

3.10.5 Running on top of the *asyncio* event loop

Prompt_toolkit has a built-in event loop of its own. However, in modern applications, you probably want to use asyncio for everything. With just one line of code, it is possible to run prompt_toolkit on top of asyncio:

```
from prompt_toolkit.eventloop import use_asyncio_event_loop

use_asyncio_event_loop()
```

This will create an adaptor between the asyncio event loop and prompt_toolkit, and register it as the underlying event loop for the prompt_toolkit application.

When doing this, remember that prompt_toolkit still has its own implementation of futures (and coroutines). A prompt_toolkit *Future* needs to be converted to an asyncio *Future* for use in an asyncio context, like asyncio's `run_until_complete`. The cleanest way is to call *to_asyncio_future()*.

So,the typical boilerplace for an asyncio application looks like this:

```
from prompt_toolkit.eventloop import use_asyncio_event_loop
from prompt_toolkit.application import Application

# Tell prompt_toolkit to use asyncio for the event loop.
use_asyncio_event_loop()

# Define application.
application = Application(
    ...
)

# Run the application, and wait for it to finish.
asyncio.get_event_loop().run_until_complete(
    application.run_async().to_asyncio_future())
```

Warning: If you want to use coroutines in your application, then using asyncio is the preferred way. It's better to avoid the built-in coroutines, because they make debugging the application much more difficult. Unless of course Python 2 support is still required.

At some point, when we drop Python 2 support, prompt_toolkit will probably use asyncio natively.

3.10.6 Input hooks

Input hooks are a tool for inserting an external event loop into the prompt_toolkit event loop, so that the other loop can run as long as prompt_toolkit is idle. This is used in applications like IPython, so that GUI toolkits can display their windows while we wait at the prompt for user input.

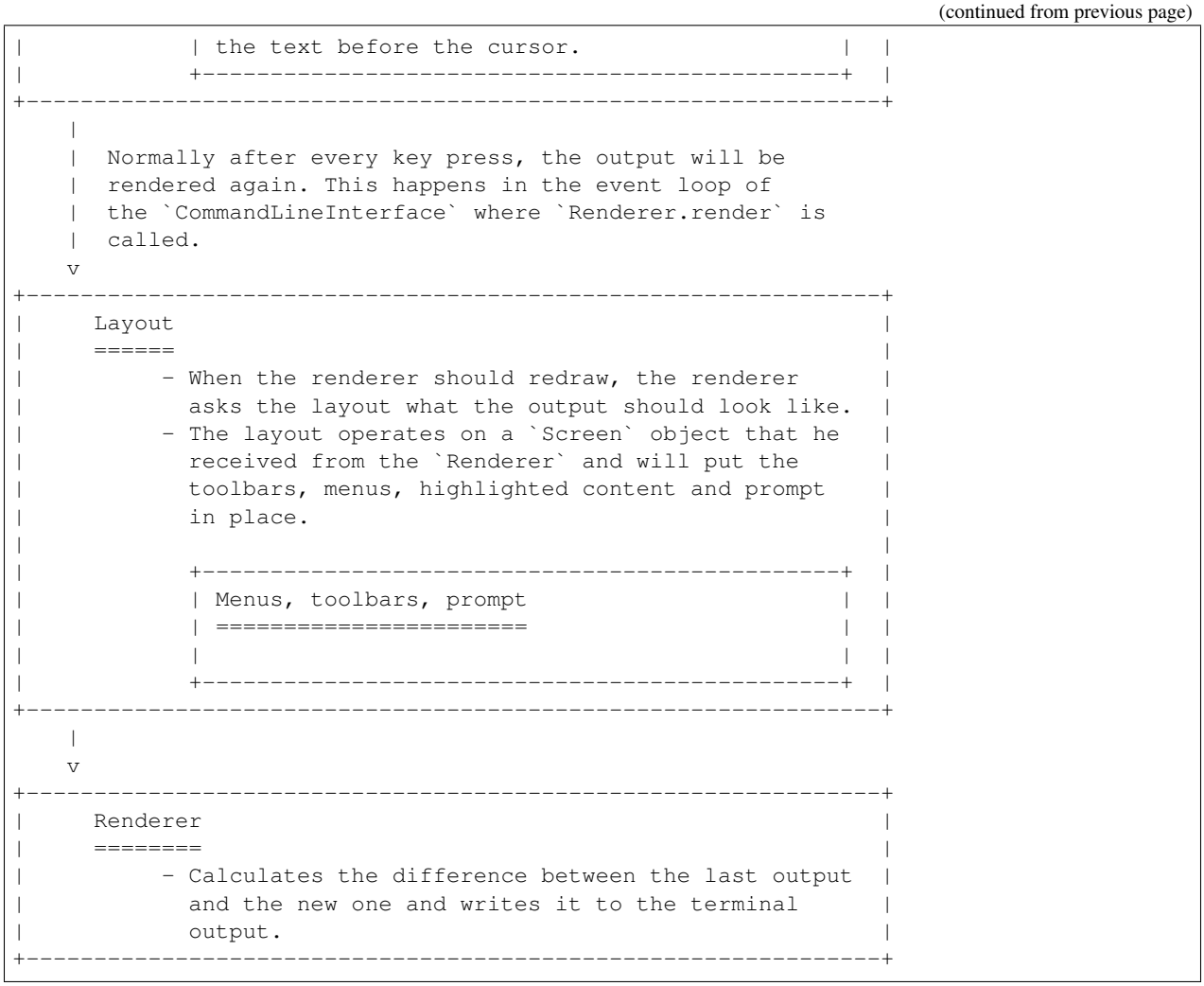
3.10.7 Architecture

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`.    |
+-------------------------------------------------------------+
           |
           v
+-------------------------------------------------------------+
|     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.
+-------------------------------------------------------------+
           |
           v
+-------------------------------------------------------------+
|     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.
    |
    v
+-------------------------------------------------------------+
|     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 |  |
```

(continues on next page)

(continued from previous page)



3.11 Reference

3.11.1 Application

class prompt_toolkit.application.**Application**(*layout=None*, *style=None*, *include_default_pygments_style=True*, *key_bindings=None*, *clipboard=None*, *full_screen=False*, *color_depth=None*, *mouse_support=False*, *enable_page_navigation_bindings=None*, *paste_mode=False*, *editing_mode=u'EMACS'*, *erase_when_done=False*, *reverse_vi_search_direction=False*, *min_redraw_interval=None*, *max_render_postpone_time=0*, *on_reset=None*, *on_invalidate=None*, *before_render=None*, *after_render=None*, *input=None*, *output=None*)

The main Application class! This glues everything together.

Parameters

- **layout** – A *Layout* instance.
- **key_bindings** – *KeyBindingsBase* 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.
- **full_screen** – When True, run the application on the alternate screen buffer.
- **color_depth** – Any *ColorDepth* value, a callable that returns a *ColorDepth* or *None* for default.
- **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.
- **min_redraw_interval** – Number of seconds to wait between redraws. Use this for applications where *invalidate* is called a lot. This could cause a lot of terminal output, which some terminals are not able to process.

  None means that every *invalidate* will be scheduled right away (which is usually fine).

  When one *invalidate* is called, but a scheduled redraw of a previous *invalidate* call has not been executed yet, nothing will happen in any case.
- **max_render_postpone_time** – When there is high CPU (a lot of other scheduled calls), postpone the rendering max x seconds. '0' means: don't postpone. '.5' means: try to draw at least twice a second.

Filters:

Parameters

- **mouse_support** – (*Filter* or boolean). When True, enable mouse support.

- **paste_mode** – *Filter* or boolean.
- **editing_mode** – EditingMode.
- **enable_page_navigation_bindings** – When *True*, enable the page navigation key bindings. These include both Emacs and Vi bindings like page-up, page-down and so on to scroll through pages. Mostly useful for creating an editor or other full screen applications. Probably, you don't want this for the implementation of a REPL. By default, this is enabled if *full_screen* is set.

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

Parameters

- **on_reset** – Called during reset.
- **on_invalidate** – Called when the UI has been invalidated.
- **before_render** – Called right before rendering.
- **after_render** – Called right after rendering.

I/O:

Parameters

- **input** – *Input* instance.
- **output** – *Output* instance. (Probably Vt100_Output or Win32Output.)

Usage:

app = Application(...) app.run()

color_depth
Active *ColorDepth*.

cpr_not_supported_callback()
Called when we don't receive the cursor position response in time.

current_buffer
The currently focused *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_search_state
Return the current SearchState. (The one for the focused *BufferControl*.)

exit(*result=None*, *exception=None*, *style=u''*)
Exit application.

Parameters

- **result** – Set this result for the application.
- **exception** – Set this exception as the result for an application. For a prompt, this is often *EOFError* or *KeyboardInterrupt*.
- **style** – Apply this style on the whole content when quitting, often this is 'class:exiting' for a prompt. (Used when *erase_when_done* is not set.)

get_used_style_strings()
Return a list of used style strings. This is helpful for debugging, and for writing a new *Style*.

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

invalidated
True when a redraw operation has been scheduled.

is_running
True when the application is currently active/running.

print_text(*text*, *style=None*)
Print a list of (style_str, 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

- **text** – List of `(style_str, text)` tuples.
- **style** – Style class to use. Defaults to the active style in the CLI.

reset()
Reset everything, for reading the next input.

run(*pre_run=None*, *set_exception_handler=True*, *inputhook=None*)
A blocking 'run' call that waits until the UI is finished.

Parameters

- **set_exception_handler** – When set, in case of an exception, go out of the alternate screen and hide the application, display the exception, and wait for the user to press ENTER.
- **inputhook** – None or a callable that takes an *InputHookContext*.

run_async(*pre_run=None*)
Run asynchronous. Return a prompt_toolkit *Future* object.

If you wish to run on top of asyncio, remember that a prompt_toolkit *Future* needs to be converted to an asyncio *Future*. The cleanest way is to call *to_asyncio_future()*. Also make sure to tell prompt_toolkit to use the asyncio event loop.

```
from prompt_toolkit.eventloop import use_asyncio_event_loop
from asyncio import get_event_loop

use_asyncio_event_loop()
get_event_loop().run_until_complete(
    application.run_async().to_asyncio_future())
```

run_system_command(*command*, *wait_for_enter=True*, *display_before_text=u''*, *wait_text=u'Press ENTER to continue...'*)
Run system command (While hiding the prompt. When finished, all the output will scroll above the prompt.)

Parameters

- **command** – Shell command to be executed.
- **wait_for_enter** – FWait for the user to press enter, when the command is finished.
- **display_before_text** – If given, text to be displayed before the command executes.

Returns A *Future* object.

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.)

prompt_toolkit.application.**get_app**(*raise_exception=False*, *return_none=False*)
Get the current active (running) Application. An *Application* is active during the *Application.run_async()* call.

We assume that there can only be one *Application* active at the same time. There is only one terminal window, with only one stdin and stdout. This makes the code significantly easier than passing around the *Application* everywhere.

If no *Application* is running, then return by default a *DummyApplication*. For practical reasons, we prefer to not raise an exception. This way, we don't have to check all over the place whether an actual *Application* was returned.

(For applications like pymux where we can have more than one *Application*, we'll use a work-around to handle that.)

Parameters

- **raise_exception** – When *True*, raise *NoRunningApplicationError* instead of returning a *DummyApplication* if no application is running.
- **return_none** – When *True*, return *None* instead of returning a *DummyApplication* if no application is running.

prompt_toolkit.application.**set_app**(**args*, ***kwds*)
Context manager that sets the given *Application* active.

(Usually, not needed to call outside of prompt_toolkit.)

exception prompt_toolkit.application.**NoRunningApplicationError**
There is no active application right now.

class prompt_toolkit.application.**DummyApplication**
When no *Application* is running, *get_app()* will run an instance of this *DummyApplication* instead.

prompt_toolkit.application.**run_in_terminal**(*func*, *render_cli_done=False*, *in_executor=False*)
Run function on the terminal above the current application or 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.
- **in_executor** – When True, run in executor. (Use this for long blocking functions, when you don't want to block the event loop.)

Returns A *Future*.

prompt_toolkit.application.**run_coroutine_in_terminal**(*async_func*, *render_cli_done=False*)
Suspend the current application and run this coroutine instead. *async_func* can be a coroutine or a function that returns a Future.

Parameters **async_func** – A function that returns either a Future or coroutine when called.

Returns A *Future*.

3.11.2 Formatted text

Many places in prompt_toolkit can take either plain text, or formatted text. For instance the `prompt()` function takes either plain text or formatted text for the prompt. The `FormattedTextControl` can also take either plain text or formatted text.

In any case, there is an input that can either be just plain text (a string), an `HTML` object, an `ANSI` object or a sequence of *(style_string, text)* tuples. The `to_formatted_text()` conversion function takes any of these and turns all of them into such a tuple sequence.

prompt_toolkit.formatted_text.**to_formatted_text**(*value*, *style=u''*, *auto_convert=False*)
Convert the given value (which can be formatted text) into a list of text fragments. (Which is the canonical form of formatted text.) The outcome is supposed to be a list of (style, text) tuples.

It can take an *HTML* object, a plain text string, or anything that implements *__pt_formatted_text__*.

Parameters

- **style** – An additional style string which is applied to all text fragments.
- **auto_convert** – If *True*, also accept other types, and convert them to a string first.

prompt_toolkit.formatted_text.**is_formatted_text**(*value*)
Check whether the input is valid formatted text (for use in assert statements). In case of a callable, it doesn't check the return type.

class prompt_toolkit.formatted_text.**Template**(*text*)
Template for string interpolation with formatted text.

Example:

```
Template(' ... {} ... ').format(HTML(...))
```

Parameters **text** – Plain text.

prompt_toolkit.formatted_text.**merge_formatted_text**(*items*)
Merge (Concatenate) several pieces of formatted text together.

class prompt_toolkit.formatted_text.**FormattedText**(*data*)
A list of `(style, text)` tuples.

class prompt_toolkit.formatted_text.**HTML**(*value*)
HTML formatted text. Take something HTML-like, for use as a formatted string.

```
# Turn something into red.
HTML('<style fg="ansired" bg="#00ff44">...</style>')

# Italic, bold and underline.
HTML('<i>...</i>')
HTML('<b>...</b>')
HTML('<u>...</u>')
```

All HTML elements become available as a "class" in the style sheet. E.g. `<username>...</username>` can be styled, by setting a style for `username`.

format(**args*, ***kwargs*)
Like *str.format*, but make sure that the arguments are properly escaped.

class prompt_toolkit.formatted_text.**ANSI**(*value*)
ANSI formatted text. Take something ANSI escaped text, for use as a formatted string. E.g.

```
ANSI('\x1b[31mhello \x1b[32mworld')
```

Characters between `\001` and `\002` are supposed to have a zero width when printed, but these are literally sent to the terminal output. This can be used for instance, for inserting Final Term prompt commands. They will be translated into a prompt_toolkit '[ZeroWidthEscape]' fragment.

class prompt_toolkit.formatted_text.**PygmentsTokens**(*token_list*)
Turn a pygments token list into a list of prompt_toolkit text fragments (`(style_str, text)` tuples).

prompt_toolkit.formatted_text.**fragment_list_len**(*fragments*)
Return the amount of characters in this text fragment list.

Parameters **fragments** – List of `(style_str, text)` or `(style_str, text, mouse_handler)` tuples.

prompt_toolkit.formatted_text.**fragment_list_width**(*fragments*)
Return the character width of this text fragment list. (Take double width characters into account.)

Parameters **fragments** – List of `(style_str, text)` or `(style_str, text, mouse_handler)` tuples.

prompt_toolkit.formatted_text.**fragment_list_to_text**(*fragments*)
Concatenate all the text parts again.

Parameters **fragments** – List of `(style_str, text)` or `(style_str, text, mouse_handler)` tuples.

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

Parameters **fragments** – List of (style_str, text) or (style_str, text, mouse_handler) tuples.

3.11.3 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.**Buffer**(*completer=None*, *auto_suggest=None*, *history=None*, *validator=None*, *tempfile_suffix=u''*, *name=u''*, *complete_while_typing=False*, *validate_while_typing=False*, *enable_history_search=False*, *document=None*, *accept_handler=None*, *read_only=False*, *multiline=True*, *on_text_changed=None*, *on_text_insert=None*, *on_cursor_position_changed=None*, *on_completions_changed=None*, *on_suggestion_set=None*)
The core data structure that holds the text and cursor position of the current input line and implements all text manipulations on top of it. It also implements the history, undo stack and the completion state.

Parameters

- **eventloop** – *EventLoop* instance.
- **completer** – *Completer* instance.
- **history** – *History* instance.

- **tempfile_suffix** – The tempfile suffix (extension) to be used for the "open in editor" function. For a Python REPL, this would be ".py", so that the editor knows the syntax highlighting to use. This can also be a callable that returns a string.
- **name** – Name for this buffer. E.g. DEFAULT_BUFFER. This is mostly useful for key bindings where we sometimes prefer to refer to a buffer by their name instead of by reference.
- **accept_handler** – Callback that takes this buffer as input. Called when the buffer input is accepted. (Usually when the user presses *enter*.)

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.)
- **on_completions_changed** – When the completions were changed. (Callable on None.)
- **on_suggestion_set** – When an auto-suggestion text has been set. (Callable on None.)

Filters:

Parameters

- **complete_while_typing** – *Filter* or *bool*. Decide whether or not to do asynchronous autocompleting while typing.
- **validate_while_typing** – *Filter* or *bool*. Decide whether or not to do asynchronous validation while typing.
- **enable_history_search** – *Filter* or *bool* to indicate when up-arrow partial string matching is enabled. It is advised 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** – *Filter*. When True, changes will not be allowed.
- **multiline** – *Filter* or *bool*. When not set, pressing *Enter* will call the *accept_handler*. Otherwise, pressing *Esc-Enter* is required.

append_to_history()
Append the current input to the history.

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.

Notice that this doesn't store the copied data on the clipboard yet. You can store it like this:

```
data = buffer.copy_selection()
get_app().clipboard.set_data(data)
```

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.

is_returnable
True when there is something handling accept.

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(*validate_and_handle=False*)
Open code in editor.

This returns a future, and runs in a thread executor.

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

reset(*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_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.

> **Warning:** When this buffer is read-only and *bypass_readonly* was not passed, the *EditReadOnlyBuffer* exception will be caught by the *KeyProcessor* and is silently suppressed. This is important to keep in mind when writing key bindings, because it won't do what you expect, and there won't be a stack trace. Use try/finally around this function if you need some cleanup code.

start_completion(*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.)

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(*set_cursor=False*)
Returns *True* if valid.

Parameters **set_cursor** – Set the cursor position, if an error was found.

validate_and_handle()
Validate buffer and handle the accept action.

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.)

3.11.4 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*

3.11.5 Clipboard

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

Text on the clipboard.

Parameters

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

class prompt_toolkit.clipboard.**DummyClipboard**

Clipboard implementation that doesn't remember anything.

class prompt_toolkit.clipboard.**DynamicClipboard**(*get_clipboard*)

Clipboard class that can dynamically returns any Clipboard.

Parameters **get_clipboard** – Callable that returns a *Clipboard* instance.

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

Default clipboard implementation. Just keep the data in memory.

This implements a kill-ring, for Emacs mode.

3.11.6 Auto completion

class prompt_toolkit.completion.**Completion**(*text*, *start_position=0*, *display=None*, *display_meta=None*, *style=u''*, *selected_style=u''*)

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. This can also be a callable that returns a string.
- **style** – Style string.
- **selected_style** – Style string, used for a selected completion. This can override the *style* parameter.

display_meta
Return meta-text. (This is lazy when using a callable).

new_completion_from_position(*position*)
(Only for internal use!) Get a new completion by splitting this one. Used by *Application* 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*)
This should be a generator that yields *Completion* instances.

If the generation of completions is something expensive (that takes a lot of time), consider wrapping this *Completer* class in a *ThreadedCompleter*. In that case, the completer algorithm runs in a background thread and completions will be displayed as soon as they arrive.

Parameters

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

get_completions_async(*document*, *complete_event*)
Asynchronous generator for completions. (Probably, you won't have to override this.)

This should return an iterable that can yield both *Completion* and *Future* objects. The *Completion* objects have to be wrapped in a *AsyncGeneratorItem* object.

If we drop Python 2 support in the future, this could become a true asynchronous generator.

class prompt_toolkit.completion.**ThreadedCompleter**(*completer=None*)
Wrapper that runs the *get_completions* generator in a thread.

(Use this to prevent the user interface from becoming unresponsive if the generation of completions takes too much time.)

The completions will be displayed as soon as they are produced. The user can already select a completion, even if not all completions are displayed.

get_completions_async(*document*, *complete_event*)
Asynchronous generator of completions. This yields both Future and Completion objects.

class prompt_toolkit.completion.**DummyCompleter**
A completer that doesn't return any completion.

class prompt_toolkit.completion.**DynamicCompleter**(*get_completer*)
Completer class that can dynamically returns any Completer.

Parameters **get_completer** – Callable that returns a *Completer* 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 explicitly 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.**merge_completers**(*completers*)

Combine several completers into one.

prompt_toolkit.completion.**get_common_complete_suffix**(*document*, *completions*)

Return the common prefix for all completions.

class prompt_toolkit.completion.**PathCompleter**(*only_directories=False*, *get_paths=None*, *file_filter=None*, *min_input_len=0*, *expanduser=False*)

Complete for Path variables.

Parameters

- **get_paths** – Callable which returns a list of directories to look into when the user enters a relative path.
- **file_filter** – Callable which takes a filename and returns whether this file should show up in the completion. `None` when no filtering has to be done.
- **min_input_len** – Don't do autocompletion when the input string is shorter.

class prompt_toolkit.completion.**ExecutableCompleter**

Complete only executable files in the current path.

class prompt_toolkit.completion.**WordCompleter**(*words*, *ignore_case=False*, *meta_dict=None*, *WORD=False*, *sentence=False*, *match_middle=False*)

Simple autocompletion on a list of words.

Parameters

- **words** – List of words or callable that returns a list of words.
- **ignore_case** – If True, case-insensitive completion.
- **meta_dict** – Optional dict mapping words to their meta-information.
- **WORD** – When True, use WORD characters.
- **sentence** – When True, don't complete by comparing the word before the cursor, but by comparing all the text before the cursor. In this case, the list of words is just a list of strings, where each string can contain spaces. (Can not be used together with the WORD option.)
- **match_middle** – When True, match not only the start, but also in the middle of the word.

3.11.7 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 occurrence.

find_all(*sub*, *ignore_case=False*)

Find all occurrences 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 occurrence.

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.

The returned upper boundary is not included in the selection, so *(0, 0)* is an empty selection. *(0, 1)*, is a one character selection.

Returns None if the selection doesn't cover this line at all.

selection_ranges()
Return a list of *(from, to)* tuples for the selection or none if nothing was selected. The upper boundary is not included.

This will yield several (from, to) tuples in case of a BLOCK selection. This will return zero ranges, like (8,8) for empty lines in 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.

3.11.8 Enums

3.11.9 History

Implementations for the history of a *Buffer*.

NOTE: Notice that there is no *DynamicHistory*. This doesn't work well, because the *Buffer* needs to be able to attach an event handler to the event when a history entry is loaded. This loading can be done asynchronously and making the history swappable would probably break this.

class prompt_toolkit.history.**History**
Base `History` class.

This also includes abstract methods for loading/storing history.

append_string(*string*)
Add string to the history.

get_item_loaded_event()
Event which is triggered when a new item is loaded.

get_strings()
Get the strings from the history that are loaded so far.

load_history_strings()
This should be a generator that yields *str* instances.

It should yield the most recent items first, because they are the most important. (The history can already be used, even when it's only partially loaded.)

load_history_strings_async()
Asynchronous generator for history strings. (Probably, you won't have to override this.)

This should return an iterable that can yield both *str* and *Future* objects. The *str* objects have to be wrapped in a *AsyncGeneratorItem* object.

If we drop Python 2 support in the future, this could become a true asynchronous generator.

start_loading()
Start loading the history.

store_string(*string*)
Store the string in persistent storage.

class prompt_toolkit.history.**ThreadedHistory**(*history=None*)
Wrapper that runs the *load_history_strings* generator in a thread.

Use this to increase the start-up time of prompt_toolkit applications. History entries are available as soon as they are loaded. We don't have to wait for everything to be loaded.

load_history_strings_async()
Asynchronous generator of completions. This yields both Future and Completion objects.

class prompt_toolkit.history.**DummyHistory**
History object that doesn't remember anything.

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

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

3.11.10 Keys

class prompt_toolkit.keys.**Keys**
List of keys for use in key bindings.

3.11.11 Style

Styling for prompt_toolkit applications.

class prompt_toolkit.styles.**Attrs**(*color*, *bgcolor*, *bold*, *underline*, *italic*, *blink*, *reverse*, *hidden*)

bgcolor
Alias for field number 1

blink
Alias for field number 5

bold
Alias for field number 2

color
Alias for field number 0

hidden
Alias for field number 7

italic
Alias for field number 4

reverse
Alias for field number 6

underline
Alias for field number 3

class prompt_toolkit.styles.**BaseStyle**
Abstract base class for prompt_toolkit styles.

get_attrs_for_style_str(*style_str*, *default=Attrs(color=u"*, *bgcolor=u"*, *bold=False*, *underline=False*, *italic=False*, *blink=False*, *reverse=False*, *hidden=False)*)
Return *Attrs* for the given style string.

Parameters

- **style_str** – The style string. This can contain inline styling as well as classnames (e.g. "class:title").
- **default** – *Attrs* to be used if no styling was defined.

invalidation_hash()
Invalidation hash for the style. When this changes over time, the renderer knows that something in the style changed, and that everything has to be redrawn.

style_rules
The list of style rules, used to create this style. (Required for *DynamicStyle* and *_MergedStyle* to work.)

class prompt_toolkit.styles.**DummyStyle**
A style that doesn't style anything.

class prompt_toolkit.styles.**DynamicStyle**(*get_style*)
Style class that can dynamically returns an other Style.

Parameters **get_style** – Callable that returns a *Style* instance.

class prompt_toolkit.styles.**Style**(*style_rules*)
Create a Style instance from a list of style rules.

The *style_rules* is supposed to be a list of ('classnames', 'style') tuples. The classnames are a whitespace separated string of class names and the style string is just like a Pygments style definition, but with a few additions: it supports 'reverse' and 'blink'.

Later rules always override previous rules.

Usage:

```
Style([
    ('title', '#ff0000 bold underline'),
    ('something-else', 'reverse'),
    ('class1 class2', 'reverse'),
])
```

The `from_dict` classmethod is similar, but takes a dictionary as input.

classmethod from_dict(*style_dict*, *priority=u'MOST_PRECISE'*)

Parameters

- **style_dict** – Style dictionary.
- **priority** – *Priority* value.

get_attrs_for_style_str(*style_str*, *default=Attrs(color=u''*, *bgcolor=u''*, *bold=False*, *underline=False*, *italic=False*, *blink=False*, *reverse=False*, *hidden=False)*)

Get *Attrs* for the given style string.

class prompt_toolkit.styles.**Priority**

The priority of the rules, when a style is created from a dictionary.

In a *Style*, rules that are defined later will always override previous defined rules, however in a dictionary, the key order was arbitrary before Python 3.6. This means that the style could change at random between rules.

We have two options:

- ***DICT_KEY_ORDER*: This means, iterate through the dictionary, and take** the key/value pairs in order as they come. This is a good option if you have Python >3.6. Rules at the end will override rules at the beginning.
- *MOST_PRECISE*: keys that are defined with most precision will get higher priority. (More precise means: more elements.)

prompt_toolkit.styles.**merge_styles**(*styles*)

Merge multiple *Style* objects.

prompt_toolkit.styles.**style_from_pygments_cls**(*pygments_style_cls*)

Shortcut to create a *Style* instance from a Pygments style class and a style dictionary.

Example:

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

Parameters **pygments_style_cls** – Pygments style class to start from.

prompt_toolkit.styles.**style_from_pygments_dict**(*pygments_dict*)

Create a *Style* instance from a Pygments style dictionary. (One that maps Token objects to style strings.)

prompt_toolkit.styles.**pygments_token_to_classname**(*token*)

Turn e.g. *Token.Name.Exception* into *'pygments.name.exception'*.

(Our Pygments lexer will also turn the tokens that pygments produces in a prompt_toolkit list of fragments that match these styling rules.)

3.11.12 Shortcuts

prompt_toolkit.shortcuts.**prompt**(*a, **kw)

Display the prompt. All the arguments are a subset of the *PromptSession* class itself.

This will raise KeyboardInterrupt when control-c has been pressed (for abort) and EOFError when control-d has been pressed (for exit).

Parameters

- **async** – When *True* return a *Future* instead of waiting for the prompt to finish.
- **accept_default** – When *True*, automatically accept the default value without allowing the user to edit the input.

class prompt_toolkit.shortcuts.**PromptSession**(*message=u''*, *default=u''*, *multiline=False*, *wrap_lines=True*, *is_password=False*, *vi_mode=False*, *editing_mode=u'EMACS'*, *complete_while_typing=True*, *validate_while_typing=True*, *enable_history_search=False*, *search_ignore_case=False*, *lexer=None*, *enable_system_prompt=False*, *enable_suspend=False*, *enable_open_in_editor=False*, *validator=None*, *completer=None*, *complete_in_thread=False*, *reserve_space_for_menu=8*, *complete_style=None*, *auto_suggest=None*, *style=None*, *color_depth=None*, *include_default_pygments_style=True*, *history=None*, *clipboard=None*, *prompt_continuation=None*, *rprompt=None*, *bottom_toolbar=None*, *mouse_support=False*, *input_processors=None*, *key_bindings=None*, *erase_when_done=False*, *tempfile_suffix=u'.txt'*, *inputhook=None*, *refresh_interval=0*, *input=None*, *output=None*)

PromptSession for a prompt application, which can be used as a GNU Readline replacement.

This is a wrapper around a lot of prompt_toolkit functionality and can be a replacement for *raw_input*.

All parameters that expect "formatted text" can take either just plain text (a unicode object), a list of (style_str, text) tuples or an HTML object.

Example usage:

```
s = PromptSession(message='>')
text = s.prompt()
```

Parameters

- **message** – Plain text or formatted text to be shown before the prompt. This can also be a callable that returns formatted text.

- **multiline** – *bool* or *Filter*. 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 *Filter*. 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 *Filter*. Enable autocompletion while typing.
- **validate_while_typing** – *bool* or *Filter*. Enable input validation while typing.
- **enable_history_search** – *bool* or *Filter*. Enable up-arrow parting string matching.
- **search_ignore_case** – *Filter*. Search case insensitive.
- **lexer** – *Lexer* to be used for the syntax highlighting.
- **validator** – *Validator* instance for input validation.
- **completer** – *Completer* instance for input completion.
- **complete_in_thread** – *bool* or *Filter*. Run the completer code in a background thread in order to avoid blocking the user interface. For `CompleteStyle.READLINE_LIKE`, this setting has no effect. There we always run the completions in the main thread.
- **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.
- **include_default_pygments_style** – *bool* or *Filter*. Tell whether the default styling for Pygments lexers has to be included. By default, this is true, but it is recommended to be disabled if another Pygments style is passed as the *style* argument, otherwise, two Pygments styles will be merged.
- **enable_system_prompt** – *bool* or *Filter*. Pressing Meta+'!' will show a system prompt.
- **enable_suspend** – *bool* or *Filter*. Enable Control-Z style suspension.
- **enable_open_in_editor** – *bool* or *Filter*. 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*)
- **rprompt** – Text or formatted text to be displayed on the right side. This can also be a callable that returns (formatted) text.
- **bottom_toolbar** – Formatted text or callable which is supposed to return formatted text.
- **prompt_continuation** – Text that needs to be displayed for a multiline prompt continuation. This can either be formatted text or a callable that takes a *width*, *line_number* and *is_soft_wrap* as input and returns formatted text.

- **complete_style** – CompleteStyle.COLUMN, CompleteStyle.MULTI_COLUMN or CompleteStyle.READLINE_LIKE.
- **mouse_support** – *bool* or *Filter* to enable mouse support.
- **default** – The default input text to be shown. (This can be edited by the user).
- **refresh_interval** – (number; in seconds) When given, refresh the UI every so many seconds.
- **inputhook** – None or an Inputhook callable that takes an *InputHookContext* object.

prompt(*message=None*, *default=u''*, *editing_mode=None*, *refresh_interval=None*, *vi_mode=None*, *lexer=None*, *completer=None*, *complete_in_thread=None*, *is_password=None*, *key_bindings=None*, *bottom_toolbar=None*, *style=None*, *color_depth=None*, *include_default_pygments_style=None*, *rprompt=None*, *multiline=None*, *prompt_continuation=None*, *wrap_lines=None*, *enable_history_search=None*, *search_ignore_case=None*, *complete_while_typing=None*, *validate_while_typing=None*, *complete_style=None*, *auto_suggest=None*, *validator=None*, *clipboard=None*, *mouse_support=None*, *input_processors=None*, *reserve_space_for_menu=None*, *enable_system_prompt=None*, *enable_suspend=None*, *enable_open_in_editor=None*, *tempfile_suffix=None*, *inputhook=None*, *async_=False*, *accept_default=False*)

Display the prompt. All the arguments are a subset of the *PromptSession* class itself.

This will raise KeyboardInterrupt when control-c has been pressed (for abort) and EOFError when control-d has been pressed (for exit).

Parameters

- **async** – When *True* return a *Future* instead of waiting for the prompt to finish.
- **accept_default** – When *True*, automatically accept the default value without allowing the user to edit the input.

prompt_toolkit.shortcuts.**confirm**(*message=u'Confirm?'*, *suffix=u' (y/n) '*)

Display a confirmation prompt that returns True/False.

class prompt_toolkit.shortcuts.**CompleteStyle**

How to display autocompletions for the prompt.

prompt_toolkit.shortcuts.**create_confirm_session**(*message*, *suffix=u' (y/n) '*)

Create a *PromptSession* object for the 'confirm' function.

prompt_toolkit.shortcuts.**clear**()

Clear the screen.

prompt_toolkit.shortcuts.**clear_title**()

Erase the current title.

prompt_toolkit.shortcuts.**print_formatted_text**(**values*, ***kwargs*)

```
print_formatted_text(*values, sep=' ', end='\n', file=None, flush=False,␣
↪style=None, output=None)
```

Print text to stdout. This is supposed to be compatible with Python's print function, but supports printing of formatted text. You can pass a *FormattedText*, *HTML* or *ANSI* object to print formatted text.

- Print HTML as follows:

```
print_formatted_text(HTML('<i>Some italic text</i> <ansired>This is red!</
↪ansired>'))

style = Style.from_dict({
    'hello': '#ff0066',
    'world': '#884444 italic',
})
print_formatted_text(HTML('<hello>Hello</hello> <world>world</world>!'),␣
↪style=style)
```

- Print a list of (style_str, text) tuples in the given style to the output. E.g.:

```
style = Style.from_dict({
    'hello': '#ff0066',
    'world': '#884444 italic',
})
fragments = FormattedText([
    ('class:hello', 'Hello'),
    ('class:world', 'World'),
])
print_formatted_text(fragments, style=style)
```

If you want to print a list of Pygments tokens, wrap it in *PygmentsTokens* to do the conversion.

Parameters

- **values** – Any kind of printable object, or formatted string.
- **sep** – String inserted between values, default a space.
- **end** – String appended after the last value, default a newline.
- **style** – *Style* instance for the color scheme.
- **include_default_pygments_style** – *bool*. Include the default Pygments style when set to *True* (the default).

prompt_toolkit.shortcuts.**set_title**(*text*)

Set the terminal title.

class prompt_toolkit.shortcuts.**ProgressBar**(*title=None*, *formatters=None*, *bottom_toolbar=None*, *style=None*, *key_bindings=None*, *file=None*, *color_depth=None*, *output=None*, *input=None*)

Progress bar context manager.

Usage

```
with ProgressBar(...) as pb:
    for item in pb(data):
        ...
```

Parameters

- **title** – Text to be displayed above the progress bars. This can be a callable or formatted text as well.
- **formatters** – List of *Formatter* instances.

- **bottom_toolbar** – Text to be displayed in the bottom toolbar. This can be a callable or formatted text.
- **style** – *prompt_toolkit.styles.BaseStyle* instance.
- **key_bindings** – *KeyBindings* instance.
- **file** – The file object used for rendering, by default *sys.stderr* is used.
- **color_depth** – *prompt_toolkit ColorDepth* instance.
- **output** – *Output* instance.
- **input** – *Input* instance.

prompt_toolkit.shortcuts.**input_dialog**(*title=u''*, *text=u''*, *ok_text=u'OK'*, *cancel_text=u'Cancel'*, *completer=None*, *password=False*, *style=None*, *async_=False*)

Display a text input box. Return the given text, or None when cancelled.

prompt_toolkit.shortcuts.**message_dialog**(*title=u''*, *text=u''*, *ok_text=u'Ok'*, *style=None*, *async_=False*)

Display a simple message box and wait until the user presses enter.

prompt_toolkit.shortcuts.**progress_dialog**(*title=u''*, *text=u''*, *run_callback=None*, *style=None*, *async_=False*)

Parameters **run_callback** – A function that receives as input a *set_percentage* function and it does the work.

prompt_toolkit.shortcuts.**radiolist_dialog**(*title=u''*, *text=u''*, *ok_text=u'Ok'*, *cancel_text=u'Cancel'*, *values=None*, *style=None*, *async_=False*)

Display a simple message box and wait until the user presses enter.

prompt_toolkit.shortcuts.**yes_no_dialog**(*title=u''*, *text=u''*, *yes_text=u'Yes'*, *no_text=u'No'*, *style=None*, *async_=False*)

Display a Yes/No dialog. Return a boolean.

prompt_toolkit.shortcuts.**button_dialog**(*title=u''*, *text=u''*, *buttons=[]*, *style=None*, *async_=False*)

Display a dialog with button choices (given as a list of tuples). Return the value associated with button.

Formatter classes for the progress bar. Each progress bar consists of a list of these formatters.

class prompt_toolkit.shortcuts.progress_bar.formatters.**Formatter**

Base class for any formatter.

class prompt_toolkit.shortcuts.progress_bar.formatters.**Text**(*text*, *style=u''*)

Display plain text.

class prompt_toolkit.shortcuts.progress_bar.formatters.**Label**(*width=None*, *suffix=u''*)

Display the name of the current task.

Parameters

- **width** – If a *width* is given, use this width. Scroll the text if it doesn't fit in this width.
- **suffix** – String suffix to be added after the task name, e.g. ': '. If no task name was given, no suffix will be added.

class prompt_toolkit.shortcuts.progress_bar.formatters.**Percentage**

Display the progress as a percentage.

class prompt_toolkit.shortcuts.progress_bar.formatters.**Bar**(*start=u'[', end=u']', sym_a=u'=', sym_b=u'>', sym_c=u' ', unknown=u'#'*)
Display the progress bar itself.

class prompt_toolkit.shortcuts.progress_bar.formatters.**Progress**
Display the progress as text. E.g. "8/20"

class prompt_toolkit.shortcuts.progress_bar.formatters.**TimeElapsed**
Display the elapsed time.

class prompt_toolkit.shortcuts.progress_bar.formatters.**TimeLeft**
Display the time left.

class prompt_toolkit.shortcuts.progress_bar.formatters.**IterationsPerSecond**
Display the iterations per second.

class prompt_toolkit.shortcuts.progress_bar.formatters.**SpinningWheel**
Display a spinning wheel.

class prompt_toolkit.shortcuts.progress_bar.formatters.**Rainbow**(*formatter*)
For the fun. Add rainbow colors to any of the other formatters.

prompt_toolkit.shortcuts.progress_bar.formatters.**create_default_formatters**()
Return the list of default formatters.

3.11.13 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 occurred.
- **message** – Text.

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

A validator is typically created in one of the following two ways:

- Either by overriding this class and implementing the *validate* method.
- Or by passing a callable to *Validator.from_callable*.

If the validation takes some time and needs to happen in a background thread, this can be wrapped in a *ThreadedValidator*.

classmethod from_callable(*validate_func*, *error_message=u'Invalid input'*, *move_cursor_to_end=False*)
Create a validator from a simple validate callable. E.g.:

```
def is_valid(text):
    return text in ['hello', 'world']
Validator.from_callable(is_valid, error_message='Invalid input')
```

Parameters

- **validate_func** – Callable that takes the input string, and returns *True* if the input is valid input.
- **error_message** – Message to be displayed if the input is invalid.
- **move_cursor_to_end** – Move the cursor to the end of the input, if the input is invalid.

get_validate_future(*document*)
Return a *Future* which is set when the validation is ready. This function can be overloaded in order to provide an asynchronous implementation.

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

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

class prompt_toolkit.validation.**ThreadedValidator**(*validator*)
Wrapper that runs input validation in a thread. (Use this to prevent the user interface from becoming unresponsive if the input validation takes too much time.)

get_validate_future(*document*)
Run the *validate* function in a thread.

class prompt_toolkit.validation.**DummyValidator**
Validator class that accepts any input.

class prompt_toolkit.validation.**DynamicValidator**(*get_validator*)
Validator class that can dynamically returns any Validator.

Parameters **get_validator** – Callable that returns a *Validator* instance.

3.11.14 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.

If you want the auto suggestions to be asynchronous (in a background thread), because they take too much time, and could potentially block the event loop, then wrap the *AutoSuggest* instance into a *ThreadedAutoSuggest*.

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(*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.

get_suggestion_future(*buff*, *document*)
Return a *Future* which is set when the suggestions are ready. This function can be overloaded in order to provide an asynchronous implementation.

class prompt_toolkit.auto_suggest.**ThreadedAutoSuggest**(*auto_suggest*)
Wrapper that runs auto suggestions in a thread. (Use this to prevent the user interface from becoming unresponsive if the generation of suggestions takes too much time.)

get_suggestion_future(*buff*, *document*)
Run the *get_suggestion* function in a thread.

class prompt_toolkit.auto_suggest.**DummyAutoSuggest**
AutoSuggest class that doesn't return any suggestion.

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.

class prompt_toolkit.auto_suggest.**DynamicAutoSuggest**(*get_auto_suggest*)
Validator class that can dynamically returns any Validator.

Parameters **get_validator** – Callable that returns a *Validator* instance.

3.11.15 Renderer

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

class prompt_toolkit.renderer.**Renderer**(*style*, *output*, *full_screen=False*, *mouse_support=False*, *cpr_not_supported_callback=None*)
Typical usage:

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

clear()
Clear screen and go to 0,0

erase(*leave_alternate_screen=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.

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.)

last_rendered_screen
The *Screen* class that was generated during the last rendering. This can be *None*.

render(*app*, *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.

We do this to calculate the minimum available height that we can consume for rendering the prompt. This is the available space below te cursor.

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.

wait_for_cpr_responses(*timeout=1*)
Wait for a CPR response.

waiting_for_cpr
Waiting for CPR flag. True when we send the request, but didn't got a response.

prompt_toolkit.renderer.**print_formatted_text**(*output*, *formatted_text*, *style*, *color_depth=None*)
Print a list of (style_str, text) tuples in the given style to the output.

3.11.16 Lexers

Lexer interface and implementations. Used for syntax highlighting.

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

invalidation_hash()
When this changes, *lex_document* could give a different output. (Only used for *DynamicLexer*.)

lex_document(*document*)
Takes a *Document* and returns a callable that takes a line number and returns a list of `(style_str, text)` tuples for that line.

XXX: Note that in the past, this was supposed to return a list of `(Token, text)` tuples, just like a Pygments lexer.

class prompt_toolkit.lexers.**SimpleLexer**(*style=u''*)
Lexer that doesn't do any tokenizing and returns the whole input as one token.

Parameters **style** – The style string for this lexer.

class prompt_toolkit.lexers.**DynamicLexer**(*get_lexer*)
Lexer class that can dynamically returns any Lexer.

Parameters **get_lexer** – Callable that returns a *Lexer* instance.

class prompt_toolkit.lexers.**PygmentsLexer**(*pygments_lexer_cls*, *sync_from_start=True*, *syntax_sync=None*)

Lexer that calls a pygments lexer.

Example:

```
from pygments.lexers.html 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_cls
from pygments.styles import get_style_by_name
style = style_from_pygments_cls(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(*document*)
Create a lexer function that takes a line number and returns the list of (style_str, text) tuples as the Pygments lexer returns for that line.

class prompt_toolkit.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.

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

class prompt_toolkit.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.)

3.11.17 Layout

The layout class itself

Command line layout definitions

The layout of a command line interface is defined by a Container instance. There are two main groups of classes here. Containers and controls:

- A container can contain other containers or controls, it can have multiple children and it decides about the dimensions.
- A control is responsible for rendering the actual content to a screen. A control can propose some dimensions, but it's the container who decides about the dimensions – or when the control consumes more space – which part of the control will be visible.

Container classes:

```
- Container (Abstract base class)
   |- HSplit (Horizontal split)
   |- VSplit (Vertical split)
   |- FloatContainer (Container which can also contain menus and other floats)
   `- Window (Container which contains one actual control
```

Control classes:

```
- UIControl (Abstract base class)
   |- FormattedTextControl (Renders formatted text, or a simple list of text␣
↪fragments)
   `- BufferControl (Renders an input buffer.)
```

Usually, you end up wrapping every control inside a *Window* object, because that's the only way to render it in a layout.

There are some prepared toolbars which are ready to use:

```
- SystemToolbar (Shows the 'system' input buffer, for entering system commands.)
- ArgToolbar (Shows the input 'arg', for repetition of input commands.)
- SearchToolbar (Shows the 'search' input buffer, for incremental search.)
- CompletionsToolbar (Shows the completions of the current buffer.)
- ValidationToolbar (Shows validation errors of the current buffer.)
```

And one prepared menu:

- CompletionsMenu

class prompt_toolkit.layout.**Layout**(*container*, *focused_element=None*)

The layout for a prompt_toolkit *Application*. This also keeps track of which user control is focused.

Parameters

- **container** – The "root" container for the layout.
- **focused_element** – element to be focused initially. (Can be anything the *focus* function accepts.)

buffer_has_focus

Return *True* if the currently focused control is a *BufferControl*. (For instance, used to determine whether the default key bindings should be active or not.)

current_buffer
The currently focused *Buffer* or *None*.

current_control
Get the *UIControl* to currently has the focus.

current_window
Return the *Window* object that is currently focused.

find_all_windows()
Find all the *UIControl* objects in this layout.

focus(*value*)
Focus the given UI element.

value can be either:

- a *UIControl*
- a *Buffer* instance or the name of a *Buffer*
- a *Window*
- Any container object. In this case we will focus the *Window* from this container that was focused most recent, or the very first focusable *Window* of the container.

focus_last()
Give the focus to the last focused control.

focus_next()
Focus the next visible/focusable Window.

focus_previous()
Focus the previous visible/focusable Window.

get_buffer_by_name(*buffer_name*)
Look in the layout for a buffer with the given name. Return *None* when nothing was found.

get_focusable_windows()
Return all the *Window* objects which are focusable (in the 'modal' area).

get_parent(*container*)
Return the parent container for the given container, or `None`, if it wasn't found.

get_visible_focusable_windows()
Return a list of *Window* objects that are focusable.

has_focus(*value*)
Check whether the given control has the focus. :param value: *UIControl* or *Window* instance.

is_searching
True if we are searching right now.

previous_control
Get the *UIControl* to previously had the focus.

search_target_buffer_control
Return the *BufferControl* in which we are searching or *None*.

update_parents_relations()
Update child->parent relationships mapping.

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

walk_through_modal_area()
Walk through all the containers which are in the current 'modal' part of the layout.

exception prompt_toolkit.layout.**InvalidLayoutError**

prompt_toolkit.layout.**walk**(*container*, *skip_hidden=False*)
Walk through layout, starting at this container.

Containers

Command line layout definitions

The layout of a command line interface is defined by a Container instance. There are two main groups of classes here. Containers and controls:

- A container can contain other containers or controls, it can have multiple children and it decides about the dimensions.
- A control is responsible for rendering the actual content to a screen. A control can propose some dimensions, but it's the container who decides about the dimensions – or when the control consumes more space – which part of the control will be visible.

Container classes:

```
- Container (Abstract base class)
   |- HSplit (Horizontal split)
   |- VSplit (Vertical split)
   |- FloatContainer (Container which can also contain menus and other floats)
   `- Window (Container which contains one actual control
```

Control classes:

```
- UIControl (Abstract base class)
   |- FormattedTextControl (Renders formatted text, or a simple list of text␣
↪fragments)
   `- BufferControl (Renders an input buffer.)
```

Usually, you end up wrapping every control inside a *Window* object, because that's the only way to render it in a layout.

There are some prepared toolbars which are ready to use:

```
- SystemToolbar (Shows the 'system' input buffer, for entering system commands.)
- ArgToolbar (Shows the input 'arg', for repetition of input commands.)
- SearchToolbar (Shows the 'search' input buffer, for incremental search.)
- CompletionsToolbar (Shows the completions of the current buffer.)
- ValidationToolbar (Shows validation errors of the current buffer.)
```

And one prepared menu:

- CompletionsMenu

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

get_children()
Return the list of child *Container* objects.

get_key_bindings()
Returns a *KeyBindings* object. These bindings become active when any user control in this container has the focus, except if any containers between this container and the focused user control is modal.

is_modal()
When this container is modal, key bindings from parent containers are not taken into account if a user control in this container is focused.

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

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

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

write_to_screen(*screen*, *mouse_handlers*, *write_position*, *parent_style*, *erase_bg*, *z_index*)
Write the actual content to the screen.

Parameters

- **screen** – *Screen*
- **mouse_handlers** – MouseHandlers.
- **parent_style** – Style string to pass to the *Window* object. This will be applied to all content of the windows. *VSplit* and *HSplit* can use it to pass their style down to the windows that they contain.
- **z_index** – Used for propagating z_index from parent to child.

class prompt_toolkit.layout.**HSplit**(*children*, *window_too_small=None*, *align=u'JUSTIFY'*, *padding=0*, *padding_char=None*, *padding_style=u''*, *width=None*, *height=None*, *z_index=None*, *modal=False*, *key_bindings=None*, *style=u''*)
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.
- **width** – When given, use this width instead of looking at the children.
- **height** – When given, use this width instead of looking at the children.
- **z_index** – (int or None) When specified, this can be used to bring element in front of floating elements. *None* means: inherit from parent.
- **style** – A style string.
- **modal** – True or False.
- **key_bindings** – None or a *KeyBindings* object.

write_to_screen(*screen*, *mouse_handlers*, *write_position*, *parent_style*, *erase_bg*, *z_index*)
Render the prompt to a *Screen* instance.

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

class prompt_toolkit.layout.**VSplit**(*children*, *window_too_small=None*, *align=u'JUSTIFY'*, *padding=Dimension(min=0, max=0, preferred=0)*, *padding_char=None*, *padding_style=u''*, *width=None*, *height=None*, *z_index=None*, *modal=False*, *key_bindings=None*, *style=u''*)
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.
- **width** – When given, use this width instead of looking at the children.
- **height** – When given, use this width instead of looking at the children.
- **z_index** – (int or None) When specified, this can be used to bring element in front of floating elements. *None* means: inherit from parent.
- **style** – A style string.
- **modal** – `True` or `False`.
- **key_bindings** – `None` or a *KeyBindings* object.

write_to_screen(*screen*, *mouse_handlers*, *write_position*, *parent_style*, *erase_bg*, *z_index*)
Render the prompt to a *Screen* instance.

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

class prompt_toolkit.layout.**FloatContainer**(*content*, *floats*, *modal=False*, *key_bindings=None*, *style=u''*, *z_index=None*)
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(...))
               ])
```

Parameters **z_index** – (int or None) When specified, this can be used to bring element in front of floating elements. *None* means: inherit from parent. This is the z_index for the whole *Float* container as a whole.

preferred_height(*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.)

class prompt_toolkit.layout.**Float**(*content=None*, *top=None*, *right=None*, *bottom=None*, *left=None*, *width=None*, *height=None*, *xcursor=None*, *ycursor=None*, *attach_to_window=None*, *hide_when_covering_content=False*, *allow_cover_cursor=False*, *z_index=1*, *transparent=False*)
Float for use in a *FloatContainer*. Except for the *content* parameter, all other options are optional.

Parameters

- **content** – *Container* instance.
- **width** – *Dimension* or callable which returns a *Dimension*.
- **height** – *Dimension* or callable which returns a *Dimension*.
- **left** – Distance to the left edge of the *FloatContainer*.
- **right** – Distance to the right edge of the *FloatContainer*.
- **top** – Distance to the top of the *FloatContainer*.
- **bottom** – Distance to the bottom of the *FloatContainer*.
- **attach_to_window** – Attach to the cursor from this window, instead of the current window.
- **hide_when_covering_content** – Hide the float when it covers content underneath.
- **allow_cover_cursor** – When *False*, make sure to display the float below the cursor. Not on top of the indicated position.
- **z_index** – Z-index position. For a Float, this needs to be at least one. It is relative to the z_index of the parent container.
- **transparent** – *Filter* indicating whether this float needs to be drawn transparently.

class prompt_toolkit.layout.**Window**(*content=None*, *width=None*, *height=None*, *z_index=None*, *dont_extend_width=False*, *dont_extend_height=False*, *ignore_content_width=False*, *ignore_content_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*, *colorcolumns=None*, *align=u'LEFT'*, *style=u''*, *char=None*)
Container that holds a control.

Parameters

- **content** – *UIControl* instance.
- **width** – *Dimension* instance or callable.
- **height** – *Dimension* instance or callable.
- **z_index** – When specified, this can be used to bring element in front of floating elements.
- **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.
- **ignore_content_width** – A *bool* or *Filter* instance. Ignore the *UIContent* width when calculating the dimensions.
- **ignore_content_height** – A *bool* or *Filter* instance. Ignore the *UIContent* height when calculating the dimensions.
- **left_margins** – A list of *Margin* instance to be displayed on the left. For instance: *NumberedMargin* 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 *Filter* 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 *Filter* 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 *Filter* instance. When True, never display the cursor, even when the user control specifies a cursor position.
- **cursorline** – A *bool* or *Filter* instance. When True, display a cursorline.
- **cursorcolumn** – A *bool* or *Filter* instance. When True, display a cursorcolumn.
- **colorcolumns** – A list of *ColorColumn* instances that describe the columns to be highlighted, or a callable that returns such a list.
- **align** – *WindowAlign* value or callable that returns an *WindowAlign* value. alignment of content.
- **style** – A style string. Style to be applied to all the cells in this window.
- **char** – (string) Character to be used for filling the background. This can also be a callable that returns a character.

preferred_height(*width*, *max_available_height*)
Calculate the preferred height for this window.

preferred_width(*max_available_width*)
Calculate the preferred width for this window.

write_to_screen(*screen*, *mouse_handlers*, *write_position*, *parent_style*, *erase_bg*, *z_index*)
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.**WindowAlign**
Alignment of Window content.

class prompt_toolkit.layout.**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** – *Filter* instance.

class prompt_toolkit.layout.**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.

class prompt_toolkit.layout.**ColorColumn**(*position*, *style='class:color-column'*)
Column for a *Window* to be colored.

prompt_toolkit.layout.**to_container**(*container*)
Make sure that the given object is a *Container*.

prompt_toolkit.layout.**to_window**(*container*)
Make sure that the given argument is a *Window*.

prompt_toolkit.layout.**is_container**(*value*)
Checks whether the given value is a container object (for use in assert statements).

class prompt_toolkit.layout.**HorizontalAlign**
Alignment for *VSplit*.

class prompt_toolkit.layout.**VerticalAlign**
Alignment for *HSplit*.

Controls

Command line layout definitions

The layout of a command line interface is defined by a Container instance. There are two main groups of classes here. Containers and controls:

- A container can contain other containers or controls, it can have multiple children and it decides about the dimensions.
- A control is responsible for rendering the actual content to a screen. A control can propose some dimensions, but it's the container who decides about the dimensions – or when the control consumes more space – which part of the control will be visible.

Container classes:

```
- Container (Abstract base class)
   |- HSplit (Horizontal split)
   |- VSplit (Vertical split)
   |- FloatContainer (Container which can also contain menus and other floats)
   `- Window (Container which contains one actual control
```

Control classes:

```
- UIControl (Abstract base class)
   |- FormattedTextControl (Renders formatted text, or a simple list of text␣
↪fragments)
   `- BufferControl (Renders an input buffer.)
```

Usually, you end up wrapping every control inside a *Window* object, because that's the only way to render it in a layout.

There are some prepared toolbars which are ready to use:

```
- SystemToolbar (Shows the 'system' input buffer, for entering system commands.)
- ArgToolbar (Shows the input 'arg', for repetition of input commands.)
- SearchToolbar (Shows the 'search' input buffer, for incremental search.)
- CompletionsToolbar (Shows the completions of the current buffer.)
- ValidationToolbar (Shows validation errors of the current buffer.)
```

And one prepared menu:

- CompletionsMenu

class prompt_toolkit.layout.**BufferControl**(*buffer=None*, *input_processors=None*, *include_default_input_processors=True*, *lexer=None*, *preview_search=False*, *focusable=True*, *search_buffer_control=None*, *menu_position=None*, *focus_on_click=False*, *key_bindings=None*)

Control for visualising the content of a *Buffer*.

Parameters

- **buffer** – The *Buffer* object to be displayed.
- **input_processors** – A list of *Processor* objects.
- **include_default_input_processors** – When True, include the default processors for highlighting of selection, search and displaying of multiple cursors.
- **lexer** – *Lexer* instance for syntax highlighting.
- **preview_search** – *bool* or *Filter*: Show search while typing. When this is *True*, probably you want to add a `HighlightIncrementalSearchProcessor` as well. Otherwise only the cursor position will move, but the text won't be highlighted.
- **focusable** – *bool* or *Filter*: Tell whether this control is focusable.
- **focus_on_click** – Focus this buffer when it's click, but not yet focused.
- **key_bindings** – a *KeyBindings* object.

create_content(*width*, *height*, *preview_search=False*)

Create a UIContent.

get_invalidate_events()

Return the Window invalidate events.

get_key_bindings()

When additional key bindings are given. Return these.

mouse_handler(*mouse_event*)

Mouse handler for this control.

preferred_width(*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.

search_state
Return the *SearchState* for searching this *BufferControl*. This is always associated with the search control. If one search bar is used for searching multiple *BufferControls*, then they share the same *SearchState*.

class prompt_toolkit.layout.**SearchBufferControl**(*buffer=None*, *input_processors=None*, *lexer=None*, *focus_on_click=False*, *key_bindings=None*, *ignore_case=False*)
BufferControl which is used for searching another *BufferControl*.

Parameters **ignore_case** – Search case insensitive.

class prompt_toolkit.layout.**DummyControl**
A dummy control object that doesn't paint any content.

Useful for filling a *Window*. (The *fragment* and *char* attributes of the *Window* class can be used to define the filling.)

class prompt_toolkit.layout.**FormattedTextControl**(*text=u''*, *style=u''*, *focusable=False*, *key_bindings=None*, *show_cursor=True*, *modal=False*, *get_cursor_position=None*)
Control that displays formatted text. This can be either plain text, an *HTML* object an *ANSI* object or a list of `(style_str, text)` tuples, depending on how you prefer to do the formatting. See `prompt_toolkit.layout.formatted_text` for more information.

(It's mostly optimized for rather small widgets, like toolbars, menus, etc...)

When this UI control has the focus, the cursor will be shown in the upper left corner of this control by default. There are two ways for specifying the cursor position:

- Pass a *get_cursor_position* function which returns a *Point* instance with the current cursor position.
- If the (formatted) text is passed as a list of `(style, text)` tuples and there is one that looks like `('[SetCursorPosition]', '')`, then this will specify the cursor position.

Mouse support:

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

Parameters

- **focusable** – *bool* or *Filter*: Tell whether this control is focusable.
- **text** – Text or formatted text to be displayed.
- **style** – Style string applied to the content. (If you want to style the whole *Window*, pass the style to the *Window* instead.)
- **key_bindings** – a *KeyBindings* object.
- **get_cursor_position** – A callable that returns the cursor position as a *Point* instance.

mouse_handler(*mouse_event*)
Handle mouse events.

(When the fragment 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(*max_available_width*)
Return the preferred width for this control. That is the width of the longest line.

class prompt_toolkit.layout.**UIControl**
Base class for all user interface controls.

create_content(*width*, *height*)
Generate the content for this user control.

Returns a *UIContent* instance.

get_invalidate_events()
Return a list of *Event* objects. This can be a generator. (The application collects all these events, in order to bind redraw handlers to these events.)

get_key_bindings()
The key bindings that are specific for this user control.

Return a *KeyBindings* object if some key bindings are specified, or *None* otherwise.

is_focusable()
Tell whether this user control is focusable.

mouse_handler(*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 **mouse_event** – *MouseEvent* instance.

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

move_cursor_up()
Request to move the cursor up.

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

Parameters

- **get_line** – Callable that takes a line number and returns the current line. This is a list of (style_str, 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.

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.

Other

Command line layout definitions

The layout of a command line interface is defined by a Container instance. There are two main groups of classes here. Containers and controls:

- A container can contain other containers or controls, it can have multiple children and it decides about the dimensions.
- A control is responsible for rendering the actual content to a screen. A control can propose some dimensions, but it's the container who decides about the dimensions – or when the control consumes more space – which part of the control will be visible.

Container classes:

```
- Container (Abstract base class)
   |- HSplit (Horizontal split)
   |- VSplit (Vertical split)
   |- FloatContainer (Container which can also contain menus and other floats)
   `- Window (Container which contains one actual control
```

Control classes:

```
- UIControl (Abstract base class)
   |- FormattedTextControl (Renders formatted text, or a simple list of text␣
↪fragments)
   `- BufferControl (Renders an input buffer.)
```

Usually, you end up wrapping every control inside a *Window* object, because that's the only way to render it in a layout.

There are some prepared toolbars which are ready to use:

```
- SystemToolbar (Shows the 'system' input buffer, for entering system commands.)
- ArgToolbar (Shows the input 'arg', for repetition of input commands.)
- SearchToolbar (Shows the 'search' input buffer, for incremental search.)
- CompletionsToolbar (Shows the completions of the current buffer.)
- ValidationToolbar (Shows validation errors of the current buffer.)
```

And one prepared menu:

- CompletionsMenu

class prompt_toolkit.layout.**Dimension**(*min=None*, *max=None*, *weight=None*, *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 *Dimension* with an exact size. (min, max and preferred set to `amount`).

is_zero()
True if this *Dimension* represents a zero size.

classmethod zero()
Create a dimension that represents a zero size. (Used for 'invisible' controls.)

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

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

Parameters

- **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(*get_ui_content*)
Return the width that this margin is going to consume.

Parameters **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.**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.**ScrollbarMargin**(*display_arrows=False*)
Margin displaying a scrollbar.

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

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

class prompt_toolkit.layout.**PromptMargin**(*get_prompt*, *get_continuation=None*)
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** – Callable returns formatted text or a list of *(style_str, type)* tuples to be shown as the prompt at the first line.
- **get_continuation** – Callable that takes three inputs. The width (int), line_number (int), and is_soft_wrap (bool). It should return formatted text or a list of *(style_str, type)* tuples for the next lines of the input.

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

class prompt_toolkit.layout.**MultiColumnCompletionsMenu**(*min_rows=3*, *suggested_max_column_width=30*, *show_meta=True*, *extra_filter=True*, *z_index=100000000*)

Container that displays the completions in several columns. When *show_meta* (a *Filter*) evaluates to True, it shows the meta information at the bottom.

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

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

class prompt_toolkit.layout.processors.**Processor**

Manipulate the fragments for a given line in a BufferControl.

apply_transformation(*transformation_input*)

Apply transformation. Returns a *Transformation* instance.

Parameters **transformation_input** – *TransformationInput* object.

class prompt_toolkit.layout.processors.**TransformationInput**(*buffer_control*, *document*, *lineno*, *source_to_display*, *fragments*, *width*, *height*)

Parameters

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

class prompt_toolkit.layout.processors.**Transformation**(*fragments*, *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 *fragments*!

Parameters

- **fragments** – The transformed fragments. 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.**DummyProcessor**

A *Processor* that doesn't do anything.

class prompt_toolkit.layout.processors.**HighlightSearchProcessor**

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

The style classes 'search' and 'search.current' will be applied to the content.

class prompt_toolkit.layout.processors.**HighlightIncrementalSearchProcessor**
Highlight the search terms that are used for highlighting the incremental search. The style class 'incsearch' will be applied to the content.

Important: this requires the *preview_search=True* flag to be set for the *BufferControl*. Otherwise, the cursor position won't be set to the search match while searching, and nothing happens.

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

class prompt_toolkit.layout.processors.**BeforeInput**(*text*, *style=u''*)
Insert text before the input.

Parameters

- **text** – This can be either plain text or formatted text (or a callable that returns any of those).
- **style** – style to be applied to this prompt/prefix.

class prompt_toolkit.layout.processors.**AfterInput**(*text*, *style=u''*)
Insert text after the input.

Parameters

- **text** – This can be either plain text or formatted text (or a callable that returns any of those).
- **style** – style to be applied to this prompt/prefix.

class prompt_toolkit.layout.processors.**AppendAutoSuggestion**(*style=u'class:auto-suggestion'*)
Append the auto suggestion to the input. (The user can then press the right arrow the insert the suggestion.)

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():
    return true_or_false

# Wrapped it in a `ConditionalProcessor` for usage in a `BufferControl`.
BufferControl(input_processors=[
```

(continues on next page)

(continued from previous page)

```
    ConditionalProcessor(HighlightSearchProcessor(),
                         Condition(highlight_enabled))])
```

Parameters

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

class prompt_toolkit.layout.processors.**ShowLeadingWhiteSpaceProcessor**(*get_char=None*, *style=u'class:leading-whitespace'*)

Make leading whitespace visible.

Parameters **get_char** – Callable that returns one character.

class prompt_toolkit.layout.processors.**ShowTrailingWhiteSpaceProcessor**(*get_char=None*, *style=u'class:training-whitespace'*)

Make trailing whitespace visible.

Parameters **get_char** – Callable that returns one character.

class prompt_toolkit.layout.processors.**TabsProcessor**(*tabstop=4*, *char1=u'|'*, *char2=u'u2508'*, *style=u'class:tab'*)

Render tabs as spaces (instead of ^I) or make them visible (for instance, by replacing them with dots.)

Parameters

- **tabstop** – Horizontal space taken by a tab. (*int* or callable that returns an *int*).
- **char1** – Character or callable that returns a character (text of length one). This one is used for the first space taken by the tab.
- **char2** – Like *char1*, but for the rest of the space.

class prompt_toolkit.layout.processors.**ReverseSearchProcessor**

Process to display the "(reverse-i-search)'...': ..." stuff around the search buffer.

Note: This processor is meant to be applied to the BufferControl that contains the search buffer, it's not meant for the original input.

class prompt_toolkit.layout.processors.**DynamicProcessor**(*get_processor*)

Processor class that can dynamically returns any Processor.

Parameters **get_processor** – Callable that returns a *Processor* instance.

prompt_toolkit.layout.processors.**merge_processors**(*processors*)

Merge multiple *Processor* objects into one.

prompt_toolkit.layout.utils.**explode_text_fragments**(*fragments*)

Turn a list of (style_str, 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 **fragments** – List of (style, text) tuples.

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

x
Alias for field number 0

y
Alias for field number 1

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 dimensional buffer of *Char* instances.

append_style_to_content(*style_str*)
For all the characters in the screen. Set the style string to the given *style_str*.

draw_all_floats()
Draw all float functions in order of z-index.

draw_with_z_index(*z_index*, *draw_func*)
Add a draw-function for a *Window* which has a >= 0 z_index. This will be postponed until *draw_all_floats* is called.

fill_area(*write_position*, *style=u''*, *after=False*)
Fill the content of this area, using the given *style*. The style is prepended before whatever was here before.

get_cursor_position(*window*)
Get the cursor position for a given window. Returns a *Point*.

get_menu_position(*window*)
Get the menu position for a given window. (This falls back to the cursor position if no menu position was set.)

set_cursor_position(*window*, *position*)
Set the cursor position for a given window.

set_menu_position(*window*, *position*)
Set the cursor position for a given window.

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

This should be considered immutable.

Parameters

- **char** – A single character (can be a double-width character).
- **style** – A style string. (Can contain classnames.)

3.11.18 Widgets

Collection of reusable components for building full screen applications. These are higher level abstractions on top of the *prompt_toolkit.layout* module.

Most of these widgets implement the __pt_container__ method, which makes it possible to embed these in the layout like any other container.

class prompt_toolkit.widgets.**TextArea**(*text=u''*, *multiline=True*, *password=False*, *lexer=None*, *completer=None*, *accept_handler=None*, *focusable=True*, *wrap_lines=True*, *read_only=False*, *width=None*, *height=None*, *dont_extend_height=False*, *dont_extend_width=False*, *line_numbers=False*, *scrollbar=False*, *style=u''*, *search_field=None*, *preview_search=True*, *prompt=u''*)

A simple input field.

This contains a prompt_toolkit *Buffer* object that hold the text data structure for the edited buffer, the *BufferControl*, which applies a *Lexer* to the text and turns it into a *UIControl*, and finally, this *UIControl* is wrapped in a *Window* object (just like any *UIControl*), which is responsible for the scrolling.

This widget does have some options, but it does not intend to cover every single use case. For more configurations options, you can always build a text area manually, using a *Buffer*, *BufferControl* and *Window*.

Parameters

- **text** – The initial text.
- **multiline** – If True, allow multiline input.
- **lexer** – *Lexer* instance for syntax highlighting.
- **completer** – *Completer* instance for auto completion.
- **focusable** – When *True*, allow this widget to receive the focus.
- **wrap_lines** – When *True*, don't scroll horizontally, but wrap lines.
- **width** – Window width. (*Dimension* object.)
- **height** – Window height. (*Dimension* object.)
- **password** – When *True*, display using asterisks.
- **accept_handler** – Called when *Enter* is pressed.
- **scrollbar** – When *True*, display a scroll bar.
- **search_field** – An optional *SearchToolbar* object.
- **style** – A style string.
- **dont_extend_height** –
- **dont_extend_width** –

class prompt_toolkit.widgets.**Label**(*text*, *style=u''*, *width=None*, *dont_extend_height=True*, *dont_extend_width=False*)

Widget that displays the given text. It is not editable or focusable.

Parameters

- **text** – The text to be displayed. (This can be multiline. This can be formatted text as well.)
- **style** – A style string.
- **width** – When given, use this width, rather than calculating it from the text size.

class prompt_toolkit.widgets.**Button**(*text*, *handler=None*, *width=12*)

Clickable button.

Parameters

- **text** – The caption for the button.

- **handler** – *None* or callable. Called when the button is clicked.
- **width** – Width of the button.

class prompt_toolkit.widgets.**Frame**(*body*, *title=u"*, *style=u"*, *width=None*, *height=None*, *key_bindings=None*, *modal=False*)

Draw a border around any container, optionally with a title text.

Changing the title and body of the frame is possible at runtime by assigning to the *body* and *title* attributes of this class.

Parameters

- **body** – Another container object.
- **title** – Text to be displayed in the top of the frame (can be formatted text).
- **style** – Style string to be applied to this widget.

class prompt_toolkit.widgets.**Shadow**(*body*)

Draw a shadow underneath/behind this container. (This applies *class:shadow* the the cells under the shadow. The Style should define the colors for the shadow.)

Parameters **body** – Another container object.

class prompt_toolkit.widgets.**Box**(*body*, *padding=None*, *padding_left=None*, *padding_right=None*, *padding_top=None*, *padding_bottom=None*, *width=None*, *height=None*, *style=u"*, *char=None*, *modal=False*, *key_bindings=None*)

Add padding around a container.

This also makes sure that the parent can provide more space than required by the child. This is very useful when wrapping a small element with a fixed size into a `VSplit` or `HSplit` object. The `HSplit` and `VSplit` try to make sure to adapt respectively the width and height, possibly shrinking other elements. Wrapping something in a `Box` makes it flexible.

Parameters

- **body** – Another container object.
- **padding** – The margin to be used around the body. This can be overridden by *padding_left*, padding_right‘, *padding_top* and *padding_bottom*.
- **style** – A style string.
- **char** – Character to be used for filling the space around the body. (This is supposed to be a character with a terminal width of 1.)

class prompt_toolkit.widgets.**VerticalLine**

A simple vertical line with a width of 1.

class prompt_toolkit.widgets.**HorizontalLine**

A simple horizontal line with a height of 1.

class prompt_toolkit.widgets.**RadioList**(*values*)

List of radio buttons. Only one can be checked at the same time.

Parameters **values** – List of (value, label) tuples.

class prompt_toolkit.widgets.**SearchToolbar**(*search_buffer=None*, *vi_mode=False*, *text_if_not_searching=u"*, *forward_search_prompt=u'I-search: '*, *backward_search_prompt=u'I-search backward: '*, *ignore_case=False*)

Parameters

- **vi_mode** – Display '/' and '?' instead of I-search.
- **ignore_case** – Search case insensitive.

class prompt_toolkit.widgets.**SystemToolbar**(*prompt=u'Shell command: '*, *enable_global_bindings=True*)

Toolbar for a system prompt.

Parameters **prompt** – Prompt to be displayed to the user.

class prompt_toolkit.widgets.**MenuContainer**(*body*, *menu_items=None*, *floats=None*, *key_bindings=None*)

Parameters

- **floats** – List of extra Float objects to display.
- **menu_items** – List of *MenuItem* objects.

3.11.19 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.

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

```
filter = has_focus('default') & ~ has_selection
```

class prompt_toolkit.filters.**Filter**

Base class for any filter to activate/deactivate a feature, depending on a condition.

The return value of `__call__` will tell if the feature should be active.

class prompt_toolkit.filters.**Condition**(*func*)

Turn any callable into a Filter. The callable is supposed to not take any arguments.

This can be used as a decorator:

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

Parameters **func** – Callable which takes no inputs and returns a boolean.

prompt_toolkit.filters.utils.**to_filter**(*bool_or_filter*)

Accept both booleans and Filters as input and turn it into a Filter.

Filters that accept a *Application* as argument.

prompt_toolkit.filters.app.**has_focus**(**a*, ***kw*)

Enable when this buffer has the focus.

prompt_toolkit.filters.app.**in_editing_mode**(**a*, ***kw*)

Check whether a given editing mode is active. (Vi or Emacs.)

3.11.20 Key binding

class prompt_toolkit.key_binding.**KeyBindingsBase**
Interface for a KeyBindings.

get_bindings_for_keys(*keys*)
Return a list of key bindings that can handle these keys. (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.

class prompt_toolkit.key_binding.**KeyBindings**
A container for a set of key bindings.

Example usage:

```
kb = KeyBindings()

@kb.add('c-t')
def _(event):
    print('Control-T pressed')

@kb.add('c-a', 'c-b')
def _(event):
    print('Control-A pressed, followed by Control-B')

@kb.add('c-x', filter=is_searching)
def _(event):
    print('Control-X pressed')  # Works only if we are searching.
```

add(**keys*, ***kwargs*)
Decorator for adding a key bindings.

Parameters

- **filter** – *Filter* to determine when this key binding is active.
- **eager** – *Filter* 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.
- **is_global** – When this key bindings is added to a *Container* or *Control*, make it a global (always active) binding.
- **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.)
- **record_in_macro** – Record these key bindings when a macro is being recorded. (True by default.)

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

Parameters

- **filter** – *Filter* to determine when this key binding is active.

- **eager** – *Filter* 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.
- **is_global** – When this key bindings is added to a *Container* or *Control*, make it a global (always active) binding.
- **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.)
- **record_in_macro** – Record these key bindings when a macro is being recorded. (True by 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(**args*)
Remove a key binding.

This expects either a function that was given to *add* method as parameter or a sequence of key bindings.

Raises *ValueError* when no bindings was found.

Usage:

```
remove(handler)  # Pass handler.
remove('c-x', 'c-a')  # Or pass the key bindings.
```

remove_binding(**args*)
Remove a key binding.

This expects either a function that was given to *add* method as parameter or a sequence of key bindings.

Raises *ValueError* when no bindings was found.

Usage:

```
remove(handler)  # Pass handler.
remove('c-x', 'c-a')  # Or pass the key bindings.
```

class prompt_toolkit.key_binding.**ConditionalKeyBindings**(*key_bindings*, *filter=True*)
Wraps around a *KeyBindings*. Disable/enable all the key bindings according to the given (additional) filter.:

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

registry = ConditionalKeyBindings(key_bindings, 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 *KeyBindings* objects.
- **filter** – *Filter* object.

prompt_toolkit.key_binding.**merge_key_bindings**(*bindings*)
Merge multiple Keybinding objects together.

Usage:

```
bindings = merge_key_bindings([bindings1, bindings2, ...])
```

class prompt_toolkit.key_binding.**DynamicKeyBindings**(*get_key_bindings*)
KeyBindings class that can dynamically returns any KeyBindings.

Parameters **get_key_bindings** – Callable that returns a *KeyBindings* instance.

Default key bindings.:

```
key_bindings = load_key_bindings()
app = Application(key_bindings=key_bindings)
```

prompt_toolkit.key_binding.defaults.**load_key_bindings**()
Create a KeyBindings object that contains the default key bindings.

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

input_mode
Get *InputMode*.

reset()
Reset state, go back to the given mode. INSERT by default.

3.11.21 Eventloop

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

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

add_win32_handle(*handle*, *callback*)
Add a Windows Handle to the event loop. (Only applied to win32 loops.)

call_exception_handler(*context*)
Call the current event loop exception handler. (Similar to asyncio.BaseEventLoop.call_exception_handler.)

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 internal 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.

create_future()
Create a *Future* object that is attached to this loop. This is the preferred way of creating futures.

default_exception_handler(*context*)
Default exception handling.

Thanks to asyncio for this function!

get_exception_handler()
Return the exception handler.

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

remove_win32_handle(*handle*)
Remove a Windows Handle from the event loop. (Only applied to win32 loops.)

run_forever(*inputhook=None*)
Run loop forever.

run_in_executor(*callback*, *_daemon=False*)
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`.

run_until_complete(*future*, *inputhook=None*)
Keep running until this future has been set. Return the Future's result, or raise its exception.

set_exception_handler(*handler*)
Set the exception handler.

prompt_toolkit.eventloop.**get_traceback_from_context**(*context*)
Get the traceback object from the context.

prompt_toolkit.eventloop.**From**(*obj*)
Used to emulate 'yield from'. (Like Trollius does.)

exception prompt_toolkit.eventloop.**Return**(*value*)
For backwards-compatibility with Python2: when "return" is not supported in a generator/coroutine. (Like Trollius.)

Instead of `return value`, in a coroutine do: `raise Return(value)`.

prompt_toolkit.eventloop.**ensure_future**(*future_or_coroutine*)
Take a coroutine (generator) or a *Future* object, and make sure to return a *Future*.

prompt_toolkit.eventloop.**create_event_loop**(*recognize_win32_paste=True*)
Create and return an `EventLoop` instance.

prompt_toolkit.eventloop.**create_asyncio_event_loop**(*loop=None*)
Returns an asyncio *EventLoop* instance for usage in a *Application*. It is a wrapper around an asyncio loop.

Parameters **loop** – The asyncio eventloop (or *None* if the default asyncioloop should be used.)

prompt_toolkit.eventloop.**use_asyncio_event_loop**(*loop=None*)
Use the asyncio event loop for prompt_toolkit applications.

prompt_toolkit.eventloop.**get_event_loop**()
Return the current event loop. This will create a new loop if no loop was set yet.

prompt_toolkit.eventloop.**set_event_loop**(*loop*)
Set the current event loop.

Parameters **loop** – *EventLoop* instance or None. (Pass *None* to clear the current loop.)

prompt_toolkit.eventloop.**run_in_executor**(*callback*, *_daemon=False*)
Run a long running function in a background thread.

prompt_toolkit.eventloop.**call_from_executor**(*callback*, *_max_postpone_until=None*)
Call this function in the main event loop.

prompt_toolkit.eventloop.**run_until_complete**(*future*, *inputhook=None*)
Keep running until this future has been set. Return the Future's result, or raise its exception.

class prompt_toolkit.eventloop.**Future**(*loop=None*)
Future object for use with the prompt_toolkit event loops. (Not by accident very similar to asyncio – but much more limited in functionality. They are however not meant to be used interchangeable.)

add_done_callback(*callback*)
Add a callback to be run when the future becomes done. (This callback will be called with one argument only: this future object.)

done()
Return True if the future is done. Done means either that a result / exception are available, or that the future was cancelled.

exception()
Return the exception that was set on this future.

classmethod fail(*result*)
Returns a Future for which the error has been set to the given result. Similar to Twisted's *Deferred.fail()*.

classmethod from_asyncio_future(*asyncio_f*, *loop=None*)
Return a prompt_toolkit *Future* from the given asyncio Future.

result()
Return the result this future represents.

set_exception(*exception*)
Mark the future done and set an exception.

set_result(*result*)
Mark the future done and set its result.

classmethod succeed(*result*)
Returns a Future for which the result has been set to the given result. Similar to Twisted's *Deferred.succeed()*.

to_asyncio_future()
Turn this *Future* into an asyncio *Future* object.

exception prompt_toolkit.eventloop.**InvalidStateError**
The operation is not allowed in this state.

class prompt_toolkit.eventloop.posix.**PosixEventLoop**(*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.

add_signal_handler(*signum*, *handler*)
Register a signal handler. Call *handler* when *signal* was received. The given handler will always be called in the same thread as the eventloop. (Like *call_from_executor*.)

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 internal 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.)

close()
Close the event loop. The loop must not be running.

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

run_in_executor(*callback*, *_daemon=False*)
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`.

run_until_complete(*future*, *inputhook=None*)
Keep running the event loop until *future* has been set.

Parameters **future** – `prompt_toolkit.eventloop.future.Future` object.

3.11.22 Input

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

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

attach(*input_ready_callback*)
Return a context manager that makes this input active in the current event loop.

close()
Close input.

closed
Should be true when the input stream is closed.

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

detach()
Return a context manager that makes sure that this input is not active in the current event loop.

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

flush()
The event loop can call this when the input has to be flushed.

flush_keys()
Flush the underlying parser. and return the pending keys. (Used for vt100 input.)

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

read_keys()
Return a list of Key objects which are read/parsed from the input.

responds_to_cpr
True if the *Application* can expect to receive a CPR response from here.

typeahead_hash()
Identifier for storing type ahead key presses.

class prompt_toolkit.input.**DummyInput**
Input for use in a *DummyApplication*

prompt_toolkit.input.**get_default_input**()
Get the input class to be used by default.

Called when creating a new Application(), when no *Input* has been passed.

prompt_toolkit.input.**set_default_input**(*input*)
Set the default *Output* class.

(Used for instance, for the telnet submodule.)

class prompt_toolkit.input.vt100.**Vt100Input**(*stdin*)
Vt100 input for Posix systems. (This uses a posix file descriptor that can be registered in the event loop.)

attach(*input_ready_callback*)
Return a context manager that makes this input active in the current event loop.

detach()
Return a context manager that makes sure that this input is not active in the current event loop.

flush_keys()
Flush pending keys and return them. (Used for flushing the 'escape' key.)

read_keys()
Read list of KeyPress.

class prompt_toolkit.input.vt100.**raw_mode**(*fileno*)

```
with raw_mode(stdin):
    ''' the pseudo-terminal stdin is now used in raw mode '''
```

We ignore errors when executing *tcgetattr* fails.

class prompt_toolkit.input.vt100.**cooked_mode**(*fileno*)
The opposite of raw_mode, used when we need cooked mode inside a *raw_mode* block. Used in *Application.run_in_terminal*.:

```
with cooked_mode(stdin):
    ''' the pseudo-terminal stdin is now used in cooked mode. '''
```

3.11.23 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.

scroll_buffer_to_prompt()
For Win32 only.

set_attributes(*attrs*, *color_depth*)
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.

class prompt_toolkit.output.**DummyOutput**
For testing. An output class that doesn't render anything.

fileno()
There is no sensible default for fileno().

class prompt_toolkit.output.**ColorDepth**
Possible color depth values for the output.

classmethod default(*term=u''*)
If the user doesn't specify a color depth, use this as a default.

prompt_toolkit.output.**create_output**(*stdout=None*)
Return an *Output* instance for the command line.

Parameters **stdout** – The stdout object

prompt_toolkit.output.**get_default_output**()
Get the output class to be used by default.

Called when creating a new Application(), when no *Output* has been passed.

prompt_toolkit.output.**set_default_output**(*output*)
Set the default *Output* class.

(Used for instance, for the telnet submodule.)

Output for vt100 terminals.

A lot of thanks, regarding outputting of colors, goes to the Pygments project: (We don't rely on Pygments anymore, because many things are very custom, and everything has been highly optimized.) http://pygments.org/

class prompt_toolkit.output.vt100.**Vt100_Output**(*stdout*, *get_size*, *term=None*, *write_binary=True*)

Parameters

- **get_size** – A callable which returns the *Size* of the output terminal.
- **stdout** – Any object with has a *write* and *flush* method + an 'encoding' property.
- **term** – The terminal environment variable. (xterm, xterm-256color, linux, ...)
- **write_binary** – Encode the output before writing it. If *True* (the default), the *stdout* object is supposed to expose an *encoding* attribute.

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

bell()
Sound bell.

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

encoding()
Return encoding used for stdout.

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 file descriptor.

flush()
Write to output stream and flush.

classmethod from_pty(*stdout*, *term=None*)
Create an Output class from a pseudo terminal. (This will take the dimensions by reading the pseudo terminal attributes.)

set_attributes(*attrs*, *color_depth*)
Create new style and output.

Parameters **attrs** – *Attrs* instance.

set_title(*title*)
Set terminal title.

write(*data*)
Write text to output. (Removes vt100 escape codes. – used for safely writing text.)

write_raw(*data*)
Write raw data to output.

3.11.24 Patch stdout

patch_stdout

This implements a context manager that ensures that print statements within it won't destroy the user interface. The context manager will replace *sys.stdout* by something that draws the output above the current prompt, rather than overwriting the UI.

Usage:

```
with patch_stdout():
    ...
    application.run()
    ...
```

Multiple applications can run in the body of the context manager, one after the other.

prompt_toolkit.patch_stdout.**patch_stdout**(**args*, ***kwds*)
Replace *sys.stdout* by an _StdoutProxy instance.

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. If no application is curring, the behaviour should be identical to writing to *sys.stdout* directly.

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

class prompt_toolkit.patch_stdout.**StdoutProxy**(*raw=False*, *original_stdout=None*)

Proxy object for stdout which captures everything and prints output above the current application.

flush()

Flush buffered output.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Prompt_toolkit was created by Jonathan Slenders.

Python Module Index

p

Index

D

E

F

G

H

I

J

K

Q

R

S

T

U

V

W

X

Y

Z